

milan@datajoin.net

<http://datajoin.net>

DataJoin

Special purpose AI

[Document subtitle]

Milan Patel

5-2-2025

Table of

Contents

Overview	2
Flowchart	3
Special purpose AI setup	5
Download models	5
Download models using python	5
Download models directly from huggingface website	5
Download models using LM Studio.....	5
How to use models in python to create prompt	6
Add documents in private chroma store	6
Delete documents in private chroma store	7
Use private chroma store to create prompt for AI	7
Special purpose AI concepts	10
Private vector store	10
Prompt size	10
Relevant documents from vector store for your question	12
Relevant documents from vector store as prompt and your question	13
To include more data from vector store	13
Vector store for prompt vs fine tuning AI	15
Inference vs test	17

Overview

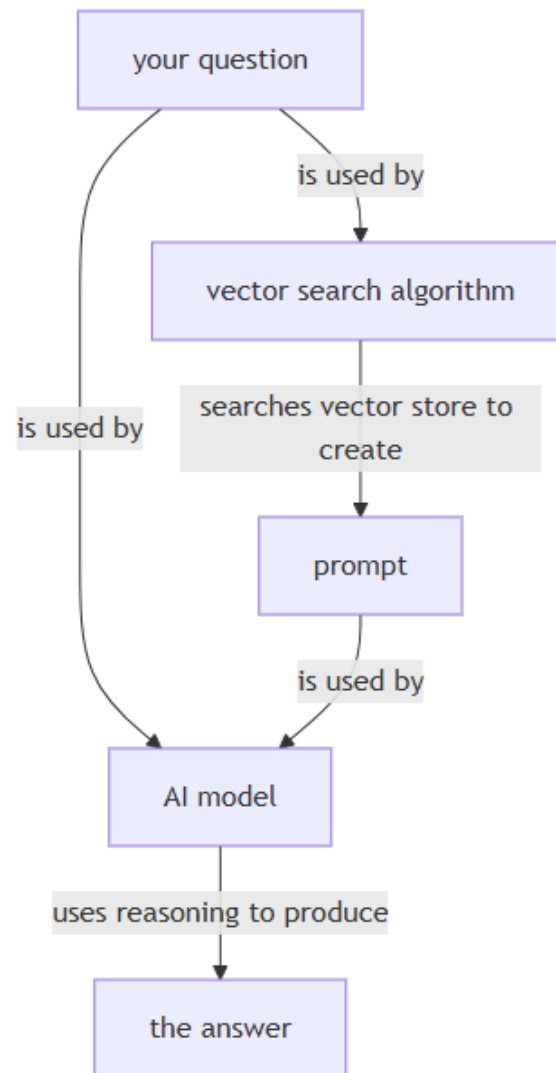
Problem: All AI Models are designed to store your questions and prompts as part of the learning process. That means your information will not be private, it will be part of global knowledge, and it can be given to other users. This is a problem for the Government or Finance or Medical industry, since it is very difficult to hide/mask personal information during interaction with AI. Also, it is very difficult to force the hosted AI model to use specific information from your organization to answer the question. AI will hallucinate and may not provide focused answers.

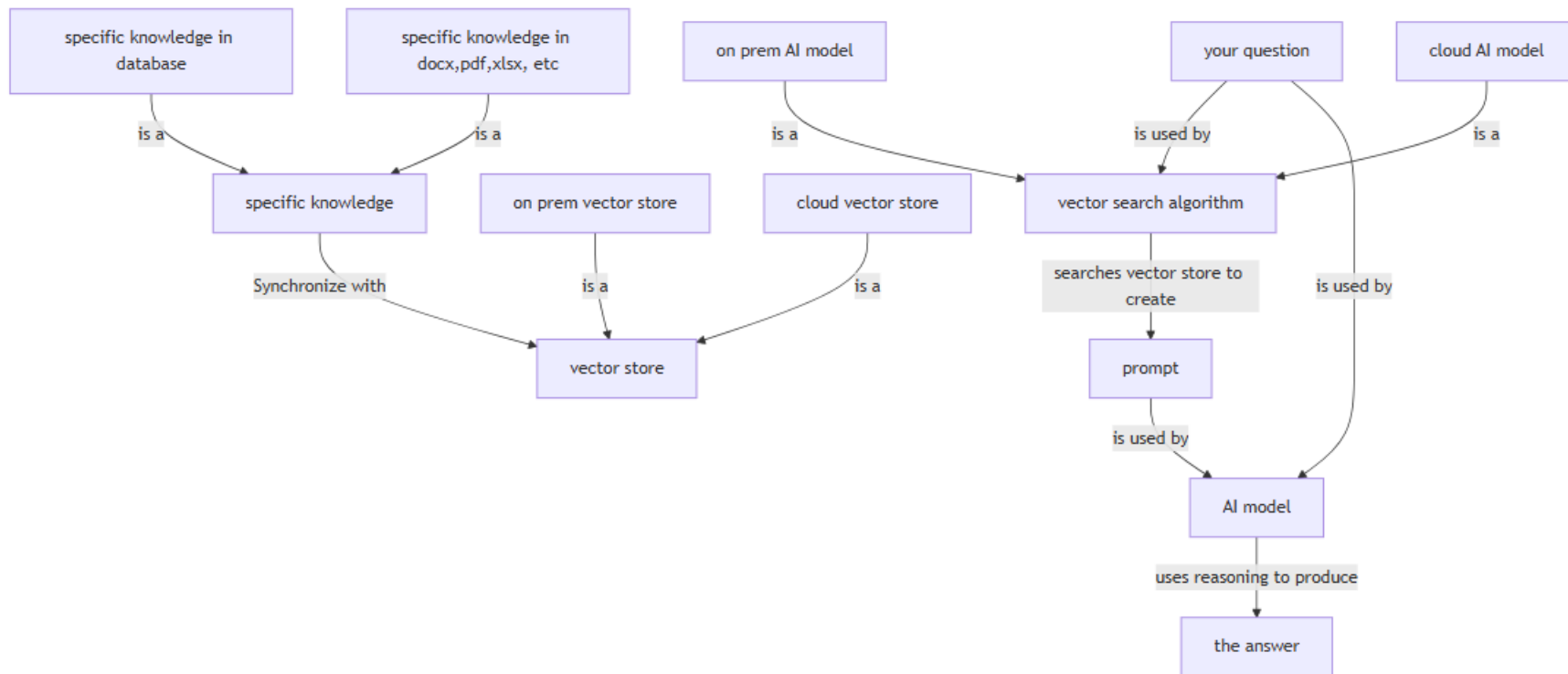
Solution: Special purpose private AI model can use your organization's knowledge to answer your question. Store your organization information in vector store. Then, use the vector store to create prompt that can be sent to local AI model or cloud AI model along with a question.

DataJoin.net provides in-depth education and consultation on special purpose AI model.

<https://github.com/milan888-design/special-purpose-ai>

Flowchart





- One can decide how to separate data in different vector stores. For example, one vector store for HR policy and another for technical information.
- One can use local AI model or hosted AI model on cloud. This depends upon how much privacy you need for your prompts and or your question to AI model.
- It is possible to use different AI models depending upon speed and accuracy needed.

Special purpose AI setup

Download models

The following bullet list came from Google's Gemini AI.

Why Download (or Local Cache Access) is Useful

- Offline Access: Once downloaded, you don't need an internet connection to load and use the model.
- Version Control: You can be absolutely sure which specific version (commit hash) of the model you are using.
- Custom Modifications: If you make any local modifications to the model or tokenizer files, you can load your modified version.
- Avoiding Re-downloads: If the model is already in your cache, pointing to that path avoids re-downloading it.

Download models using python

Use `test_vector_download_model.py` to download a model. It is not necessary to download models. You can simply reference the model in `test_vector_retrieve_chuck_v2.py`

This document describes how to download from huggingface website. You must have login id with huggingface and created token. You should run visual studio as administrator (Right click on visual studio code and use run as administrator

There will be prompt to provide the token of the huggingface, paste the token and hit enter. Answer Y for add as git token

Define a directory to save the model (optional, defaults to Hugging Face cache)

`download_dir = "./models--google--gemma-3-1b-it"` This will create the following in the C drive.

`C:\Users\NAME\.cache\huggingface\hub\models--google--gemma-3-1b-it\snapshots\dcc83ea841ab6100d6b47a070329e1ba4cf78752`

Download models directly from huggingface website

go the "<https://huggingface.co/google/gemma-3-1b-it/tree/main>" and download all the file in a folder, and then, use the folder name to access the model for the python code such as following.

Download models using LM Studio

<https://lmstudio.ai/>

How to use models in python to create prompt

Method 1: Reference specific folder where you have downloaded the model

```
#model_name = " C:\Users\NAME\.cache\huggingface\hub\models--google--gemma-3-1b-it\snapshots\dcc83ea841ab6100d6b47a070329e1ba4cf78752"
```

Method 2: Reference the hosted model in the python code.

```
#model_name = "google/gemma-3-1b-it"
```

AutoTokenizer.from_pretrained() and AutoModelForCausalLM.from_pretrained() references model

```
# 1. Load your Gemma model and tokenizer
# Specify the path to your locally downloaded Gemma model
model_name = "C:/Users/NAME/.cache/huggingface/hub/models--google--gemma-3-1b-it/snapshots/093f9f388b31de276ce2de164bdc2081324b9767"

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.bfloat16, # Use bfloat16 for better performance on compatible hardware
    # Add trust_remote_code=True if needed for custom model architectures
)
```

Add documents in private chroma store

Use test_vector_create_chroma.py to add documents to vector store. The following is a hard coded example in the python file. However, it is possible to get data from SQL or JSON databases. If you get data from other sources, then, it is important to make sure that id is unique id from the source system such as primary key in SQL table. There is no update data operation in vector store, thus, any update has to be “delete” the old document using the id, and insert new document with the same key.

```
documents_raw = [
    {"id": "doc_C", "title":"To reduce crime", "content": "Government is needed to reduce crime. Less corrupt government is more effective to reduce crime."},
]
```

```
{ "id": "doc_D", "title": "Benefits of FREE PRESS", "content": "Government corruption can be reduced by FREE PRESS. Less corrupt government can reduce crime." }
]
```

```
persist_directory = "c:/pydatacroma"
```

Delete documents in private chroma store

Use test_vector_delete_chroma.py for deleting a document. There is no update data operation in vector store, thus, any update has to be “delete” the old document using the id, and insert new document with the same key.

```
# Define a directory to store the ChromaDB data
persist_directory = "c:/pydatacroma"
```

```
if ids_to_delete:
    vectorstore.delete(ids=ids_to_delete)
    print(f"Deleted {len(ids_to_delete)} chunks associated with 'doc_A'.")
    # It's good practice to persist changes after deletion
    vectorstore.persist()
else:
    print("'doc_A' not found for deletion or has no associated chunks.")
```

Use private chroma store to create prompt for AI

Use test_vector_retrieve_chuck_v2.py to test prompt from vector store and AI models. The following describes primary functions within the python file.

Specify the path to your locally downloaded Gemma model

```
# Specify the path to your locally downloaded Gemma model
model_name = "C:/Users/milan/.cache/huggingface/hub/models--google--gemma-3-1b-it/snapshots/dcc83ea841ab6100d6b47a070329e1ba4cf78752"
```


Specify model and python will use model hosted on huggingface

```
# Specify the path to Gemma model
model_name = "google/gemma-3-1b-it"
```

Specify vector store (previously created) path

```
# 4. Create embeddings and load the vector database (also part of one-time setup)
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")

persist_directory = "c:/pydatachroma"

# Load the existing Chroma vectorstore from the specified directory
vectorstore = Chroma(persist_directory=persist_directory, embedding_function=embedding_model)
print(f"ChromaDB loaded from {persist_directory}")
```

Specify prompt template that will instruct AI model to use data provided in “context/prompt” from vector store

```
# 5. Define the RAG prompt template
rag_prompt_template = """<start_of_turn>user
Answer the question based *only* on the following context. If you cannot answer the question from the context, please say "I
don't have enough information to answer that."

Context:
{context}

Question: {question}<end_of_turn>
<start_of_turn>model
"""
```

Configure retriever to decide number of documents to use for prompt

```
# Configure the retriever to fetch a specified number of relevant documents
retriever = vectorstore.as_retriever(search_kwargs={"k": 6})
```

The following is used to create data for context/prompt



```
# 6. Build the RAG chain
# This chain orchestrates the retrieval and generation steps.
rag_chain = (
    # First, retrieve relevant context documents based on the question
    {"context": retriever, "question": RunnablePassthrough()}
    # Then, format the retrieved context and question into the defined prompt
    | rag_prompt
    # Pass the formatted prompt to the Language Model (LLM) for generation
    | llm # Use the wrapped LangChain LLM object initialized above
    # Finally, parse the LLM's output into a simple string
    | StrOutputParser()
)
```

To type a question and hit enter

```
while True:
    question = input("\nYour Question: ")
    if question.lower() in ["exit", "quit"]:
        print("Ending session.")
        break

    # Get a RAG-augmented answer for the user's question
    response = rag_chain.invoke(question)
    print(f"Answer: {response}")
```

Special purpose AI concepts

Private vector store

The following directory can be in cloud or local store. Since the purpose to create special purpose AI, the use of separate stores forces AI to provide specific answer based on the vector store used.

```
persist_directory = "c:/chroma1"
persist_directory = "c:/chroma2"
```

The following is the format to segregate data in vector stores. Document id can be primary key from SQL database row. It is desirable to separate content in different document. The following is the python list format. It can be key value pair or primary key and attribute.

```
documents_raw = [
    {"id": "doc_A", "title": "To reduce the cost of education", "content": "Universities should control non education related expenses."},
    {"id": "doc_A", "title": "To reduce the cost of education", "content": "Compress college degree period from 4 years to 3 years."},
    {"id": "doc_A", "title": "To reduce the cost of education", "content": "Promote trade and skill paths as an alternative to four-year degree."},
    {"id": "doc_A", "title": "To reduce the cost of education", "content": "Provide lower rates for student loan."},
    {"id": "doc_A", "title": "To reduce the cost of education", "content": "Provide loan payback by service to country or charity."},
    {"id": "doc_B", "title": "PUBLIC education benefits", "content": "PUBLIC EDUCATION has done many good things such as affordability"},
    {"id": "doc_B", "title": "PUBLIC education benefits", "content": "PUBLIC EDUCATION has done many good things such as common education across multiple economic level."}
]
```

Prompt size

The following information came from Google's Gemini AI.

The data size that can be given to your `rag_chain` context is primarily limited by the **context window** of the underlying Large Language Model (LLM), which in your case is **Gemma 3.1B-IT**.

Here's a breakdown based on the search results:

- **Gemma 3.1B-IT Context Window:** While some larger Gemma 3 variants (like 4B, 12B, and 27B) boast a **128K token context window**, the **Gemma 3.1B (the "1B" variants)** specifically have a context window of **32K tokens**.

- This 32K token limit applies to the *total* input to the model, which includes your prompt, the retrieved context from your `retriever`, and any chat history if you were maintaining one.
- **max_new_tokens in pipeline:** You have `max_new_tokens=500` set in your `hf_pipeline`. This parameter controls the *maximum length of the generated response, not the input context*. The total number of tokens (input context + generated output) must remain within the model's overall context window.
- **LangChain and Transformers Integration:** LangChain's `HuggingFacePipeline` essentially wraps the Transformers pipeline. It will respect the underlying model's and pipeline's limits. If the combined input tokens (context from retriever + your question) exceed the model's context window, you will likely encounter an error (e.g., "Token indices sequence length is longer than the specified maximum sequence length").
- **RAG Best Practices (Chunking):** In RAG systems, it's common to break down large documents into smaller "chunks" before embedding them and storing them in the vector database. Your `retriever` then fetches a certain number of these chunks (`search_kwargs={"k": 6}`). This chunking strategy is crucial to ensure that the retrieved context, even if it comes from a large original document, fits within the LLM's context window.

In summary, for your `rag_chain` using Gemma 3.1B-IT, the effective maximum data size for the context (retrieved documents + your question) is approximately 32,000 tokens.

It's important to consider that while the model has a large context window, feeding it excessively long contexts can sometimes lead to:

- **"Lost in the Middle" phenomenon:** The model might struggle to identify the most relevant information when surrounded by a very large context.
- **Increased inference time and memory usage:** Longer inputs require more computational resources.

Therefore, while the technical limit is 32K tokens for Gemma 3.1B, practical considerations around relevance and performance might lead you to keep your retrieved context size (controlled by `k` in your retriever) to a more manageable amount that is well within the model's capabilities.

In summary, for your `rag_chain` using Gemma 3.1B-IT, the effective maximum data size for the context (retrieved documents + your question) is approximately 32,000 tokens.

Therefore, while the technical limit is 32K tokens for Gemma 3.1B, practical considerations around relevance and performance might lead you to keep your retrieved context size (controlled by `k` in your retriever) to a more manageable amount that is well within the model's capabilities.

so how many megabyte of data is 32000 tokens. Based on common approximations for English text tokenization:

- 1 token is roughly equivalent to 4 characters.
- Assuming UTF-8 encoding, a typical English character takes 1 byte.

Therefore, if we estimate that 1 token is about 4 bytes:

$32,000 \text{ tokens} \times 4 \text{ bytes/token} = 128,000 \text{ bytes}$

To convert bytes to megabytes:

$128,000 \text{ bytes} \div (1024 \times 1024) \text{ bytes/MB} \approx 0.122 \text{ MB}$

So, 32,000 tokens of English text is approximately 0.122 megabytes of data.

Keep in mind that this is an estimation, as the exact byte size per token can vary based on the specific tokenizer used and the language of the text (e.g., non-English characters often require more bytes per character in UTF-8).

Relevant documents from vector store for your question

The following information came from Google's Gemini AI.

Question to Gemini:

How does the following python knows what are the relevant document to retrieve. Does it use words used in the question to retrieve specific documents?. Here is the python statement: `retriever = vectorstore.as_retriever(search_kwargs={"k": 6})`

Answer from Gemini:

You're asking about how the `retriever` in your Canvas, specifically the line `retriever = vectorstore.as_retriever(search_kwargs={"k": 6})`, identifies relevant documents.

You're right to wonder if it uses words from the question, but the process is actually more sophisticated than simple keyword matching. Here's how it works:

1. **Question to Vector (Embeddings):** When you ask a question, the `retriever` doesn't just look for exact words. Instead, it uses the `embedding_model` you defined earlier (`HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")`) to convert your question into a numerical representation called a "vector" or "embedding." This vector captures the *semantic meaning* of your question.
2. **Vector Store Search:** Your `vectorstore` (ChromaDB) contains embeddings for all the document chunks you previously stored. When the `retriever` gets the vector representation of your question, it then performs a similarity search within the `vectorstore`. It looks for document embeddings that are "closest" in numerical space to your question's embedding. This "closeness" implies semantic similarity, meaning documents about similar topics or ideas will be retrieved, even if they don't use the exact same words as your question.

3. **Top k Retrieval:** The `search_kwargs={"k": 6}` part tells the `retriever` to fetch the top 6 most semantically similar document chunks from your `vectorstore`. These 6 chunks are then provided as `context` to the Large Language Model in your RAG chain.

So, while words in your question are fundamental, it's their conversion into a numerical embedding and the subsequent vector similarity search that enables the `retriever` to find genuinely *relevant* documents, rather than just documents containing specific keywords.

Relevant documents from vector store as prompt and your question

The following information came from Google's Gemini AI.

Question to Gemini:

So after retriever gets the data for the `rag_chain` context and `rag_chain` has the question. Does it then, searches the entire AI model or just the Chroma vector store only.

Answer from Gemini:

To clarify, after the `retriever` gets the data for the `rag_chain` context and `rag_chain` has the question, the process unfolds as follows:

1. **Retrieval (Chroma Vector Store Only):** The `retriever` (which uses your `vectorstore`, ChromaDB) performs a similarity search *only* within the pre-indexed document embeddings stored in the Chroma database. It uses the semantic meaning of your question to find the top k (in your case, 6) most relevant document chunks.
2. **Augmentation and Generation (AI Model with Provided Context):** These top 6 relevant document chunks are then passed *along with your original question* to the Large Language Model (your Gemma model, represented by `llm`). The LLM's task is then to generate an answer **based *only* on this provided context** (the 6 retrieved chunks and your question). It does not perform a new search across its entire knowledge base or the entire Chroma vector store. Its role is to synthesize an answer from the specific, targeted information it has been given.

So, the AI model (Gemma) does not search the entire Chroma vector store itself. The vector store's job is to efficiently find and provide only the most relevant snippets to the LLM.

To include more data from vector store

The following information came from Google's Gemini AI.

Question to Gemini:

So how do I expand the search to include entire LLM or entire chroma vector store?

Answer from Gemini:



You're asking how to expand the search to include either the "entire LLM" or the "entire Chroma vector store" based on the line `retriever = vectorstore.as_retriever(search_kwargs={"k": 6})`.

Let's clarify how the RAG (Retrieval Augmented Generation) system in your Canvas works:

1. **The LLM (Large Language Model - Gemma in your case) is not a search engine.** Its function is to *generate* text based on the input (the "context") it receives. It doesn't perform real-time searches across its entire pre-trained knowledge or external databases. Its internal knowledge is embedded in its parameters during training, and it uses this to reason over the specific context provided to it. So, you cannot "expand the search to include the entire LLM" in the way you might search a database.
2. **The retriever already searches the *entire* Chroma vector store.** When you call `vectorstore.as_retriever()`, the underlying vector database (Chroma) is queried across all the document embeddings it contains. The `search_kwargs={"k": 6}` parameter means that from this complete search of the `vectorstore`, it will return the **top 6 most relevant** document chunks.

Therefore, the `retriever` *is* searching the entire Chroma vector store to find the most relevant documents.

If your goal is to provide *more* context to the LLM (Gemma) from your Chroma database, here's how you can approach it:

- **Increase the `k` value:** The most direct way to get more documents from your `vectorstore` into the LLM's context is to increase the `k` value in `search_kwargs={"k": X}`. For example, `search_kwargs={"k": 10}` would retrieve the top 10 most relevant chunks.
 - **Important Consideration:** The ultimate limit here is the **context window of your Gemma 3.1B-IT model**, which is 32,000 tokens. You cannot provide more context than this limit. As you increase `k`, ensure that the total tokens of all retrieved documents, plus your question and the prompt template, do not exceed 32,000 tokens. Exceeding this will result in errors or truncation, potentially reducing the quality of the generated response.
- **Refine your document chunking strategy:** If your original documents are very large, you might need to reconsider how you're splitting them into chunks before adding them to Chroma. Smaller, more focused chunks can sometimes lead to more precise retrieval.
- **Advanced RAG techniques (beyond this current setup):** For very large datasets or extremely long documents, more advanced strategies exist, such as:
 - **Re-ranking:** Retrieve more documents (e.g., `k=50`), then use a separate, smaller re-ranking model to select the absolute best `k` (e.g., 6) documents to pass to the LLM.
 - **Summarization:** If retrieved chunks are still too long, you could summarize them before passing them to the LLM.
 - **Multi-stage retrieval:** Implement a more complex retrieval process that might involve multiple queries or different types of searches.

In summary, the `retriever` already searches the entire Chroma vector store. The key is managing how much of that retrieved information ultimately fits and is effectively utilized within your Gemma model's context window.

Vector store for prompt vs fine tuning AI

The following information came from Google's Gemini AI.

Question to Gemini:

What is difference between populating chroma vector store and using it in as for rag context vs. fine tuning a model with new content?

Answer from Gemini:

You're asking about two fundamental approaches to making a Large Language Model (LLM) knowledgeable about new information:

1. **Retrieval Augmented Generation (RAG) with a Vector Store (like ChromaDB):** This involves providing external, relevant information to the LLM at the time of inference (when you ask a question).
2. **Fine-tuning a model with new content:** This involves actually updating the internal parameters (weights) of a pre-trained LLM by training it on a new dataset.

Here's a breakdown of the differences, advantages, and disadvantages of each:

1. RAG with a Chroma Vector Store (Populating and Using as Context)

How it works:

You take your new content (e.g., documents, articles, internal knowledge base).

You "chunk" this content into smaller, manageable pieces.

You use an embedding model (like `sentence-transformers/all-MiniLM-L6-v2` in your code) to convert each chunk into a numerical vector (embedding).

These embeddings are stored in a vector database (like ChromaDB).

At inference time, when a user asks a question, the question is also converted into an embedding.

The vector database performs a similarity search to find the most relevant document chunks based on the question's embedding.

These retrieved chunks are then added to the prompt as "context" and sent to the pre-trained LLM (Gemma).

The LLM generates an answer based on its general knowledge *and* the specific context provided.

Advantages:

Cost-Effective (Inference): Much cheaper than fine-tuning for adding new knowledge, as you're not retraining the entire model. The LLM itself remains static.

Fast Updates: Easily update the knowledge by adding, removing, or modifying documents in the vector store. No need for lengthy re-training.



Factuality & Reduced Hallucinations: The LLM is explicitly told to answer "only" from the provided context, which significantly reduces the risk of it "hallucinating" or generating incorrect information based on its general training data. You can also cite sources (the retrieved documents).

Scalability for Large Knowledge Bases: Can handle extremely large amounts of information by simply adding more data to the vector store.

Transparency: You can inspect the retrieved documents to see *why* the LLM generated a particular answer.

Disadvantages:

Context Window Limits: The amount of context you can provide to the LLM in a single prompt is limited by its context window size (e.g., 32K tokens for Gemma 3.1B-IT). This means you can only retrieve and pass a certain number of relevant chunks.

Retrieval Quality Dependent: The quality of the answer heavily depends on the quality of the retrieval. If the relevant documents aren't retrieved, the LLM won't be able to answer correctly.

No Model Behavior Change: The LLM's inherent style, tone, and reasoning capabilities remain unchanged. It doesn't "learn" new ways of thinking or specific jargon from your new content; it just uses the content as reference.

Latency: There's a slight added latency due to the retrieval step before the LLM generates a response.

2. Fine-tuning a Model with New Content

How it works:

You take a pre-trained LLM (Gemma).

You prepare a dataset of new content, often in a question-answer, instruction-response, or conversational format (`<instruction>` `<response>`).

You then perform additional training (fine-tuning) on this dataset, using the pre-trained LLM as a starting point.

During fine-tuning, the LLM's internal weights are adjusted to incorporate the patterns and knowledge from your new data.

The result is a *new version* of the LLM that has absorbed this specific information directly into its parameters.

Advantages:

Deeper Knowledge Integration: The model truly "learns" the new information and incorporates it into its internal representation. It can generalize better to unseen questions within the domain of the fine-tuning data.

Behavioral Change: Can be used to make the model adopt a specific tone, style, or persona, or to improve its performance on specific types of tasks (e.g., code generation, summarization, specific domain reasoning).

Potentially Better Fluency/Cohesion: Responses might be more natively integrated and fluent, as the knowledge is directly part of the model.

Disadvantages:

High Computational Cost: Fine-tuning requires significant computational resources (GPUs) and time, especially for larger models and datasets.

Updates are Slow & Expensive: If your knowledge changes frequently, you'd need to re-fine-tune the model, which is a slow and costly process.

Risk of Catastrophic Forgetting: Fine-tuning on new data can sometimes cause the model to "forget" some of its original general knowledge.

Higher Hallucination Risk (if not combined with RAG): The model might still generate plausible but incorrect information if the fine-tuning data wasn't comprehensive or if it's asked questions outside the scope of its fine-tuning.

Less Transparent: It's harder to trace *why* the model generated a particular answer, as the knowledge is implicitly stored in its weights.

When to use which:

Use RAG (Vector Store) when:

Your knowledge base is dynamic and changes frequently.

Factuality and citing sources are critical.

You have a large volume of content that might exceed the LLM's context window if all were to be fine-tuned.

You want the LLM to retain its general knowledge and simply reference external documents for specific facts.

Computational resources for training are limited.

Use Fine-tuning when:

You want to change the LLM's style, tone, or specific conversational behavior.

You need the model to learn a new skill or adapt to a very specific domain's reasoning patterns.

Your new content is relatively static and doesn't change often.

You have sufficient computational resources and time for training.

The knowledge is fundamental to the model's core function, not just supplemental.

Often, the best approach is a hybrid: You can fine-tune a model to adopt a certain style or understand specific domain terminology, and then use RAG on top of that fine-tuned model to provide it with up-to-date, factual information from your vector store. This combines the strengths of both methods.

Inference vs test

The following information came from Google's Gemini AI.

In machine learning, the terms "inference script" and "test script" are related but refer to slightly different phases or purposes:

1. Inference Script:

- **Purpose:** To use a **trained** model to make predictions or generate outputs on **new, unseen data**. The focus is on applying the model.
- **Data:** Typically operates on single inputs or a stream of new data, not necessarily a pre-defined "test set" with known answers. It's about getting the model's output for practical use.
- **Evaluation:** While you might manually inspect the outputs for quality, an inference script *itself* doesn't usually calculate quantitative metrics (like accuracy, F1-score, etc.) because it might not have the "ground truth" labels for the new data.
- **Example:** Your `inference_script.py` is a perfect example. You provide a prompt ("Categorize the following product description...") and get the model's prediction. You're not comparing it to a known correct answer in an automated way.

2. Test Script:

- **Purpose:** To **evaluate the performance** of a trained model on a dedicated **test dataset** that has not been seen during training. The primary goal is to quantitatively measure how well the model generalizes to new data.
- **Data:** Uses a pre-defined "test set" or "evaluation set" which includes both the input data and the corresponding "ground truth" labels.
- **Evaluation:** Calculates specific metrics (e.g., accuracy, precision, recall, F1-score, BLEU score, ROUGE score, etc.) by comparing the model's predictions against the known ground truth.

what does inference mean in the python code you provided:

In the provided Python code snippet, "inference" refers to the process of using a trained machine learning model (specifically, a large language model in this case) to make predictions or generate new content based on new, unseen input data.

Here's a breakdown of what that means in the context of the code:

1. **Applying a Trained Model:** The core idea of inference is to take a model that has already learned patterns and relationships from a large dataset (during a process called "training") and then feed it new information to see what it outputs.
2. **Generating Output:** In this specific code, the model is a "causal language model" (like those used for text generation). When you provide it with an `inference_prompt` (e.g., "Categorize the following product description..."), the model uses its learned knowledge to predict the most probable sequence of words that should follow the prompt.
3. **No Further Learning:** During inference, the model's weights (the internal parameters it learned during training) are fixed. It's not learning anything new or updating its understanding of the data. It's simply applying what it already knows.
4. **Steps in the Code for Inference:**
 - **Loading the Model:** `peft_model_for_inference = AutoModelForCausalLM.from_pretrained(...)` and `peft_model_for_inference = PeftModel.from_pretrained(...)` are loading the pre-trained language model, potentially with PEFT (Parameter-Efficient Fine-Tuning) adapters, which allow for efficient loading and usage of a fine-tuned model without loading the entire base model.

- **Preparing Input:** `tokenizer.apply_chat_template(chat, tokenize=False, add_generation_prompt=True)` and `inputs = tokenizer(formatted_prompt, return_tensors="pt").to(peft_model_for_inference.device)` prepare your `inference_prompt` into a format that the model can understand (tokenized numerical representations).
- **Generation:** `output = peft_model_for_inference.generate(**inputs, max_new_tokens=100, ...)` is the crucial step where the model takes the prepared input and generates a sequence of new tokens (words/subwords) based on its training.
- **Decoding Output:** `decoded_output = tokenizer.decode(output[0], skip_special_tokens=True)` converts the numerical output from the model back into human-readable text.

In summary, "inference" in this context is the practical application of the trained language model to generate a categorization for the given product description, demonstrating its ability to "reason" or "understand" the input based on its prior training.