

# Batching upserts

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE

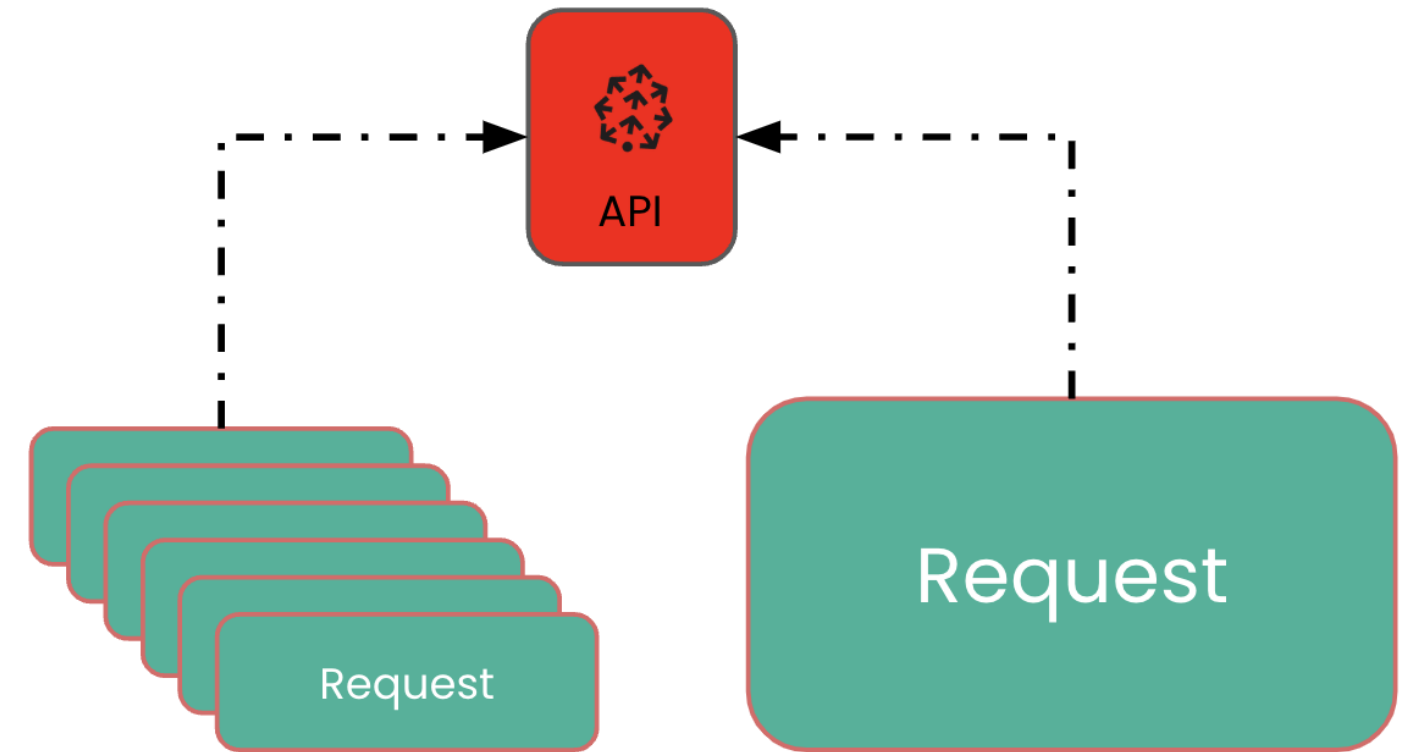


**James Chapman**

Curriculum Manager, DataCamp

# Upserting limitations

1. Rate of requests
  2. Size of requests
- **Batching:** breaking requests up into smaller *chunks*



<sup>1</sup> <https://docs.pinecone.io/reference/quotas-and-limits#rate-limits>

# Defining a chunking function

```
def chunks(iterable, batch_size=100):  
    it = iter(iterable)  
    chunk = tuple(itertools.islice(it, batch_size))  
    while chunk:  
        yield chunk  
        chunk = tuple(itertools.islice(it, batch_size))
```

# Sequential batching

- Splitting requests and sending them sequentially *one-by-one*

```
pc.Pinecone(api_key="YOUR API KEY")
index = pc.Index('datacamp-index')

for chunk in chunks(vectors):
    index.upsert(vectors=chunk)
```

## Pros:

- Solve rate and size limiting

## Cons:

- Really slow!

# Parallel batching

- Splitting requests and sending them in *parallel*

```
pc = Pinecone(api_key="YOUR_API_KEY", pool_threads=30)

with pc.Index('datacamp-index', pool_threads=30) as index:
    async_results = [index.upsert(vectors=chunk, async_req=True)
                      for chunk in chunks(vectors, batch_size=100)]

[async_result.get() for async_result in async_results]
```

# Let's practice!

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE

# Multitenancy and namespaces

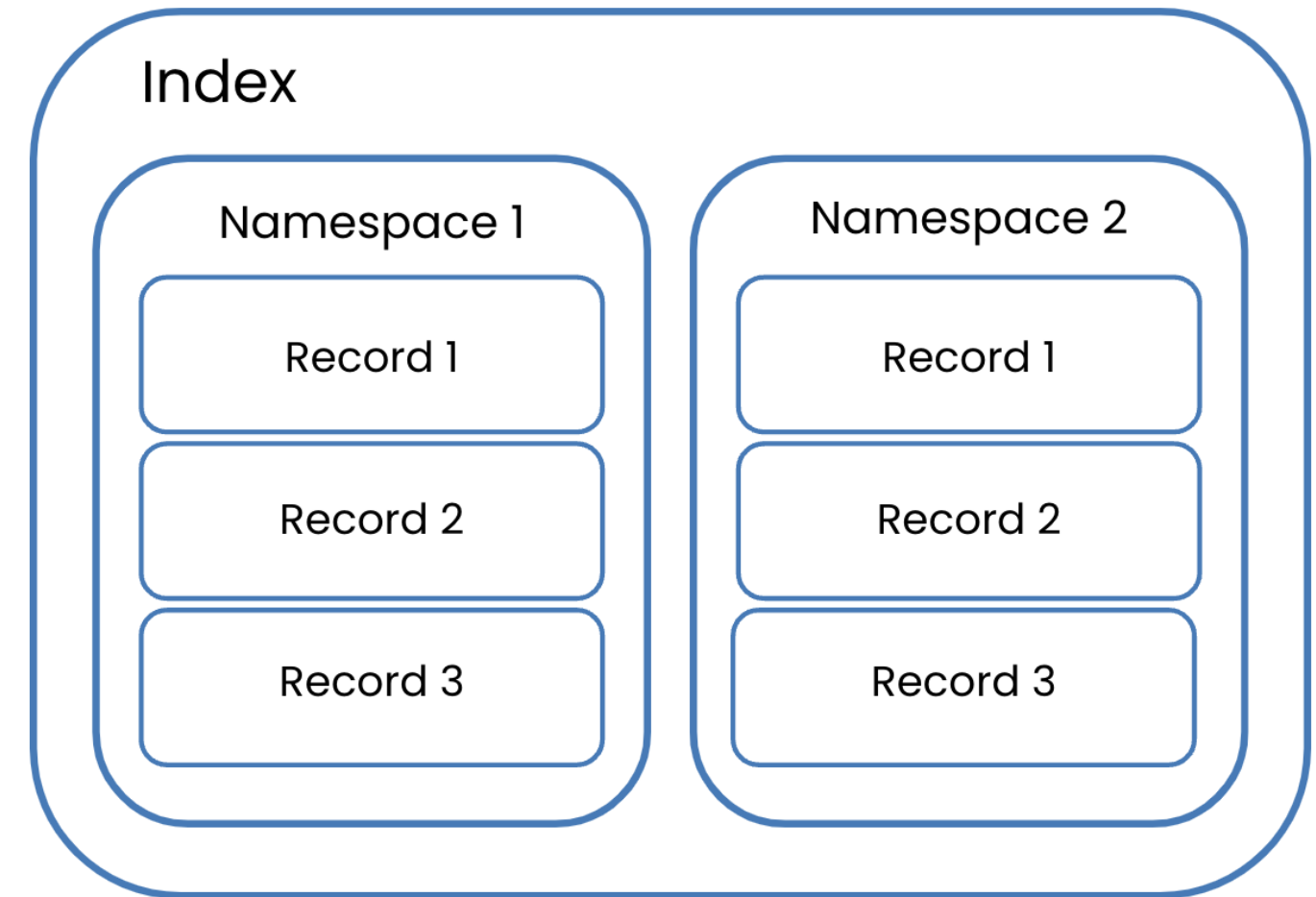
VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE



**James Chapman**  
Curriculum Manager, DataCamp

# Multitenancy

- Serve multiple tenants in *isolation*
- Separate different customers' data
  - **Security and privacy**
- Reduce **query latency**





# Multitenancy strategies

## 1. Namespaces

- **Advantages:** Reduces the need for additional indexes
- **Disadvantages:** Tenants share resources, complex data

## 2. Metadata Filtering

- **Advantages:** Allows querying across multiple tenants
- **Disadvantages:** Shared resources, challenging cost tracking

## 3. Separate Indexes

- **Advantages:** Physically separates tenants, allocates individual resources
- **Disadvantages:** Requires more effort and cost

# Namespaces

- Created *implicitly* during upsertion if they don't exist

```
index.upsert(  
    vectors=vector_set1, namespace="namespace1"  
)
```

```
index.upsert(  
    vectors=vector_set2, namespace="namespace2"  
)
```

# Inspecting namespaces

```
index.describe_index_stats()
```

```
{'dimension': 1536,  
 'index_fullness': 0.0,  
 'namespaces': {'namespace1': {'vector_count': 5},  
                 'namespace2': {'vector_count': 5}},  
 'total_vector_count': 10}
```

# Querying vectors from namespaces

```
query_result = index.query(  
    vector=vector,  
    namespace='namespace1',  
    top_k=3  
)
```

# Deleting vectors from namespaces

```
index.delete(  
    ids=["1", "2"],  
    namespace='namespace1'  
)
```

# Let's practice!

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE

# Semantic search with Pinecone

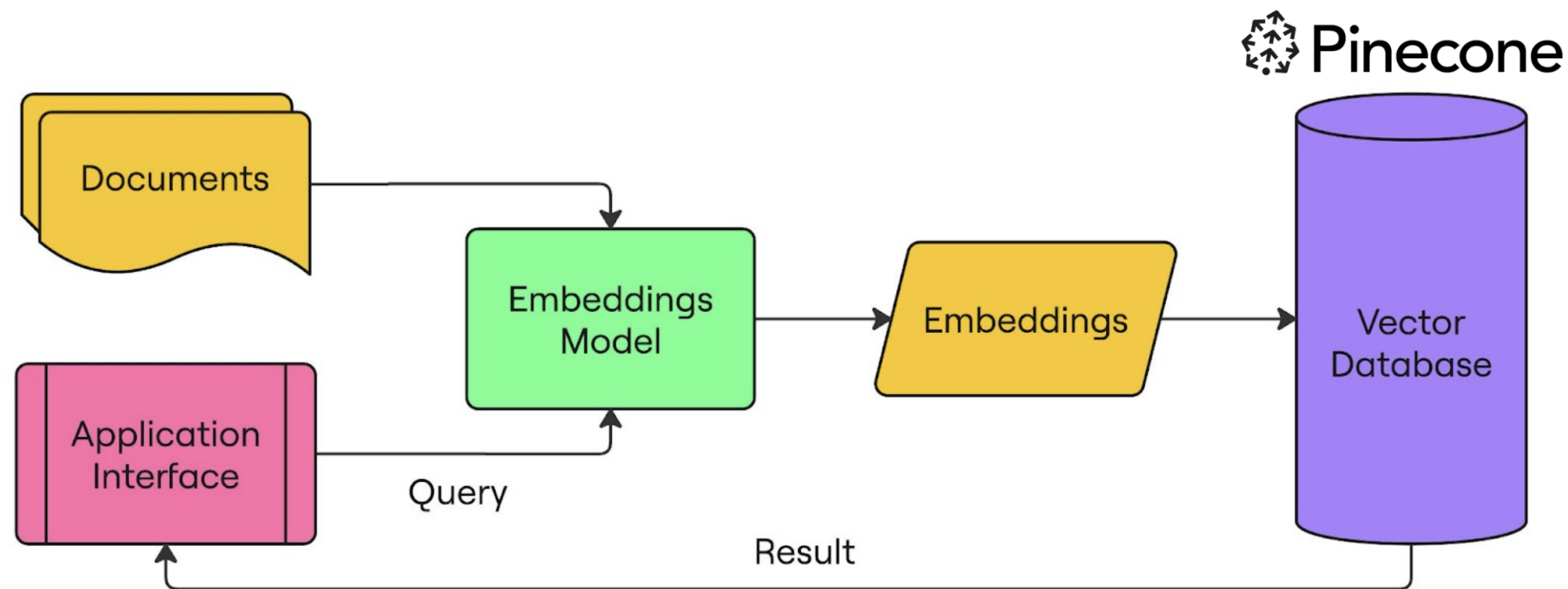
VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE



**James Chapman**  
Curriculum Manager, DataCamp

# Semantic search engines

1. Embed and ingest documents into a Pinecone index
2. Embed a user query
3. Query the index with the embedded user query





# Setting up Pinecone and OpenAI for semantic search

```
from openai import OpenAI
from pinecone import Pinecone, ServerlessSpec

client = OpenAI(api_key="OPENAI_API_KEY")
pc = Pinecone(api_key="PINECONE_API_KEY")

pc.create_index(
    name="semantic-search-datacamp",
    dimension=1536,
    spec=ServerlessSpec(cloud='aws', region='us-east-1')
)
index = pc.Index("semantic-search-datacamp")
```

# Ingesting documents to Pinecone index

```
import pandas as pd
import numpy as np
from uuid import uuid4

df = pd.read_csv('squad_dataset.csv')
```

| id | text  | title             |
|----|---|-------------------|
| 1  | Architecturally, the school has a Catholic cha... | University of ... |
| 2  | The College of Engineering was established in.... | University of ... |
| 3  | Following the disbandment of Destiny's Child in.. | Beyonce           |
| 4  | Architecturally, the school has a Catholic cha... | University of ... |

# Ingesting documents to Pinecone index

```
batch_limit = 100

for batch in np.array_split(df, len(df) / batch_limit):
    metadatas = [{"text_id": row['id'], "text": row['text'], "title": row['title']}
                  for _, row in batch.iterrows()]
    texts = batch['text'].tolist()
    ids = [str(uuid4()) for _ in range(len(texts))]

    response = client.embeddings.create(input=texts, model="text-embedding-3-small")
    embeds = [np.array(x.embedding) for x in response.data]

    index.upsert(vectors=zip(ids, embeds, metadatas), namespace="squad_dataset")
```

# Ingesting documents to Pinecone index

```
index.describe_index_stats()
```

```
{'dimension': 1536, 'index_fullness': 0.02,  
  'namespaces': {'squad_dataset': {'vector_count': 2000}},  
  'total_vector_count': 2000}
```

# Querying with Pinecone

```
query = "To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France?"

query_response = client.embeddings.create(
    input=query,
    model="text-embedding-3-small")

query_emb = query_response.data[0].embedding

retrieved_docs = index.query(vector=query_emb,
                             top_k=3,
                             namespace=namespace,
                             include_metadata=True)
```

# Querying with Pinecone

```
for result in retrieved_docs['matches']:
    print(f"{round(result['score'], 2)}: {result['metadata']['text']}")
```

0.41: Architecturally, the school has a Catholic character. Atop the Main Building gold dome is a golden statue of the Virgin Mary...

0.3: Because of its Catholic identity, a number of religious buildings stand on campus. The Old College building has become one of two seminaries...

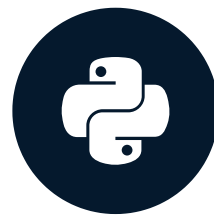
0.29: Within the white inescutcheon, the five quinas (small blue shields) with their five white bezants representing the five wounds...

# Time to build!

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE

# RAG chatbot with Pinecone and OpenAI

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE

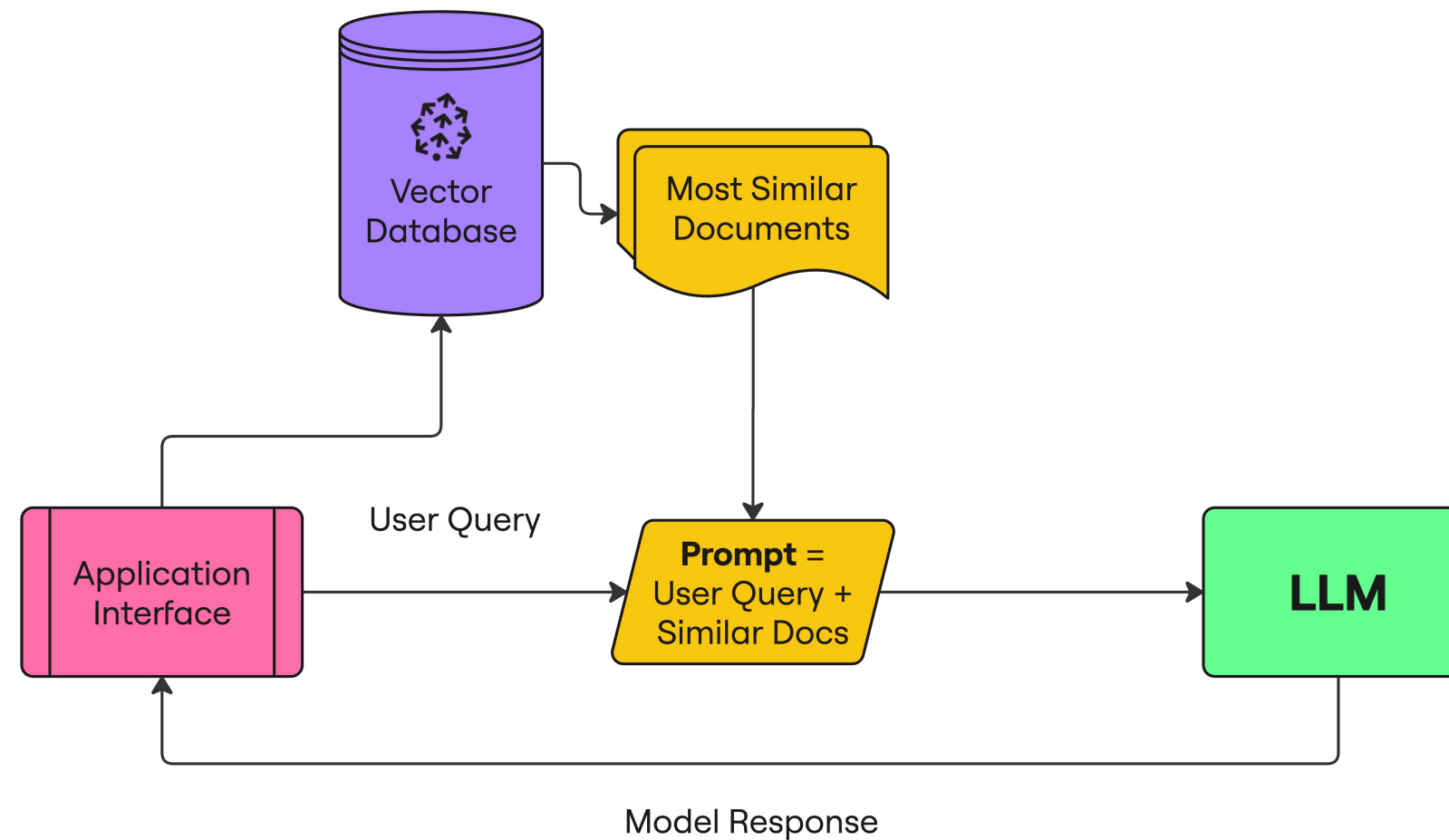


**James Chapman**  
Curriculum Manager, DataCamp



# Retrieval Augmented Generation (RAG)

1. Embed user query
2. Retrieve similar documents
3. Added documents to **prompt**



# Initialize Pinecone and OpenAI

```
from openai import OpenAI
from pinecone import Pinecone
import pandas as pd
from uuid import uuid4

client = OpenAI(api_key="OPENAI_API_KEY")
pc = Pinecone(api_key="PINECONE_API_KEY")

index = pc.Index("semantic-search-datacamp")
```

# YouTube transcripts

```
youtube_df = pd.read_csv('youtube_rag_data.csv')
```

|      |       |            |      |           |       |       |       |       |
|------|-------|------------|------|-----------|-------|-------|-------|-------|
| id   | blob  | channel_id | end  | published | start | text  | title | url   |
| ---- | ----- | -----      | ---- | -----     | ----- | ----- | ----- | ----- |
| int  | dict  | str        | int  | datetime  | int   | str   | str   | str   |

# Ingesting documents

```
batch_limit = 100

for batch in np.array_split(youtube_df, len(youtube_df) / batch_limit):
    metadatas = [{"text_id": row['id'], "text": row['text'],
                  "title": row['title'], "url": row['url'],
                  "published": row['published']} for _, row in batch.iterrows()]
    texts = batch['text'].tolist()
    ids = [str(uuid4()) for _ in range(len(texts))]

    response = client.embeddings.create(input=texts, model="text-embedding-3-small")
    embeds = [np.array(x.embedding) for x in response.data]

    index.upsert(vectors=zip(ids, embeds, metadatas), namespace='youtube_rag_dataset')
```

# Retrieval function

```
def retrieve(query, top_k, namespace, emb_model):
    query_response = client.embeddings.create(input=query, model=emb_model)
    query_emb = query_response.data[0].embedding

    retrieved_docs = []
    sources = []
    docs = index.query(vector=query_emb, top_k=top_k,
                       namespace='youtube_rag_dataset', include_metadata=True)

    for doc in docs['matches']:
        retrieved_docs.append(doc['metadata']['text'])
        sources.append((doc['metadata']['title'], doc['metadata']['url']))

    return retrieved_docs, sources
```

# Retrieval output

```
query = "How to build next-level Q&A with OpenAI"  
documents, sources = retrieve(query, top_k=3, namespace='youtube_rag_dataset',  
                             emb_model="text-embedding-3-small")
```

Document: To use for Open Domain Question Answering. We're going to start...

Source: How to build a Q&A AI in Python [...], <https://youtu.be/w1dMEWm7jBc>

Document: Over here we have Google and we can ask Google questions...

Source: How to build next-level Q&A with OpenAI, <https://youtu.be/coaaSxys5so>

Document: We need vector database to enhance the quality of Q&A systems...

Source: How to Build Custom Q&A Transfo [...], <https://youtu.be/ZIRmXKHp0-c>

# Prompt with context builder function

```
def prompt_with_context_builder(query, docs):  
    delim = '\n\n---\n\n'  
    prompt_start = 'Answer the question based on the context below.\n\nContext:\n'  
    prompt_end = f'\n\nQuestion: {query}\nAnswer:'  
  
    prompt = prompt_start + delim.join(docs) + prompt_end  
    return prompt
```

# Prompt with context builder output

```
query = "How to build next-level Q&A with OpenAI"  
context_prompt = prompt_with_context_builder(query, documents)
```

Answer the question based on the context below.

Context:

to use for Open Domain Question Answering. We're going to start with a few examples. Over here we have Google and we can ask Google questions like we would a normal person. So we can say, how do I tie my shoelaces? So what we have right here is three components to the question and answer. And I want you to remember these because these are relevant for what we are going to be building. We have the query at the top. We have what we can refer to as a context, which is the video, which is where we're getting this small, more specific answer from. And we can ask another question. Is Google SkyNet? So we have our question at the top. We have this paragraph, which is our context. And then we have the answer, which is yes, which is highlighted here. So it's slightly different to the previous one where we had the video. This time we have actual text, which is our context. And this is more aligned with what we will see throughout this video as well. Now, what we really want to be asking here is one, how does Google do that?

Question: How to build next-level Q&A with OpenAI

Answer:



# Question-answering function

```
def question_answering(prompt, sources, chat_model):
    sys_prompt = "You are a helpful assistant that always answers questions."
    res = client.chat.completions.create(
        model=chat_model,
        messages=[{"role": "system", "content": sys_prompt},
                  {"role": "user", "content": prompt}
    ], temperature=0)
    answer = res.choices[0].message.content.strip()
    answer += "\n\nSources:"
    for source in sources:
        answer += "\n" + source[0] + ": " + source[1]
    return answer
```

# Question-answering output

```
query = "How to build next-level Q&A with OpenAI"  
answer = question_answering(prompt_with_context, sources,  
                             chat_model='gpt-4o-mini')
```

To build next-level Q&A with OpenAI, you can start by fine-tuning a Q&A Transformer model using relevant datasets such as the SQuAD dataset (Stanford Question Answering Dataset). This involves downloading the dataset, organizing it in a folder, and then using it to train your model. Additionally, you can utilize OpenAI's generative models like GPT-2 to enhance the question-answering capabilities by providing context and prompting the model to generate intelligent responses. By following these steps and leveraging transformer-based solutions, you can create a more advanced Q&A system with OpenAI technology.

## Sources:

How to build a Q&A AI in Python (Open-domain Question-Answering): <https://youtu.be/w1dMEWm7jBc>

How to build next-level Q&A with OpenAI: <https://youtu.be/coaaSxys5so>

How to Build Custom Q&A Transformer Models in Python: <https://youtu.be/ZIRmXkHp0-c>

# Putting it all together

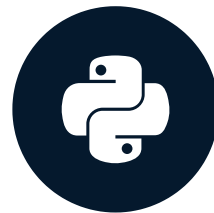
```
query = "How to build next-level Q&A with OpenAI"
documents, sources = retrieve(query, top_k=3,
                              namespace='youtube_rag_dataset',
                              emb_model="text-embedding-3-small")
prompt_with_context = prompt_with_context_builder(query, documents)
answer = question_answering(prompt_with_context, sources,
                              chat_model='gpt-4o-mini')
```

# Let's practice!

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE

# Congratulations!

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE

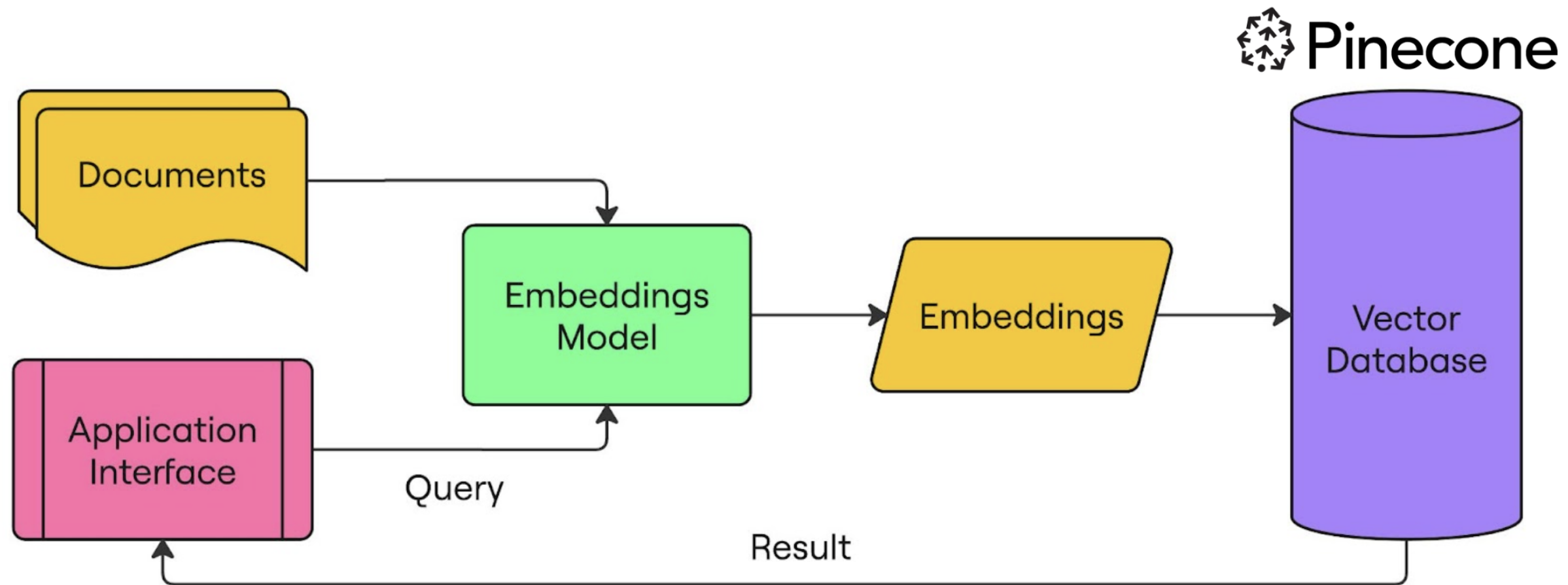


**James Chapman**

Curriculum Manager, DataCamp

# Chapter 1 - Introduction to Pinecone

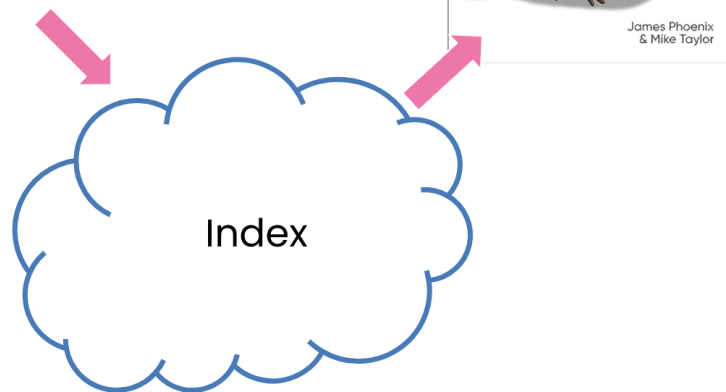
- Created an index → `pc.create_index()`
- Connect to it → `pc.Index()`
- Ingested vectors → `index.upsert()`



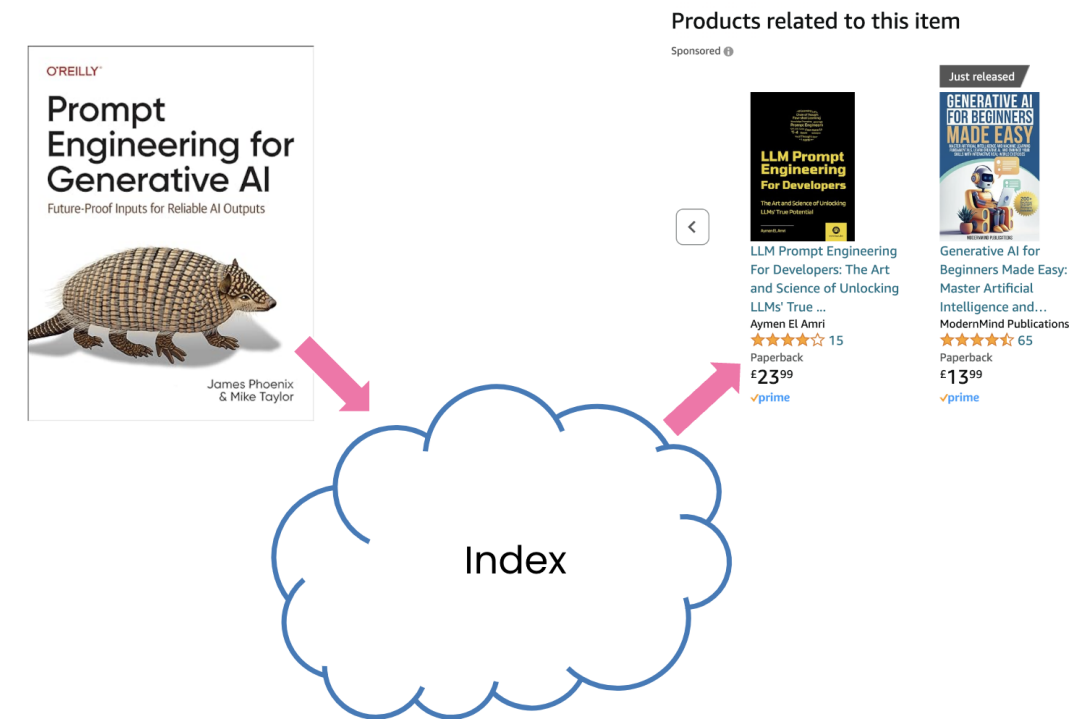
# Chapter 2 - Pinecone Vector Manipulation in Python

## Fetching

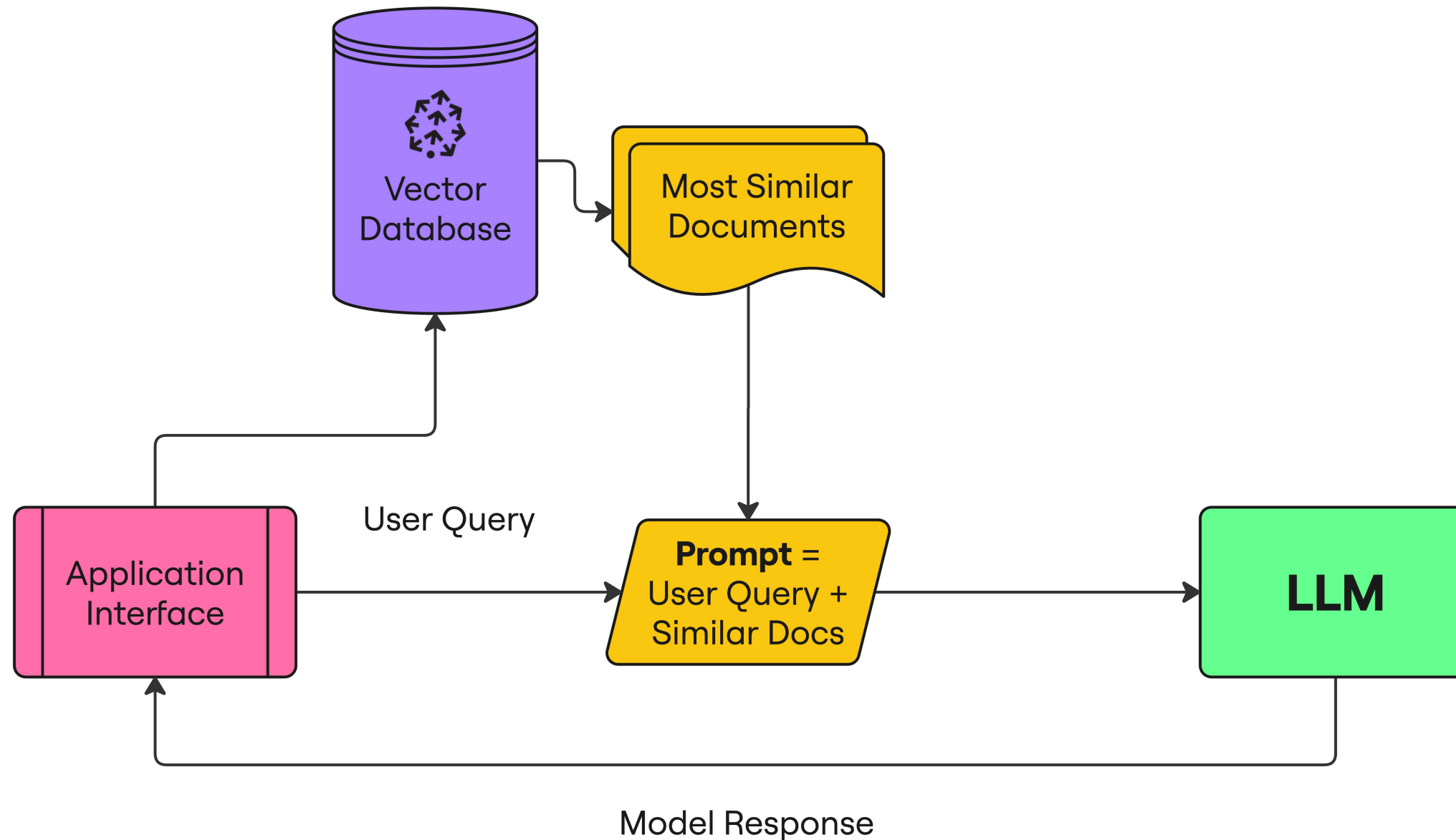
ISBN  
978-1-09-815343-4



## Querying



# Chapter 3 - Performance Tuning and AI Applications





# What's next?

## Integrations

- Setting up your Knowledge Base for Amazon Bedrock
- Retrieval Augmentation in LangChain
- Hugging Face Inference Endpoints



# Congratulations!

VECTOR DATABASES FOR EMBEDDINGS WITH PINECONE