

## Local UNLIMITED Memory Ai Agent | Ollama RAG Crash Course

Ai Austin  
21.1K subscribers

Subscribe

2.8K



Share

Ask

Download

...

75,181 views Jul 11, 2024 #ai #aiagents #programming

Learn how to create powerful Ai agents with Python in this easy to follow along crash course on Ollama RAG. In this video we build a RAG agent that stores every conversation in a PostgreSQL database, converts the SQL database into a vector embedding database on program startup using Ollama to run the open source Nomic embedding model and then logically retrieves multiples needles from the haystack of context before generating a response with the open source LLM of your choice.



```
assistant.py > ...
1  import ollama
2  import chromadb
3  import psycopg
4  import ast
5  from psycopg.rows import dict_row
6  from tqdm import tqdm
7  from colorama import Fore
8
9  client = chromadb.Client()
10 > system_prompt = ...
11  convo = [{"role": "system", "content": system_prompt}]
12
13 > DB_PARAMS = ...
14
15 > def connect_db(): ...
16
17 > def fetch_conversations(): ...
18
19 > def create_vector_db(conversations): ...
20
21 > def create_queries(prompt): ...
22
23 > def classify_embedding(query, context): ...
24
25 > def retrieve_embeddings(queries, results_per_query=2): ...
26
27 > def store_conversation(prompt, response): ...
28
29 > def remove_last_conversation(): ...
30
31 > def stream_response(prompt): ...
32
33 > def recall(prompt): ...
```

We will be building a RAG system with local DB, storage, embeddings and open source LLM for inference.

# First Principles Retrieval Augmented Generation

```
o  o  o
austindobbins@Austins-Laptop ollama_recall % python3 assistant.py
USER:
/recall using everything you know about me, what should i create youtube video on next?

Vector database queries: ["What are your hobbies?", "What topics do you enjoy learning about?", "Are there any areas where you're interested in sharing your expertise or experiences?", "Have you recently learned something new that you're excited to share with others?", "What type of content do you typically consume on YouTube?" ]

Processing queries to vector database: 100% | 5/5 [00:31<00:00, 6.32s/it]

5 message:response embeddings added for context.

ASSISTANT:
Based on your interests and preferences, I think a great topic for your next YouTube video could be exploring the applications of large language models (LLMs) in creative fields. You've mentioned being interested in AI and having a realistic approach to creating useful AI applications.

You could create a video that showcases how LLMs can be used to generate music, poetry, or even stories, and how these tools can be used creatively. This topic aligns with your interest in lofi music and old-school rap/rock, as well as your goal of helping people see the potential of AI without overselling it.

Additionally, you could also create a video that compares different LLMs, their strengths, and weaknesses, providing a balanced view for your audience. This would be an informative and engaging video that showcases your expertise in AI and programming.

What do you think? Would this be a topic you'd enjoy exploring further on your channel?

USER:
```



Get up and running with large language models.

Run [Llama\\_3](#), [Phi 3](#), [Mistral](#), [Gemma](#), and other models. Customize and create your own.

[Download ↓](#)

Available for macOS, Linux, and Windows (preview)

Build a Local Ai Agent to Memorize ...

# Build a Local Ai Agent to Memorize EVERYTHING | Ollama RAG Crash Course

## PRO Tutorials

'PRO Tutorial' for people who want to fully understand the code  
'LAZY PRO Tutorial' for people who want to run the program with no explanation of how it works

- ▶ PRO Tutorial
- ▶ LAZY PRO Tutorial
- ▼ Source Code

 assistant.py 7.6KB  
 requirements.txt 0.1KB

### Program Overview:

In this tutorial we use Python & PostgreSQL to create a first principle RAG agent that stores and can retrieve any conversation you had with the agent in the past. This program is 100% a local, open source model agent that give you 100% data privacy from big tech.

Microsoft Recall and even ChatGPT's memory feature rolled out and activated on all users without telling us, shows that big tech is becoming more and more interested in using these LLM's to create AI's that know us, maybe better than we want big tech to know us.

This program shows how to create a memory agent that outperforms big tech's tools with smaller open source models. All conversations are stored on your device and processed with local Ollama embedding models and LLM's.

### Program Requirements:

- Python 3.11 or 3.12
- At least 8gb of RAM

python.org

Python PSF Docs PyPI Jobs Community

python™

Download the latest version for macOS

Download Python 3.12.4

Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other

Want to help test development versions of Python 3.13? Pre-releases, Docker images



Active Python Releases

local\_rag\_agent

EXPLORER LOCAL\_RAG\_AGENT

requirements.txt

```
1 psycopg==3.1.19
2 colorama==0.4.6
3 chromadb==0.5.0
4 tqdm==4.66.4
5 ollama==0.2.1
6 |
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS zsh +

```
austindobbins@Austins-Laptop local_rag_agent % python3 -m pip install -r requirements.txt
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS zsh + ⌘ ⌘ ⌘ ⌘ ⌘ ⌘ ⌘ ⌘ ⌘ ⌘

```
b<1.0,>=0.16.4->tokenizers>=0.13.2->chromadb==0.5.0->r requirements.txt (line 3)) (2024.6.0)
Requirement already satisfied: zipp>=0.5 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from importlib-metadata<7.1,>=6.0->opentelemetry-api>=1.2.0->chromadb==0.5.0->r requirements.txt (line 3)) (3.19.2)
Requirement already satisfied: MarkupSafe>=2.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from jinja2>=2.11.2->fastapi>=0.95.2->chromadb==0.5.0->r requirements.txt (line 3)) (2.1.5)
Requirement already satisfied: markdown-it-py>=2.2.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from rich>=11.0->typer>=0.9.0->chromadb==0.5.0->r requirements.txt (line 3)) (3.0.0)
Requirement already satisfied: pygments>=3.0.0,>=2.13.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from rich>=10.11.0->typer>=0.9.0->chromadb==0.5.0->r requirements.txt (line 3)) (2.18.0)
Requirement already satisfied: humanfriendly>=9.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from coloredlogs->onnxruntime>=1.14.1->chromadb==0.5.0->r requirements.txt (line 3)) (10.0)
Requirement already satisfied: mpmath<1.4.0,>=1.1.0 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from sympy->onnxruntime>=1.14.1->chromadb==0.5.0->r requirements.txt (line 3)) (1.3.0)
Requirement already satisfied: mdurl<=0.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from markdown-it-py>=2.2.0->rich>=10.11.0->typer>=0.9.0->chromadb==0.5.0->r requirements.txt (line 3)) (0.1.2)
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages (from pyasn1-modules>=0.2.1->google-auth>=1.0.1->kubernetes>=28.1.0->chromadb==0.5.0->r requirements.txt (line 3)) (0.6.0)

[notice] A new release of pip is available: 24.0 -> 24.1
[notice] To update, run: pip3 install --upgrade pip
austindobbins@Austins-Laptop local_rag_agent %
```

## llama3

Meta Llama 3: The most capable openly available LLM to date

8B 708

↳ 4M Pulls Updated 5 weeks ago



## CLI

Open the terminal and run `ollama run llama3`

## API

Example using curl:

```
curl -X POST http://localhost:11434/api/generate -d '{
    "model": "llama3",
    "prompt": "Why is the sky blue?"
}'
```

[API documentation](#)

## Model variants

Instruct is fine-tuned for chat/dialogue use cases.

Example:

`ollama run llama3`

8B 8x22B

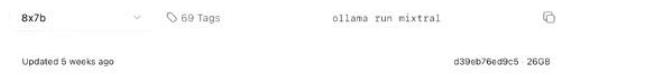
↳ 309.4K Pulls Updated 5 weeks ago

## mixtral

A set of Mixture of Experts (MoE) model with open weights by Mistral AI in 8x7b and 8x22b parameter sizes.

8x7B 8x22B

↳ 309.4K Pulls Updated 5 weeks ago



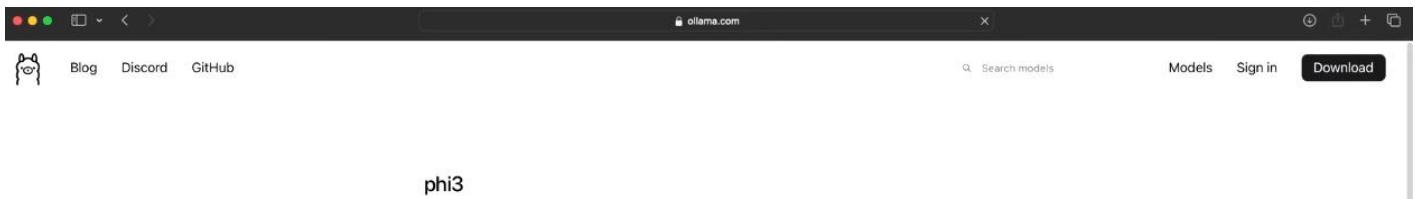


The Mistral large Language Models (LLM) are a set of pretrained generative Sparse Mixture of Experts.

#### Sizes

- mixtral:8x22b
- mixtral:8x7b

#### Mixtral 8x22b



#### phi3

Phi-3 is a family of lightweight 3B (Mini) and 14B (Medium) state-of-the-art open models by Microsoft.

3B 14B

2M Puffs Updated 3 weeks ago

3.8b

70 Tags

ollama run phi3



Updated 4 weeks ago

64c1188f2485 · 2.4GB



Phi-3 is a family of open AI models developed by Microsoft.

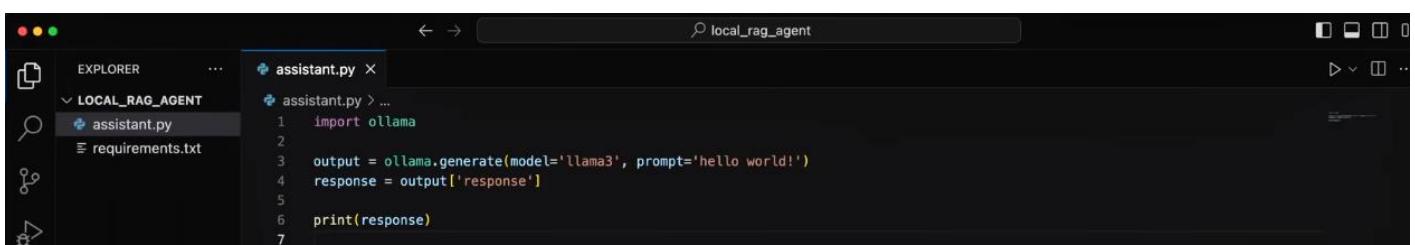
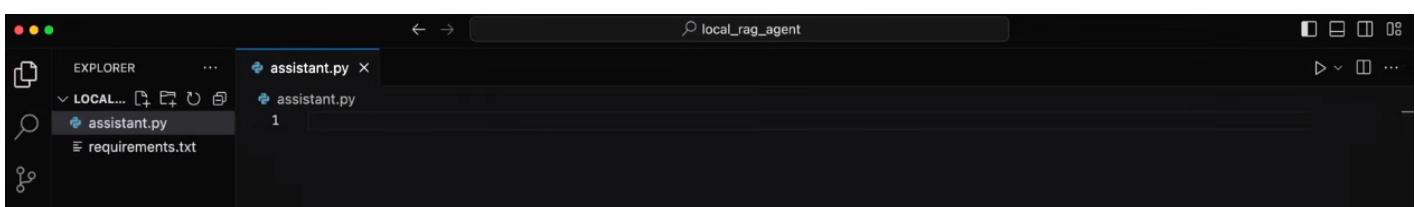
#### Parameter sizes

- Phi-3 Mini – 3B parameters – ollama run phi3:mini
- Phi-3 Medium – 14B parameters – ollama run phi3:medium

#### Context window sizes

Note: the 128k version of this model requires [Ollama 0.1.39](#) or later.

- 4k ollama run phi3:mini ollama run phi3:medium
- 128k ollama run phi3:medium-128k



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
● austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
Hello there! It's great to see you! Hello World indeed, it's a classic phrase that never gets old. How are you today? What brings you here? I'm all ears (or in this case, all text) and ready to chat!
○ austindobbins@Austins-Laptop local_rag_agent %
```

This works! Next, we need to make the program fully conversational

```
assistant.py > ...
1 import ollama
2
3 convo = []
4
5 while True:
6     prompt = input('USER: \n')
7     convo.append({'role': 'user', 'content': prompt})
8
9     output = ollama.chat(model='llama3', messages=convo)
10    response = output['message'][0]['content']
11
12    print(f'ASSISTANT: \n{response} \n')
13    convo.append({'role': 'assistant', 'content': response})
14
```

```
EXPLORER ... 🔍 assistant.py ✎
assistant.py > ...
1 import ollama
2
3 convo = []
4
5 while True:
6     prompt = input('USER: \n')
7     convo.append({'role': 'user', 'content': prompt})
8
9     output = ollama.chat(model='llama3', messages=convo)
10    response = output['message'][0]['content']
11
12    print(f'ASSISTANT: \n{response} \n')
13    convo.append({'role': 'assistant', 'content': response})
14
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
/usr/local/bin/python3 /Users/austindobbins/Desktop/local\_rag\_agent/assistant.py
○ austindobbins@Austins-Laptop local\_rag\_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local\_rag\_agent/assistant.py
USER:
hello my name is ai austin!
ASSISTANT:
Nice to meet you, Ai Austin! What brings you here today? Do you have any questions, topics you'd like to discuss, or just looking for some friendly conversation? I'm all ears (or rather, all text)!

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
○ austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
USER:
hello my name is ai austin!
ASSISTANT:
Nice to meet you, Ai Austin! What brings you here today? Do you have any questions, topics you'd like to discuss, or just looking for some friendly conversation? I'm all ears (or rather, all text)!

USER:
say my name 10 times!
ASSISTANT:
Here it goes:

Ai Austin, Ai Austin!
```

Implementing a conversation works! Next, let us implement Ollama response streaming to reduce response latency.

The screenshot shows the VS Code interface with the following details:

- EXPLORER**: Shows a folder named "LOCAL\_RAG\_AGENT" containing "assistant.py" and "requirements.txt".
- ASSISTANT**: A sidebar on the left.
- TERMINAL**: The title bar says "local\_rag\_agent".
  - Content:

```
assistant.py > ...
1 import ollama
2
3 convo = []
4
5 def stream_response(prompt):
6     convo.append({'role': 'user', 'content': prompt})
7     response = ''
8     stream = ollama.chat(model='llama3', messages=convo, stream=True)
9     print('\nASSISTANT:')
10
11     for chunk in stream:
12         content = chunk['message']['content']
13         response += content
14         print(content, end='', flush=True)
15
16     print('\n')
17     convo.append({'role': 'assistant', 'content': response})
18
19 while True:
20     prompt = input('USER: \n')
21     stream_response(prompt=prompt)
```
  - Bottom status bar: PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (highlighted), PORTS. Python icon.

The terminal window shows the following interaction:

```
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
USER:
what are you?

ASSISTANT:
```

The terminal window shows the following detailed AI response:

```
ASSISTANT:
I am LLaMA, an AI assistant developed by Meta AI that can understand and respond to human input in a conversational manner. I'm a type of large language model (LLM) trained on a massive dataset of text from the internet, books, and other sources.

My primary function is to provide helpful and accurate responses to your questions, engage in natural-sounding conversations, and even generate text based on prompts or topics. I can discuss various subjects, including but not limited to:
```

1. General knowledge: History, science, technology, culture, and more.
2. Entertainment: Movies, TV shows, music, books, and video games.
3. Education: Academic topics, research, and career guidance.
4. Personal development: Self-improvement, mental health, relationships, and lifestyle.

I'm constantly learning and improving my language understanding and generation capabilities, so please feel free to ask me anything or engage in a conversation!

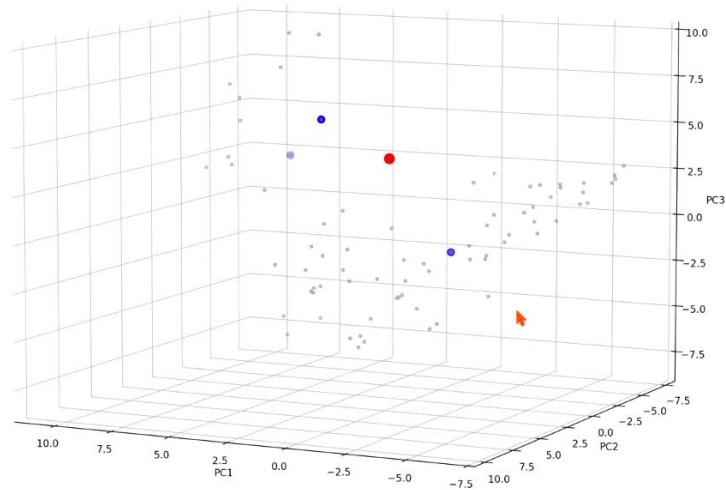
(Note: I'm not perfect and may make mistakes. If you...

Streaming the response works! Next, we need to be able to create vector embeddings of our previous conversations for storage and use in future recalls. We will be using **nomic-embed-text** and **ChromaDB**

# 3d visualization of my conversations vector database

Prompt and All Embeddings Visualization (Total: 75)

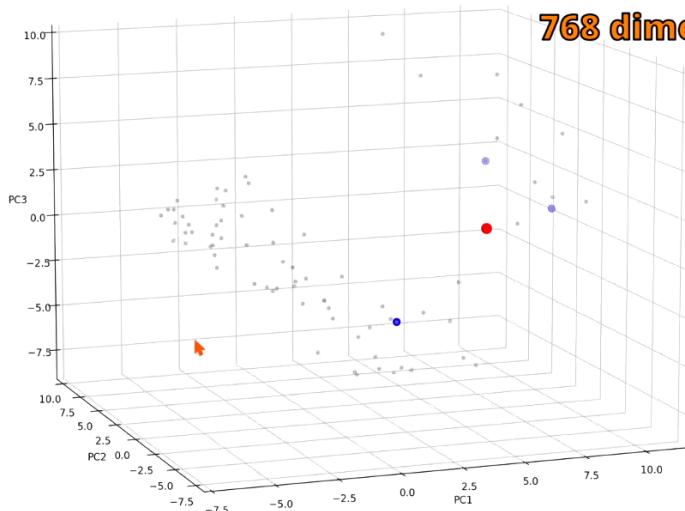
All embeddings  
Closest embeddings  
Prompt



Prompt and All Embeddings Visualization (Total: 75)

All embeddings  
Closest embeddings  
Prompt

The embedding model  
we use is actually  
768 dimensions



PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python

```
austindobbins@Austins-Laptop local_rag_agent % ollama pull nomic-embed-text
pulling manifest
pulling 970aa74c0a90... 100%
pulling c71d239df917... 100%
pulling ce4a164fc046... 100%
pulling 31df23ea7daa... 100%
verifying sha256 digest
writing manifest
removing any unused layers
success
austindobbins@Austins-Laptop local_rag_agent %
```

274 MB  
11 KB  
17 B  
420 B

The screenshot shows the VS Code interface with the title bar 'local\_rag\_agent'. The left sidebar has icons for Explorer, Search, Open, and Outline. The Explorer view shows a folder 'LOCAL\_RAG\_AGENT' containing 'assistant.py' and 'requirements.txt'. The main editor area displays the 'assistant.py' code. The code defines a 'stream\_response' function that takes a prompt, sends it to Ollama, and prints the response. It also defines a 'create\_vector\_db' function that creates a vector database named 'conversations'. The code uses the chromadb library to interact with the database.

```
assistant.py > ...
1 import ollama
2 import chromadb
3
4 client = chromadb.Client()
5 message_history = [
6     {
7         'id': 1,
8         'prompt': 'What is my name?',
9         'response': 'Your name is Austin, known online as Ai Austin.'
10    },
11    {
12        'id': 2,
13        'prompt': 'What is the square root of 9876',
14        'response': '99.3780659904'
15    },
16    {
17        'id': 3,
18        'prompt': 'What kind of dog do I have.',
19        'response': 'Your dog Roxy is an American Pitbull Terrier breed. She is a light brindle blue nose pitbull that you adopted from a local shelter in 2018.'
20    }
21]
22
23 convo = []
24
25 def stream_response(prompt):
26     convo.append({'role': 'user', 'content': prompt})
27     response = ''
28     stream = ollama.chat(model='llama3', messages=convo, stream=True)
29     print('\nASSISTANT: ')
30
31     for chunk in stream:
32         content = chunk['message']['content']
33         response += content
34         print(content, end='', flush=True)
35
36     print('\n')
37     convo.append({'role': 'assistant', 'content': response})
38
39 def create_vector_db(conversations):
40     vector_db_name = 'conversations'
```

The screenshot shows the VS Code interface with the title bar 'local\_rag\_agent'. The left sidebar has icons for Explorer, Search, Open, and Outline. The Explorer view shows a folder 'LOCAL\_RAG\_AGENT' containing 'assistant.py' and 'requirements.txt'. The main editor area displays the 'assistant.py' code. The code now includes a 'try' block to handle a 'ValueError' when deleting a collection. It also adds code to serialize the conversation history, calculate embeddings using the 'nomic-embed-text' model, and add the embeddings to the vector database. The 'create\_vector\_db' function is called with 'message\_history' as an argument.

```
assistant.py > ...
25     def stream_response(prompt):
26         for chunk in stream:
27             content = chunk['message']['content']
28             response += content
29             print(content, end='', flush=True)
30
31         print('\n')
32         convo.append({'role': 'assistant', 'content': response})
33
34     def create_vector_db(conversations):
35         vector_db_name = 'conversations'
36
37         try:
38             client.delete_collection(name=vector_db_name)
39         except ValueError:
40             pass
41
42         vector_db = client.create_collection(name=vector_db_name)
43
44         for c in conversations:
45             serialized_convo = f'prompt: {c["prompt"]} response: {c["response"]}'
46             response = ollama.embeddings(model='nomic-embed-text', prompt=serialized_convo)
47             embedding = response['embedding']
48
49             vector_db.add(
50                 ids=[str(c['id'])],
51                 embeddings=[embedding],
52                 documents=[serialized_convo]
53             )
54
55         create_vector_db(conversations=message_history)
56
57     while True:
58         prompt = input('USER: \n')
59         stream_response(prompt=prompt)
```

Next, let us now get the program to create a vector representation of each prompt input and mathematically retrieve the most relevant embedding from our database.

```

assistant.py > retrieve_embeddings
39 def create_vector_db(conversations):
40     embeddings=[embedding],
41     documents=[serialized_convo]
42
43
44
45 def retrieve_embeddings(prompt):
46     response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
47     prompt_embedding = response['embedding']
48
49
50     vector_db = client.get_collection(name='conversations')
51     results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
52     best_embedding = results['documents'][0][0]
53
54
55     return best_embedding
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70 create_vector_db(conversations=message_history)
71
72 while True:
73     prompt = input('USER: \n')
74     stream_response(prompt=prompt)
75

```

```

assistant.py > ...
68     return best_embedding
69
70
71
72
73
74
75
76
77

```

```

assistant.py > ...
1 import ollama
2 import chromadb
3
4 client = chromadb.Client()
5 message_history = [
6     {
7         'id': 1,
8         'prompt': 'What is my name?',
9         'response': 'Your name is Austin, known online as Ai Austin.'
10    },
11    {
12        'id': 2,
13        'prompt': 'What is the square root of 9876',
14        'response': '99.3780659904'
15    },
16    {
17        'id': 3,
18        'prompt': 'What kind of dog do I have.',
19        'response': 'Your dog Roxy is an American Pitbull Terrier breed. She is a light brindle blue nose pitbull that you adopted'
20    }
21 ]

```

The screenshot shows the VS Code interface with the 'assistant.py' file open in the editor. The code implements a local RAG agent using Chromadb and Ollama. It defines a 'message\_history' list and a 'create\_vector\_db' function. A 'while True' loop reads user input, retrieves embeddings, and streams responses.

This works! We can now implement long-term storage instead of using the python list as above. We will use Postgres SQL as our vector database for CRUD.

# PostgreSQL Installation & Setup

▼ Step by Step (mac)

1. Install PostgreSQL:

```
brew install postgresql@16
brew services start postgresql@16
brew install libpq
brew link --force libpq
```

► Step by Step (linux)

► Step by Step (windows)

|Write something, or press 'space' for AI, '/' for commands...

I



```
austindobbins@austins-Laptop ~ % psql -U postgres
psql (16.3)
Type "help" for help.

postgres=# CREATE USER example_user WITH PASSWORD '123456' SUPERUSER;
CREATE ROLE
postgres=# CREATE DATABASE memory_agent;
CREATE DATABASE
postgres=# GRANT ALL PRIVILEGES ON SCHEMA public TO example_user;
GRANT
postgres=# GRANT ALL PRIVILEGES ON DATABASE memory_agent TO example_user;
GRANT
postgres=# \c memory_agent
You are now connected to database "memory_agent" as user "postgres".
memory_agent=# CREATE TABLE conversations (
memory_agent(# id SERIAL PRIMARY KEY,
memory_agent(# timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
memory_agent(# prompt TEXT NOT NULL,
memory_agent(# response TEXT NOT NULL
memory_agent(# );
```

```
Type "help" for help.

postgres=# CREATE USER example_user WITH PASSWORD '123456' SUPERUSER;
CREATE ROLE
postgres=# CREATE DATABASE memory_agent;
CREATE DATABASE
postgres=# GRANT ALL PRIVILEGES ON SCHEMA public TO example_user;
GRANT
postgres=# GRANT ALL PRIVILEGES ON DATABASE memory_agent TO example_user;
GRANT
postgres=# \c memory_agent
You are now connected to database "memory_agent" as user "postgres".
memory_agent=# CREATE TABLE conversations (
memory_agent(# id SERIAL PRIMARY KEY,
memory_agent(# timestamp TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
memory_agent(# prompt TEXT NOT NULL,
memory_agent(# response TEXT NOT NULL
memory_agent(# );
CREATE TABLE
memory_agent=# INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, 'what is my name?', 'Your name is Austin. Known online as Ai Austin.')
```

```
austindobbins - more +psql -U postgres - 92x21
memory_agent=# INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, 'explain what you are', 'I am a local Ollama memory agent, capable of running local embedding models and large language models and recalling any previous messages with my user.');
INSERT 0 1
memory_agent=# SELECT * FROM conversations;
 id |      timestamp      |           prompt           |          response
----+-----+-----+-----+
 1 | 2024-07-02 11:29:31.750539 | what is my name? | Your name is Austin. Known online as Ai Austin.
 2 | 2024-07-02 11:29:34.090198 | What is 3355 / 15 | 223.666667
 3 | 2024-07-02 11:29:36.154907 | explain what you are | I am a local Ollama memory agent, capable of running local embedding models and large language models and recalling any previous messages with my user.
(3 rows)

(END)
```

```
austindobbins - zsh - 92x21
TAMP, 'explain what you are', 'I am a local Ollama memory agent, capable of running local embedding models and large language models and recalling any previous messages with my user.');

INSERT 0 1
memory_agent=# SELECT * FROM conversations;
 id |      timestamp      |           prompt           |          response
----+-----+-----+-----+
 1 | 2024-07-02 11:29:31.750539 | what is my name? | Your name is Austin. Known online as Ai Austin.
 2 | 2024-07-02 11:29:34.090198 | What is 3355 / 15 | 223.666667
 3 | 2024-07-02 11:29:36.154907 | explain what you are | I am a local Ollama memory agent, capable of running local embedding models and large language models and recalling any previous messages with my user.
(3 rows)

memory_agent=# \q
austindobbins@Austins-Laptop ~ %
```

We now have a table in the SQL database to store every message in our conversation session

```
assistant.py > ...
1 import ollama
2 import chromadb
3
4 client = chromadb.Client()
5 convo = []
6
7 def stream_response(prompt):
8     convo.append({'role': 'user', 'content': prompt})
9     response = ''
10    stream = ollama.chat(model='llama3', messages=convo, stream=True)
11    print('\nASSISTANT:')
12
13    for chunk in stream:
14        content = chunk['message']['content']
15        response += content
16        print(content, end='', flush=True)
17
18    print('\n')
19    convo.append({'role': 'assistant', 'content': response})
20
21 def create_vector_db(conversations):
22     vector_db_name = 'conversations'
```

```
assistant.py > ...
1 import ollama
2 import chromadb
3 import psycopg
4 from psycopg.rows import dict_row
5
6 client = chromadb.Client()
7 convo = []
8 DB_PARAMS = {
9     'dbname': 'memory_agent',
10    'user': 'example_user',
11    'password': '123456',
12    'host': 'localhost',
13    'port': '5432'
14 }
15
16 def connect_db():
17     conn = psycopg.connect(**DB_PARAMS)
18     return conn
19
```

```
assistant.py > ...
16 def connect_db():
17     return conn
18
19
20 def fetch_conversations():
21     conn = connect_db()
22     with conn.cursor(row_factory=dict_row) as cursor:
23         cursor.execute('SELECT * FROM conversations')
24         conversations = cursor.fetchall()
25     conn.close()
26     return conversations
27
28 def stream_response(prompt):
29     convo.append({'role': 'user', 'content': prompt})
30     response = ''
31     stream = ollama.chat(model='llama3', messages=convo, stream=True)
32     print('\nASSISTANT:')
33
34     for chunk in stream:
35         content = chunk['message']['content']
36         response += content
37         print(content, end='', flush=True)
38
39     print('\n')
```

The screenshot shows the Visual Studio Code interface with the file 'assistant.py' open in the editor. The code is a Python script for a local RAG agent. It imports ollama, chromadb, and psycopg. It defines a 'connect\_db' function that returns a PostgreSQL connection. It also defines 'fetch\_conversations' and 'stream\_response' functions. The 'fetch\_conversations' function connects to a PostgreSQL database named 'memory\_agent' and retrieves all rows from the 'conversations' table. The 'stream\_response' function takes a prompt, adds it to a conversation list, and then uses the ollama library to generate a response, printing each chunk of the response as it arrives.

```
assistant.py > ...
1 import ollama
2 import chromadb
3 import psycopg
4 from psycopg.rows import dict_row
5
6 client = chromadb.Client()
7 convo = []
8 DB_PARAMS = {
9     'dbname': 'memory_agent',
10    'user': 'example_user',
11    'password': '123456',
12    'host': 'localhost',
13    'port': '5432'
14 }
15
16 def connect_db():
17     conn = psycopg.connect(**DB_PARAMS)
18     return conn
19
20 def fetch_conversations():
21     conn = connect_db()
22     with conn.cursor(row_factory=dict_row) as cursor:
23         cursor.execute('SELECT * FROM conversations')
24         conversations = cursor.fetchall()
25     conn.close()
26     return conversations
27
28 def stream_response(prompt):
29     convo.append({'role': 'user', 'content': prompt})
30     response = ''
31     stream = ollama.chat(model='llama3', messages=convo, stream=True)
32     print('\nASSISTANT:')
33
34     for chunk in stream:
35         content = chunk['message']['content']
36         response += content
37         print(content, end='', flush=True)
38
39     print('\n')
40     convo.append({'role': 'assistant', 'content': response})
```

The screenshot shows the Visual Studio Code interface with the following details:

- Explorer View:** Shows the workspace named "LOCAL\_RAG\_AGENT" containing files "assistant.py" and "requirements.txt".
- Search Bar:** Displays the text "local\_rag\_agent".
- Code Editor:** The "assistant.py" file is open, showing Python code for creating a vector database and retrieving embeddings. The code includes imports from "ollama", defines functions like "create\_vector\_db" and "retrieve\_embeddings", and uses a loop to interact with the user.

```
assistant.py > ...
42 def create_vector_db(conversations):
43     vector_db_name = 'conversations'
44
45     try:
46         client.delete_collection(name=vector_db_name)
47     except ValueError:
48         pass
49
50     vector_db = client.create_collection(name=vector_db_name)
51
52     for c in conversations:
53         serialized_convo = f'prompt: {c["prompt"]} response: {c["response"]}'
54         response = ollama.embeddings(model='nomic-embed-text', prompt=serialized_convo)
55         embedding = response['embedding']
56
57         vector_db.add(
58             ids=[str(c['id'])],
59             embeddings=[embedding],
60             documents=[serialized_convo]
61         )
62
63     def retrieve_embeddings(prompt):
64         response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
65         prompt_embedding = response['embedding']
66
67         vector_db = client.get_collection(name='conversations')
68         results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
69         best_embedding = results['documents'][0][0]
70
71     return best_embedding
72
73 #create_vector_db(conversations=message_history)
74 print(fetch_conversations())
75
76 while True:
77     prompt = input('USER: \n')
78     context = retrieve_embeddings(prompt=prompt)
79     prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
80     stream_response(prompt=prompt)
81
82
```

The screenshot shows the continuation of the "assistant.py" file in the code editor:

```
assistant.py > ...
62
63     def retrieve_embeddings(prompt):
64         response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
65         prompt_embedding = response['embedding']
66
67         vector_db = client.get_collection(name='conversations')
68         results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
69         best_embedding = results['documents'][0][0]
70
71     return best_embedding
72
73 conversations = fetch_conversations()
74 create_vector_db(conversations=conversations)
75 print(fetch_conversations())
76
77 while True:
78     prompt = input('USER: \n')
79     context = retrieve_embeddings(prompt=prompt)
80     prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
81     stream_response(prompt=prompt)
82
```

Next, we can store each response in our DB after its generated

```
assistant.py > store_conversations
20     def fetch_conversations():
21         conversations = cursor.fetchall()
22         conn.close()
23         return conversations
24
25     def store_conversations(prompt, response):
26         conn = connect_db()
27         with conn.cursor() as cursor:
28             cursor.execute(
29                 'INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, %s, %s)',
30                 (prompt, response)
31             )
32             conn.commit()
33         conn.close()
34
35     def stream_response(prompt):
36         convo.append({'role': 'user', 'content': prompt})
37         response = ''
38         stream = ollama.chat(model='llama3', messages=convo, stream=True)
39         print('\nASSISTANT: ')
40
41         for chunk in stream:
42             content = chunk['message']['content']
43             response += content
44             print(content, end='', flush=True)
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
```

```
assistant.py > ...
37
38     def stream_response(prompt):
39         convo.append({'role': 'user', 'content': prompt})
40         response = ''
41         stream = ollama.chat(model='llama3', messages=convo, stream=True)
42         print('\nASSISTANT: ')
43
44         for chunk in stream:
45             content = chunk['message']['content']
46             response += content
47             print(content, end='', flush=True)
48
49
50         print('\n')
51         store_conversations(prompt=prompt, response=response)
52         convo.append({'role': 'assistant', 'content': response})
53
54     def create_vector_db(conversations):
55         vector_db_name = 'conversations'
56
57         try:
58             client.delete_collection(name=vector_db_name)
59         except ValueError:
60             pass
61
62         vector_db = client.create_collection(name=vector_db_name)
```

We now have a RAG agent capable of storing every conversation we have with the LLM Assistant in a session, it retrieves the single most relevant embedding to our prompt every time and sends it to the LLM Assistant for a better response.

The screenshot shows a light-themed code editor interface with a sidebar on the left containing icons for file operations like Open, Save, Find, and Refresh. The main area displays a Python script named `assistant.py`. The code imports `ollama`, `chromadb`, and `psycopg`, and defines functions for connecting to a database, fetching conversations, storing them, and streaming responses.

```
assistant.py
import ollama
import chromadb
import psycopg
from psycopg.rows import dict_row

client = chromadb.Client()
convo = []
DB_PARAMS = {
    'dbname': 'memory_agent',
    'user': 'example_user',
    'password': '123456',
    'host': 'localhost',
    'port': '5432'
}

def connect_db():
    conn = psycopg.connect(**DB_PARAMS)
    return conn

def fetch_conversations():
    conn = connect_db()
    with conn.cursor(row_factory=dict_row) as cursor:
        cursor.execute('SELECT * FROM conversations')
        conversations = cursor.fetchall()
    conn.close()
    return conversations

def store_conversations(prompt, response):
    conn = connect_db()
    with conn.cursor() as cursor:
        cursor.execute(
            'INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, %s, %s)',
            (prompt, response)
        )
    conn.commit()
    conn.close()

def stream_response(prompt):
    convo.append({'role': 'user', 'content': prompt})
    response = ''
```

This screenshot shows the same Python script `assistant.py` in a dark-themed code editor. The code continues from the previous snippet, adding logic to handle AI-generated responses, store them in a vector database, and retrieve embeddings for specific prompts.

```
def stream_response(prompt):
    convo.append({'role': 'user', 'content': prompt})
    response = ''
    stream = ollama.chat(model='llama3', messages=convo, stream=True)
    print('\nASSISTANT:')

    for chunk in stream:
        content = chunk['message']['content']
        response += content
        print(content, end='', flush=True)

    print('\n')
    store_conversations(prompt=prompt, response=response)
    convo.append({'role': 'assistant', 'content': response})

def create_vector_db(conversations):
    vector_db_name = 'conversations'

    try:
        client.delete_collection(name=vector_db_name)
    except ValueError:
        pass

    vector_db = client.create_collection(name=vector_db_name)

    for c in conversations:
        serialized_convo = f'prompt: {c["prompt"]} response: {c["response"]}'
        response = ollama.embeddings(model='nomic-embed-text', prompt=serialized_convo)
        embedding = response['embedding']

        vector_db.add(
            id=str(c['id']),
            embeddings=[embedding],
            documents=[serialized_convo]
        )

    def retrieve_embeddings(prompt):
        response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
        prompt_embedding = response['embedding']
```

```

assistant.py
53 def create_vector_db(conversations):
54     vector_db = client.create_collection(name=vector_db_name)
55
56     for c in conversations:
57         serialized_convo = f'prompt: {c["prompt"]} response: {c["response"]}'
58         response = ollama.embeddings(model='nomic-embed-text', prompt=serialized_convo)
59         embedding = response['embedding']
60
61         vector_db.add(
62             ids=[str(c['id'])],
63             embeddings=[embedding],
64             documents=[serialized_convo]
65         )
66
67     return vector_db
68
69
70 def retrieve_embeddings(prompt):
71     response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
72     prompt_embedding = response['embedding']
73
74     vector_db = client.get_collection(name='conversations')
75     results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
76     best_embedding = results['documents'][0][0]
77
78     return best_embedding
79
80
81 conversations = fetch_conversations()
82 create_vector_db(conversations=conversations)
83 print(fetch_conversations())
84
85 while True:
86     prompt = input('USER: \n')
87     context = retrieve_embeddings(prompt=prompt)
88     prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
89     stream_response(prompt=prompt)
90
91
92
93

```

Next, let us add functionality for a much logical retrieval system capable of pulling context from multiple topics within our database.

```

assistant.py > ...
1 import ollama
2 import chromadb
3 import psycopg
4 from psycopg.rows import dict_row
5
6 client = chromadb.Client()
7
8 system_prompt = (
9     'You are an AI assistant that has memory of every conversation you have ever had with this user. '
10    'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
11    'If any embedded previous conversations are attached, use them for context to responding to the user, '
12    'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
13    'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
14    'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
15 )
16 convo = [{"role": "system", "content": system_prompt}]
17
18 DB_PARAMS = {
19     'dbname': 'memory_agent',
20     'user': 'example_user',
21     'password': '123456',
22     'host': 'localhost',
23     'port': '5432'
24 }
25
26 def connect_db():
27     conn = psycopg.connect(**DB_PARAMS)
28     return conn
29

```

First, we give the assistant a system prompt on how to properly use the attached context

```

assistant.py > ⌂ create_queries
84 def retrieve_embeddings(prompt):
88     vector_db = client.get_collection(name='conversations')
89     results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
90     best_embedding = results['documents'][0][0]
91
92     return best_embedding
93
94 def create_queries(prompt):
95     [
96
97     conversations = fetch_conversations()
98     create_vector_db(conversations=conversations)
99     print(fetch_conversations())
100
101 while True:
102     prompt = input('USER: \n')
103     context = retrieve_embeddings(prompt=prompt)
104     prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
105     stream_response(prompt=prompt)
106

```

```

○ ● ● ollama_recall - Python - Python assistant.py - 127x35
austindobbins@Austins-Laptop ollama_recall % python3 assistant.py
USER:
/recall using everything you know about me, what should i create youtube video on next?

Vector database queries: ["What are your hobbies?", "What topics do you enjoy learning about?", "Are there any areas where you're interested in sharing your expertise or experiences?", "Have you recently learned something new that you're excited to share with others?", "What type of content do you typically consume on YouTube?" ]

Processing queries to vector database: 100% [ 5/5 [00:31<00:00, 6.32s/it]

5 message:response embeddings added for context.

ASSISTANT:
Based on your interests and preferences, I think a great topic for your next YouTube video could be exploring the applications of large language models (LLMs) in creative fields. You've mentioned being interested in AI and having a realistic approach to creating useful AI applications.

You could create a video that showcases how LLMs can be used to generate music, poetry, or even stories, and how these tools can be used creatively. This topic aligns with your interest in lofi music and old-school rap/rock, as well as your goal of helping people see the potential of AI without overselling it.

Additionally, you could also create a video that compares different LLMs, their strengths, and weaknesses, providing a balanced view for your audience. This would be an informative and engaging video that showcases your expertise in AI and programming.

What do you think? Would this be a topic you'd enjoy exploring further on your channel?

USER:

```

```

○ ● ● ollama_recall - Python - Python assistant.py - 127x35
austindobbins@Austins-Laptop ollama_recall % python3 assistant.py
USER:
/recall using everything you know about me, what should i create youtube video on next?

Vector database queries: ["What are your hobbies?", "What topics do you enjoy learning about?", "Are there any areas where you're interested in sharing your expertise or experiences?", "Have you recently learned something new that you're excited to share with others?", "What type of content do you typically consume on YouTube?" ] →

Processing queries to vector database: 100% [ 5/5 [00:31<00:00, 6.32s/it]

5 message:response embeddings added for context.

ASSISTANT:

```

```
assistant.py > ⚙️ create_queries
84  def retrieve_embeddings(prompt):
88      vector_db = client.get_collection(name='conversations')
89      results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
90      best_embedding = results['documents'][0][0]
91
92      return best_embedding
93
94  def create_queries(prompt):
95      query_msg = [
96          'You are a first principle reasoning search query AI agent. '
97          'Your list of search queries will be ran on an embedding database of all your conversations '
98          'you have ever had with the user. With first principles create a Python list of queries to '
99          'search the embeddings database for any data that would be necessary to have access to in '
100         'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
101         'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
102     ]
103
104    conversations = fetch_conversations()
105    create_vector_db(conversations=conversations)
106    print(fetch_conversations())
107
108    while True:
109        prompt = input('USER: \n')
110        context = retrieve_embeddings(prompt=prompt)
111        prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
112        stream_response(prompt=prompt)
113
```

```
assistant.py > ...
94  def create_queries(prompt):
95      query_msg = [
96          'You are a first principle reasoning search query AI agent. '
97          'Your list of search queries will be ran on an embedding database of all your conversations '
98          'you have ever had with the user. With first principles create a Python list of queries to '
99          'search the embeddings database for any data that would be necessary to have access to in '
100         'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
101         'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
102     ]
103
104      query_convo = [
105          {'role': 'system', 'content': query_msg},
106          {'role': 'user', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower my rates.'},
107          {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "What is the users zip code?"]'},
108          {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead of gtts'},
109          {'role': 'assistant', 'content': '["Llama3 voice assistant", "Python voice assistant", "OpenAI TTS", "openai speak"]'},
110          {'role': 'user', 'content': prompt}
111      ]
112
113      response = ollama.chat(model='llama3', messages=query_convo)
114
115    conversations = fetch_conversations()
116    create_vector_db(conversations=conversations)
117    print(fetch_conversations())
118
119    while True:
120        prompt = input('USER: \n')
121        context = retrieve_embeddings(prompt=prompt)
122        prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
```

```
assistant.py > ⚙ create_queries
94  def create_queries(prompt):
95      query_msg = (
96          "You are a first principle reasoning search query AI agent. "
97          "Your list of search queries will be ran on an embedding database of all your conversations "
98          "you have ever had with the user. With first principles create a Python list of queries to "
99          "search the embeddings database for any data that would be necessary to have access to in "
100         "order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. "
101         "Do not explain anything and do not ever generate anything but a perfect syntax Python list"
102     )
103     query_convo = [
104         {'role': 'system', 'content': query_msg},
105         {'role': 'user', 'content': 'Write an email to my car insurance company and create a pursuasive request for them to lower my rates. I am a current customer and have been with them for over 10 years.'},
106         {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "What is the users zip code?"]'},
107         {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead of google text to speech?'},
108         {'role': 'assistant', 'content': '["Llama3 voice assistant", "Python voice assistant", "OpenAI TTS", "openai speak"]'}
109     ]
110
111     conversations = fetch_conversations()
112     create_vector_db(conversations=conversations)
113     print(fetch_conversations())
114
115     while True:
116         prompt = input('USER: \n')
117         context = retrieve_embeddings(prompt=prompt)
118         prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
119         stream_response(prompt=prompt)
120
121
122
123
124
125
```

```
assistant.py > ...
94  def create_queries(prompt):
95      |     {'role': 'user', 'content': prompt}
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112     response = ollama.chat(model='llama3', messages=query_convo)
113     print(f'\nVector database queries: {response["message"]["content"]} \n')
114
115
116     conversations = fetch_conversations()
117     create_vector_db(conversations=conversations)
118     print(fetch_conversations())
119
120     while True:
121         prompt = input('USER: \n')
122         context = retrieve_embeddings(prompt=prompt)
123         prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
124         stream_response(prompt=prompt)
125
```

```
assistant.py ● | example.py ●
example.py > ...
1  llm_output = ["query1", "query2", "query3"]
2  we_need = ['query1', 'query2', 'query3']
3  |
```

```
assistant.py > ...
1 import ollama
2 import chromadb
3 import psycopg
4 import ast
5 from psycopg.rows import dict_row
6
7 client = chromadb.Client()
8
9 system_prompt = (
10     'You are an AI assistant that has memory of every conversation you have ever had with this user. '
11     'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
12     'If any embedded previous conversations are attached, use them for context to responding to the user, '
13     'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
14     'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
15     'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
16 )
17 convo = [{'role': 'system', 'content': system_prompt}]
18
19 DB_PARAMS = {
20     'dbname': 'memory_agent',
21     'user': 'example_user',
22     'password': '123456',
23     'host': 'localhost',
24     'port': '5432'
25 }
26
27 def connect_db():
28     conn = psycopg.connect(**DB_PARAMS)
29     return conn
```

```
assistant.py > ...
95     def create_queries(prompt):
96         {'role': 'user', 'content': prompt}
97     ]
98
99     response = ollama.chat(model='llama3', messages=query_convo)
100    print(f'\nVector database queries: {response["message"]["content"]}\n')
101
102    try:
103        return ast.literal_eval(response['message']['content'])
104    except:
105        return [prompt]
106
107    conversations = fetch_conversations()
108    create_vector_db(conversations=conversations)
109    print(fetch_conversations())
110
111    while True:
112        prompt = input('USER: \n')
113        context = retrieve_embeddings(prompt=prompt)
114        prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
115        stream_response(prompt=prompt)
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
```

This **ast** library function will help convert the string variable into a python list so that we can use it.

```
assistant.py > ...
1 import ollama
2 import chromadb
3 import psycopg
4 import ast
5 from psycopg.rows import dict_row
6
7 client = chromadb.Client()
8
9 system_prompt = (
10     'You are an AI assistant that has memory of every conversation you have ever had with this user. '
11     'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
12     'If any embedded previous conversations are attached, use them for context to responding to the user, '
13     'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
14     'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
15     'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
16 )
17 convo = [{"role": "system", "content": system_prompt}]
18
19 DB_PARAMS = {
20     'dbname': 'memory_agent',
21     'user': 'example_user',
22     'password': '123456',
23     'host': 'localhost',
24     'port': '5432'
25 }
26
27 def connect_db():
28     conn = psycopg.connect(**DB_PARAMS)
29     return conn
```

```
assistant.py > ...
27 def connect_db():
28     conn = psycopg.connect(**DB_PARAMS)
29     return conn
30
31 def fetch_conversations():
32     conn = connect_db()
33     with conn.cursor(row_factory=dict_row) as cursor:
34         cursor.execute('SELECT * FROM conversations')
35         conversations = cursor.fetchall()
36     conn.close()
37     return conversations
38
39 def store_conversations(prompt, response):
40     conn = connect_db()
41     with conn.cursor() as cursor:
42         cursor.execute(
43             'INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, %s, %s)',
44             (prompt, response)
45         )
46     conn.commit()
47     conn.close()
48
49 def stream_response(prompt):
50     convo.append({'role': 'user', 'content': prompt})
51     response = ''
52     stream = ollama.chat(model='llama3', messages=convo, stream=True)
53     print('\nASSISTANT:')
54     for chunk in stream:
```

```
assistant.py > ...
49 def stream_response(prompt):
50     # Stream response
51
52     for chunk in stream:
53         content = chunk['message']['content']
54         response += content
55         print(content, end='', flush=True)
56
57     print('\n')
58     store_conversations(prompt=prompt, response=response)
59     convo.append({'role': 'assistant', 'content': response})
60
61
62
63
64 def create_vector_db(conversations):
65     vector_db_name = 'conversations'
66
67     try:
68         client.delete_collection(name=vector_db_name)
69     except ValueError:
70         pass
71
72     vector_db = client.create_collection(name=vector_db_name)
73
74     for c in conversations:
75         serialized_convo = f'prompt: {c["prompt"]} response: {c["response"]}'
76         response = ollama.embeddings(model='nomic-embed-text', prompt=serialized_convo)
77         embedding = response['embedding']
78
79         vector_db.add(
80             ids=[str(c['id'])],
81             embeddings=[embedding],
```

```
assistant.py > ...
64 def create_vector_db(conversations):
65     vector_db.add(
66         ids=[str(c['id'])],
67         embeddings=[embedding],
68         documents=[serialized_convo]
69     )
70
71
72
73
74
75 def retrieve_embeddings(prompt):
76     response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
77     prompt_embedding = response['embedding']
78
79     vector_db = client.get_collection(name='conversations')
80     results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
81     best_embedding = results['documents'][0][0]
82
83     return best_embedding
84
85
86
87
88
89
90
91
92
93
94
95 def create_queries(prompt):
96     query_msg = (
97         'You are a first principle reasoning search query AI agent. '
98         'Your list of search queries will be ran on an embedding database of all your conversations '
99         'you have ever had with the user. With first principles create a Python list of queries to '
100         'search the embeddings database for any data that would be necessary to have access to in '
101         'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
102         'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
103     )
104     query_convo = [
105         {'role': 'system', 'content': query_msg},
106         {'role': 'user', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower'}
107         # The rest of the conversation is cut off by the code editor's scroll bar
```

```

assistant.py > ...
94
95 def create_queries(prompt):
96     query_msg = (
97         'You are a first principle reasoning search query AI agent. '
98         'Your list of search queries will be ran on an embedding database of all your conversations '
99         'you have ever had with the user. With first principles create a Python list of queries to '
100        'search the embeddings database for any data that would be necessary to have access to in '
101        'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
102        'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
103    )
104    query_convo = [
105        {'role': 'system', 'content': query_msg},
106        {'role': 'user', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower my rates'}
107        {'role': 'assistant', 'content': '[{"What is the users name?", "What is the users current auto insurance provider?", "What is the users address?"}]'}
108        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead'}
109        {'role': 'assistant', 'content': '[{"Llama3 voice assistant", "Python voice assistant", "OpenAI TTS", "openai speak"}]'}
110        {'role': 'user', 'content': prompt}
111    ]
112
113    response = ollama.chat(model='llama3', messages=query_convo)
114    print(f'\nVector database queries: {response["message"]["content"]}\n')
115
116    try:
117        return ast.literal_eval(response['message']['content'])
118    except:
119        return [prompt]
120
121 conversations = fetch_conversations()
122 create_vector_db(conversations=conversations)
123 print(fetch_conversations())

```

```

assistant.py > ...
95 def create_queries(prompt):
96     query_msg = (
97         'You are a first principle reasoning search query AI agent. '
98         'Your list of search queries will be ran on an embedding database of all your conversations '
99         'you have ever had with the user. With first principles create a Python list of queries to '
100        'search the embeddings database for any data that would be necessary to have access to in '
101        'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
102        'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
103    )
104    query_convo = [
105        {'role': 'system', 'content': query_msg},
106        {'role': 'user', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower my rates'}
107        {'role': 'assistant', 'content': '[{"What is the users name?", "What is the users current auto insurance provider?", "What is the users address?"}]'}
108        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead'}
109        {'role': 'assistant', 'content': '[{"Llama3 voice assistant", "Python voice assistant", "OpenAI TTS", "openai speak"}]'}
110        {'role': 'user', 'content': prompt}
111    ]
112
113    response = ollama.chat(model='llama3', messages=query_convo)
114    print(f'\nVector database queries: {response["message"]["content"]}\n')
115
116    try:
117        return ast.literal_eval(response['message']['content'])
118    except:
119        return [prompt]
120
121 conversations = fetch_conversations()
122 create_vector_db(conversations=conversations)
123 print(fetch_conversations())

```

We can then improve the number of results we need per query as below on line 85

```

assistant.py > retrieve_embeddings
64  def create_vector_db(conversations):
65      embeddings=[embedding],
66      documents=[serialized_convo]
67
68
69  def retrieve_embeddings(queries, results_per_query=2):
70      response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
71      prompt_embedding = response['embedding']
72
73
74  vector_db = client.get_collection(name='conversations')
75  results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
76  best_embedding = results['documents'][0][0]
77
78
79  return best_embedding
80
81
82  def create_queries(prompt):
83      query_msg = (
84          'You are a first principle reasoning search query AI agent. '
85          'Your list of search queries will be ran on an embedding database of all your conversations '
86          'you have ever had with the user. With first principles create a Python list of queries to '
87          'search the embeddings database for any data that would be necessary to have access to in '
88          'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
89          'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
90      )
91
92      query_convo = [
93          {'role': 'system', 'content': query_msg},
94          {'role': 'user', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower my rates'},
95          {'role': 'assistant', 'content': ['"What is the users name?", "What is the users current auto insurance provider?", "What is the users car model?"']},
96          {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead of espeak'}
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140

```

```

assistant.py > retrieve_embeddings
64  def create_vector_db(conversations):
65      embeddings=[embedding],
66      documents=[serialized_convo]
67
68
69  def retrieve_embeddings(queries, results_per_query=2):
70      embeddings = set()
71
72
73  for query in queries:
74      response = ollama.embeddings(model='nomic-embed-text', prompt=query)
75      query_embedding = response['embedding']
76
77
78  vector_db = client.get_collection(name='conversations')
79  results = vector_db.query(query_embeddings=[query_embedding], n_results=results_per_query)
80  best_embeddings = results['documents'][0]
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140

```

```

assistant.py > ...
105  def create_queries(prompt):
106      except:
107          return [prompt]
108
109
110  conversations = fetch_conversations()
111  create_vector_db(conversations=conversations)
112  print(fetch_conversations())
113
114
115  while True:
116      prompt = input('USER: \n')
117      context = retrieve_embeddings(prompt=prompt)
118      prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
119      stream_response(prompt=prompt)
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140

```

We now can classify the embedding as below

```
assistant.py > classify_embedding
105 def create_queries(prompt):
126     try:
127         return ast.literal_eval(response['message']['content'])
128     except:
129         return [prompt]
130
131 def classify_embedding(query, context):
132     classify_msg = (
133         'You are an embedding classification AI agent. Your input will be a prompt and one embedded chunk of text. '
134         'You will not respond as an AI assistant. You only respond "yes" or "no". '
135         'Determine whether the context contains data that directly is related to the search query. '
136         'If the context is seemingly exactly what the search query needs, respond "yes" if it is anything but directly '
137         'related respond "no". Do not respond "yes" unless the content is highly relevant to the search query.'
138     )
139     classify_convo = [
140         {'role': 'system', 'content': classify_msg},
141         {'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?' },
142         {'role': 'assistant', 'content': 'yes'},
143         {'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS. The content is highly relevant to the search query.' },
144         {'role': 'assistant', 'content': 'no'},
145         {'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
146     ]
147
148 conversations = fetch_conversations()
149 create_vector_db(conversations=conversations)
150 print(fetch_conversations())
151
152 while True:
153     prompt = input('USER: \n')
```

```
assistant.py > classify_embedding
105
126 response['message']['content']
128
129
130
131 intext):
132
133 classification AI agent. Your input will be a prompt and one embedded chunk of text. '
134 : an AI assistant. You only respond "yes" or "no". '
135 :context contains data that directly is related to the search query. '
136 :ngly exactly what the search query needs, respond "yes" if it is anything but directly '
137 :o not respond "yes" unless the content is highly relevant to the search query.'
138
139
140 :ent': classify_msg},
141 :it': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?' },
142 :content': 'yes'},
143 :it': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS. The content is highly relevant to the search query.' },
144 :content': 'no'},
145 :it': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
146
147
148 .ons()
149 :conversations)
150
```

```

assistant.py > classify_embedding
105 def create_queries(prompt):
106     try:
107         return ast.literal_eval(response['message']['content'])
108     except:
109         return [prompt]
110
111 def classify_embedding(query, context):
112     classify_msg = (
113         'You are an embedding classification AI agent. Your input will be a prompt and one embedded chunk of text. '
114         'You will not respond as an AI assistant. You only respond "yes" or "no". '
115         'Determine whether the context contains data that directly is related to the search query. '
116         'If the context is seemingly exactly what the search query needs, respond "yes" if it is anything but directly '
117         'related respond "no". Do not respond "yes" unless the content is highly relevant to the search query.'
118     )
119     classify_convo = [
120         {'role': 'system', 'content': classify_msg},
121         {'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help you today?'},
122         {'role': 'assistant', 'content': 'yes'},
123         {'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant that can answer questions and perform tasks.'},
124         {'role': 'assistant', 'content': 'no'},
125         {'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
126     ]
127
128     response = ollama.chat(model='llama3', messages=classify_convo)
129
130     return response['message']['content'].strip().lower()
131
132 conversations = fetch_conversations()
133 create_vector_db(conversations=conversations)
134
135 print(create_queries('What is the capital of France?'))

```

We can now use the `classify_embedding` function as below

```

assistant.py > ...
84
85 def retrieve_embeddings(queries, results_per_query=2):
86     embeddings = set()
87
88     for query in queries:
89         response = ollama.embeddings(model='nomic-embed-text', prompt=query)
90         query_embedding = response['embedding']
91
92         vector_db = client.get_collection(name='conversations')
93         results = vector_db.query(query_embeddings=[query_embedding], n_results=results_per_query)
94         best_embeddings = results['documents'][0]
95
96         for best in best_embeddings:
97             if best not in embeddings:
98                 if 'yes' in classify_embedding(query=query, context=best):
99                     embeddings.add(best)
100
101     return embeddings
102
103 def create_queries(prompt):
104     query_msg = (
105         'You are a first principle reasoning search query AI agent. '
106         'Your list of search queries will be ran on an embedding database of all your conversations '
107         'you have ever had with the user. With first principles create a Python list of queries to '
108         'search the embeddings database for any data that would be necessary to have access to in '
109         'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
110         'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
111     )
112     query_convo = [

```

```

assistant.py > ...
129 def classify_embedding(query, context):
130     response = ollama.chat(model='llama3', messages=classify_convo)
131
132     return response['message']['content'].strip().lower()
133
134
135 def recall(prompt):
136     queries = create_queries(prompt=prompt)
137     embeddings = retrieve_embeddings(queries=queries)
138     convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
139     print(f'\n{len(embeddings)} message:response embeddings added for context.')
140
141
142 conversations = fetch_conversations()
143 create_vector_db(conversations=conversations)
144 print(fetch_conversations())
145
146 while True:
147     prompt = input('USER: \n')
148     context = retrieve_embeddings(prompt=prompt)
149     prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
150     stream_response(prompt=prompt)
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
```

With the recall function done, we can update the `stream_response` function and the while loop as below

```

assistant.py > ⌂ stream_response
39 def store_conversations(prompt, response):
40     conn.close()
41
42
43 def stream_response(prompt):
44     convo.append({'role': 'user', 'content': prompt})
45     response = ''
46     stream = ollama.chat(model='llama3', messages=convo, stream=True)
47     print('\nASSISTANT:')
48
49     for chunk in stream:
50         content = chunk['message']['content']
51         response += content
52         print(content, end='', flush=True)
53
54     print('\n')
55     store_conversations(prompt=prompt, response=response)
56     convo.append({'role': 'assistant', 'content': response})
57
58
59
60
61
62
63
64 def create_vector_db(conversations):
65     vector_db_name = 'conversations'
66
67     try:
68         client.delete_collection(name=vector_db_name)
69     except ValueError:
70         pass
71
72     vector_db = client.create_collection(name=vector_db_name)
73
74     for c in conversations:
```

```
assistant.py > stream_response
39 def store_conversations(prompt, response):
40     )
41     conn.commit()
42 conn.close()
43
44 def stream_response(prompt):
45     response = ''
46     stream = ollama.chat(model='llama3', messages=convo, stream=True)
47     print('\nASSISTANT:')
48
49     for chunk in stream:
50         content = chunk['message']['content']
51         response += content
52         print(content, end='', flush=True)
53
54     print('\n')
55     store_conversations(prompt=prompt, response=response)
56     convo.append({'role': 'assistant', 'content': response})
57
58
59 def create_vector_db(conversations):
60     vector_db_name = 'conversations'
61
62     try:
63         client.delete_collection(name=vector_db_name)
64     except ValueError:
65         pass
66
67     vector_db = client.create_collection(name=vector_db_name)
68
```

```
assistant.py > ...
155 conversations = fetch_conversations()
156 create_vector_db(conversations=conversations)
157 print(fetch_conversations())
158
159 while True:
160     prompt = input('USER: \n') I
161     context = retrieve_embeddings(prompt=prompt)
162     prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
163     stream_response(prompt=prompt)
164
```

```
assistant.py > ...
149 def recall(prompt):
150     convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
151     print(f'\n{len(embeddings)} message:response embeddings added for context.')
152
153 conversations = fetch_conversations()
154 create_vector_db(conversations=conversations)
155
156 while True:
157     prompt = input('USER: \n')
158     recall(prompt=prompt)
159     stream_response(prompt=prompt)
160
161
162
```

This completes our work on an AI agent that stores its conversations as embeddings in a database.

```
assistant.py ●
assistant.py > ...
1 import ollama
2 import chromadb
3 import psycopg
4 import ast
5 from psycopg.rows import dict_row
6
7 client = chromadb.Client()
8
9 system_prompt = (
10     "You are an AI assistant that has memory of every conversation you have ever had with this user."
11     "On every prompt from the user, the system has checked for any relevant messages you have had with the user."
12     "If any embedded previous conversations are attached, use them for context to responding to the user."
13     "if the context is relevant and useful to responding. If the recalled conversations are irrelevant,"
14     "disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations."
15     "Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant."
16 )
17 convo = [ {'role': 'system', 'content': system_prompt}]
18
19 DB_PARAMS = {
20     'dbname': 'memory_agent',
21     'user': 'example_user',
22     'password': '123456',
23     'host': 'localhost',
24     'port': '5432'
25 }
26
27 def connect_db():
28     conn = psycopg.connect(**DB_PARAMS)
29     return conn
30
31 def fetch_conversations():
32     conn = connect_db()
33     with conn.cursor(row_factory=dict_row) as cursor:
34         cursor.execute('SELECT * FROM conversations')
35         conversations = cursor.fetchall()
36     conn.close()
37     return conversations
38
39 def store_conversations(prompt, response):
40     conn = connect_db()
```

```
assistant.py ●
assistant.py > ...
31 def fetch_conversations():
32     conn.close()
33     return conversations
34
35 def store_conversations(prompt, response):
36     conn = connect_db()
37     with conn.cursor() as cursor:
38         cursor.execute(
39             'INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, %s, %s)',
40             (prompt, response)
41         )
42     conn.commit()
43     conn.close()
44
45 def stream_response(prompt):
46     response = ''
47     stream = ollama.chat(model='llama3', messages=convo, stream=True)
48     print('\nASSISTANT:\n')
49
50     for chunk in stream:
51         content = chunk['message']['content']
52         response += content
53         print(content, end='', flush=True)
54
55     print('\n')
56     store_conversations(prompt=prompt, response=response)
57     convo.append({'role': 'assistant', 'content': response})
58
59 def create_vector_db(conversations):
60     vector_db_name = 'conversations'
61
62     try:
63         client.delete_collection(name=vector_db_name)
64     except ValueError:
65         pass
66
67     vector_db = client.create_collection(name=vector_db_name)
68
69     for c in conversations:
70         serialized_conv = f'prompt: {c["prompt"]}, response: {c["response"]}'
```

The screenshot shows a code editor window with a dark theme. The title bar says "local\_rag\_agent". The left sidebar has icons for file operations like Open, Save, Find, and others. The main area displays the following Python code:

```
assistant.py

def create_vector_db(conversations):
    vector_db_name = 'conversations'

    try:
        client.delete_collection(name=vector_db_name)
    except ValueError:
        pass

    vector_db = client.create_collection(name=vector_db_name)

for c in conversations:
    serialized_convo = f'prompt: {c["prompt"]} response: {c["response"]}'
    response = ollama.embeddings(model='nomic-embed-text', prompt=serialized_convo)
    embedding = response['embedding']

    vector_db.add(
        ids=[str(c['id'])],
        embeddings=[embedding],
        documents=[serialized_convo]
    )

def retrieve_embeddings(queries, results_per_query=2):
    embeddings = set()

    for query in queries:
        response = ollama.embeddings(model='nomic-embed-text', prompt=query)
        query_embedding = response['embedding']

        vector_db = client.get_collection(name='conversations')
        results = vector_db.query(query_embeddings=[query_embedding], n_results=results_per_query)
        best_embeddings = results['documents'][0]

        for best in best_embeddings:
            if best not in embeddings:
                if 'yes' in classify_embedding(query=query, context=best):
                    embeddings.add(best)

    return embeddings

# def create_queries(prompt):
```

The screenshot shows a continuation of the Python code in the same code editor window. The title bar still says "local\_rag\_agent". The code starts with a function definition for "retrieve\_embeddings" and then moves on to "create\_queries".

```
assistant.py

def retrieve_embeddings(queries, results_per_query=2):
    return embeddings

def create_queries(prompt):
    query_msg = (
        'You are a first principle reasoning search query AI agent. '
        'Your list of search queries will be ran on an embedding database of all your conversations '
        'you have ever had with the user. With first principles create a Python list of queries to '
        'search the embeddings database for any data that would be necessary to have access to in '
        'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
        'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
    )
    query_convo = [
        {'role': 'system', 'content': query_msg},
        {'role': 'user', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower my monthly rate.'},
        {'role': 'assistant', 'content': ['What is the users name?', "What is the users current auto insurance provider?", "What is the monthly rate of the users car insurance?"]},
        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead of the openai tts'},
        {'role': 'assistant', 'content': ['Llama3 voice assistant', "Python voice assistant", "OpenAI TTS", "openai speak"]},
        {'role': 'user', 'content': prompt}
    ]

    response = ollama.chat(model='llama3', messages=query_convo)
    print(f'\nVector database queries: {response["message"]["content"]}\n')

    try:
        return ast.literal_eval(response['message']['content'])
    except:
        return [prompt]

def classify_embedding(query, context):
    classify_msg = (
        'You are an embedding classification AI agent. Your input will be a prompt and one embedded chunk of text. '
        'You will not respond as an AI assistant. You only respond "yes" or "no". '
        'Determine whether the context contains data that directly is related to the search query. '
        'If the context is seemingly exactly what the search query needs, respond "yes" if it is anything but directly '
        'related respond "no". Do not respond "yes" unless the content is highly relevant to the search query.'
    )
    classify_convo = [
        {'role': 'system', 'content': classify_msg},
        {'role': 'user', 'content': f'Search QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'}
```

```
assistant.py
assistant.py > ...
101
102
103
104
105     r conversations '
106     of queries to '
107     access to in '
108     st with no syntax errors. '
109     tax Python list'
110
111
112
113     create a persuasive request for them to lower my monthly rate.'},
114     sers current auto insurance provider?", "What is the monthly rate the user currently pays for auto insurance?"}],
115     a3 python voice assistant to use pyttsx instead of the openai tts api?'],
116     ssistant", "OpenAI TTS", "openai speak"]',
117
```

```
assistant.py
assistant.py > ...
126
127
128
129
130     I chunk of text.'
131
132     '
133     iything but directly '
134     earch query.'
135
136
137
138     'ou are Ai Austin. How can I help today Austin?'},
139
140     TEXT: Siri is a voice assistant on Apple iOS and Mac OS. The voice assistant is designed to take voice prompts and help the user complete simple tasks or
141
142
```

```
136     classify_convo = [
137         {'role': 'system', 'content': classify_msg},
138         {'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
139         {'role': 'assistant', 'content': 'yes'},
140         {'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac O
141         {'role': 'assistant', 'content': 'no'},
142         {'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
143     ]
144
145     response = ollama.chat(model='llama3', messages=classify_convo)
146
147     return response['message']['content'].strip().lower()
148
149 def recall(prompt):
150     queries = create_queries(prompt=prompt)
151     embeddings = retrieve_embeddings(queries=queries)
152     convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
153     print(f'\n{len(embeddings)} message:response embeddings added for context.')
154
155     conversations = fetch_conversations()
156     create_vector_db(conversations=conversations)
157
158     while True:
159         prompt = input('USER: \n')
160         recall(prompt=prompt)
161         stream_response(prompt=prompt)
162
```

```
assistant.py •
assistant.py > ...
101
102
103
104
105     r conversations '
106     of queries to '
107     access to in '
108     st with no syntax errors. '
109     tax Python list'
110
111
112
113     create a persuasive request for them to lower my monthly rate.'),
114     sers current auto insurance provider?", "What is the monthly rate the user currently pays for auto insurance?"}],
115     a3 python voice assistant to use pyttsx3 instead of the openai tts api?'),
116     ssistant", "OpenAI TTS", "openai speak"]),
117
118
119
120
121
122
123
124
125
126
127
128
129
130     nd one embedded chunk of text.'
131
132     search query.
133     es" if it is anything but directly '
134     evant to the search query.'
135
136
137
138     DDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
139
140     \nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS. The voice assistant is desioned to take voice prompts and help the user complete
0 0 0 % 0
```

Ln 162, Col 1 Spaces: 4 UTF-8 LF ↗ Python 3.12.4

```
assistant.py •
assistant.py > ...
126
127
128     :
129
130     .ation AI agent. Your input will be a prompt and one embedded chunk of text.'
131     : assistant. You only respond "yes" or "no".
132     : contains data that directly is related to the search query.
133     : exactly what the search query needs, respond "yes" if it is anything but directly '
134     respond "yes" unless the content is highly relevant to the search query.'
135
136
137     classify_msg),
138     SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
139     :: 'yes',
140     SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS. The voice assistant is designed to
141     :: 'no',
142     SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
143
144
145     a3', messages=classify_conv)
146
147     rt'].strip().lower()
148
149
150     'ompt)
151     ueries=queries)
152     ntent': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}')}
153     je:response embeddings added for context.')
154
155
156     iations)
157
```

The screenshot shows the VS Code interface. The code editor displays the file 'assistant.py' with the following content:

```
126
127
128
129
130
131
132
133
134
135
136
137
138     "ow can I help today Austin?'},
139
140     "e assistant on Apple iOS and Mac OS. The voice assistant is designed to take voice prompts and help the user complete simple tasks on the device.'},
141
```

The terminal below shows a user session:

```
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
○ austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
USER:
lets test if you can recall multiple memories about me.. what is my name? what is my dogs breed?
```

The terminal output shows the user's query and the system's response:

```
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
○ austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
USER:
lets test if you can recall multiple memories about me.. what is my name? what is my dogs breed?
```

The terminal output shows the user's query and the system's response:

```
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
○ austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
USER:
lets test if you can recall multiple memories about me.. what is my name? what is my dogs breed?

Vector database queries: ["What are the user's previous conversations?", "Retrieve information about the user's dog", "User's dog breed", "User's name"]
```

The terminal output shows the user's query and the system's response:

```
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
○ austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
USER:
lets test if you can recall multiple memories about me.. what is my name? what is my dogs breed?

Vector database queries: ["What are the user's previous conversations?", "Retrieve information about the user's dog", "User's dog breed", "User's name"]

5 message:response embeddings added for context.

ASSISTANT:
```

The terminal output shows the user's query and the system's response:

```
/usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
○ austindobbins@Austins-Laptop local_rag_agent % /usr/local/bin/python3 /Users/austindobbins/Desktop/local_rag_agent/assistant.py
USER:
lets test if you can recall multiple memories about me.. what is my name? what is my dogs breed?

Vector database queries: ["What are the user's previous conversations?", "Retrieve information about the user's dog", "User's dog breed", "User's name"]

5 message:response embeddings added for context.

ASSISTANT:
I think I can help you with that!

Your name is Austin, also known online as Ai Austin.

As for your dog Roxy, she's a Blue Nose American Pitbull.

USER:
```

Let us implement notifications to see what is happening while waiting with a loading bar

```

assistant.py > ...
1  import ollama
2  import chromadb
3  import psycopg
4  import ast
5  from tqdm import tqdm
6  from psycopg.rows import dict_row
7
8  client = chromadb.Client()
9
10 system_prompt = (
11     'You are an AI assistant that has memory of every conversation you have ever had with this user'
12     'On every prompt from the user, the system has checked for any relevant messages you have had w'
13     'If any embedded previous conversations are attached, use them for context to responding to the'
14     'if the context is relevant and useful to responding. If the recalled conversations are irrelev'
15     'disregard speaking about them and respond normally as an AI assistant. Do not talk about recal'
16     'Just use any useful data from the previous conversations and respond normally as an intelligen'
17 )
18 convo = [{'role': 'system', 'content': system_prompt}]
19
20 DB_PARAMS = {
21     'dbname': 'memory_agent',
22     'user': 'example_user',
23     'password': '123456',

```

```

assistant.py > ⚡ retrieve_embeddings
83
84
85 def retrieve_embeddings(queries, results_per_query=2):
86     embeddings = set()
87
88     for query in tqdm(queries, desc='Processing queries to vector database'):
89         response = ollama.embeddings(model='nomic-embed-text', prompt=query)
90         query_embedding = response['embedding']
91
92         vector_db = client.get_collection(name='conversations')
93         results = vector_db.query(query_embeddings=[query_embedding], n_results=results_per_query)
94         best_embeddings = results['documents'][0]
95
96         for best in best_embeddings:
97             if best not in embeddings:
98                 if 'yes' in classify_embedding(query=query, context=best):
99                     embeddings.add(best)
100
101     return embeddings
102
103 def create_queries(prompt):
104     query_msg = (
105         'You are a first principle reasoning search query AI agent. '

```

We can make it clear using some log and /recall command as below

```

assistant.py > ...
156     conversations = fetch_conversations()
157     create_vector_db(conversations=conversations)
158
159     while True:
160         prompt = input('USER: \n')
161
162         if prompt[:7].lower() == '/recall':
163             prompt = prompt[8:]
164             recall(prompt=prompt)
165             stream_response(prompt=prompt)
166         else:
167             convo.append({'role': 'user', 'content': prompt})
168             stream_response(prompt=prompt)
169
170             stream_response(prompt=prompt)
171

```

Let us now add a **/forget** command as below

```

assistant.py > ...
40     def store_conversations(prompt, response):
41         conn = connect_db()
42         cursor = conn.cursor()
43         cursor.execute('INSERT INTO conversations (prompt, response) VALUES (?, ?)', (prompt, response))
44         conn.commit()
45     conn.close()
46
47 def remove_last_conversation():
48     conn = connect_db()
49     with conn.cursor() as cursor:
50         cursor.execute('DELETE FROM conversations WHERE id = (SELECT MAX(id) FROM conversations)')
51         cursor.commit()
52     conn.close()
53
54 def stream_response(prompt):
55     response = ''
56     stream = ollama.chat(model='llama3', messages=convo, stream=True)
57     print('\nASSISTANT:')
58
59     for chunk in stream:
60         content = chunk['message']['content']
61         response += content
62         print(content, end='', flush=True)
63
64
65
66

```

```

assistant.py > ...
164     create_vector_db(conversations=conversations)
165
166 while True:
167     prompt = input('USER: \n')
168
169     if prompt[:7].lower() == '/recall':
170         prompt = prompt[8:]
171         recall(prompt=prompt)
172         stream_response(prompt=prompt)
173     elif prompt[:7] == '/forget':
174         remove_last_conversation()
175         convo = convo[:-2]
176         print('\n')
177     else:
178         convo.append({'role': 'user', 'content': prompt})
179         stream_response(prompt=prompt)
180
181     stream_response(prompt=prompt)
182

```

Next, we can implement a `/memorize` function to allow us store our prompts without generating a response to attach to the vector embedding

```

assistant.py > ...
166 while True:
167     prompt = input('USER: \n')
168
169     if prompt[:7].lower() == '/recall':
170         prompt = prompt[8:]
171         recall(prompt=prompt)
172         stream_response(prompt=prompt)
173     elif prompt[:7].lower() == '/forget':
174         remove_last_conversation()
175         convo = convo[:-2]
176         print('\n')
177     elif prompt[:9].lower() == '/memorize':
178         prompt = prompt[10:]
179         store_conversations(prompt=prompt, response='Memory stored.')
180         print('\n')
181     else:
182         convo.append({'role': 'user', 'content': prompt})
183         stream_response(prompt=prompt)
184
185     stream_response(prompt=prompt)
186

```

Next, we will colorize all of our print statements to make them clear

```
assistant.py > ...
1  import ollama
2  import chromadb
3  import psycopg
4  import ast
5  from colorama import Fore
6  from tqdm import tqdm
7  from psycopg.rows import dict_row
8  |
9  client = chromadb.Client()
10 |
11 system_prompt = (
12     "You are an AI assistant that has memory of every conversation you have ever had with this user."
13     "On every prompt from the user, the system has checked for any relevant messages you have had with them."
14     "If any embedded previous conversations are attached, use them for context to responding to the user's message."
15     "if the context is relevant and useful to responding. If the recalled conversations are irrelevant, "
16     "disregard speaking about them and respond normally as an AI assistant. Do not talk about recalled conversations."
17     "Just use any useful data from the previous conversations and respond normally as an intelligent agent."
18 )
19 convo = [ {'role': 'system', 'content': system_prompt}]
20 |
21 DB_PARAMS = {
22     'dbname': 'memory_agent',
23     'user': 'example_user',
```

```
assistant.py > ⌂ stream_response
56     conn.close()
57
58 def stream_response(prompt):
59     response = ''
60     stream = ollama.chat(model='llama3', messages=convo, stream=True)
61     print(Fore.LIGHTGREEN_EX + '\nASSISTANT: ')
62
63     for chunk in stream:
64         content = chunk['message']['content']
65         response += content
66         print(content, end='', flush=True)
67
68     print('\n')
69     store_conversations(prompt=prompt, response=response)
70     convo.append({'role': 'assistant', 'content': response})
71
72 def create_vector_db(conversations):
73     vector_db_name = 'conversations'
74
75     try:
76         client.delete_collection(name=vector_db_name)
77     except ValueError:
78         pass
```

```
assistant.py > ⌂ create_queries
111 def create_queries(prompt):
112     [ {'role': 'user', 'content': prompt}]
113
114 response = ollama.chat(model='llama3', messages=query_convo)
115 print(Fore.YELLOW + f'\nVector database queries: {response["message"]["content"]}\n')
116
117 try:
118     return ast.literal_eval(response['message']['content'])
119 except:
120     return [prompt]
121
122 def classify_embedding(query, context):
123     classify_msg = (
124         "You are an embedding classification AI agent. Your input will be a prompt and one embedded vector at a time." +
125         "You will not respond as an AI assistant. You only respond \"yes\" or \"no\"." +
126         "Determine whether the context contains data that directly is related to the search query." +
127         "If the context is seemingly exactly what the search query needs, respond \"yes\" if it is an exact match." +
128         "related respond \"no\". Do not respond \"yes\" unless the content is highly relevant to the search query." +
129     )
130     classify_convo = [
131         {'role': 'system', 'content': classify_msg},
132         {'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: {query}'}
```

```

assistant.py > ...
163
164 conversations = fetch_conversations()
165 create_vector_db(conversations=conversations)
166
167 while True:
168     prompt = input(Fore.WHITE + 'USER: \n')
169
170     if prompt[:7].lower() == '/recall':
171         prompt = prompt[8:]
172         recall(prompt=prompt)
173         stream_response(prompt=prompt)
174     elif prompt[:7].lower() == '/forget':
175         remove_last_conversation()
176         convo = convo[:-2]
177         print('\n')
178     elif prompt[:9].lower() == '/memorize':
179         prompt = prompt[10:]
180         store_conversations(prompt=prompt, response='Memory stored.')
181         print('\n')
182     else:
183         convo.append({'role': 'user', 'content': prompt})
184

```

This completes the local AI agent with persistent memory

```

assistant.py > ...
assistant.py > ...
1 import ollama
2 import chromadb
3 import psycopg
4 import ast
5 from colorama import Fore
6 from tqdm import tqdm
7 from psycopg.rows import dict_row
8
9 client = chromadb.Client()
10
11 system_prompt = (
12     'You are an AI assistant that has memory of every conversation you have ever had with this user. '
13     'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
14     'If any embedded previous conversations are attached, use them for context to responding to the user. '
15     'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
16     'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
17     'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
18 )
19 convo = [ {'role': 'system', 'content': system_prompt}]
20
21 DB_PARAMS = {
22     'dbname': 'memory_agent',
23     'user': 'example_user',
24     'password': '123456',
25     'host': 'localhost',
26     'port': '5432'
27 }
28
29 def connect_db():
30     conn = psycopg.connect(**DB_PARAMS)
31     return conn
32
33 def fetch_conversations():
34     conn = connect_db()
35     with conn.cursor(row_factory=dict_row) as cursor:
36         cursor.execute('SELECT * FROM conversations')
37         conversations = cursor.fetchall()
38     conn.close()
39     return conversations
40

```

```
assistant.py > ...
33     conn.close()
34     return conversations
35
36 def store_conversations(prompt, response):
37     conn = connect_db()
38     with conn.cursor() as cursor:
39         cursor.execute(
40             'INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, %s, %s)',
41             (prompt, response)
42         )
43         conn.commit()
44     conn.close()
45
46 def remove_last_conversation():
47     conn = connect_db()
48     with conn.cursor() as cursor:
49         cursor.execute('DELETE FROM conversations WHERE id = (SELECT MAX(id) FROM conversations)')
50         cursor.commit()
51     conn.close()
52
53 def stream_response(prompt):
54     response = ''
55     stream = ollama.chat(model='llama3', messages=convo, stream=True)
56     print(Fore.LIGHTGREEN_EX + '\nASSISTANT:')
57
58     for chunk in stream:
59         content = chunk['message']['content']
60         response += content
61         print(content, end='', flush=True)
62
63     print('\n')
64     store_conversations(prompt=prompt, response=response)
65     convo.append({'role': 'assistant', 'content': response})
66
67 def create_vector_db(conversations):
68     vector_db_name = 'conversations'
69
70     try:
71         client.delete_collection(name=vector_db_name)
72     except ValueError:
73         pass
74
75     vector_db = client.create_collection(name=vector_db_name)
76
```

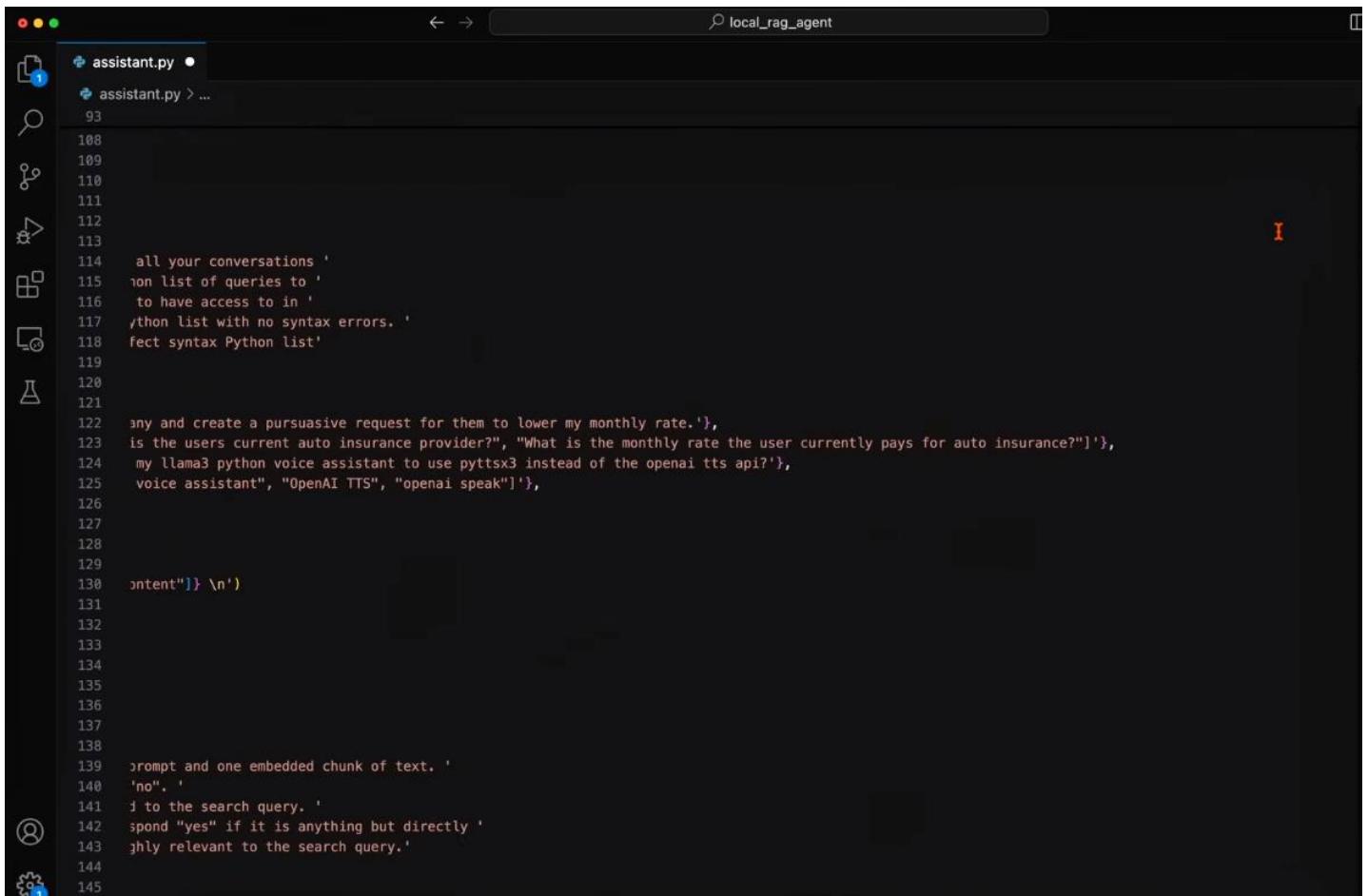
```
assistant.py > ...
72     def create_vector_db(conversations):
73         vector_db_name = 'conversations'
74
75         try:
76             client.delete_collection(name=vector_db_name)
77         except ValueError:
78             pass
79
80         vector_db = client.create_collection(name=vector_db_name)
81
82         for c in conversations:
83             serialized_convo = f'prompt: {c["prompt"]} response: {c["response"]}'
84             response = ollama.embeddings(model='nomic-embed-text', prompt=serialized_convo)
85             embedding = response['embedding']
86
87             vector_db.add(
88                 ids=[str(c['id'])],
89                 embeddings=[embedding],
90                 documents=[serialized_convo]
91             )
92
93     def retrieve_embeddings(queries, results_per_query=2):
94         embeddings = set()
95
96         for query in tqdm(queries, desc='Processing queries to vector database'):
97             response = ollama.embeddings(model='nomic-embed-text', prompt=query)
98             query_embedding = response['embedding']
99
100            vector_db = client.get_collection(name='conversations')
101            results = vector_db.query(query_embeddings=[query_embedding], n_results=results_per_query)
102            best_embeddings = results['documents'][0]
103
104            for best in best_embeddings:
105                if best not in embeddings:
106                    if 'yes' in classify_embedding(query=query, context=best):
107                        embeddings.add(best)
108
109    return embeddings
110
```

assistant.py

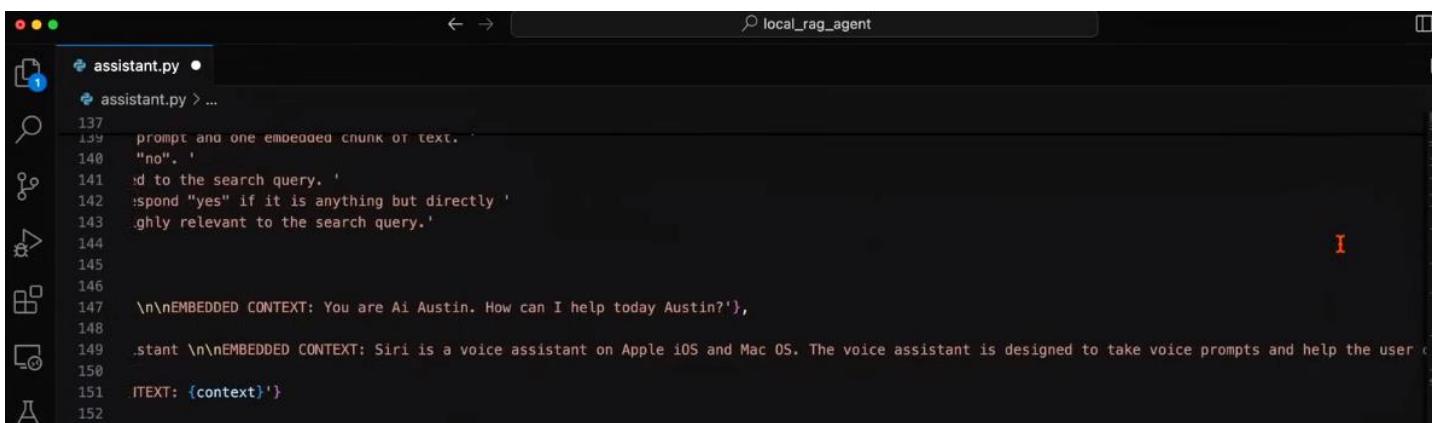
```
assistant.py > ...
93     def retrieve_embeddings(queries, results_per_query=2):
105         if best not in embeddings:
106             if 'yes' in classify_embedding(query=query, context=best):
107                 embeddings.add(best)
108
109     return embeddings
110
111 def create_queries(prompt):
112     query_msg = (
113         'You are a first principle reasoning search query AI agent. '
114         'Your list of search queries will be ran on an embedding database of all your conversations '
115         'you have ever had with the user. With first principles create a Python list of queries to '
116         'search the embeddings database for any data that would be necessary to have access to in '
117         'order to correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
118         'Do not explain anything and do not ever generate anything but a perfect syntax Python list'
119     )
120     query_convo = [
121         {'role': 'system', 'content': query_msg},
122         {'role': 'user', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower my monthly rate.'},
123         {'role': 'assistant', 'content': ['What is the users name?', "What is the users current auto insurance provider?", "What is the monthly rate the user currently pays?"]},
124         {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead of the openai tts api?'},  
125         {'role': 'assistant', 'content': ['Llama3 voice assistant", "Python voice assistant", "OpenAI TTS", "openai speak"]},
126         {'role': 'user', 'content': prompt}
127     ]
128
129     response = ollama.chat(model='llama3', messages=query_convo)
130     print(Fore.YELLOW + f'\nVector database queries: {response["message"]["content"]}\n')
131
132     try:
133         return ast.literal_eval(response['message']['content'])
134     except:
135         return [prompt]
136
137 def classify_embedding(query, context):
138     classify_msg = (
139         'You are an embedding classification AI agent. Your input will be a prompt and one embedded chunk of text. '
140         'You will not respond as an AI assistant. You only respond "yes" or "no". '
141         'Determine whether the context contains data that directly is related to the search query. '
142         'If the context is seemingly exactly what the search query needs, respond "yes" if it is anything but directly '
143         'related respond "no". Do not respond "yes" unless the content is highly relevant to the search query.'
144
145     : [
```

assistant.py

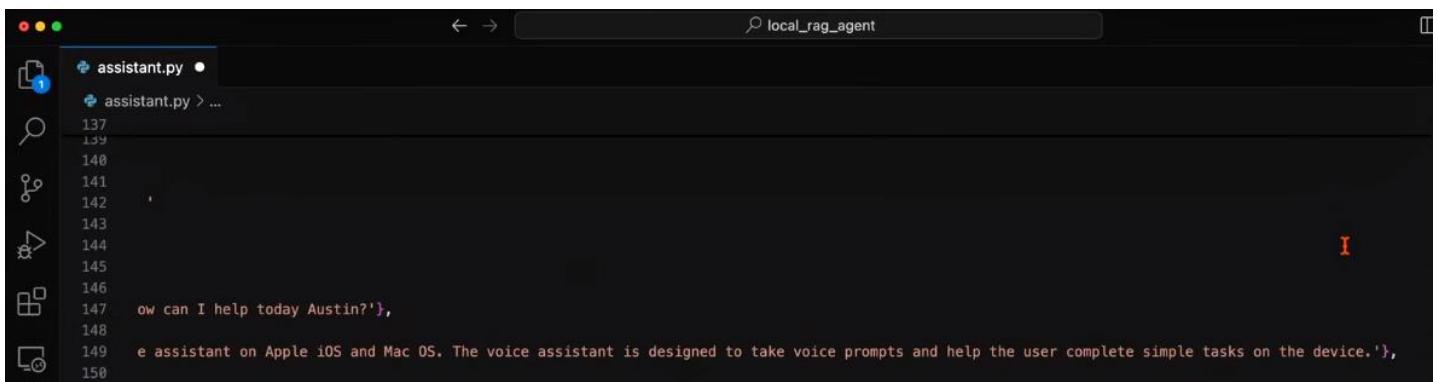
```
assistant.py > ...
93     .args(queries, results_per_query=2):
108
109     ls
110
111     prompt):
112
113     'first principle reasoning search query AI agent. '
114     if search queries will be ran on an embedding database of all your conversations '
115     ever had with the user. With first principles create a Python list of queries to '
116     embeddings database for any data that would be necessary to have access to in '
117     correctly respond to the prompt. Your response must be a Python list with no syntax errors. '
118     .ain anything and do not ever generate anything but a perfect syntax Python list'
119
120
121     'stem', 'content': query_msg},
122     'er', 'content': 'Write an email to my car insurance company and create a persuasive request for them to lower my monthly rate.'},
123     'sistant', 'content': ['What is the users name?', "What is the users current auto insurance provider?", "What is the monthly rate the user currently pays?"],
124     'er', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead of the openai tts api?'},  
125     'sistant', 'content': ['Llama3 voice assistant", "Python voice assistant", "OpenAI TTS", "openai speak"]},
126     'er', 'content': prompt}
127
128
129     ia.chat(model='llama3', messages=query_convo)
130     IW + f'\nVector database queries: {response["message"]["content"]}\n'
131
132
133     .literal_eval(response['message']['content'])
134
135     apt]
136
137     .ng(query, context):
138
139     'embedding classification AI agent. Your input will be a prompt and one embedded chunk of text. '
140     't respond as an AI assistant. You only respond "yes" or "no". '
141     'whether the context contains data that directly is related to the search query. '
142     'text is seemingly exactly what the search query needs, respond "yes" if it is anything but directly '
143     'pond "no". Do not respond "yes" unless the content is highly relevant to the search query.'
144
145     : [
```



```
assistant.py
assistant.py > ...
93
108
109
110
111
112
113
114     all your conversations '
115     ion list of queries to '
116     to have access to in '
117     ython list with no syntax errors. '
118     ect syntax Python list'
119
120
121
122     any and create a persuasive request for them to lower my monthly rate.'},
123     is the users current auto insurance provider?", "What is the monthly rate the user currently pays for auto insurance?"'],
124     my llama3 python voice assistant to use pyttsx3 instead of the openai tts api?'},
125     voice assistant", "OpenAI TTS", "openai speak"]),
126
127
128
129
130     content"]} \n')
131
132
133
134
135
136
137
138
139     prompt and one embedded chunk of text. '
140     'no'.
141     d to the search query. '
142     spond "yes" if it is anything but directly '
143     ghly relevant to the search query.'
144
145
146
147     \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
148
149     stant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS. The voice assistant is designed to take voice prompts and help the user complete simple tasks on the device.'},
150
151     ITTEXT: {context})
152
```



```
assistant.py
assistant.py > ...
137
138     prompt and one embeaded chunk of text.
139     "no".
140     d to the search query. '
141     spond "yes" if it is anything but directly '
142     ghly relevant to the search query.'
143
144
145
146
147     \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
148
149     stant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS. The voice assistant is designed to take voice prompts and help the user complete simple tasks on the device.'},
150
151     ITTEXT: {context})
152
```



```
assistant.py
assistant.py > ...
137
138
139
140
141     ,
142
143
144
145
146
147     ow can I help today Austin?'},
148
149     e assistant on Apple iOS and Mac OS. The voice assistant is designed to take voice prompts and help the user complete simple tasks on the device.'},
150
```

```
assistant.py
assistant.py > ...
137 def classify_embedding(query, context):
138     {'role': 'system', 'content': classify_msg},
139     {'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
140     {'role': 'assistant', 'content': 'yes'},
141     {'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS X.'},
142     {'role': 'assistant', 'content': 'no'},
143     {'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
144 ]
153
154 response = ollama.chat(model='llama3', messages=classify_convo)
155
156 return response['message']['content'].strip().lower()
157
158 def recall(prompt):
159     queries = create_queries(prompt=prompt)
160     embeddings = retrieve_embeddings(queries=queries)
161     convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
162     print(f'\n{len(embeddings)} message:response embeddings added for context.')
163
164 conversations = fetch_conversations()
165 create_vector_db(conversations=conversations)
166
167 while True:
168     prompt = input(Fore.WHITE + 'USER: \n')
169
170     if prompt[:7].lower() == '/recall':
171         prompt = prompt[8:]
172         recall(prompt=prompt)
173         stream_response(prompt=prompt)
174     elif prompt[:7].lower() == '/forget':
175         remove_last_conversation()
176         convo = convo[:-2]
177         print('\n')
178     elif prompt[:9].lower() == '/memorize':
179         prompt = prompt[10:]
180         store_conversations(prompt=prompt, response='Memory stored.')
181         print('\n')
182     else:
183         convo.append({'role': 'user', 'content': prompt})
184         stream_response(prompt=prompt)
```

```
assistant.py
assistant.py > ...
137 def classify_embedding(query, context):
138     {'role': 'system', 'content': classify_msg},
139     {'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
140     {'role': 'assistant', 'content': 'yes'},
141     {'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS X.'},
142     {'role': 'assistant', 'content': 'no'},
143     {'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
144 ]
153
154 response = ollama.chat(model='llama3', messages=classify_convo)
155
156 return response['message']['content'].strip().lower()
157
158 def recall(prompt):
159     queries = create_queries(prompt=prompt)
160     embeddings = retrieve_embeddings(queries=queries)
161     convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
162     print(f'\n{len(embeddings)} message:response embeddings added for context.')
163
164 conversations = fetch_conversations()
165 create_vector_db(conversations=conversations)
166
167 while True:
168     prompt = input(Fore.WHITE + 'USER: \n')
169
170     if prompt[:7].lower() == '/recall':
171         prompt = prompt[8:]
172         recall(prompt=prompt)
173         stream_response(prompt=prompt)
174     elif prompt[:7].lower() == '/forget':
175         remove_last_conversation()
176         convo = convo[:-2]
177         print('\n')
178     elif prompt[:9].lower() == '/memorize':
179         prompt = prompt[10:]
180         store_conversations(prompt=prompt, response='Memory stored.')
181         print('\n')
182     else:
183         convo.append({'role': 'user', 'content': prompt})
184         stream_response(prompt=prompt)
```