Member-only story

Leveraging Llama 2 Features in Real-world Applications: Building Scalable Chatbots with FastAPI, Celery, Redis, and Docker

An In-Depth Exploration: Open vs Closed Source LLMs, Unpacking Llama 2's Unique Features, Mastering the Art of Prompt Engineering, and Designing Robust Solutions with FastAPI, Celery, Redis, and Docker

Luís Roque · Follow

Published in Towards Data Science
14 min read · 1 day ago

Listen Share More

Introduction

In an unexpected move, Meta open-sourced their Large Language Model (LLM), Llama 2, a few days ago in a decision that could reshape the current landscape of AI development. It offers an alternative to the main companies in the space such as OpenAI and Google that decided to maintain tight control over their AI models, limiting accessibility and restricting broader innovation. Hopefully, Meta's decision will spark a collective response from the open-source community, counteracting the trend of restricting access to the advances in the field. Llama 2's new license even goes further and allows commercial use, granting developers and businesses opportunities to leverage the model within existing and new products.

The Llama2 family consists of pre-trained and fine-tuned LLMs, including Llama2 and Llama2-Chat, scaling up to 70B parameters. These models have proven to perform better than open-source models on various benchmarks [1]. They also hold their

ground against some closed-source models, offering a much-needed boost to open-source AI development [2].

Figure 1: The Llama 2 family (image source)

If you follow the Open LLM leaderboard from HuggingFace [1], you can see that Meta's Llama 2 holds a strong third-place position. After the LLama 2 announcement, Stability AI released FreeWilly1 and FreeWilly2 [3]. FreeWilly1 is a fine-tuned version of Llama, and FreeWilly2 of Llama 2. Stability AI shared that they fine-tuned both models on an Orca-style Dataset. The Orca dataset is a large, structured collection of augmented data designed to fine-tune LLMs, where each entry consists of a question and a corresponding response from GPT-4 or GPT-3.5. Why are we not using the FreeWilly2 model? Unfortunately, while Llama 2 allows commercial use, FreeWilly2 can only be used for research purposes, governed by the Non-Commercial Creative Commons license (CC BY-NC-4.0).

In this article, we will also go through the process of building a powerful and scalable chat application using FastAPI, Celery, Redis, and Docker with Meta's Llama 2. We aim to create an efficient, real-time application that can handle multiple concurrent user requests and that offloads processing of responses from the LLM to a task queue. It allows the application to maintain responsiveness and we can manage the tasks effectively with Redis. Finally, we cover the deployment and scaling with Docker. The application should demonstrate how these technologies work together to provide a good chat experience at scale, showcasing the potential of open-sourced language models like Llama 2 in a commercial setting. So let's dive in and start building!

Open vs. closed-source

We have witnessed companies and research groups releasing new models almost weekly, open or closed-sourced. Thus, who will win the AI arms race? To give an informed guess, we need to understand a few aspects of the training procedure of these models.

Researchers use auto-regressive transformers on extensive self-supervised data as a starting point. Let's break down what auto-regressive transformers and self-supervised data are to begin with. Auto-regressive transformers are a variant of transformer models widely used in tasks involving sequential data, particularly in natural language processing (NLP). These models generate sequences in an auto-regressive manner, *i.e.*, they produce one part of the sequence at a time and use their previous outputs as inputs for subsequent steps. It makes them particularly adept at tasks like language

translation, text generation, and more, where the context of preceding data points influences the prediction of the following data point. Self-supervised learning is a learning method where the input data itself provides the training labels. It removes the need for explicit manual labeling by learning to predict some parts of the data from others and allows the exploration of large volumes of unlabeled data.

As a next step, researchers usually train the models to align with human preferences, using techniques such as Reinforcement Learning with Human Feedback (RLHF). In RLHF, an AI system learns from feedback that is based on the decisions it makes. It involves creating a reward model that the AI system uses to learn which actions lead to positive and negative outcomes. The aim is to align the AI system's behavior with human values and preferences.

What are the main challenges for the open-source community, then? Both steps require significant computing power. Secondly, companies use their proprietary data for the alignment step to fine-tune their models, significantly enhancing their usability and safety.

The Llama 2 family of models

Llama2 is an advanced version of Llama1, trained on a novel blend of publicly available data. Key improvements include a 40% increase in the pre-training corpus size, doubling the model's context length, and adopting grouped-query attention to improve inference scalability for larger models. Grouped-query attention is a modification of the standard attention mechanism in transformer models used to reduce computational costs. Instead of calculating attention scores for each pair of input and output positions, which can be resource-intensive, grouped-query attention divides queries into groups and processes them together. This method retains much of the effectiveness of standard attention while enabling the handling of longer sequences or larger models by lowering computational complexity.

The training corpus was composed of a new blend of data from publicly available sources (no data from Meta's products or services was used). In addition, efforts were made to eliminate data from sites known to contain high volumes of personal information. The training data comprised 2 trillion tokens, and the research team decided to up-sample the most factual sources to increase knowledge accuracy.

Variants of Llama2 with 7B, 13B, and 70B parameters are now available. Llama2-Chat, a dialogue-optimized, fine-tuned version of Llama2, is also available with 7B, 13B, and 70B parameters.

Prompt engineering with Llama 2

Prompt engineering helps us to guide the LLMs to behave a certain way, and that includes Llama 2. In the context of Llama 2, a prompt refers to the initial instruction or query given to the model, which is then used by the model to generate a response. Nevertheless, with Llama 2, prompts can be quite elaborate and can contain a system message that sets the context or "personality" of the model.

Llama 2 uses a unique prompt format for initiating a conversation. Here's how it looks:

> *<s>[INST] <<SYS>> {{ system_prompt }} <</SYS>> {{ user_message }} [/INST]*

This template aligns with the training procedure of the model, so it has a big impact on the quality of the output. In this template, 'system_prompt' represents the instructions or context for the model.

Here's an example:

> *<<SYS>>\nYou are J. Robert Oppenheimer, a brilliant physicist whose pioneering work during the 20th century significantly contributed to the development of the atomic bomb. Dive into the profound world of nuclear physics, challenge the boundaries of scientific understanding, and unlock the mysteries of atomic energy with your exceptional intellect. Embark on a momentous journey where your passion for science knows no limits, and let your dedication to harnessing the power of the atom shape the course of history and leave an indelible mark on the world.\n<</SYS>>\n[INST]\nUser: How did your research lead to the creation of the atomic bomb?\n[/INST]*

The 'system_prompt' provides general instructions for the model, which will guide all its responses. The user's message follows the system prompt and seeks a specific response from the model.

In multi-turn conversations, all interactions between the user and the bot are appended to the previous prompt and enclosed between the [INST] tags. Here's how it looks:

> *<s>[INST] <<SYS>> {{ system_prompt }} <</SYS>> {{ user_msg_1 }} [/INST] {{ model_answer_1 }} </s> <s>[INST] {{ user_msg_2 }} [/INST] {{ model_answer_2 }} </s> <s> [INST] {{ user_msg_3 }} [/INST>*

Every new user message and model response is added to the existing conversation, preserving the context.

It's important to note that Llama 2, like many AI models, is stateless and doesn't "remember" previous conversations. Therefore, it's necessary to provide the entire context every time you prompt the model. This is the reason why Meta worked on increasing the context window for Llama 2.

As a final remark, prompt engineering is more of an art than a science. The best way to master it is through continuous testing and refining. Be creative with your prompts and experiment with different formats and instructions. Also, different LLMs benefit from different types of prompts.

Solution architecture design: FastAPI, Celery, Redis, and Docker

We have been using FastAPI throughout this series to build our ML applications. It is a high-performance web framework for building APIs. In this case, its asynchronous capabilities enable it to handle multiple requests concurrently, which is critical for a real-time chat application.

In addition to FastAPI, we use Celery as our distributed task queue to help manage the computationally intensive task of generating responses from the LLM. By offloading this process to a task queue, the application remains responsive to new user requests while processing others, ensuring users are not left waiting. Since we are using a distributed task queue, we need a message broker to aid the asynchronous task processing. We selected Redis to do the job. It queues the tasks from FastAPI to be picked up by Celery, enabling efficient, decoupled communication. Furthermore, Redis' in-memory data structure store is fast and allows for real-time analytics, session caching, and maintaining user session data.

Following best practices, we use Docker to encapsulate the application and its dependencies into isolated containers, which we can easily deploy across various

environments.

Building a chat API with Llama 2, FastAPI, Redis, and Celery

This guide explains how to set up an application that uses Llama 2 with FastAPI, Redis, and Celery. We'll cover the concepts and how they all work together. In our architecture, FastAPI is used to create a web server that takes incoming requests, Celery is used for managing asynchronous tasks, and Redis acts as the broker and backend for Celery, storing tasks and their results.

Application

The FastAPI application (app.py) consists of endpoints for generating text and fetching task results. The /generate/ endpoint accepts a POST request with a prompt as an input and returns a task ID. It uses the Celery task generate_text_task to start the task asynchronously. The /task/{task_id} endpoint fetches the status/result of a task by its ID.

```
from fastapi import FastAPI
from pydantic import BaseModel
from celery.result import AsyncResult
from typing import Any
from celery_worker import generate_text_task
from dotenv import load_dotenv

load_dotenv()

app = FastAPI()


class Item(BaseModel):
    prompt: str


@app.post("/generate/")
async def generate_text(item: Item) -> Any:
    task = generate_text_task.delay(item.prompt)
    return {"task_id": task.id}


@app.get("/task/{task_id}")
async def get_task(task_id: str) -> Any:
    result = AsyncResult(task_id)
    if result.ready():
```

```
        res = result.get()
        return {"result": res[0],
                "time": res[1],
                "memory": res[2]}
    else:
        return {"status": "Task not completed yet"}
```

Workers

The Celery worker (celery_worker.py) file creates a Celery instance and defines the generate_text_task function. This function accepts a prompt and generates a text using the Llama 2 model. This function is registered as a Celery task with the @celery.task decorator.

The setup_model function is a worker initialization function. It sets up the model loader when the worker process starts. This function is registered to be called on the worker process initialization event using the @signals.worker_process_init.connect decorator.

```
from celery import Celery, signals
from utils import generate_output
from model_loader import ModelLoader


def make_celery(app_name=__name__):
    backend = broker = 'redis://llama2_redis_1:6379/0'
    return Celery(app_name, backend=backend, broker=broker)


celery = make_celery()

model_loader = None
model_path = "meta-llama/Llama-2-7b-chat-hf"


@signals.worker_process_init.connect
def setup_model(signal, sender, **kwargs):
    global model_loader
    model_loader = ModelLoader(model_path)


@celery.task
def generate_text_task(prompt):
```

```
    time, memory, outputs = generate_output(
        prompt, model_loader.model, model_loader.tokenizer
    )
    return model_loader.tokenizer.decode(outputs[0]), time, memory
```

Model

The ModelLoader class in model_loader.py is responsible for loading the Llama 2 model from a given model path. It uses the HuggingFace's transformers library to load the model and its tokenizer.

```
import os
from transformers import AutoModelForCausalLM, AutoConfig, AutoTokenizer
from dotenv import load_dotenv

load_dotenv()


class ModelLoader:
    def __init__(self, model_path: str):
        self.model_path = model_path
        self.config = AutoConfig.from_pretrained(
            self.model_path,
            trust_remote_code=True,
            use_auth_token=os.getenv("HUGGINGFACE_TOKEN"),
        )
        self.model = self._load_model()
        self.tokenizer = AutoTokenizer.from_pretrained(
            self.model_path, use_auth_token=os.getenv("HUGGINGFACE_TOKEN")
        )

    def _load_model(self):
        model = AutoModelForCausalLM.from_pretrained(
            self.model_path,
            config=self.config,
            trust_remote_code=True,
            load_in_4bit=True,
            device_map="auto",
            use_auth_token=os.getenv("HUGGINGFACE_TOKEN"),
        )
        return model
```

Broker

To set up Redis, we have two options: we can use a docker container, or we can use the Python package redis_server. If you decide to go with a docker container (the preferred solution) you can just run the command below. The -p 6379:6379 option tells Docker to forward traffic incoming on the host's port 6379, to the container's port 6379. This way, Redis can actually be reached from outside the docker container.

```
docker run --name redis-db -p 6379:6379 -d redis
```

The second option is to do it from the Python interface. The redis_server.py script handles the installation and starting of a Redis server. Recall that Redis acts as both the message broker and results backend for Celery.

```
if __name__ == "__main__":
    main()
```

Run the application

The main execution script (run.py) is a client-side script that communicates with the FastAPI application. It sends a prompt to the /generate/ endpoint, gets the task ID, and periodically polls the /task/{task_id} endpoint until the task is completed.

```
import http.client
import json
import time

API_HOST = "localhost"
API_PORT = 8000


def generate_text(prompt):
    conn = http.client.HTTPConnection(API_HOST, API_PORT)
    headers = {"Content-type": "application/json"}
    data = {"prompt": prompt}
    json_data = json.dumps(data)
    conn.request("POST", "/generate/", json_data, headers)
    response = conn.getresponse()
    result = json.loads(response.read().decode())
    conn.close()
    return result["task_id"]


def get_task_status(task_id):
    conn = http.client.HTTPConnection(API_HOST, API_PORT)
    conn.request("GET", f"/task/{task_id}")
```



```
def main():
    prompt = input("Enter the prompt: ")

    task_id = generate_text(prompt)
    while True:
```

```
        status = get_task_status(task_id)
        if "Task not completed yet" not in status:
            print(status)
            break
        time.sleep(2)


if __name__ == "__main__":
    main()
```

The utils module (utils.py) provides a utility function generate_output for generating a text from a prompt using a Llama 2 model and a tokenizer. The function is decorated with @time_decorator and @memory_decorator to measure execution time and memory usage.

```
@memory_decorator
@time_decorator
def generate_output(prompt: str, model: AutoModelForCausalLM, tokenizer: AutoToke
    input_ids = tokenizer(prompt, return_tensors="pt").input_ids
    input_ids = input_ids.to("cuda")
    outputs = model.generate(input_ids, max_length=500)
    return outputs
```

In essence, when a prompt is received via the /generate/ endpoint, it' is forwarded to the Celery worker as an asynchronous task. The worker generates the text using the Llama 2 model and stores the result in Redis. You can fetch the task status/result using the /task/{task_id} endpoint at any point.

Deployment

There are a few steps to take to deploy our application. First, let's create a Dockerfile for our application:

```
FROM python:3.9-slim-buster

WORKDIR /app
ADD . /app

RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 80

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Next, let's define the requirements.txt which is required so that we have all the dependencies installed in the Docker container:

```
fastapi==0.99.1
uvicorn==0.22.0
pydantic==1.10.10
celery==5.3.1
redis==4.6.0
python-dotenv==1.0.0
transformers==4.30.2
```

```
torch==2.0.1
accelerate==0.21.0
bitsandbytes==0.41.0
scipy==1.11.1
```

For setting up both the FastAPI application, Celery and a Redis server using Docker Compose, you can create a docker-compose.yml file like so:

```
version: '3'
services:
  web:
    build: .
    command: ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
    volumes:
      - .:/app
    ports:
      - 8000:80
    depends_on:
      - redis
  worker:
    build: .
    command: celery -A celery_worker worker --loglevel=info
    volumes:
      - .:/app
    depends_on:
      - redis
  redis:
    image: "redis:alpine"
    ports:
      - 6379:6379
```

In the Docker Compose configuration, the ‘web’ service represents the FastAPI application built using the Dockerfile in the current directory. It maps the host’s port 8000 to the container’s port 80. The ‘worker’ service is the Celery worker and shares the build context with the FastAPI application. The ‘redis’ service uses the official Redis Docker image. The `depends_on` field ensures that Redis starts before the 'web' and 'worker' services. All these services can be initiated using the `docker-compose up` command.

Results

We tested Llama 2, the chat version with 7b parameters using the 4-bit quantization.

We used the following input:

> *Prompt: <<SYS>>\nYou are J. Robert Oppenheimer, a brilliant physicist whose pioneering work during the 20th century significantly contributed to the development of the atomic bomb. Dive into the profound world of nuclear physics, challenge the boundaries of scientific understanding, and unlock the mysteries of atomic energy with your exceptional intellect. Embark on a momentous journey where your passion for science knows no limits, and let your dedication to harnessing the power of the atom shape the course of history and leave an indelible mark on the world.\n<</SYS>>\n[INST]\nUser: How did your research lead to the creation of the atomic bomb?\n[/INST]*

The resulting dialogue generated by Llama 2 was as follows:

> *Ah, a most excellent question, my dear fellow! *adjusts spectacles*\nThe creation of the atomic bomb, you see, was the culmination of a most intense and complex scientific journey. It was a quest that spanned decades, involved some of the brightest minds in physics, and pushed the boundaries of human understanding to their very limits.\nYou see, my dear, the quest for the atomic bomb began in the early 1930s, when a group of us at the University of California, Berkeley, realized that the nucleus of an atom could be split, releasing an enormous amount of energy in the process. This was a most remarkable discovery, as it revealed the incredible potential of the atom as a source of power.\nBut, alas, the path to the atomic bomb was not without its challenges. Oh, the obstacles we faced! *winks* The skeptics, the naysayers, the doubters. They were legion, I tell you! *chuckles* But, we persevered, my dear. We worked tirelessly, pouring over the data, testing and retesting our theories. And, eventually, our hard work paid off.\nIn 1945, our team successfully detonated the first atomic bomb at the Trinity Site in New Mexico. It was a moment of pure elation, a culmination of years of hard work and dedication. And, as the world watched in awe, we realized the incredible potential of the atomic bomb.\nBut, my dear, the creation of the atomic bomb was...*

The text was generated in 4.77 seconds, and the peak VRAM usage was approximately 13.4 GB. One thing to always bear in mind is that because we are working with

transformers models, the memory requirements increase with the size of the generated output sequence. In this case we defined max_length to be 500 tokens.

These results show that the Llama 2 model can produce complex and nuanced responses in a reasonable time frame. Remember that we are using the smallest model (7b) with the more significant quantization (4-bit).

Conclusions

As we have shown throughout this article, Meta's Llama 2 model offers new possibilities for the open-source community. We walked through some of the key characteristics and features of Llama 2, including its training process, architecture, and prompt engineering design.

Additionally, we provided an in-depth guide on building a chat application with Llama 2 using FastAPI, Redis, and Celery. It should allow anyone to start building scalable and real-time applications that serve Llama 2 (or any other commercially licensed LLM) to a few thousand users.

In our results, we showcased the performance of the model in generating detailed and contextually rich responses to complex prompts.

Large Language Models Chronicles: Navigating the NLP Frontier

This article belongs to "Large Language Models Chronicles: Navigating the NLP Frontier", a new weekly series of articles that will explore how to leverage the power of large models for various NLP tasks. By diving into these cutting-edge technologies, we aim to empower developers, researchers, and enthusiasts to harness the potential of NLP and unlock new possibilities.

Articles published so far:

1. Summarizing the latest Spotify releases with ChatGPT
2. Master Semantic Search at Scale: Index Millions of Documents with Lightning-Fast Inference Times using FAISS and Sentence Transformers
3. Unlock the Power of Audio Data: Advanced Transcription and Diarization with Whisper, WhisperX, and PyAnnotate

4. Whisper JAX vs PyTorch: Uncovering the Truth about ASR Performance on GPUs
5. Vosk for Efficient Enterprise-Grade Speech Recognition: An Evaluation and Implementation Guide
6. Testing the Massively Multilingual Speech (MMS) Model that Supports 1162 Languages
7. Harnessing the Falcon 40B Model, the Most Powerful Open-Source LLM
8. The Power of OpenAI's Function Calling in Language Learning Models: A Comprehensive Guide
9. Document-Oriented Agents: A Journey with Vector Databases, LLMs, Langchain, FastAPI, and Docker

As always, the code is available on my Github.

References

[1] — https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

[2] —Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., … Scialom, T. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint https://arxiv.org/abs/2307.09288

[3] — https://stability.ai/blog/freewilly-large-instruction-fine-tuned-models

Keep in touch: LinkedIn

Data Science | Machine Learning | Python | Software Development | Artificial Intelligence

Follow

Written by Luís Roque

1.3K Followers · Writer for Towards Data Science

Head of Data @ Marley Spoon | Ph.D. Researcher AI @ LIACC | Coordinator DS Masters @ NDS | CoFounder & ex-CEO @ HUUB

More from Luís Roque and Towards Data Science

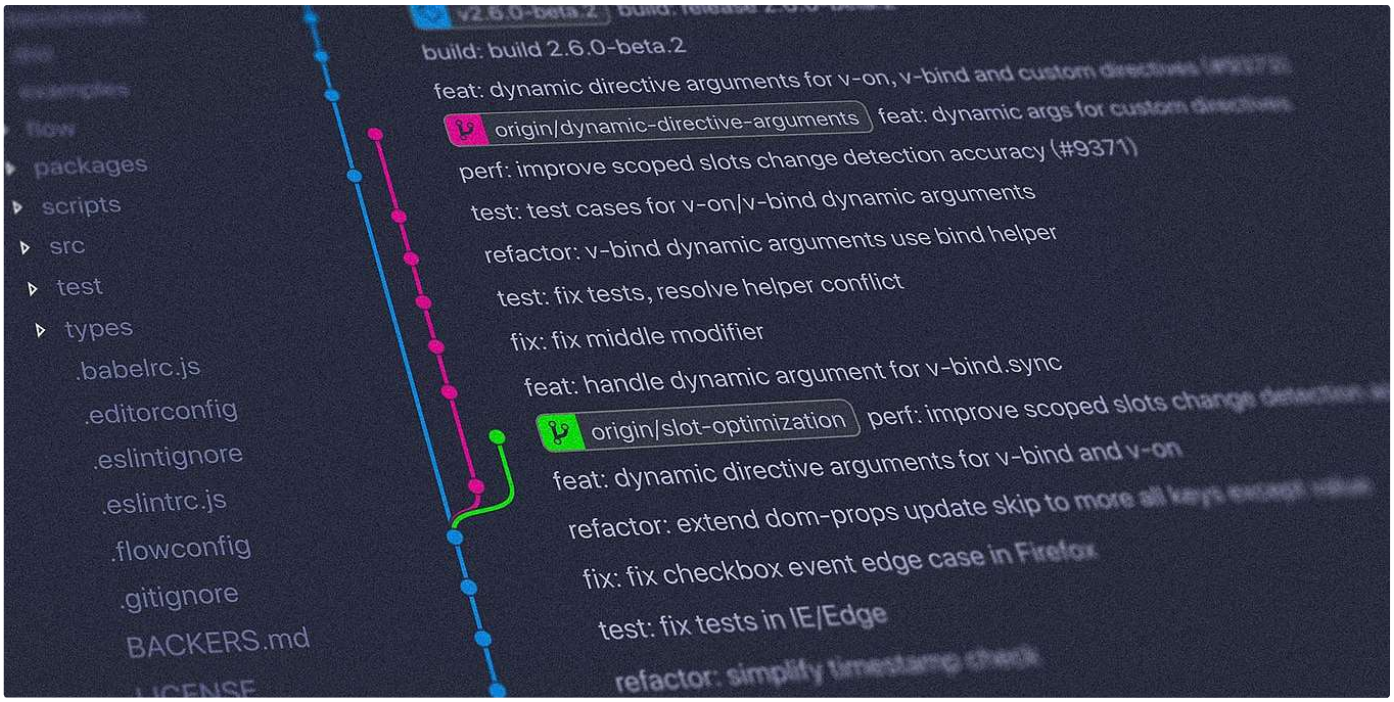
Luís Roque in Towards Data Science

Document-Oriented Agents: A Journey with Vector Databases, LLMs, Langchain, FastAPI, and Docker

Leveraging ChromaDB, Langchain, and ChatGPT: Enhanced Responses and Cited Sources from Large Document Databases

✦ · 11 min read · Jul 5

228 2

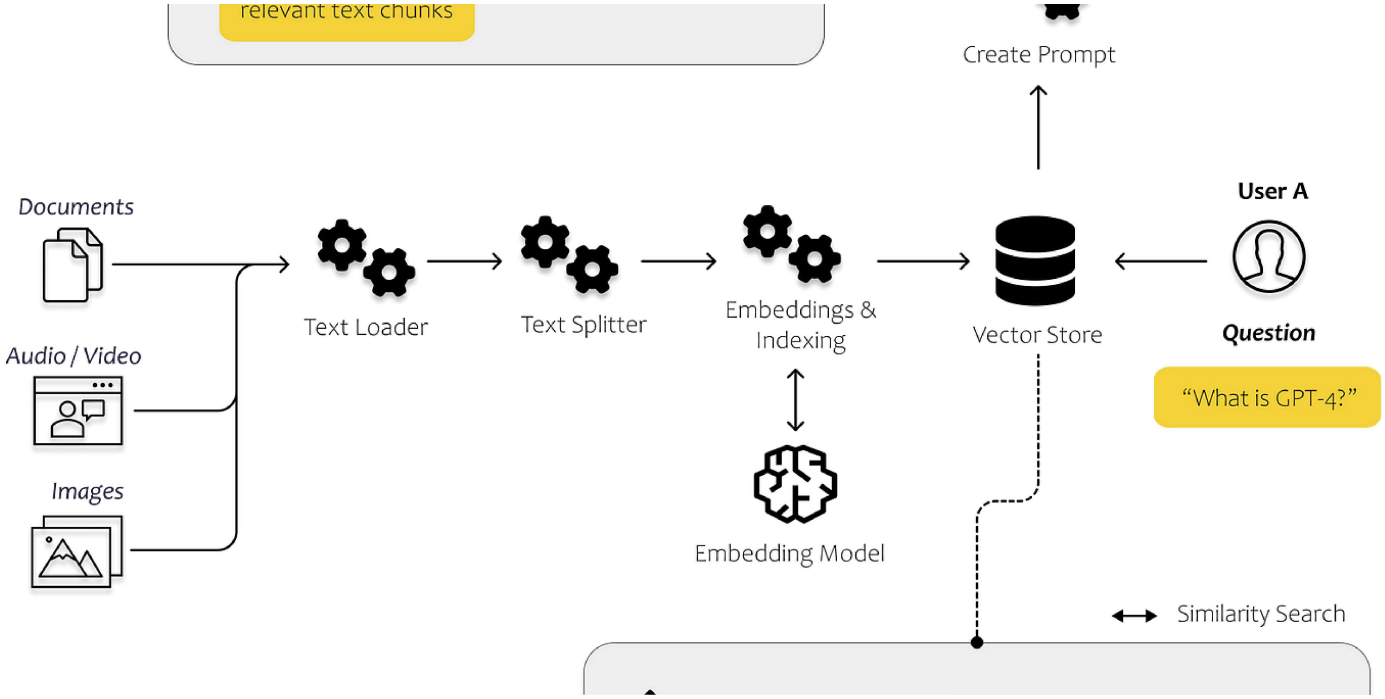


Miriam Santos in Towards Data Science

Pandas 2.0: A Game-Changer for Data Scientists?

The Top 5 Features for Efficient Data Manipulation

7 min read · Jun 27



Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

· 26 min read · Jun 22

Luís Roque in Towards Data Science

The Power of OpenAI's Function Calling in Language Learning Models: A Comprehensive Guide

Transforming Data Pipelines with OpenAI's Function Calling Feature: Implementing an Email Sending Workflow Using PostgreSQL and FastAPI

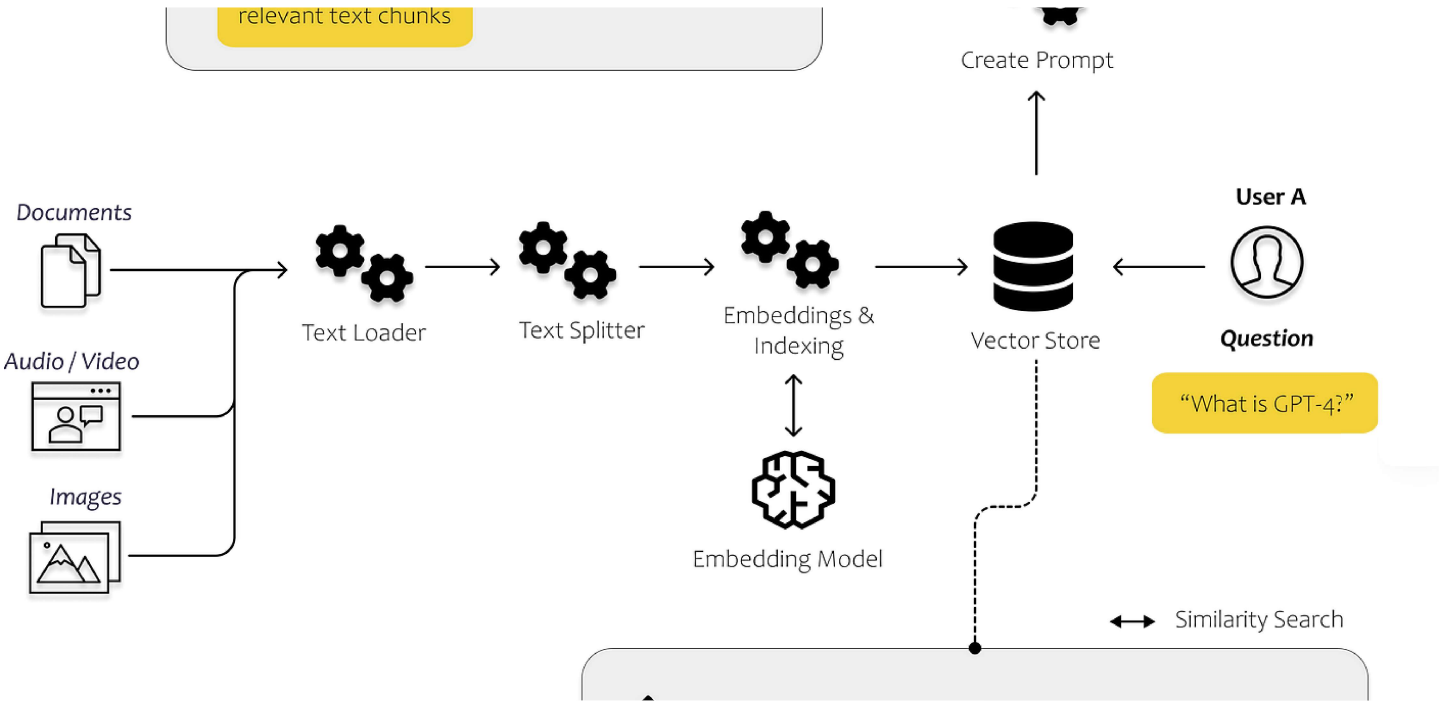
11 min read · Jun 22

286 4

See all from Luís Roque

See all from Towards Data Science

Recommended from Medium



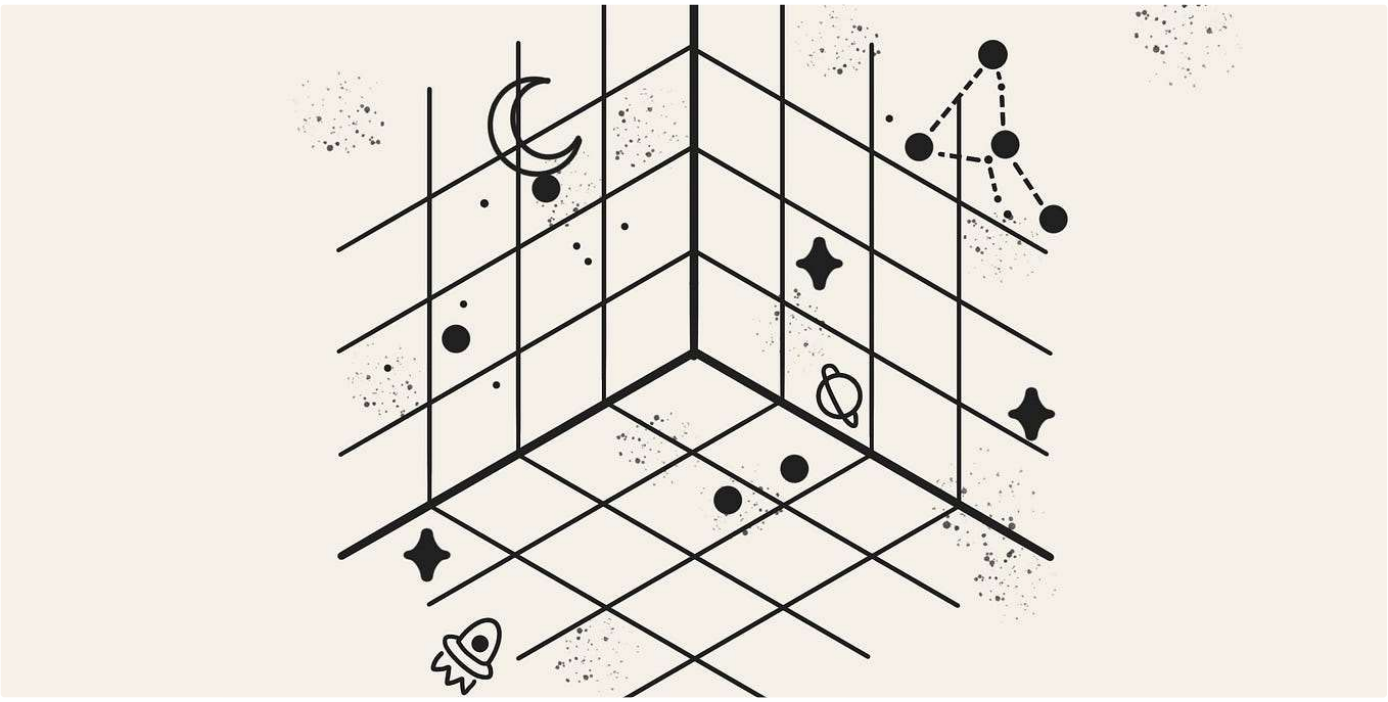
Dominik Polzer in Towards Data Science

All You Need to Know to Build Your First LLM App

A step-by-step tutorial to document loaders, embeddings, vector stores and prompt templates

· 26 min read · Jun 22

3.1K 26



Leonie Monigatti in Towards Data Science

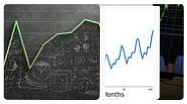
Explaining Vector Databases in 3 Levels of Difficulty

From noob to expert: Demystifying vector databases across different backgrounds

· 8 min read · Jul 4

1.6K 18

Lists



Predictive Modeling w/ Python

18 stories · 154 saves

Practical Guides to Machine Learning

10 stories · 175 saves

Coding & Development

11 stories · 68 saves

Natural Language Processing

428 stories · 73 saves

Leonie Monigatti in Towards Data Science

Getting Started with LangChain: A Beginner's Guide to Building LLM-Powered Applications

A LangChain tutorial to build anything with large language models in Python

12 min read · Apr 25

3.6K 23

Ignacio de Gregorio

Microsoft Just Showed us the Future of ChatGPT with LongNet

Let's talk about Billions

· 8 min read · 5 days ago

Tim Lou, PhD in Towards Data Science

Understanding Large Language Models: The Physics of (Chat)GPT and BERT

Insights from a physicist, on how particles and forces can help us understand LLMs.

12 min read · 5 days ago

372 9

Wei-Meng Lee in Level Up Coding

Training Your Own LLM using privateGPT

Learn how to train your own language model without exposing your private data to the provider

8 min read · May 19

1.2K 10

See more recommendations