Using LLaMA 2.0, FAISS and LangChain for Question-Answering on Your Own Data

Murtuza Kazmi · Follow
9 min read · Jul 24

Listen Share

Source: venturebeat.com

Over the past few weeks, I have been playing around with several large language models (LLMs) and exploring their potential with all sorts of methods available on the internet, but now it's time for me to share what I have learned so far!

I was super excited to know that Meta released the next generation of its open-source large language model, LLaMA 2 (on 18th July 2023) and the most interesting

part of the release was, they made it available free of charge for commercial use to the public. Therefore, I decided to try it out and see how its performs.

In this article, I'm going share on how I performed Question-Answering (QA) like a chatbot using Llama-2–7b-chat model with LangChain framework and FAISS library over the documents which I fetched online from Databricks documentation website.

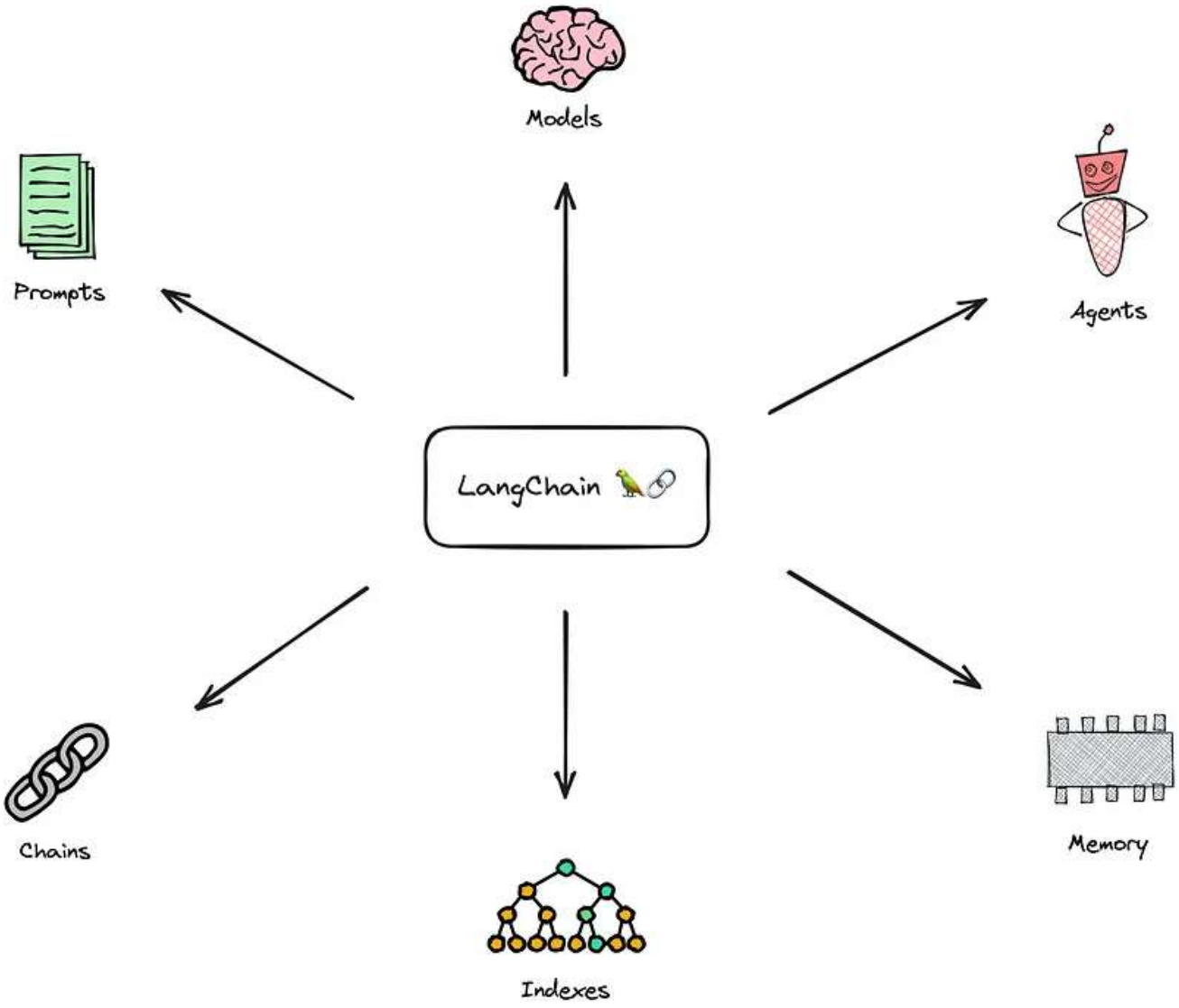
Introduction

LLaMA 2 model is pretrained and fine-tuned with 2 Trillion 🚀 tokens and 7 to 70 Billion parameters which makes it one of the powerful open source models. It comes in three different model sizes (i.e. 7B, 13B and 70B) with significant improvements over the Llama 1 models, including being trained on 40% more tokens, having a much longer context length (4k tokens 🤯), and using grouped-query attention for fast inference of the 70B model 🔥. It outperforms other open source LLMs on many external benchmarks, including reasoning, coding, proficiency, and knowledge tests.

Model	Size	Code	Commonsense Reasoning	World Knowledge	Reading Comprehension	Math	MMLU	BBH	AGI Eval
Llama 1	7B	14.1	60.8	46.2	58.5	6.95	35.1	30.3	23.9
Llama 1	13B	18.9	66.1	52.6	62.3	10.9	46.9	37.0	33.9
Llama 1	33B	26.0	70.0	58.4	67.6	21.4	57.8	39.8	41.7
Llama 1	65B	30.7	70.7	60.5	68.6	30.8	63.4	43.5	47.6
Llama 2	7B	16.8	63.9	48.9	61.3	14.6	45.3	32.6	29.3
Llama 2	13B	24.5	66.9	55.4	65.8	28.7	54.8	39.4	39.1
Llama 2	70B	**37.5**	**71.9**	**63.6**	**69.4**	**35.2**	**68.9**	**51.2**	**54.2**

Llama 1 vs Llama 2 Benchmarks — Source: huggingface.co

LangChain is a powerful, open-source framework designed to help you develop applications powered by a language model, particularly a large language model (LLM). The core idea of the library is that we can *"chain"* together different components to create more advanced use cases around LLMs. LangChain consists of multiple components from several modules.



LangChain Modules

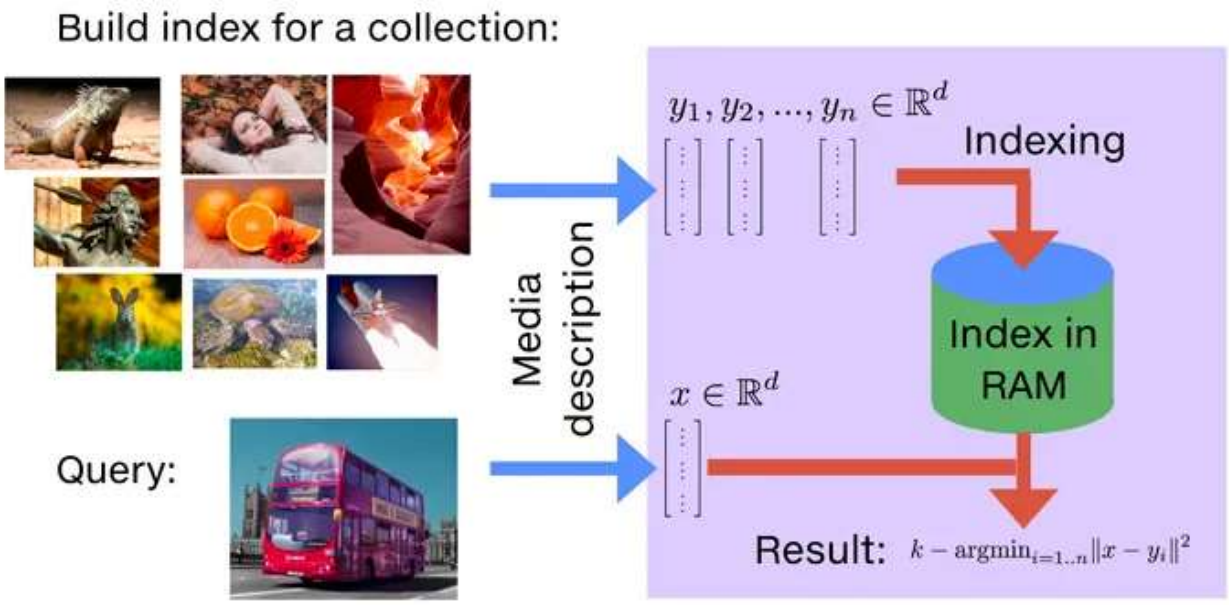
Modules:

- **Prompts:** This module allows you to build dynamic prompts using templates. It can adapt to different LLM types depending on the context window size and input variables used as context, such as conversation history, search results, previous answers, and more.
- **Models:** This module provides an abstraction layer to connect to most available third- party LLM APIs. It has API connections to ~40 public LLMs, chat and

embedding models.

- **Memory:** This gives the LLMs access to the conversation history.
- **Indexes:** Indexes refer to ways to structure documents so that LLMs can best interact with them. This module contains utility functions for working with documents and integration to different vector databases.
- **Agents:** Some applications require not just a predetermined chain of calls to LLMs or other tools, but potentially to an unknown chain that depends on the user's input. In these types of chains, there is an agent with access to a suite of tools. Depending on the user's input, the agent can decide which — if any — tool to call.
- **Chains:** Using an LLM in isolation is fine for some simple applications, but many more complex ones require the chaining of LLMs, either with each other, or other experts. LangChain provides a standard interface for Chains, as well as some common implementations of chains for ease of use.

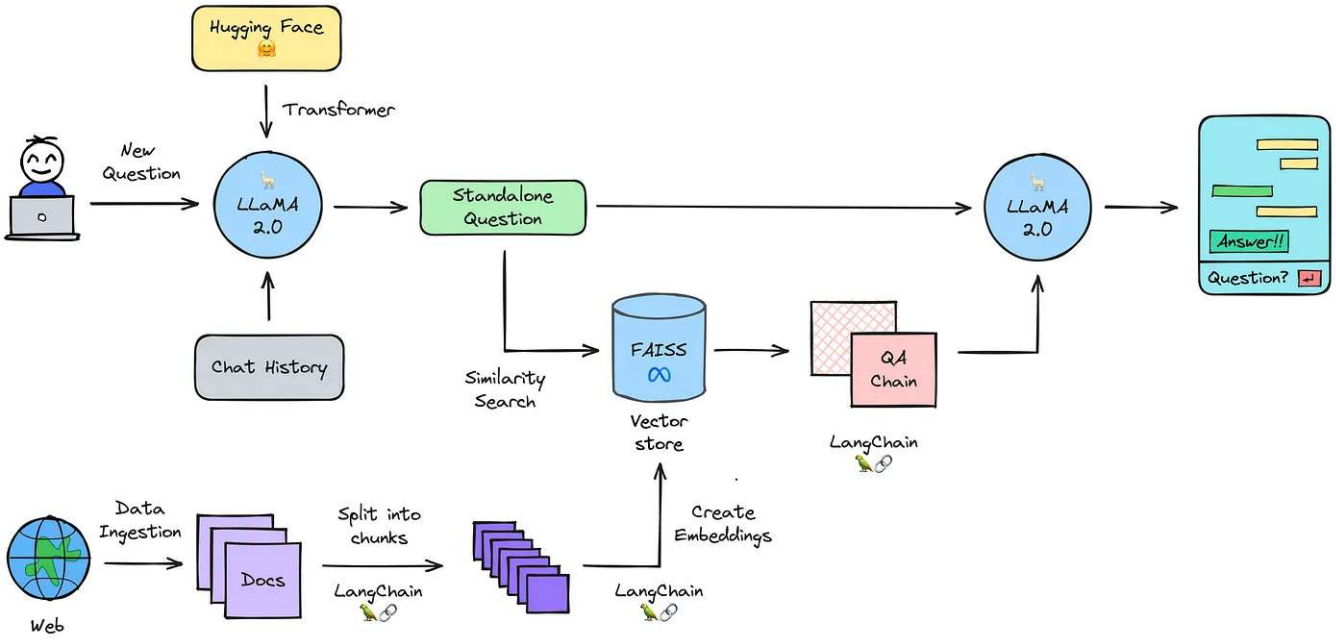
FAISS (Facebook AI Similarity Search) is a library for efficient similarity search and clustering of dense vectors. It can search multimedia documents (e.g. images) in ways that are inefficient or impossible with standard database engines (SQL). It contains algorithms that search in sets of vectors of any size, up to ones that possibly do not fit in RAM. It also contains supporting code for evaluation and parameter tuning.



FAISS Indexing and Similarity Search — Source: engineering.fb.com

Process Flow

In this section, I will briefly describe each part of the process flow.



Process Flow Diagram

1. **Initialize model pipeline**: initializing text-generation pipeline with Hugging Face transformers for the pretrained **Llama-2-7b-chat-hf** model.
2. **Ingest data:** loading the data from arbitrary sources in the form of text into the document loader.

3. **Split into chunks:** splitting the loaded text into smaller chunks. It is necessary to create small chunks of text because language models can handle limited amount of text.

4. **Create embeddings**: converting the chunks of text into numerical values, also known as embeddings. These embeddings are used to search and retrieve similar or relevant documents quickly in large databases, as they represent the semantic meaning of the text.

5. **Load embeddings into vector store**: loading the embeddings into a vector store i.e. "FAISS" in this case. Vector stores perform extremely well in similarity search using text embeddings compared to the traditional databases.

6. **Enable memory:** combing chat history with a new question and turn them into a single standalone question is quite important to enable the ability to ask follow up questions.

7. **Query data:** searching for the relevant information stored in vector store using the embeddings.

8. **Generate answer:** passing the standalone question and the relevant information to the question-answering chain where the language model is used to generate an answer.

Code Walkthrough

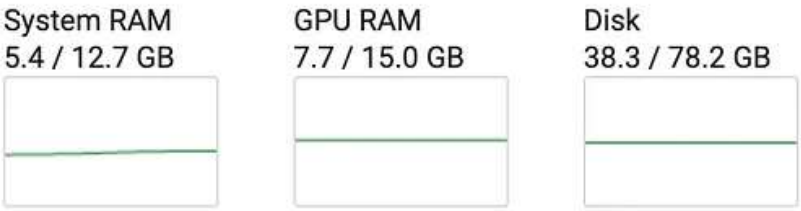
In this section, I will go through the code to explain you each step in detail.

Getting Started

You can use the open source **Llama-2-7b-chat** model in both Hugging Face transformers and LangChain. However, you have to first request access to Llama 2 models via Meta website and also accept to share your account details with Meta on Hugging Face website. It typically takes a few minutes or hours to get the access.

🚨 *Note that your Hugging Face account email **MUST** match the email you provided on the Meta website, or your request will not be approved.*

If you're using Google Colab to run the code. In your notebook, go to *Runtime* > *Change runtime type* > *Hardware accelerator* > *GPU* > *GPU type* > *T4*. You will need ~8GB of GPU RAM for inference and running on CPU is practically impossible.



Notebook Resources in Google Colab

Installing the Libraries

First of all, let's start by installing all required libraries using `pip install`.

```
!pip install -qU transformers accelerate einops langchain xformers bitsandbytes
```

Initializing the Hugging Face Pipeline

You have to initialize a `text-generation` pipeline with Hugging Face transformers. The pipeline requires the following three things that you must initialize:

- A LLM, in this case it will be `meta-llama/Llama-2-7b-chat-hf`.
- The respective tokenizer for the model.
- A stopping criteria object.

You have to initialize the model and move it to CUDA-enabled GPU. Using Colab, this can take 5–10 minutes to download and initialize the model.

Also, you need to generate an access token to allow downloading the model from Hugging Face in your code. For that, go to your Hugging Face Profile > Settings > Access Token > New Token > Generate a Token. Just copy the token and add it in the below code.

```
from torch import cuda, bfloat16
import transformers
```

```
model_id = 'meta-llama/Llama-2-7b-chat-hf'

device = f'cuda:{cuda.current_device()}' if cuda.is_available() else 'cpu'

# set quantization configuration to load large model with less GPU memory
# this requires the `bitsandbytes` library
bnb_config = transformers.BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type='nf4',
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=bfloat16
)

# begin initializing HF items, you need an access token
hf_auth = '<add your access token here>'
model_config = transformers.AutoConfig.from_pretrained(
    model_id,
    use_auth_token=hf_auth
)

model = transformers.AutoModelForCausalLM.from_pretrained(
    model_id,
    trust_remote_code=True,
    config=model_config,
    quantization_config=bnb_config,
    device_map='auto',
    use_auth_token=hf_auth
)

# enable evaluation mode to allow model inference
model.eval()

print(f"Model loaded on {device}")
```

The pipeline requires a tokenizer which handles the translation of human readable plaintext to LLM readable token IDs. The Llama 2 7B models were trained using the Llama 2 7B tokenizer, which can be initialized with this code:

```
tokenizer = transformers.AutoTokenizer.from_pretrained(
    model_id,
    use_auth_token=hf_auth
)
```

Now, we need to define the *stopping criteria* of the model. The stopping criteria allows us to specify *when* the model should stop generating text. If we don't provide

a stopping criteria the model just goes on a bit tangent after answering the initial question.

```
stop_list = ['\nHuman:', '\n```\n']

stop_token_ids = [tokenizer(x)['input_ids'] for x in stop_list]
stop_token_ids
```

You have to convert these stop token ids into `LongTensor` objects.

```
import torch

stop_token_ids = [torch.LongTensor(x).to(device) for x in stop_token_ids]
stop_token_ids
```

You can do a quick spot check that no `<unk>` token IDs (`0`) appear in the `stop_token_ids` — there are none so we can move on to building the stopping criteria object that will check whether the stopping criteria has been satisfied — meaning whether any of these token ID combinations have been generated.

```
from transformers import StoppingCriteria, StoppingCriteriaList

# define custom stopping criteria object
class StopOnTokens(StoppingCriteria):
    def __call__(self, input_ids: torch.LongTensor, scores: torch.FloatTensor,
        for stop_ids in stop_token_ids:
            if torch.eq(input_ids[0][-len(stop_ids):], stop_ids).all():
                return True
        return False

stopping_criteria = StoppingCriteriaList([StopOnTokens()])
```

You are ready to initialize the Hugging Face pipeline. There are a few additional parameters that we must define here. Comments are included in the code for further explanation.

```
generate_text = transformers.pipeline(
    model=model,
    tokenizer=tokenizer,
    return_full_text=True,  # langchain expects the full text
    task='text-generation',
    # we pass model parameters here too
    stopping_criteria=stopping_criteria,  # without this model rambles during c
    temperature=0.1,  # 'randomness' of outputs, 0.0 is the min and 1.0 the max
    max_new_tokens=512,  # max number of tokens to generate in the output
    repetition_penalty=1.1  # without this output begins repeating
)
```

Run this code to confirm that everything is working fine.

```
res = generate_text("Explain me the difference between Data Lakehouse and Data
print(res[0]["generated_text"])
```

Implementing HF Pipeline in LangChain

Now, you have to implement the Hugging Face pipeline in LangChain. You will still get the same output as nothing different is being done here. However, this code will allow you to use LangChain's advanced agent tooling, chains, etc, with **Llama 2.**

```
from langchain.llms import HuggingFacePipeline

llm = HuggingFacePipeline(pipeline=generate_text)

# checking again that everything is working fine
llm(prompt="Explain me the difference between Data Lakehouse and Data Warehouse
```

Ingesting Data using Document Loader

You have to ingest data using `WebBaseLoader` document loader which collects data by scraping webpages. In this case, you will be collecting data from Databricks documentation website.

```
from langchain.document_loaders import WebBaseLoader

web_links = ["https://www.databricks.com/","https://help.databricks.com","https

loader = WebBaseLoader(web_links)
documents = loader.load()
```

Splitting in Chunks using Text Splitters

You have to make sure to split the text into small pieces. You will need to initialize `RecursiveCharacterTextSplitter` and call it by passing the documents.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=2
all_splits = text_splitter.split_documents(documents)
```

Creating Embeddings and Storing in Vector Store

You have to create embeddings for each small chunk of text and store them in the vector store (i.e. FAISS). You will be using `all-mpnet-base-v2` Sentence Transformer to convert all pieces of text in vectors while storing them in the vector store.

```
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS

model_name = "sentence-transformers/all-mpnet-base-v2"
model_kwargs = {"device": "cuda"}

embeddings = HuggingFaceEmbeddings(model_name=model_name, model_kwargs=model_kw

# storing embeddings in the vector store
vectorstore = FAISS.from_documents(all_splits, embeddings)
```

Initializing Chain

You have to initialize `ConversationalRetrievalChain`. This chain allows you to have a chatbot with memory while relying on a vector store to find relevant information

from your document.

Additionally, you can return the source documents used to answer the question by specifying an optional parameter i.e. `return_source_documents=True` when constructing the chain.

```
from langchain.chains import ConversationalRetrievalChain

chain = ConversationalRetrievalChain.from_llm(llm, vectorstore.as_retriever(),
```

Now, it's time to do some Question-Answering on your own data!

```
chat_history = []

query = "What is Data lakehouse architecture in Databricks?"
result = chain({"question": query, "chat_history": chat_history})

print(result['answer'])
```

Output:

```
In Databricks, data lakehouse architecture refers to the way data is organized and managed within
the Databricks Lakehouse Platform. This includes the use of Delta Lake, a cloud-native data
storage system that allows for flexible and scalable data management. Additionally, the medallion
lakehouse architecture provides a structured approach to managing data and AI assets, including
data catalogs, data lineage, and data quality control. By using this architecture, organizations
can ensure their data is well-governed, secure, and easily accessible for analysis and machine
learning.
```

This time your previous question and answer will be included as a chat history which will enable the ability to ask follow up questions.

```
chat_history = [(query, result["answer"])]

query = "What are Data Governance and Interoperability in it?"
result = chain({"question": query, "chat_history": chat_history})
```

```
print(result['answer'])
```

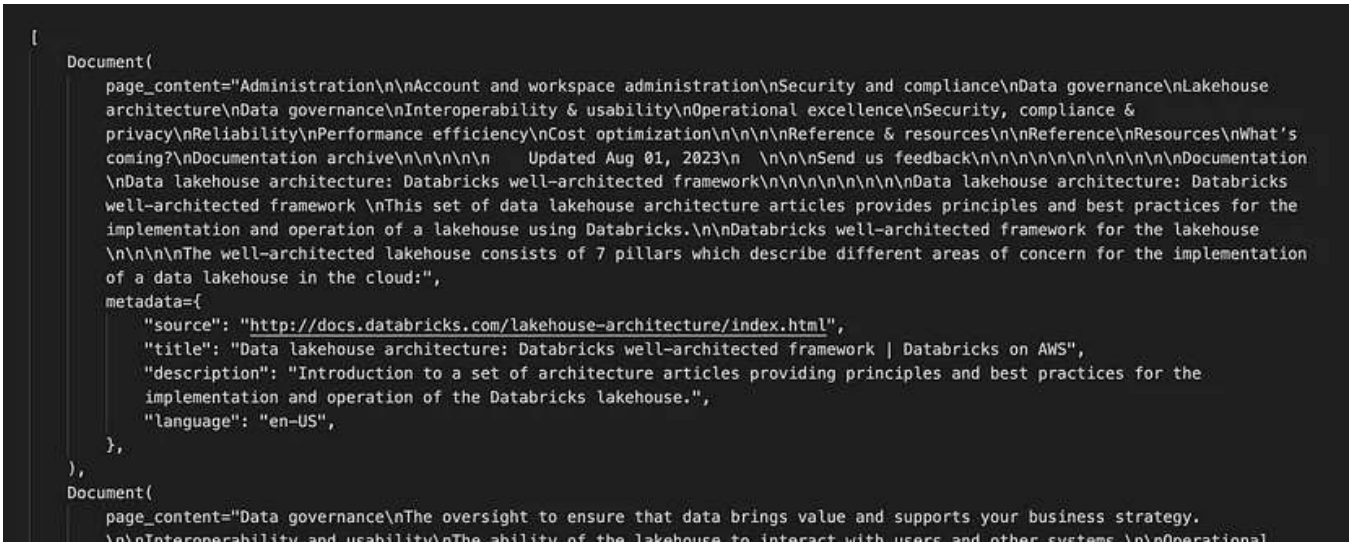


```
In the context of Data Lakehouse Architecture in Databricks, Data Governance refers to the
policies and practices implemented to securely manage the data assets within an organization. It
encompasses the centralized management of data across various systems, ensuring data quality,
security, and compliance with regulatory requirements.
```

You can also see the source of the information used to generate the answer.

```
print(result['source_documents'])
```

Output:



Finally...

Et voilà! You have now the capability to do question-answering on your on data using a powerful language model. Additionally, you can further develop it into a chatbot application using Streamlit.

Source: giphy.com

References

[1] https://huggingface.co/blog/llama2

[2] https://venturebeat.com/ai/llama-2-how-to-access-and-use-metas-versatile-open-source-chatbot-right-now/

[3] https://www.pinecone.io/learn/series/langchain/langchain-intro/

[4] https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

[5] https://ai.meta.com/tools/faiss/

[6] https://blog.bytebytego.com/p/how-to-build-a-smart-chatbot-in-10

[7] https://newsletter.theaiedge.io/p/deep-dive-building-a-smart-chatbot

[8] https://www.youtube.com/watch?v=6iHVJyX2e50

[9] https://github.com/pinecone-io/examples/blob/master/learn/generation/llm-field-guide/llama-2/llama-2-70b-chat-agent.ipynb

Llamas | Langchain | Large Language Models | Deep Learning | Machine Learning

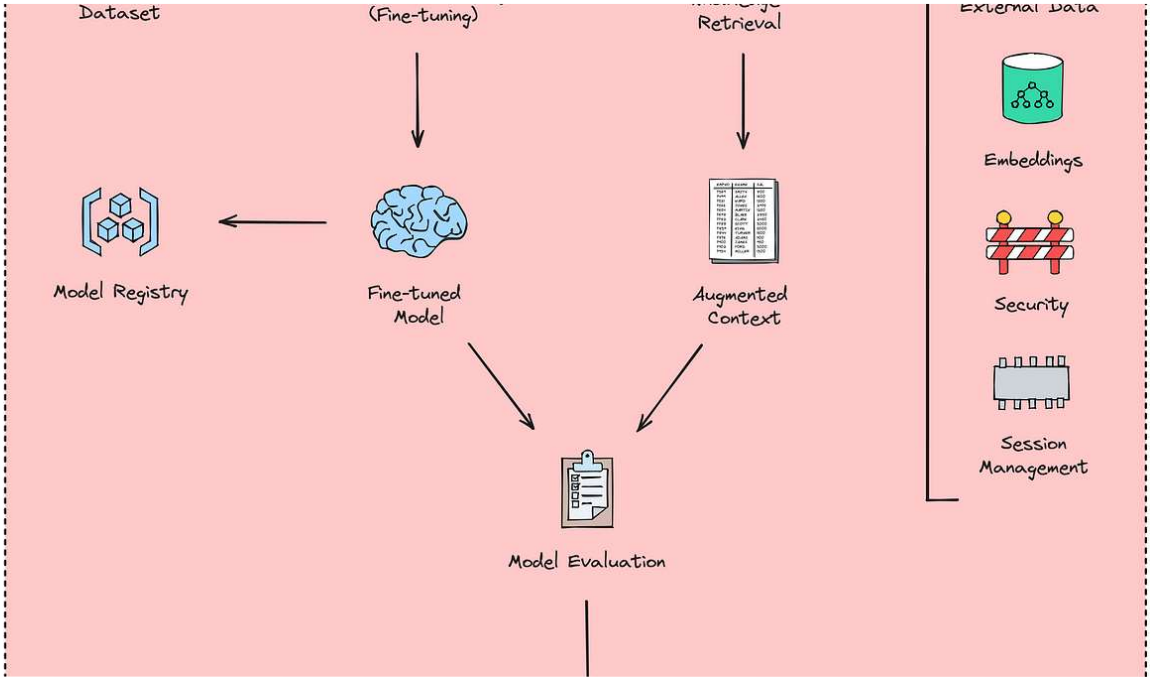
Follow

Written by Murtuza Kazmi

377 Followers

Crafting Data + AI Solutions @ adidas 🪶 linkedin.com/in/imurtuza

More from Murtuza Kazmi



Murtuza Kazmi

How Is LLMOps Different From MLOps?

A new methodology, known as “LLMOps”, has evolved and become the talk of every ML community to streamline how we should operationalize...

4 min read · Aug 11

162

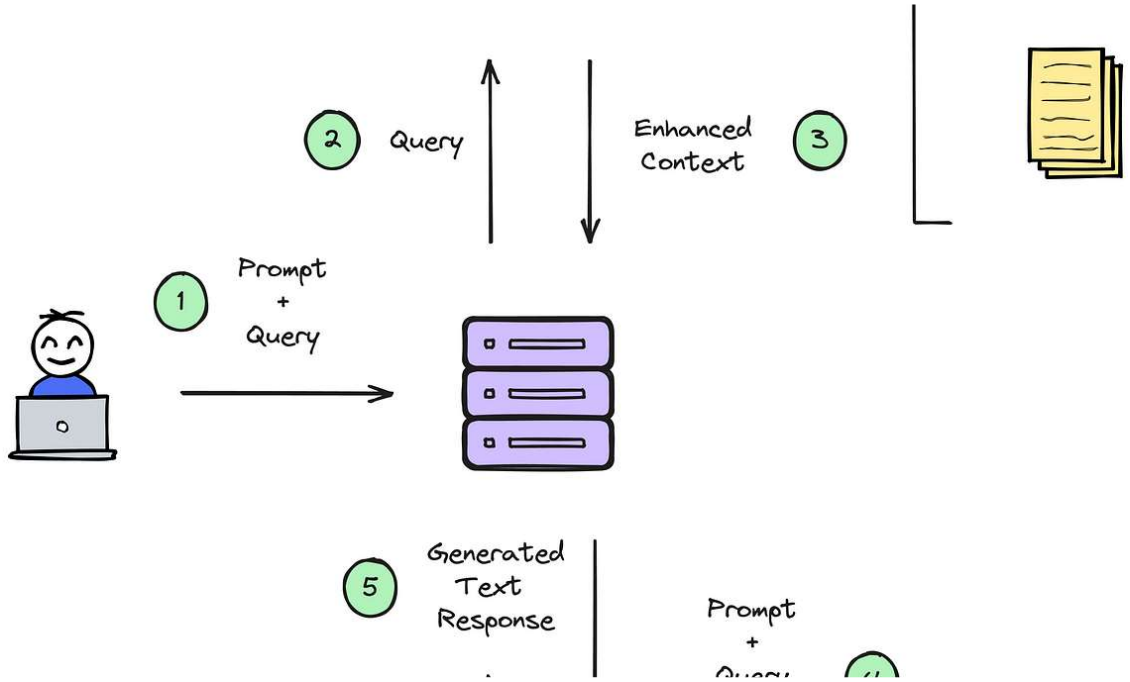
Murtuza Kazmi

How I Passed Databricks Data Engineer Associate Exam in 7 Days

"In my opinion, certification is one of the best ways of gauging your skills and knowledge about a particular technology and its...

3 min read · Jul 8

108 2

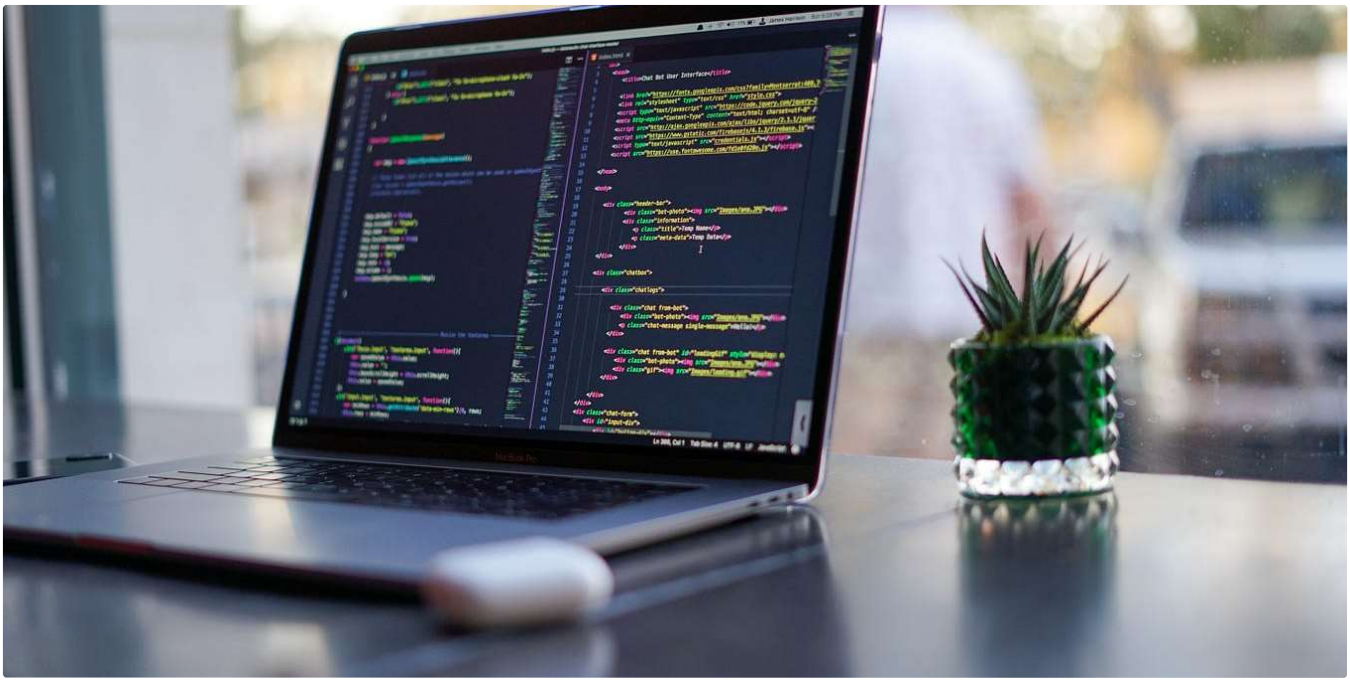


Murtuza Kazmi in GoPenAI

Enrich LLMs with Retrieval Augmented Generation (RAG)

2 min read · Jul 31

102 1



Murtuza Kazmi

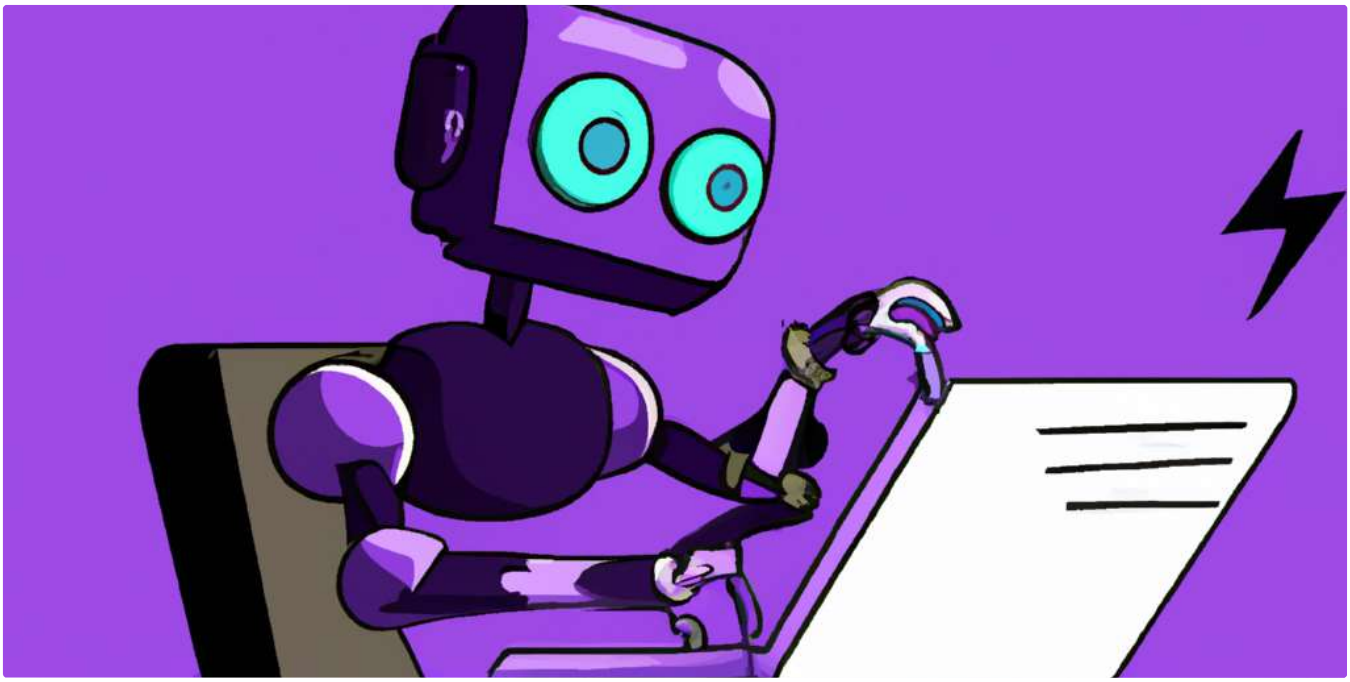
How I Built a Scalable Machine Learning Platform in Just 5 Hours—A Complete Guide

21 min read · Jan 7

22

See all from Murtuza Kazmi

Recommended from Medium



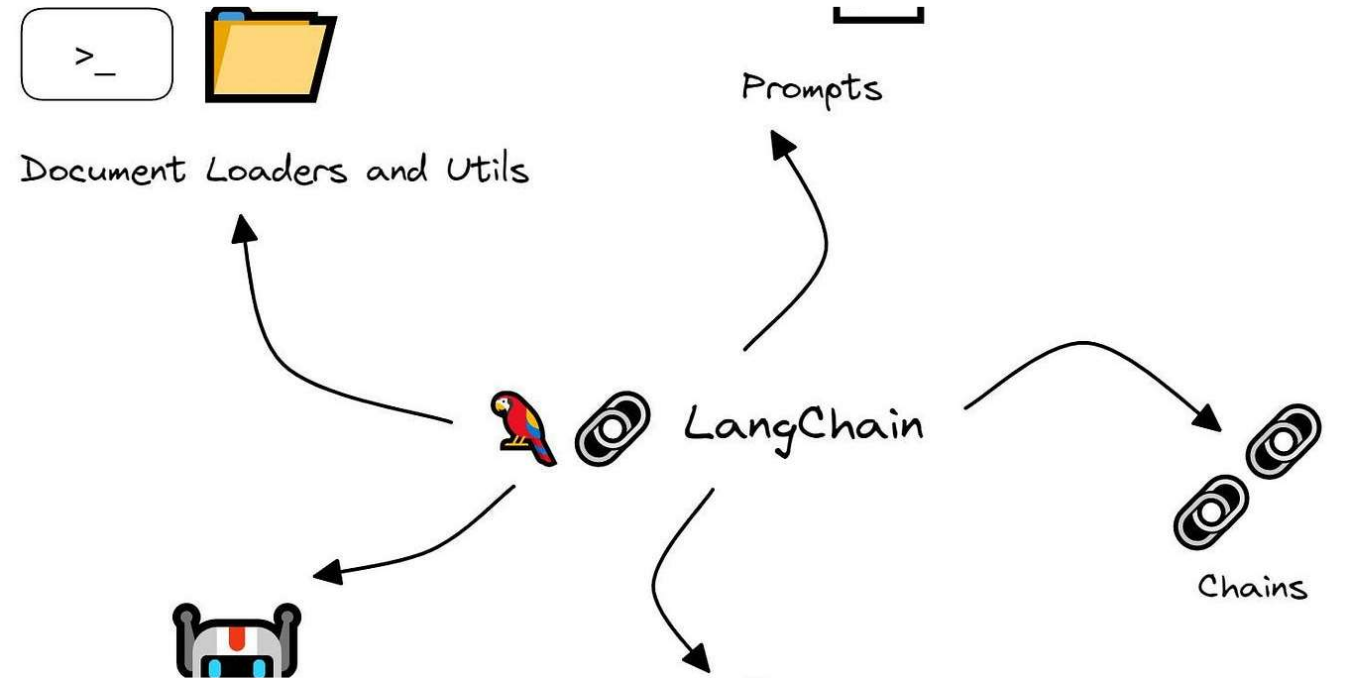
Woyera in Artificial Intelligence in Plain English

How to use Llama 2 with an API on AWS to power your AI apps

A quick start guide how to use the most worthy competitor yet to ChatGPT

9 min read · Jul 19

248 13



Vishnu Sivan in CoinsBench

Chat with your databases using LangChain

The rise of Large Language Models (LLMs) has brought about a significant shift in technology, empowering developers to create applications...

10 min read · May 9

975 16

Lists

Predictive Modeling w/ Python

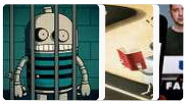
20 stories · 562 saves

Natural Language Processing

800 stories · 367 saves

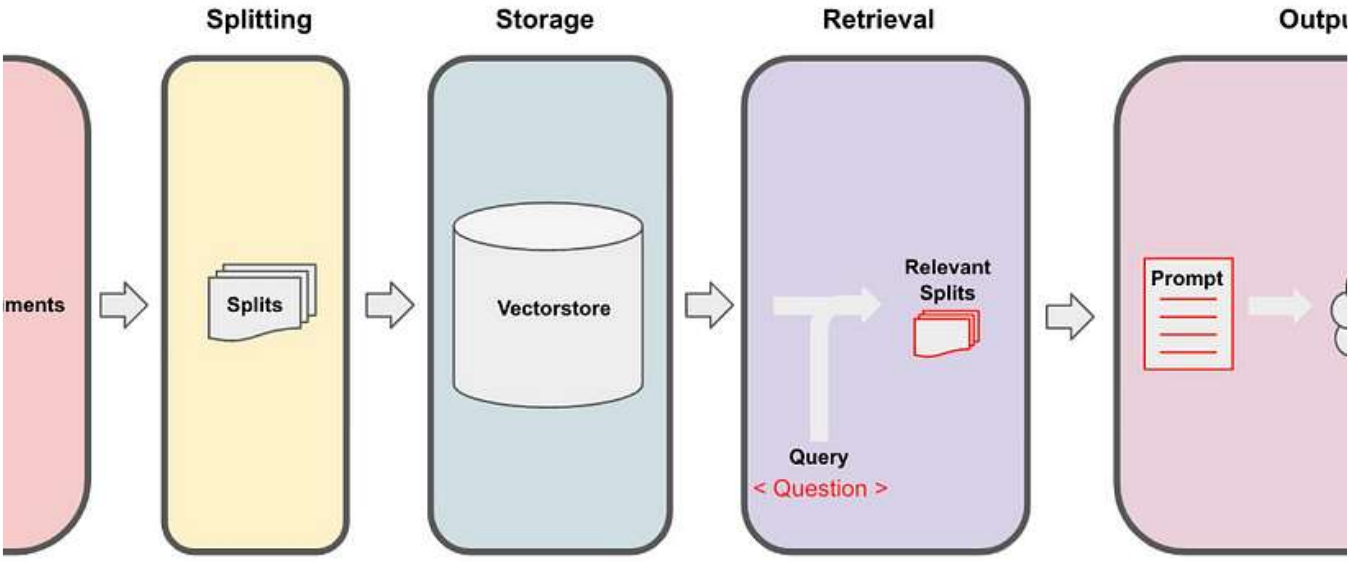
Practical Guides to Machine Learning

10 stories · 641 saves



AI Regulation

6 stories · 174 saves



Onkar Mishra

Using langchain for Question Answering on own data

Step-by-step guide to using langchain to chat with own data

23 min read · Aug 7

801 10

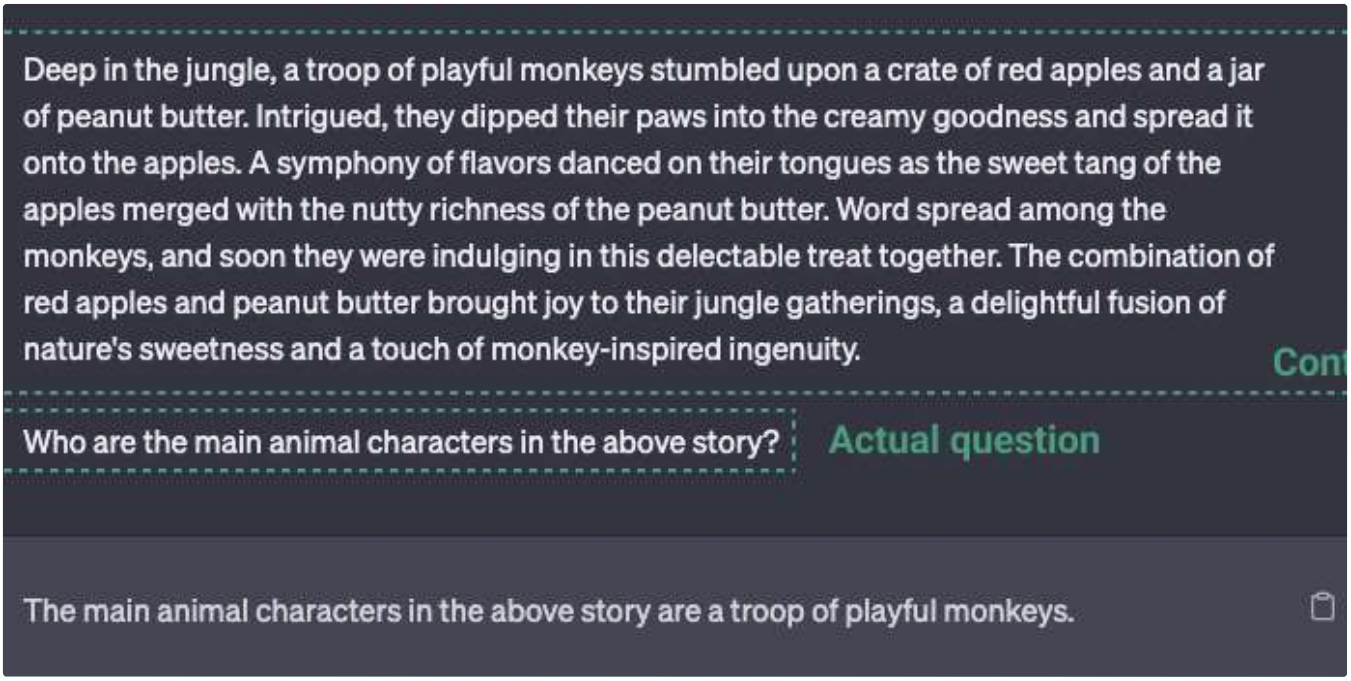
Gathnex

Fine-Tuning Llama-2 LLM on Google Colab: A Step-by-Step Guide.

Llama, Llama, Llama: 🦙 A Highly Speakable Model in Recent Times. 🗣️ Llama 2: 🌟 It's like the rockstar of language models, developed by...

12 min read · Sep 18

246 4



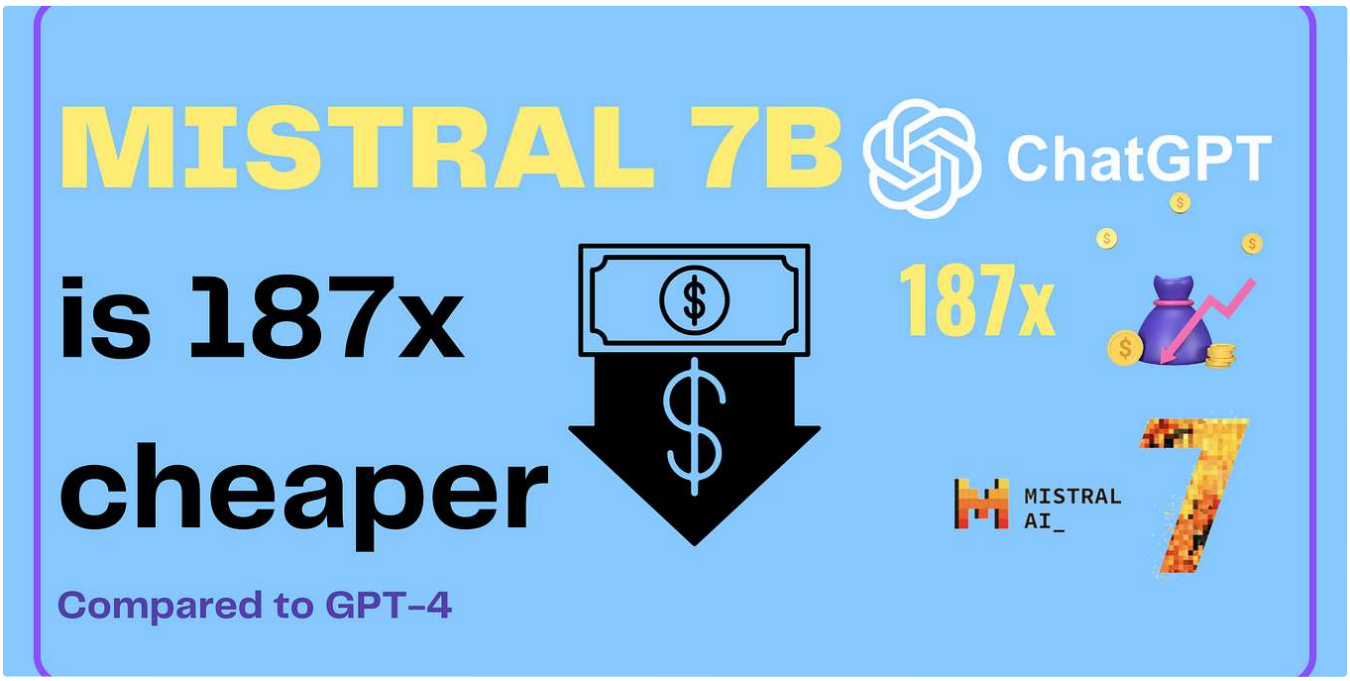
Sami Maameri in Better Programming

Building a Multi-document Reader and Chatbot With LangChain and ChatGPT

The best part? The chatbot will remember your chat history

17 min read · May 20

1.1K 10



Mastering LLM (Large Language Model)

Mistral 7B is 187x cheaper compared to GPT-4

Find how Mistral AI 7B model can be a great alternative to GPT 3.5 or 4 models with 187x cheaper in cost.

3 min read · Oct 18

183 6