# RAG's Anatomy

## Building a RAG app from scratch using Postgres and pgvector

**GÜLÇİN YILDIRIM JELÍNEK**
Staff Database Engineer

# SELECT * FROM ME ;

## Gülçin Yıldırım Jelínek
## Staff Engineer @ EnterpriseDB

Host @ **The Builders: A Postgres Podcast**
Co-Founder @ **Prague PostgreSQL Meetup**
Co-Founder & General Coordinator @ **Kadın Yazılımcı**

## Social

X: **@apatheticmagpie, @postgrespodcast, @kadinyazilimci, @PrahaPostgreSQL, @divaconference**

Linkedin: https://www.linkedin.com/in/gulcinyildirim

# Agenda

- What is RAG?
- Motivation behind the app
- Limitations with RAG apps
- Application Architecture

# What is RAG?

**RAG** is an architectural approach that can improve the efficacy of **large language model (LLM)** applications by leveraging **custom data**.

This is done by **retrieving data/documents** relevant to a **question or a task** and providing them as **context** for the **LLM**. RAG has shown success in support chatbots and Q&A systems that need to maintain up-to-date information or access **domain-specific** knowledge.

# The motivation behind the app

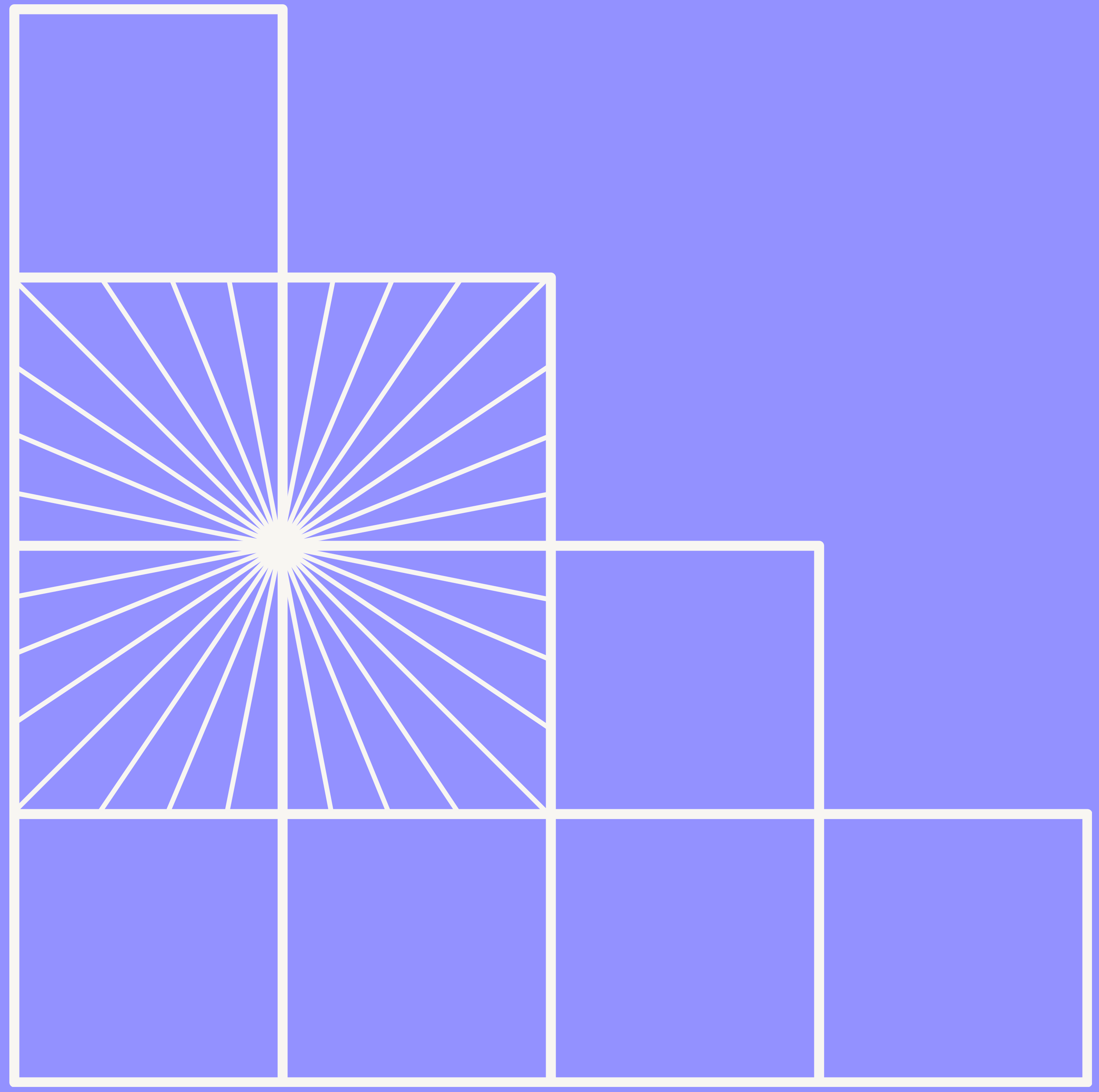
What do people want?

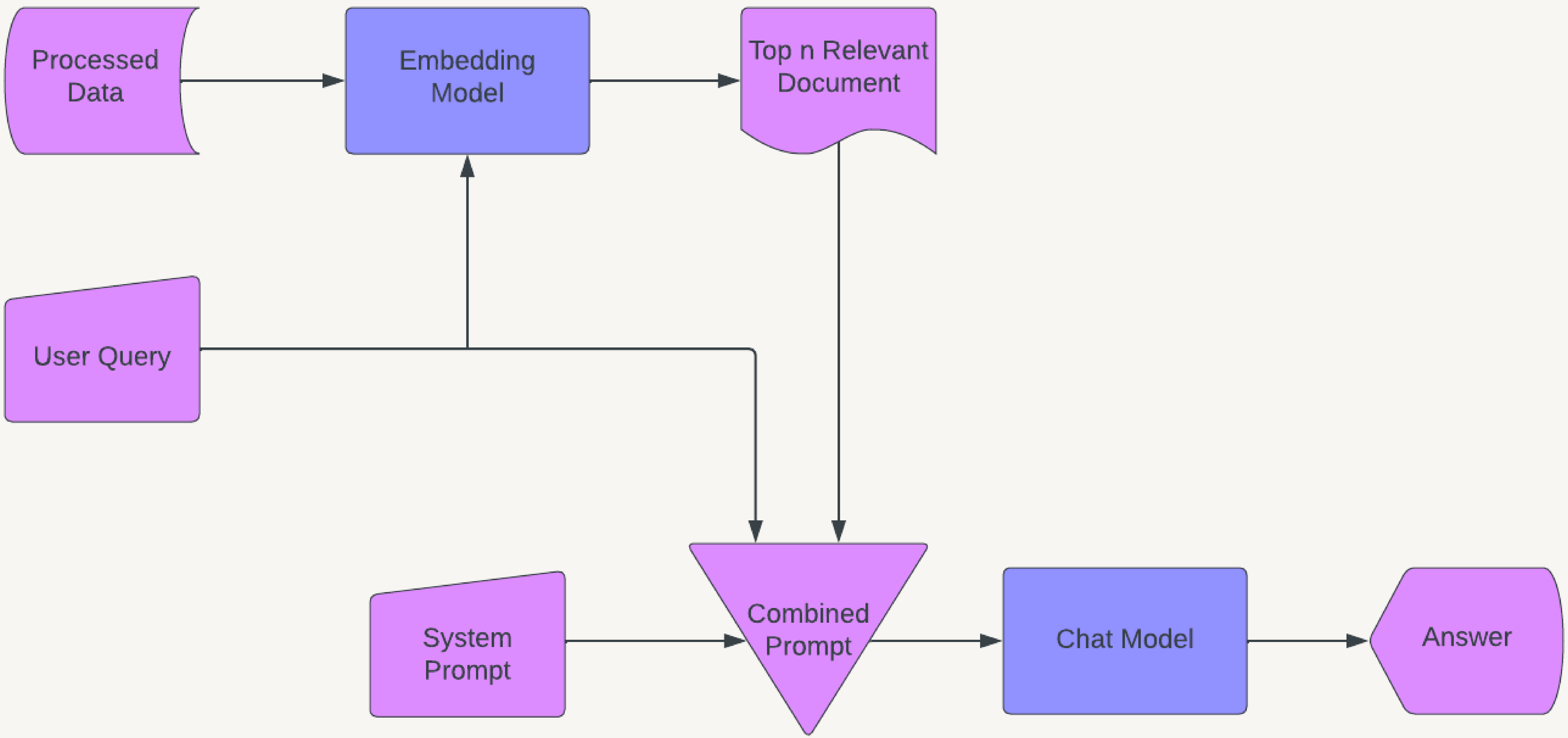
- Ability to ingest **different types data sources**
- **Data privacy concerns**, some would like to run their **LLM locally**
- Store and query vector data **directly in Postgres** using pgvector
- **Control and regulate** the vector search based on **roles/privileges**

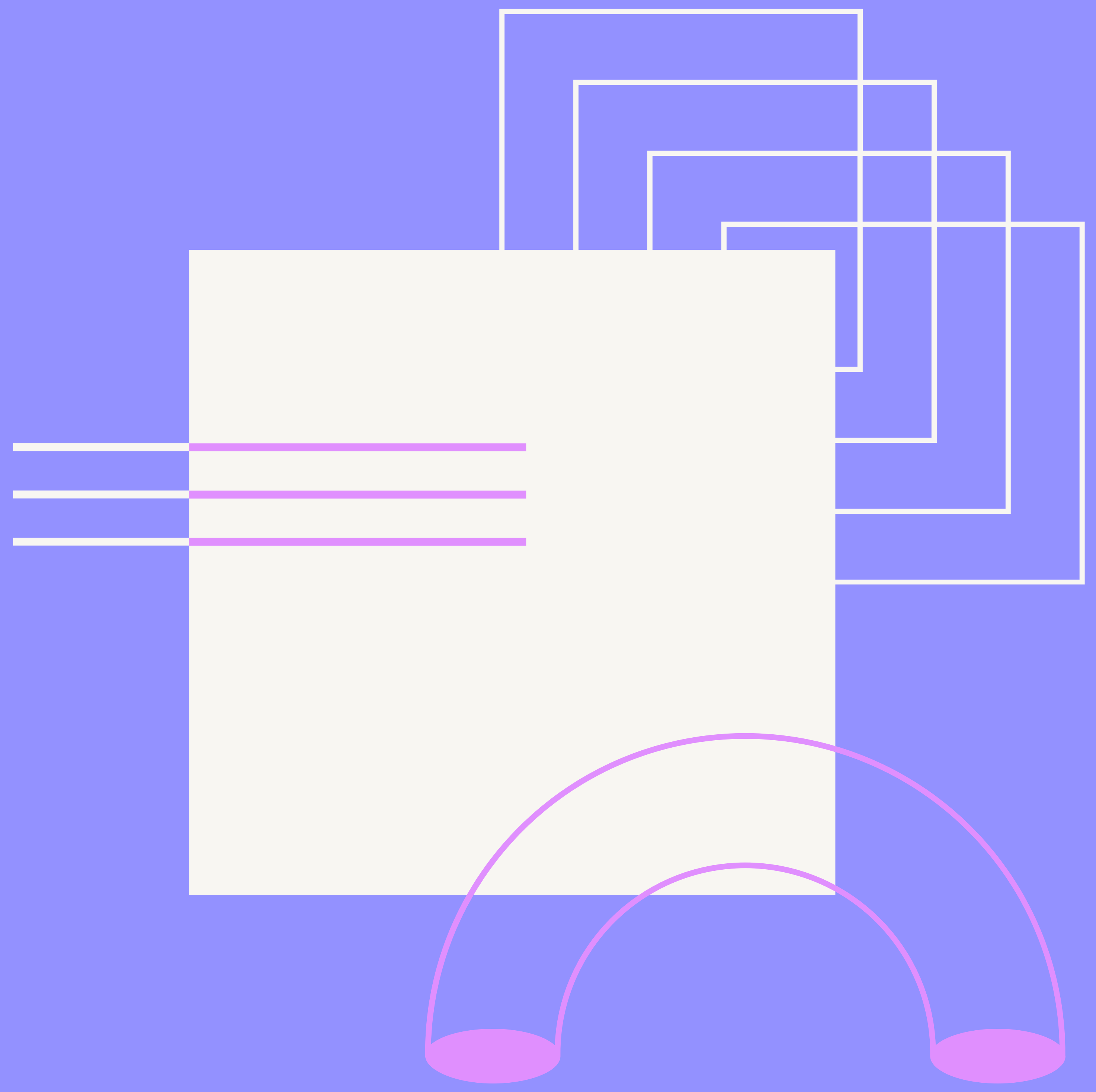
# Limitations with RAG apps

- Running LLMs on **CPU**, most models are optimized for **GPUs**
- Development and testing **locally** takes time
- Instructions of the RAG are limited by the **context window**
- **Scaling** the system to handle increased loads or larger models
- **Cost** of the environment

# Process flow of RAGs




Processed Data
Embedding Model
Top n Relevant Document
User Query
System Prompt
Combined Prompt
Chat Model
Answer



# Architecture

# Code is available here:

github.com/gulcin/pgvector-rag-app

```
(env) ubuntu@ip-172-31-2-121:~/pgvector/pgvector-rag-main$ pyth
```

# Requirements: Postgres, pgvector, Python3

app

rag

db

embeddings

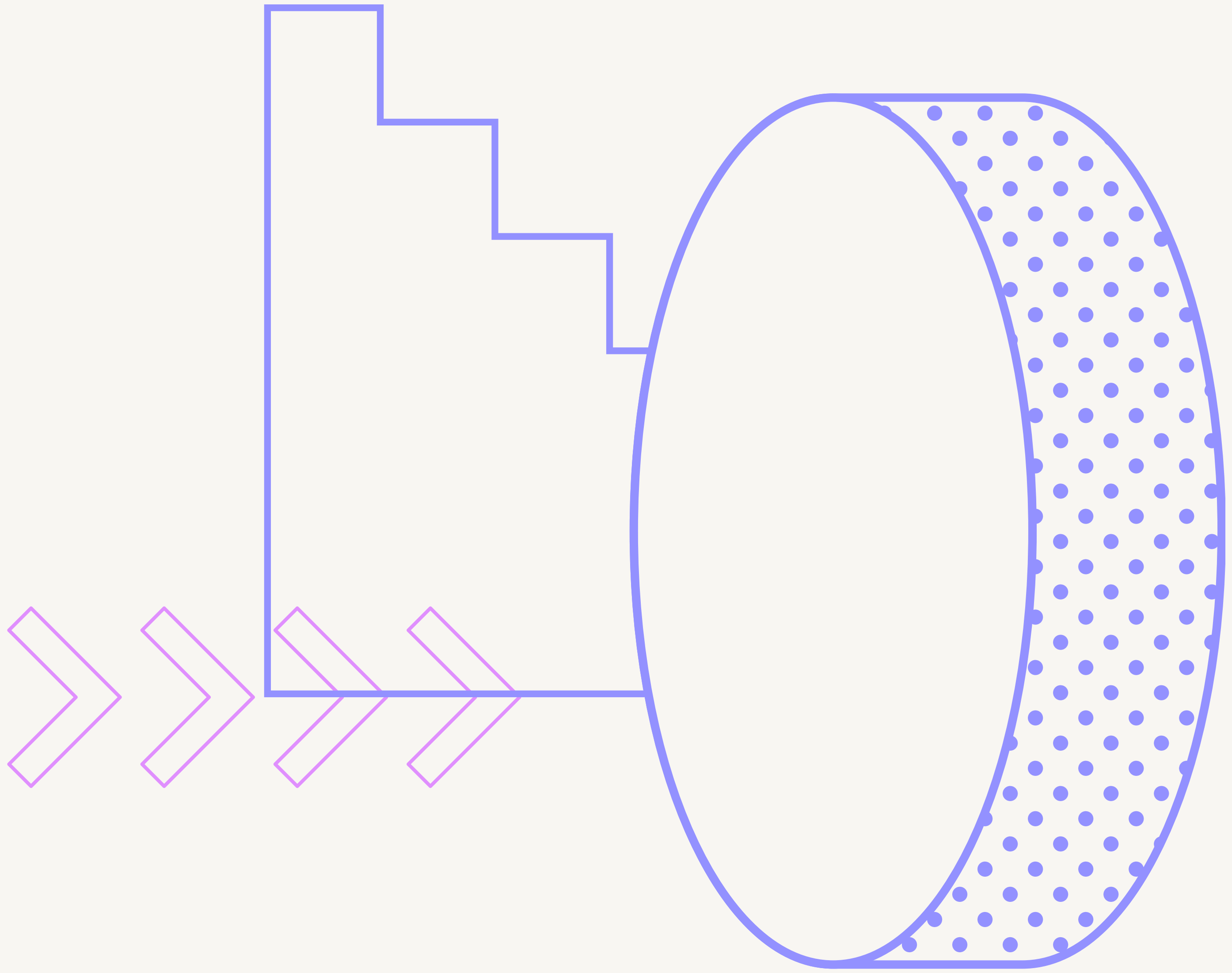
```
python app.py --help

usage: app.py [-h] {create-db,import-data,chat} ...

Application Description

options:
  -h, --help            show this help message and exit

Subcommands:
  {create-db,import-data,chat}
                        Display available subcommands
    create-db           Create a database
    import-data         Import data
    chat                Use chat feature
```

# create_db.py

# create_db.py

- Create database (if not exists) using **ENV** parameters, **DB_USER**, **DB_PASSWORD**, **DB_HOST**, **DB_PORT**
- Create pgvector extension if not exists
- Create embeddings table if not exists

```
CREATE EXTENSION IF NOT EXISTS vector;
CREATE TABLE IF NOT EXISTS embeddings (id serial PRIMARY KEY, doc_fragment text, embeddings vector(4096));
```



# import_data.py

# import_data.py

- Connect to DB
- Read pdf files
  - Takes a PDF file path as input, reads the text content of each page in the PDF file, splits it into lines, and returns the lines as a list
- Generate embeddings
  - Takes input text, tokenizes it, passes it through the LLM, retrieves the hidden states from the model's output, calculates the mean embedding, and returns both the original text and its corresponding embedding vector
- Store embeddings in the database
  - Store the document fragments and their embeddings in the database

# chat.py

# chat.py

Define a chat function that facilitates an interactive chat with a user. It continuously prompts the user for questions, generates responses using a specified model, and displays the response to the user until they choose to exit the chat.

```
def chat(args, model, device, tokenizer):
```

```
answer = rag_query(tokenizer=tokenizer, model=model, device=device, query=question)
```

# rag.py

```python
template = """<s>[INST]
You are a friendly documentation search bot.
Use following piece of context to answer the question.
If the context is empty, try your best to answer without it.
Never mention the context.
Try to keep your answers concise unless asked to provide details.

Context: {context}
Question: {question}
[/INST]</s>
Answer:
"""
```

# get_retrieval_condition

- Take query embedding and a threshold value as input
- Convert embedding into a string format suitable for SQL queries
- Construct an SQL condition for retrieving embeddings similar to the query embedding based on cosine similarity

```
def get_retrieval_condition(query_embedding, threshold=0.7):
    # Convert query embedding to a string format for SQL query
    query_embedding_str = ",".join(map(str, query_embedding))

    # SQL condition for cosine similarity
    condition = f"(embeddings <=> '{query_embedding_str}') < {threshold} ORDER BY embeddings <=> 
'{query_embedding_str}'"
    return condition
```

# rag_query

```
def rag_query(tokenizer, model, device, query):
    # Generate query embedding
    query_embedding = generate_embeddings(
        tokenizer=tokenizer, model=model, device=device, text=query
    )[1]

    # Retrieve relevant embeddings from the database
    retrieval_condition = get_retrieval_condition(query_embedding)

    conn = get_connection()
    register_vector(conn)
    cursor = conn.cursor()
    cursor.execute(
        f"SELECT doc_fragment FROM embeddings WHERE {retrieval_condition} LIMIT 5"
    )
    retrieved = cursor.fetchall()

    rag_query = ' '.join([row[0] for row in retrieved])

    query_template = template.format(context=rag_query, question=query)

    input_ids = tokenizer.encode(query_template, return_tensors="pt")

    # Generate the response
    generated_response = model.generate(input_ids.to(device), max_new_tokens=50,
pad_token_id=tokenizer.eos_token_id)
    return tokenizer.decode(generated_response[0][input_ids.shape[-1]:], skip_special_tokens=True)
```

# app.py

```python
if hasattr(args, "func"):
    if torch.cuda.is_available():
        device = "cuda"
        bnb_config = BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_use_double_quant=True,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_compute_dtype=torch.bfloat16
        )
    else:
        device = "cpu"
        bnb_config = None

    tokenizer = AutoTokenizer.from_pretrained(
        os.getenv("TOKENIZER_NAME"),
        token=os.getenv("HUGGING_FACE_ACCESS_TOKEN"),
    )
    model = AutoModelForCausalLM.from_pretrained(
        os.getenv("MODEL_NAME"),
        token=os.getenv("HUGGING_FACE_ACCESS_TOKEN"),
        quantization_config=bnb_config,
        device_map=device,
        torch_dtype=torch.float16,
    )

    args.func(args, model, device, tokenizer)
else:
    print("Invalid command. Use '--help' for assistance.")
```

# Further reading

- https://www.enterprisedb.com/blog/what-is-pgvector
- https://www.enterprisedb.com/blog/rag-app-postgres-and-pgvector

# Teşekkürler!

X: @apatheticmagpie @divaconference
@kadinyazilimci @PrahaPostgreSQL