# Crafting Context-Driven AI Chatbots: A Data Engineer’s Guide to Vector Databases and Embeddings

## **1. Introduction**

### 1.1 Purpose of the tutorial

In today’s data-driven world, it’s more important than ever to be able to sort through a lot of information and find what’s important. As a data engineer with a background in data science, I’ve seen how much semantic search could help solve this problem. Semantic search, unlike traditional keyword-based searches, digs into the meanings of words to find relevant documents in a more detailed and accurate way. The idea of “embeddings,” which are mathematical representations of words or papers in vector space, is at the heart of this ability.

Even though embeddings are becoming more important, many people still don’t understand what they are. Because they are abstract and contain a lot of math, they can be hard to understand. This tutorial is meant to make embeddings less mysterious by showing how they can be made, saved, and used with a vector database like Pinecone. By the end of this guide, readers will not only understand the basics of embeddings, but they will also know how to use them in a vector database to make semantic searches more powerful. Basically, I want to bridge the gap between complicated data science ideas and real-world engineering uses so that you can use semantic search in your projects.

### 1.2 Overview of the process: Documents -> Embeddings -> Vector Database -> Contextual LLM Prompt

Going from raw documents to AI-driven answers that make sense and are based on context requires a step-by-step process that combines the power of embeddings, vector databases, and large language models. Let’s break this trip down into steps:

* **Documents Base Creation**: We start the process with a selected set of thirty documents about a made-up company called StarTech. These papers are the main source of information and context for our whole system. They are the foundation on which everything else will be built.
* **Embeddings Generation**: We switch to embeddings to make sense of the text in these documents in a way that machines can understand. Each document is transformed into a vector of decimals. Even though these vectors look abstract, they capture the main ideas, concepts, and nuances of the document’s content in a way that can be used by programs.
* **Vector Database Integration with Pinecone**: Once the embeddings are made, we need a complex storage system that is optimized for vector operations. Comes Pinecone, a vector library that is made for this kind of work. By putting our document vectors in Pinecone, we can not only store them, but also run complex searches. When a query is run, Pinecone sorts through these vectors to find the “closest vector neighbors.” In simple words, it means looking for documents that best match the meaning of the query. This makes sure that the results are related semantically.
* **Crafting Contextual LLM Prompts**: The last step is to connect our system to a state-of-the-art language model, which in our case is LLama v2. We make a context-rich prompt for the LLama model based on documents that Pinecone finds. This lets the model come up with answers that aren’t just generic, but instead have a deep connection to the particular context given by the relevant StarTech documents.

This process is basically an ensemble of multiple technologies working together. It makes sure that when someone interacts with our system, they don’t just get a generic AI answer, but one that is backed up by specific, relevant information from our StarTech document base.

## **2. Preparing the Document Database**

Creating an accurate document database is certainly the crucial step of our project. Given that our AI chatbot will rely significantly on this database to generate relevant, context-driven responses, this repository must be created with care and precision. Let’s get into the details of this fundamental step:

* **The Importance of Quality Over Quantity**: While collecting a large collection of documents may be appealing, it is critical to ensure that each entry is succinct, clear, and complete. The goal is to provide our LLM with well-defined, structured information. Ambiguity or unnecessary details might obscure the context, resulting in less clear chatbot responses.
* **The Database’s Dynamic Nature**: One of the approach’s distinguishing features is its inherent flexibility. Unlike traditional models, where updates may demand complete retraining, our approach is built for scalability. As new information becomes available or the organization evolves, new documents can be easily added to the database. This means that our chatbot is always up to date and in sync with the newest company developments, without the need for time-consuming recalibrations. However, The document database, like any other repository of information, is not a’set-it-and-forget-it’ asset. Regular audits are recommended to verify information accuracy and relevancy.

For the purposes of this tutorial, I’ve created imaginary documents about StarTech using ChatGPT. It will be sufficient for readers to understand core concepts.

## **3. Generating Embeddings**

### 3.1 Introduction to embeddings and their relevance in AI chatbots

Embeddings are a way to turn words, phrases, or even whole documents into vectors of numbers, capturing their semantic meaning. The technique is based on sophisticated algorithms and neural networks trained on large datasets. Specific numbers in embeddings, contrary to common misconceptions, are not directly interpretable (like a linear regression for instance). It is instead about the relative locations in high-dimensional space: words or sentences with comparable meanings will be closer together.

It is difficult to build a model that generates reliable embeddings. It necessitates extensive training and a large volume of data. As a result, from an engineering standpoint, it frequently makes sense to leverage pre-existing, validated models. These provide reliable embeddings, so you don’t have to go through the complicated process of building one from scratch.

Embeddings are extremely useful for AI chatbots. They enable the chatbot to assess semantic similarity, ensuring that responses are relevant to the user’s goal. The good news is that, while the inner workings may appear complex, engineers do not need to know all of the details. The main lesson is that embeddings allow us to discover and exploit semantic linkages in language, allowing our chatbots to answer with greater relevance and context awareness.

As we will see later, the process of generating embeddings is straightforward when we use a pre-trained model.

### 3.2 Tools and frameworks for generating embeddings

Data professionals have many options when it comes to obtaining embeddings. While the notion of building a model from scratch may be appealing, it is a daunting task that is beyond the scope of this guide. Instead, we’ll look into simpler and ready-made options that promise efficiency and reliability.

1. **Open Source Models**: Using open-source models is one feasible option. For example, all-mpnet-base-v2 is a well-known model that may be configured to run locally on your workstation. This strategy is flexible and might be especially handy for individuals who do not want to rely on external servers or services.
2. **Open Source Models on Remote Servers**: These open-source models can be hosted by services that provide an API for simpler access. Hugging Face and Replicate are two noteworthy possibilities. Users can run embeddings on these platforms without the hassle of local setups. See for instance [all-mpnet-base-v2 on Replicate](https://replicate.com/replicate/all-mpnet-base-v2), [all-mpnet-base-v2 on Hugging Face](https://huggingface.co/sentence-transformers/all-mpnet-base-v2) or [all-MiniLM-L6-v2 on Hugging Face](https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2)
3. **Text-Embedding Model of OpenAI**: Finally, there’s OpenAI’s text-embedding-ada-002 model. It is a proprietary model designed for the generation of embeddings. This was chosen for our tutorial because it has a good balance between cost and performance. Its integration is easy and the outcomes are reliable.

### 3.3 Running embeddings on your document database

In this section, we’ll go over how to turn the text from our document database (documents.xlsx) into embeddings. The full code can be found in the companion github repository in src/01\_create\_database.py. Please if you want to follow along, get an openai token and store it in config.yaml after cloning the repository. Read README.md to get more indications on how to run the code.

Let’s break the code contained in this file.

1. **Imports**: The module begins by importing the dependencies required for this stage. Notably, we import the openai library, which will be used to produce embeddings from the Excel file’s documents.
2. **Loading the Excel File**: The Excel file is loaded into a pandas dataframe. This spreadsheet only contains the index, title and document columns. The index column is a unique identifiers for each row and the document column contains the document content.
3. **The get\_embedding function**:

def get\_embedding(text, model="text-embedding-ada-002"):  
 text = text.replace("\n", " ")  
 return openai.Embedding.create(input=[text], model=model)["data"][0]["embedding"]

The purpose of the get\_embedding function is to generate the embedding of the text that is given as input. This is directly taken from the openai docs with no modifications. A few things worth noting: \* The newline characters in the text are deleted to make sure the entry is clean. \* The function calls  OpenAI’s API to create the embedding for the given text, using the text-embedding-ada-002 model.

1. **Data processing**: The embeddings are stored in a new column of our pandas dataframe. After that, we may permanently store the dataframe:

df["embeddings"] = df.apply(lambda x: get\_embedding(x["document"]), axis=1)  
 df.to\_csv("data/documents\_processed.csv", index=False)

By using this simple approach, you can easily turn your document collection into a file that includes embeddings. The beauty is in how simple it is: with just a few lines of code, each document in your original collection is added to with its own embedding. This embedded data is now the foundation of our AI chatbot’s ability to understand context as we will see in the next steps.

## **4. Uploading Vectors to Pinecone: A Vector Database**

### 4.1 Introduction to Vector Databases

In the age of the AI, which is changing businesses and bringing about new ideas that have never been seen before, the need for efficient and specialized data processing is important. This is especially true for apps that take advantage of the power of big language models, semantic search, and generative AI. All of these AI tools use vector embeddings, which is what connects them together. These are special ways of representing data that are full of meaningful insights. They give AIs a repository of knowledge and a long-term memory that are important for some tasks.

Vector databases are specialized storage systems that have been carefully made to hold vectors. In contrast to other databases, a vector database is based on its ability to manage and query these numerical representations. When an index is created in this kind of database, the length of the vectors it can hold is set ahead of time and stays the same throughout the index. Further, these databases can use mathematical measures like Euclidean or cosine distances to figure out how far away two vectors are from each other. Notably, these databases use complex algorithms like the approximate nearest neighbors to find the ‘n’ vectors that are most similar to a given vector almost instantly.

Some examples of these cutting-edge vector libraries that are making waves in the AI world are:

* [Chroma](https://www.trychroma.com/)
* [Milvus](https://milvus.io/)
* [Pinecone](https://www.pinecone.io/)
* [Qdrant](https://qdrant.tech/)

Not only do these databases store vectors, but they are also highly tuned to support fast, efficient retrieval, which is important for AI applications.

### 4.2 Uploading Vectors to Pinecone

Pinecone is a vector database that is both reliable and simple to use. Since it is fully managed, much of the operational overhead is taken care of for you. This makes it a great choice for this lesson. Pinecone is easy to use, and it comes with a large free tier, which is great for new users. But if you want to follow along, you should know that you will need an account and, as a result, a unique token for verification. The full code can be found in the file src/02\_create\_records\_in\_pynecone.py

Let’s look at the code that allows us to send vectors to Pinecone:

1. **Initialization and Configuration**: This part imports the required modules, reads the file that we created during the previous step, and sets up Pinecone with the required API key and environment setting. Specifically, we import the pinecone Python library to communicate with the database.

import pandas as pd  
 import pinecone  
 from .load\_config import load  
  
 df = pd.read\_csv("data/documents\_processed.csv")  
 pinecone\_token = load("config.yaml")["tokens"]["pinecone"]  
 pinecone\_env = load("config.yaml")["parameters"]["pinecone\_env"]  
 pinecone.init(api\_key=pinecone\_token, environment=pinecone\_env)

1. **Create Index in Pinecone**: Here, an attempt is made to make an index named “startech” with a certain size (1536) and associated with the cosine metric to measure the distance between vectors. If the index already exists, the code will just show a message. Then, it gets the newly made (or existing) index.

try:  
 pinecone.create\_index("startech", dimension=1536, metric="cosine")  
 except:  
 print("index already exists")  
  
 print(pinecone.list\_indexes())  
 index = pinecone.Index("startech")

1. **Convert Embeddings to List**: Since our embeddings are saved as strings that look like lists, we need a function to turn them back into real lists of floats for processing. That’s what this function does.

def create\_a\_list\_from\_list\_as\_string(x):  
 ll = x.replace("[", "").replace("]", "").split(",")  
 return [float(element) for element in ll]   
  
 df["embeddings\_as\_list"] = df["embeddings"].apply(create\_a\_list\_from\_list\_as\_string)

1. **Upload Records**: This part gets our records ready for insertion (or “upsertion”). It goes through our dataframe, combining each document’s index with its embedding, and then adds them to the Pinecone index.

records = []  
 for row in df.iterrows():  
 records.append((str(row[1]["index"]), row[1]["embeddings\_as\_list"]))  
  
 index.upsert(records)

## **5. Querying the Vector Database**

In this part, we see how to ask the vector database for the most relevant documents based on a user prompt. We were able to do this by putting together OpenAI for embeddings and Pinecone for the vector database.

The approach starts with a question from the user, like “What is StarTech’s market share?” Our goal is to find the most relevant documents in our vector database for this prompt.

We first use the OpenAI model to turn the user’s input into a vector (or embedding). Then, we send this vector to Pinecone and ask for the top-k vectors in the database that have the shortest distance with it (using cosine in our case). The Pinecone database gives us the IDs of these vectors, which are the documents from our original dataset.

The relevant file is located at src/03\_query\_vector\_database.py

Embedding the prompt is done with the same function as when we generated embeddings on our documents collection:

prompt\_embedding = openai.Embedding.create(input = [prompt], model = "text-embedding-ada-002")['data'][0]['embedding']

Then, we connect to the “startech” Pinecone index and do a query using the prompt’s vector to find the three vectors in the database that are most like it:

index = pinecone.Index("startech")  
result = index.query(  
 vector=prompt\_embedding,  
 top\_k=3,  
 include\_values=False  
)  
print(result)

Here the three most relevant documents are stored in results:

{'matches': [{'id': '26', 'score': 0.895502687, 'values': []},  
 {'id': '4', 'score': 0.89512372, 'values': []},  
 {'id': '8', 'score': 0.849597216, 'values': []}],  
 'namespace': ''}

## **6. Integrating with Large Language Models (LLM)**

Now that we have these documents, it is easy to use an LLM of your choice with this additional context added for the model to use in its answer.

For this part, this guide will use Llama v2 which is an open source model released by Meta on July 18, 2023. To avoid the hassle of running it locally on a GPU machine, we can use [Replicate.com](https://replicate.com/) which is a service that runs open source models through a rest apis. To use Replicate, create an account and get a token as with OpenAI and Pinecone. Of course it is also possible to run it locally but you will need to download the model first.

The code in src/03\_query\_vector\_database.py. It is still the same script but the second part is dedicated to interacting with the LLM.

We start by retrieving relevant documents thanks to the documents IDs from the previous step:

doc1 = int(result["matches"][0]["id"])  
doc2 = int(result["matches"][1]["id"])  
doc3 = int(result["matches"][2]["id"])  
df = pd.read\_csv("data/documents\_processed.csv")  
doc\_1\_content = df.loc[df["index"]==doc1, "document"].values[0]  
doc\_2\_content = df.loc[df["index"]==doc2, "document"].values[0]  
doc\_3\_content = df.loc[df["index"]==doc3, "document"].values[0]

Please note that this basic example use pandas to get the content of these documents from a csv. In a real world deployment, a proper database will be used.

The data obtained is then added to the prompt:

prompt = prompt + "\n" + doc\_1\_content  
prompt = prompt + "\n" + doc\_2\_content  
prompt = prompt + "\n" + doc\_3\_content

Finally we can query the LLM:

output = replicate.run(  
 "replicate/llama-2-70b-chat:2c1608e18606fad2812020dc541930f2d0495ce32eee50074220b87300bc16e1",  
 input={"prompt": prompt, "system\_prompt": system\_prompt}  
)  
print("Output:")  
print("".join(list(output)))

Note that the system prompt is a general prompt that sets the general behavior of the model. It is possible to modify it in this file: src/load\_config.py

The model can now answer questions with relevant information given to it just-in-time:

Prompt: Is StarTech a company that try to limit its impact on the environment ?  
  
Output:  
Yes, StarTech appears to be a company that prioritizes limiting its impact on the environment. The company has implemented sustainable manufacturing practices that have reduced carbon emissions by 40% since 2018, and has initiatives in place to promote gender equality and diversity. Additionally, StarTech is committed to reducing waste and implementing cleaner production methods. The company's commitment to sustainability is evident in their efforts to expand their global footprint, increase R&D investments, and diversify their product range while aiming to minimize their environmental impact.

Here are copies of the documents that the model used to answer the question:

Document 1:  
Modern investors prioritize ESG values, and StarTech is at the forefront. Our sustainable manufacturing practices have reduced carbon emissions by 40% since 2018. Furthermore, our workforce initiatives ensure gender equality and promote diversity, making us a responsible investment choice.  
=========================================  
Document 2:  
Understanding the environmental footprint of aerospace, StarTech is committed to sustainable manufacturing practices. Our recycling programs have significantly reduced waste, and our push for cleaner production methods is making a noticeable difference in our carbon footprint.  
=========================================  
Document 3:  
As we soar into the next decade, StarTech aims to expand its global footprint, increase R&D investments, and diversify our product range. With plans to penetrate the space tourism sector and develop components for reusable rockets, the sky is not the limit for us.

## **7. Conclusion**

This small experiment shows how powerful a flexible this approach can be to leverage semantic search and using your own knowledge with LLMs. By using language models that have already been trained and vector databases, it is possible to make systems that are useful in your context and can process and understand a lot of text data. One of the best things about this method is that you don’t have to keep retraining models. By not retraining, we don’t have to deal with the storage and processing problems that come with big model weights.

This method is also easy to try out and change because it is made up of separate parts. We used the Llama v2 model in our test because it is new and showed promise. But it is possible to experiment with any model. Readers can easily try out different language models, like those from OpenAI, to see how well they do on the job. The code and tools are also flexible enough to be used in different ways and with different data sets.

Also, by combining a vector database with large language models, we make a scalable system that can handle even the biggest collections of documents. The vector database does the hard work of quickly finding relevant documents, while the large language models provide the semantic understanding needed to make sense of the material. This strong combination makes sure that our method will still work even as the amount of data keeps growing.

## **Further reading and resources**

[Embeddings - OpenAI API](https://platform.openai.com/docs/guides/embeddings/what-are-embeddings)  
[Multi-modal ML with OpenAI’s CLIP | Pinecone](https://www.pinecone.io/learn/series/image-search/clip/)