Local UNLIMITED Memory Ai Agent | Ollama RAG Crash Course

Ai Austin
21.1K subscribers
Subscribe

2.8K | Share | Ask | Download | ...

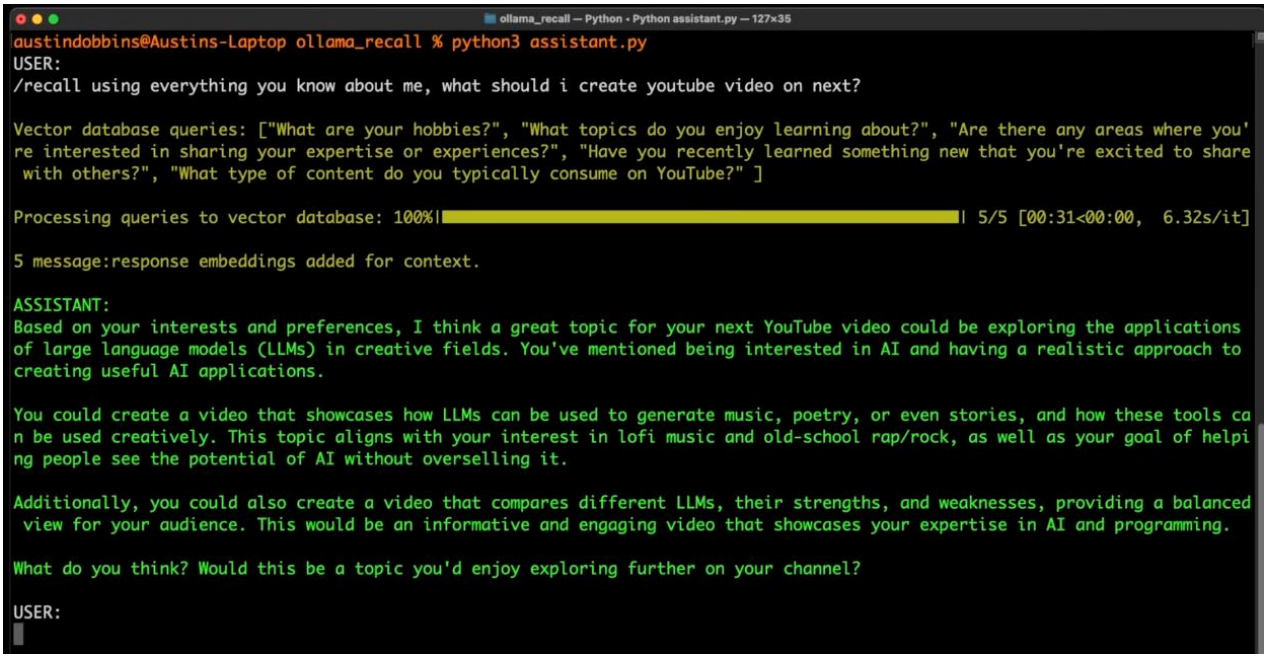
75,181 views Jul 11, 2024 #ai #aiagents #programming
Learn how to create powerful Ai agents with Python in this easy to follow along crash course on Ollama RAG. In this video we build a RAG agent that stores every conversation in a PostgreSQL database, converts the SQL database into a vector embedding 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.

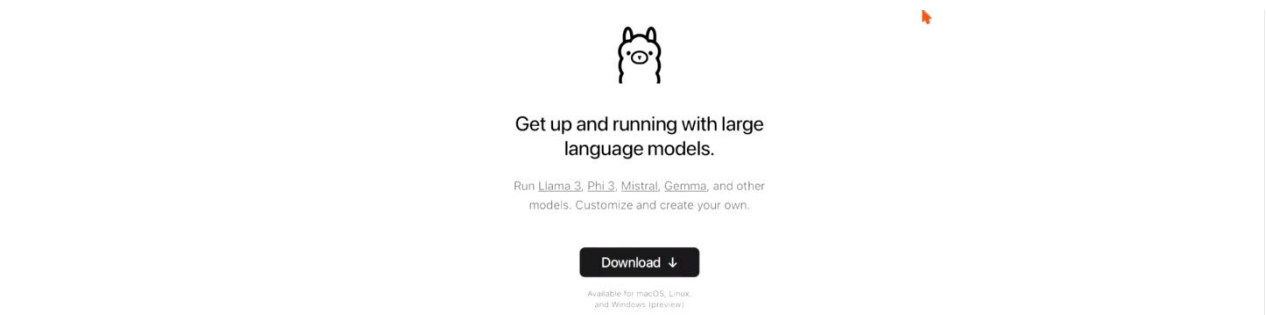
assistant.py > ...

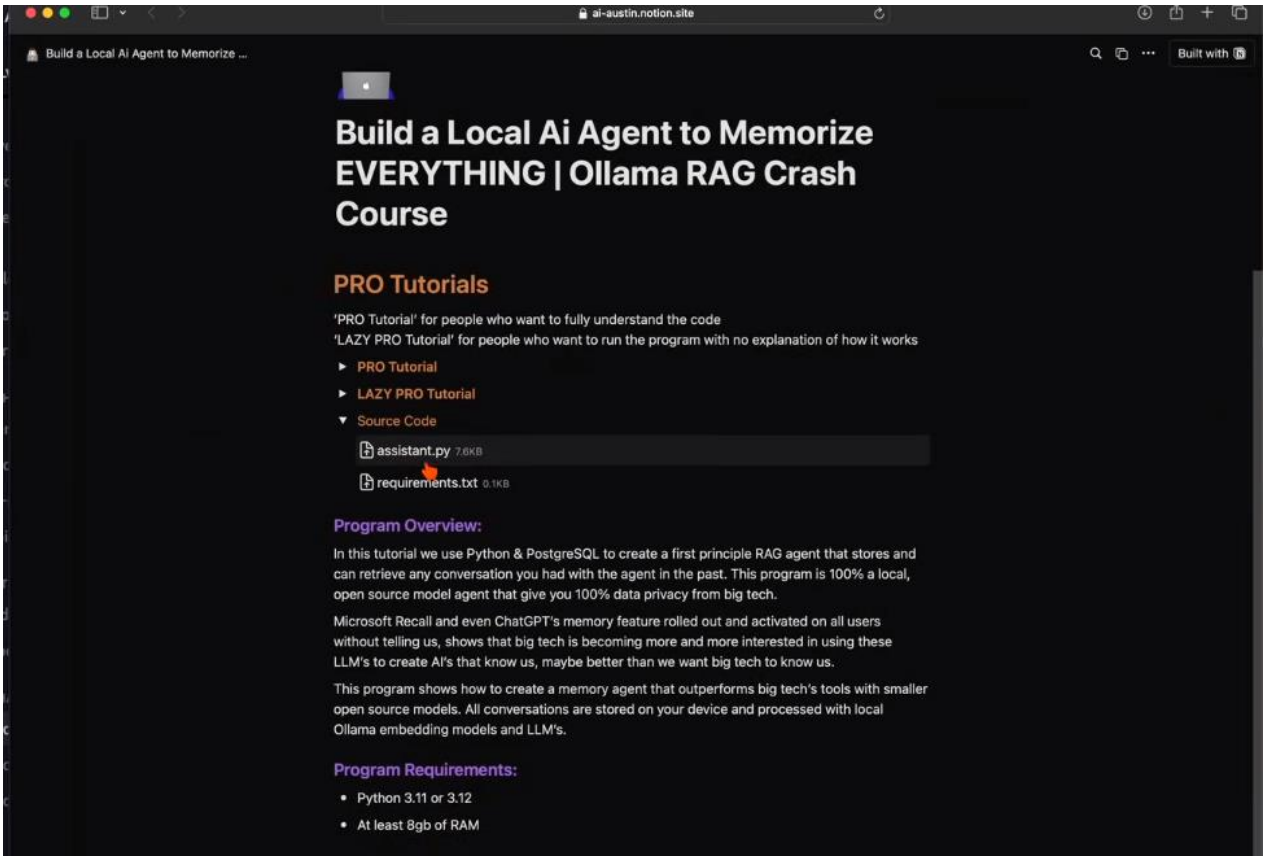
```
1   import ollama
2   import chromadb
3   import psycopg
4   import ast
5   from psycopg.rows import dict_row
6   from tqdm import tqdm
7   from colorama import Fore
8
9   client = chromadb.Client()
10  system_prompt = (...
18  convo = [{'role': 'system', 'content': system_prompt}]
19
20  DB_PARAMS = {...
27
28  def connect_db():...
31
32  def fetch_conversations():...
39
40  def create_vector_db(conversations):...
60
61  def create_queries(prompt):...
85
86  def classify_embedding(query, context):...
106
107 def retrieve_embeddings(queries, results_per_query=2):...
123
124 def store_conversation(prompt, response):...
130
131 def remove_last_conversation():...
137
138 def stream_response(prompt):...
152
153 def recall(prompt):...
```

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

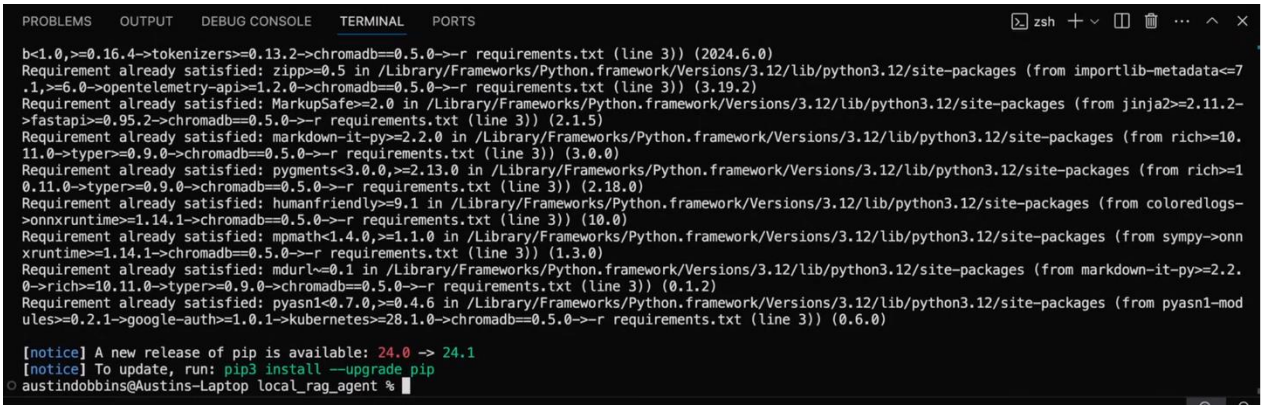
First Principles Retrieval Augmented Generation

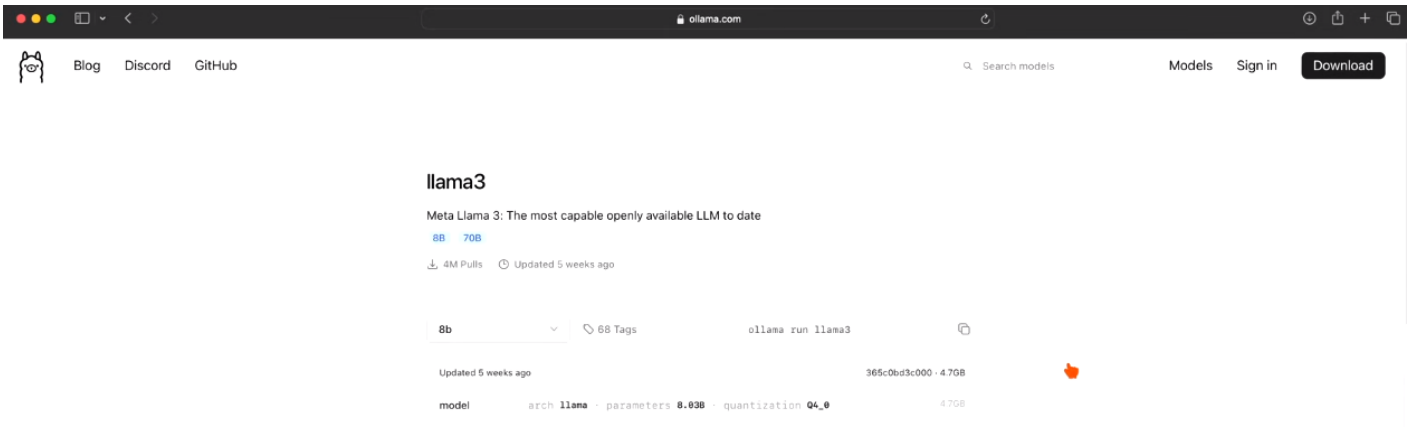






Active Python Releases





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?"
 }'
```

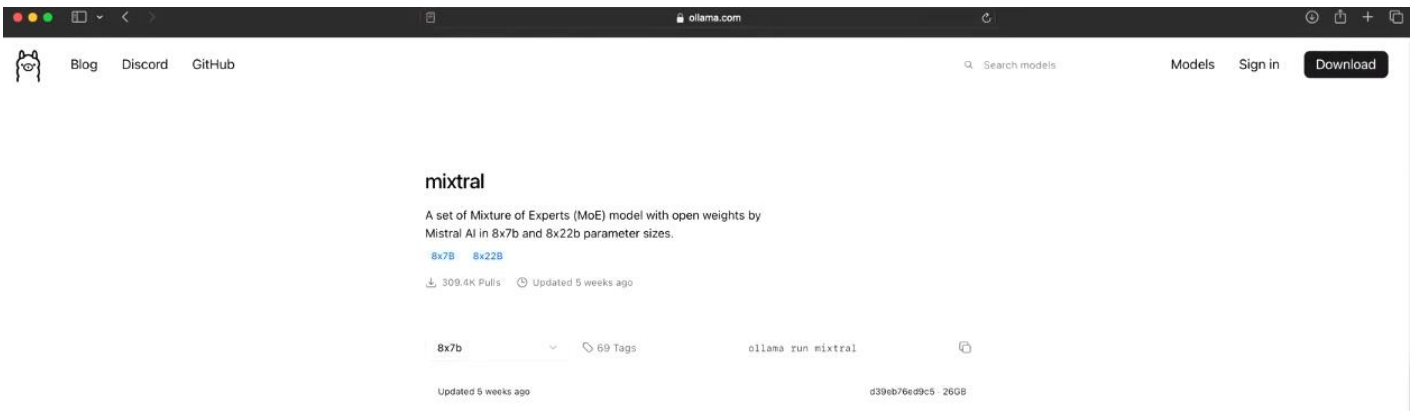
API documentation

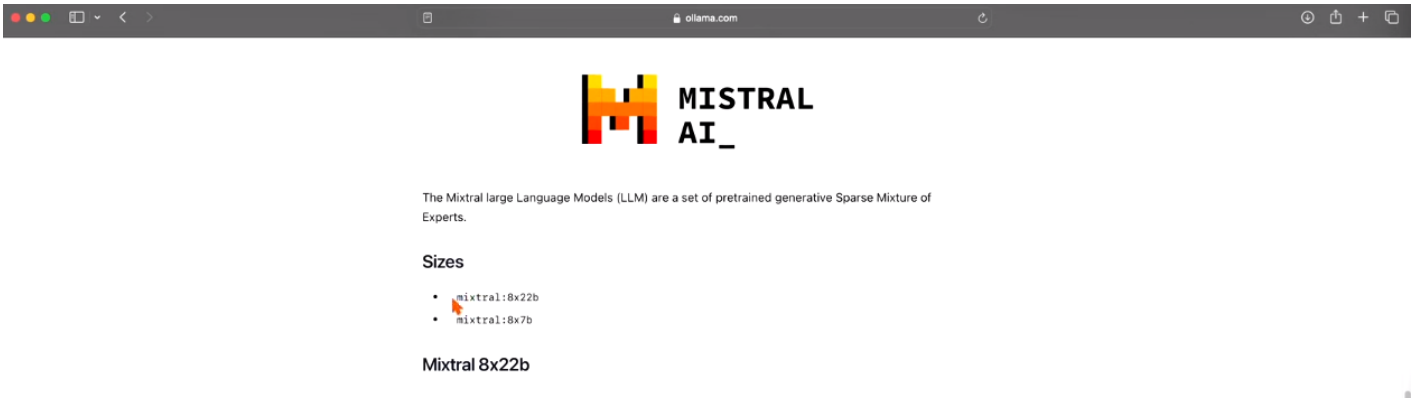
Model variants

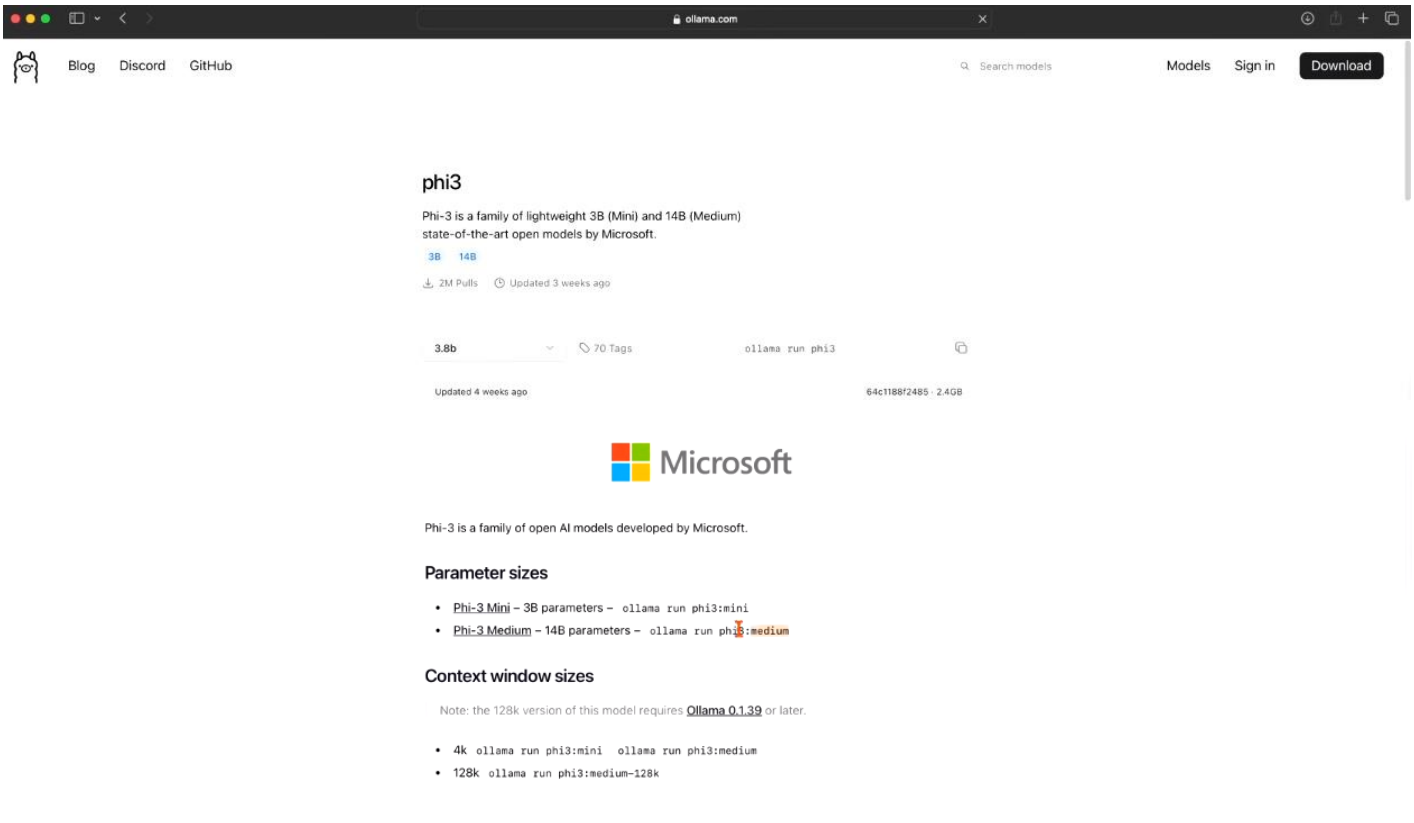
Instruct is fine-tuned for chat/dialogue use cases.

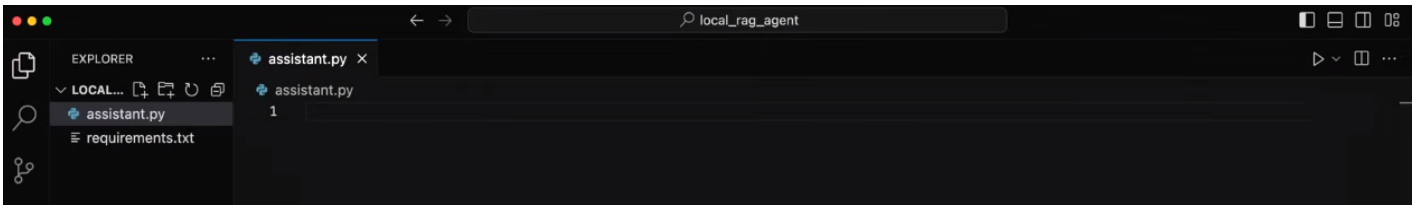
Example:

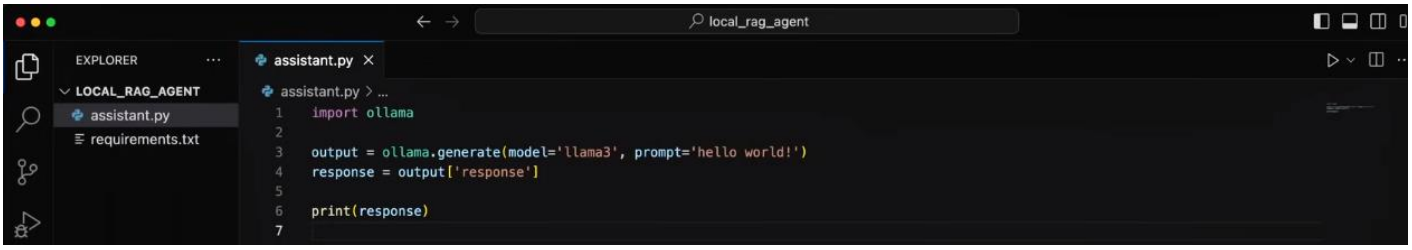
`ollama run llama3`

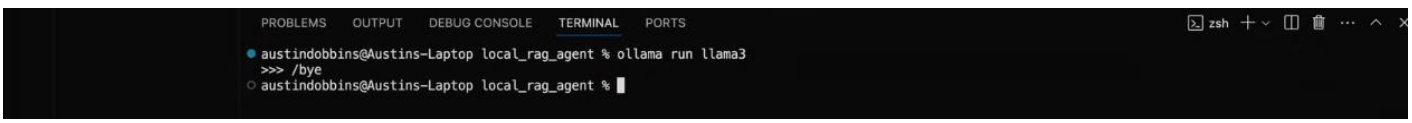


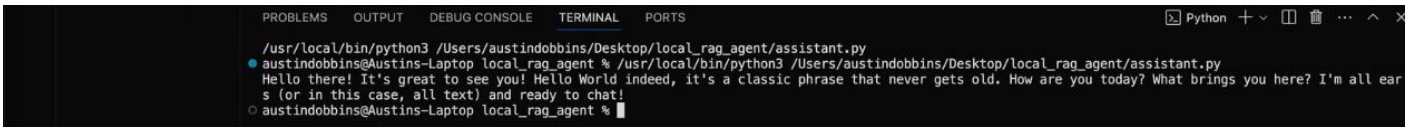

ollama.com
MISTRAL AI_
The Mixtral large Language Models (LLM) are a set of pretrained generative Sparse Mixture of Experts.
Sizes
mixtral:8x22b
mixtral:8x7b
Mixtral 8x22b


ollama.com
Blog
Discord
GitHub
Search models
Models
Sign in
Download
phi3
Phi-3 is a family of lightweight 3B (Mini) and 14B (Medium) state-of-the-art open models by Microsoft.
3B
14B
2M Pulls
Updated 3 weeks ago
3.8b
70 Tags
ollama run phi3
Updated 4 weeks ago
64c1188f2485 · 2.4GB
Microsoft
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

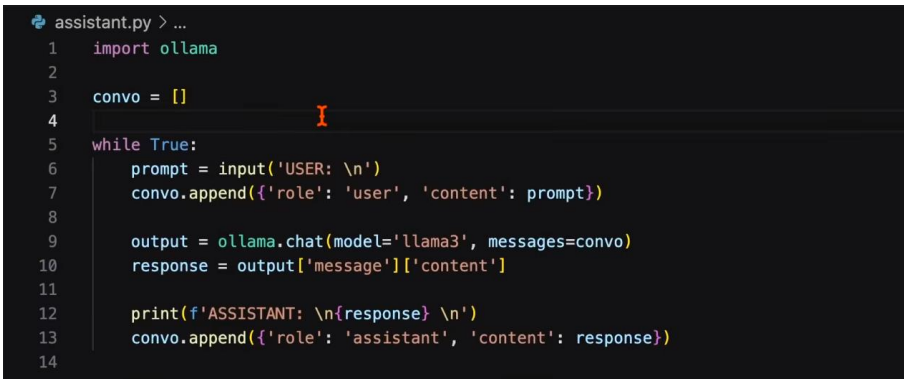

local_rag_agent
EXPLORER
assistant.py
LOCAL...
assistant.py
requirements.txt
assistant.py
1

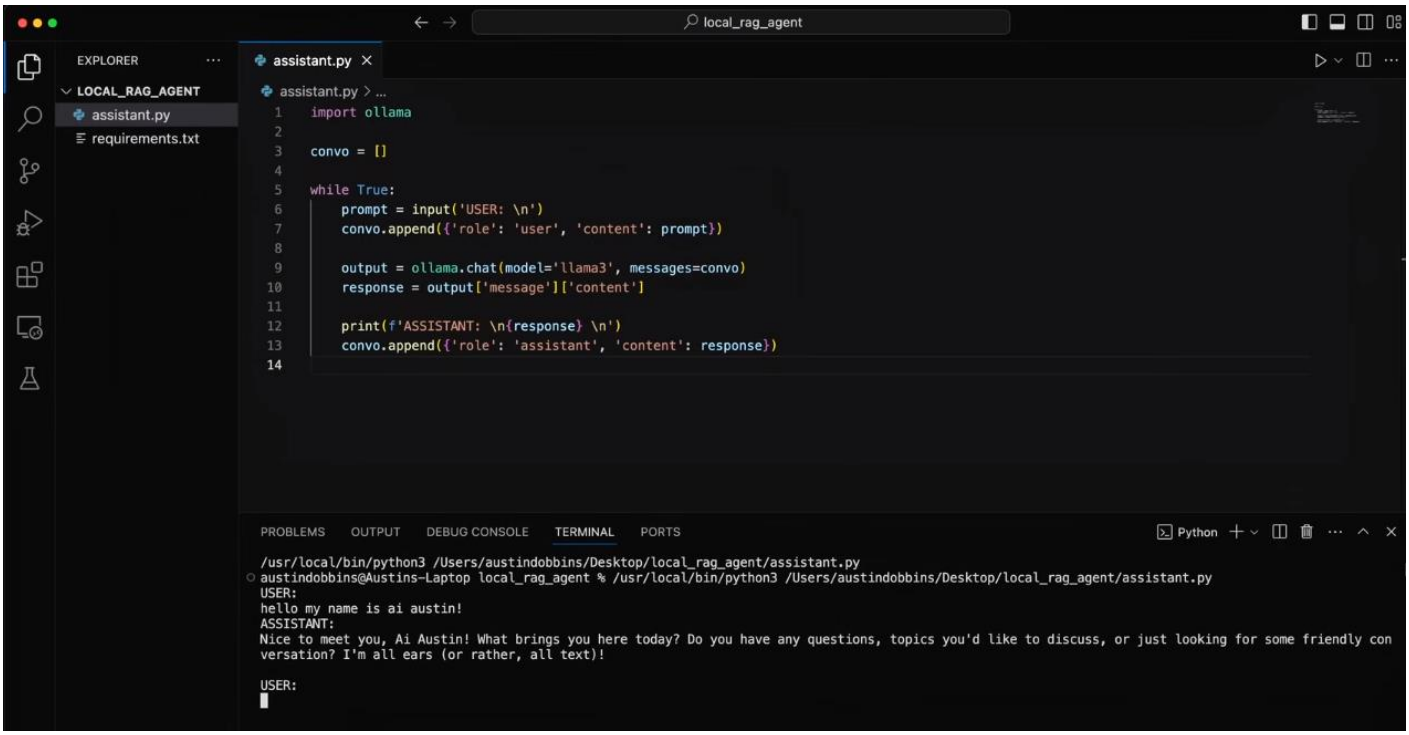

local_rag_agent
EXPLORER
assistant.py
LOCAL_RAG_AGENT
assistant.py
requirements.txt
assistant.py > ...
1 import ollama
2
3 output = ollama.generate(model='llama3', prompt='hello world!')
4 response = output['response']
5
6 print(response)
7

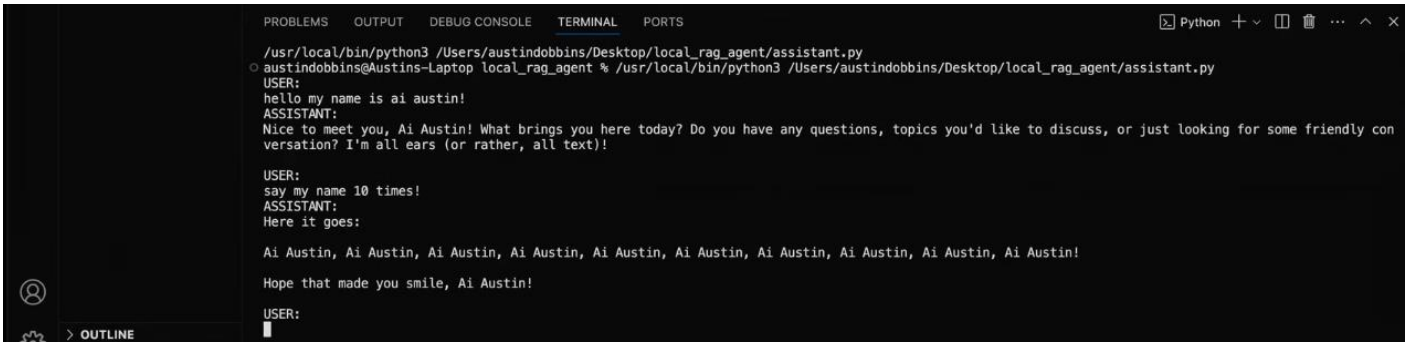

PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
zsh
austindobbins@Austins-Laptop local_rag_agent % ollama run llama3
>>> /bye
austindobbins@Austins-Laptop local_rag_agent %



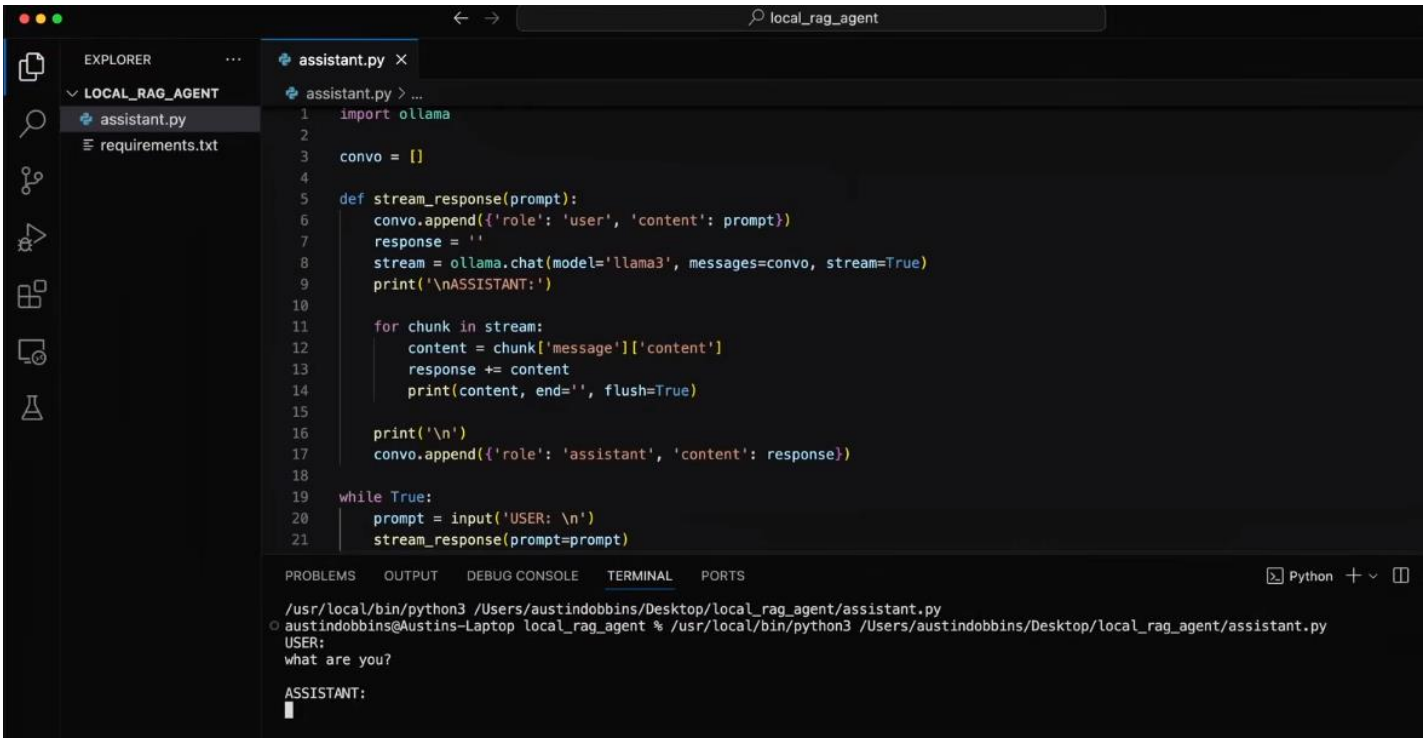
This works! Next, we need to make the program fully conversational

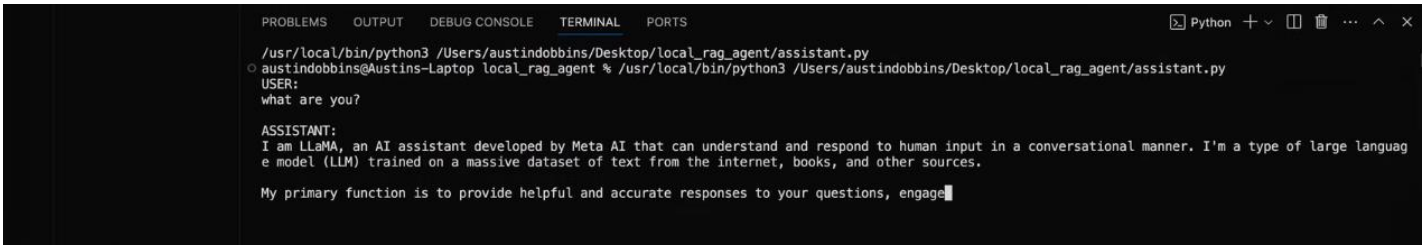


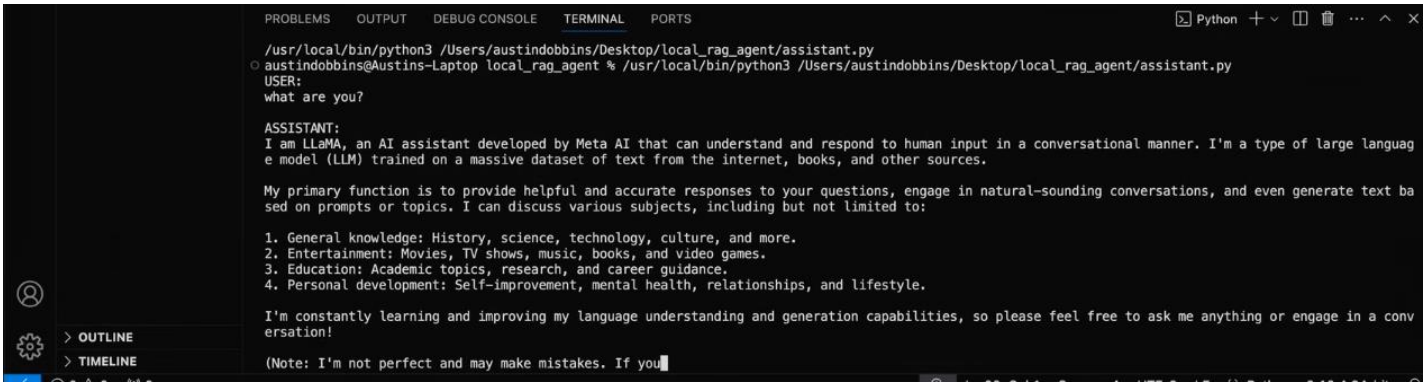




Implementing a conversation works! Next, let us implement Ollama response streaming to reduce response latency.

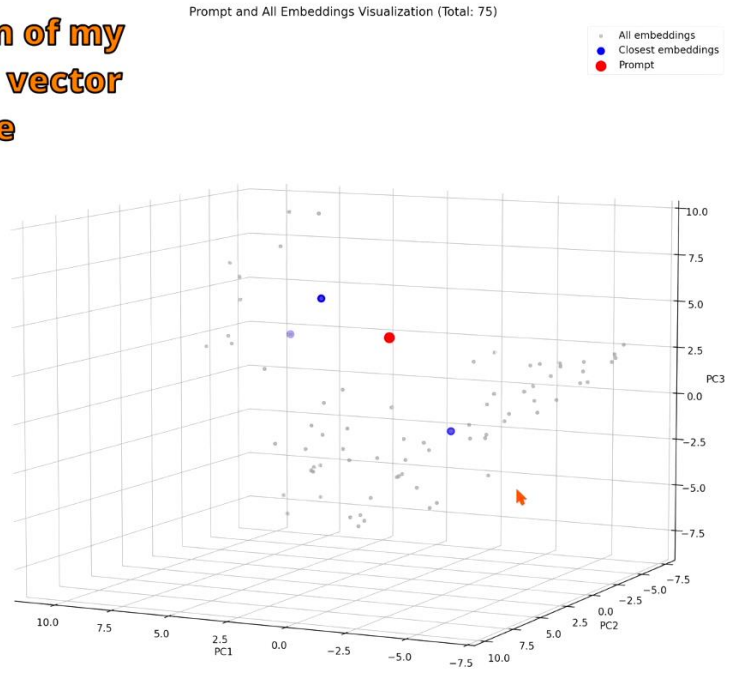


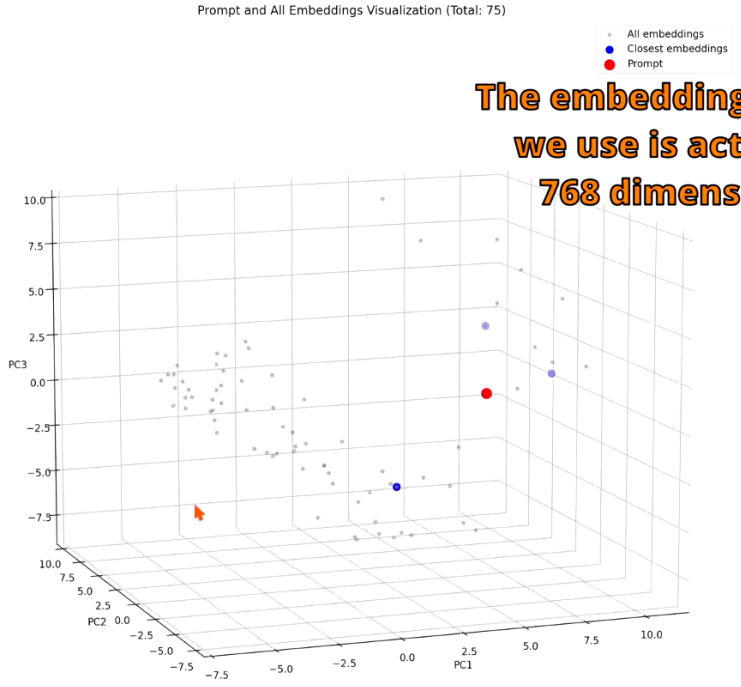




Streaming the response works! Next, we need to be able to create vector embeddings of our previous conversations for storage and use in future recalls. We will be using **nomic-embed-text** and **ChromaDB**

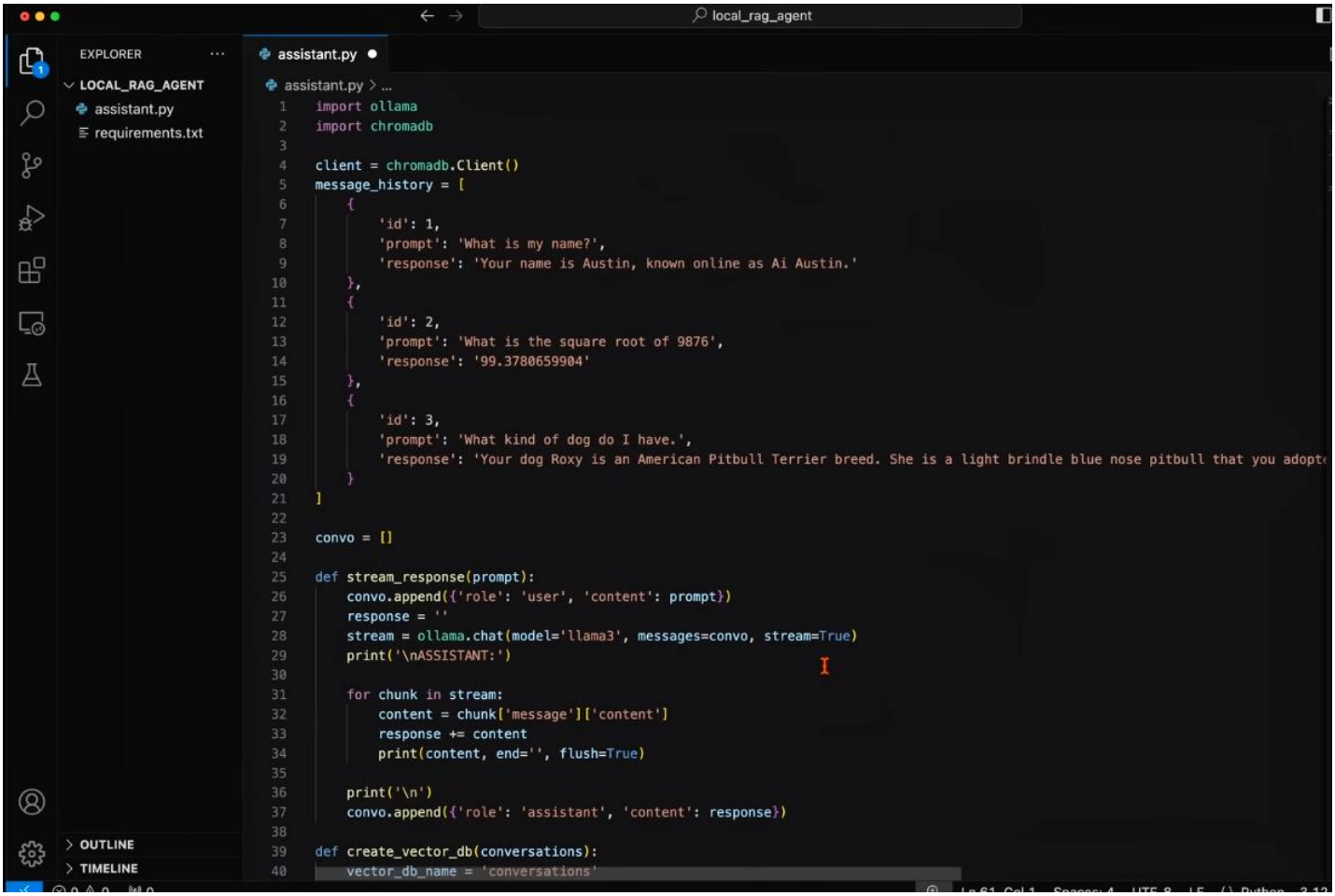
3d visualization of my conversations vector database

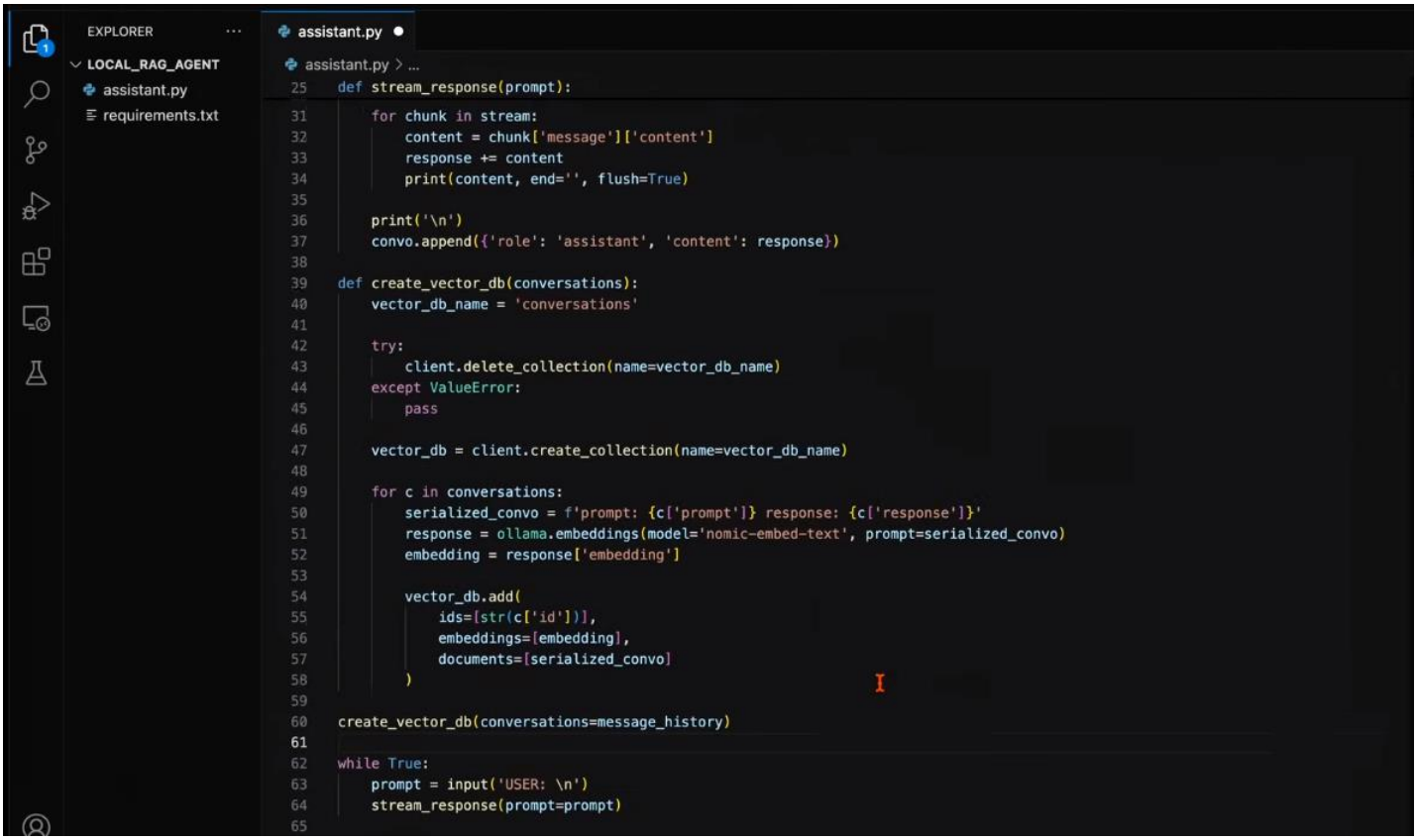




The embedding model we use is actually 768 dimensions

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    Python

austindobbins@Austins-Laptop local_rag_agent % ollama pull nomic-embed-text
pulling manifest
pulling 970aa74c0a90... 100%  274 MB
pulling c71d239df917... 100%   11 KB
pulling ce4a164fc046... 100%   17 B
pulling 31df23ea7daa... 100%  420 B
verifying sha256 digest
writing manifest
removing any unused layers
success
austindobbins@Austins-Laptop local_rag_agent %
```

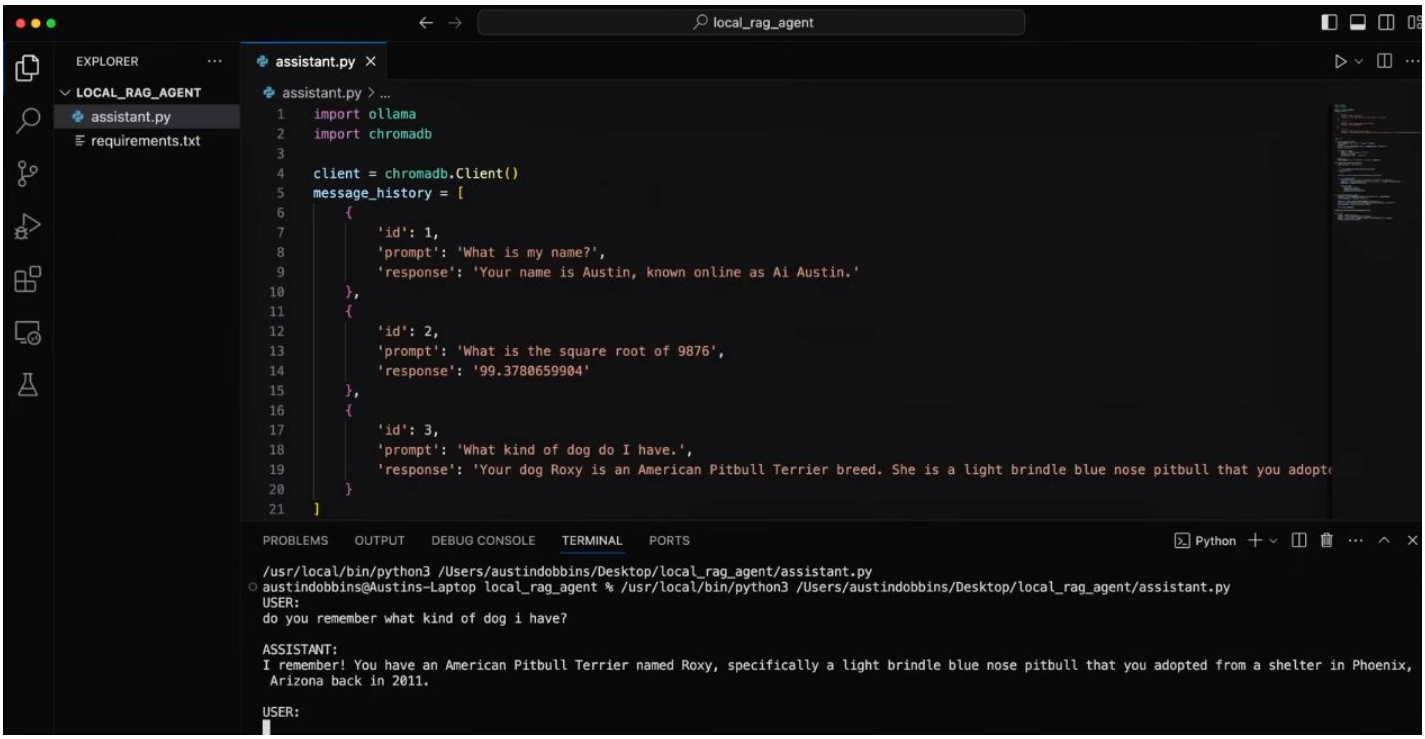




Next, let us now get the program to create a vector representation of each prompt input and mathematically retrieve the most relevant embedding from our database.

```python
assistant.py > retrieve_embeddings
    def create_vector_db(conversations):
                embeddings=[embedding],
                documents=[serialized_convo]
            )

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

        vector_db = client.get_collection(name='conversations')
        results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
        best_embedding = results['documents'][0][0]

        return best_embedding

    create_vector_db(conversations=message_history)

    while True:
        prompt = input('USER: \n')
        stream_response(prompt=prompt)
```

```python
assistant.py > ...
        return best_embedding

    create_vector_db(conversations=message_history)

    while True:
        prompt = input('USER: \n')
        context = retrieve_embeddings(prompt=prompt)
        prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS: {context}'
        stream_response(prompt=prompt)
```



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:

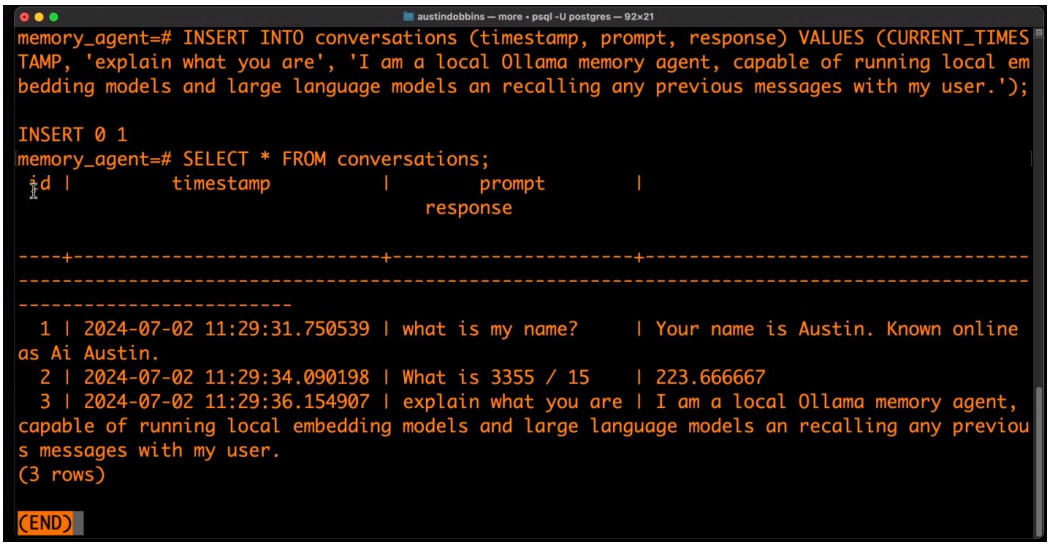
```
brew install postgresql@16
brew services start postgresql@16
brew install libpq
brew link --force libpq
```

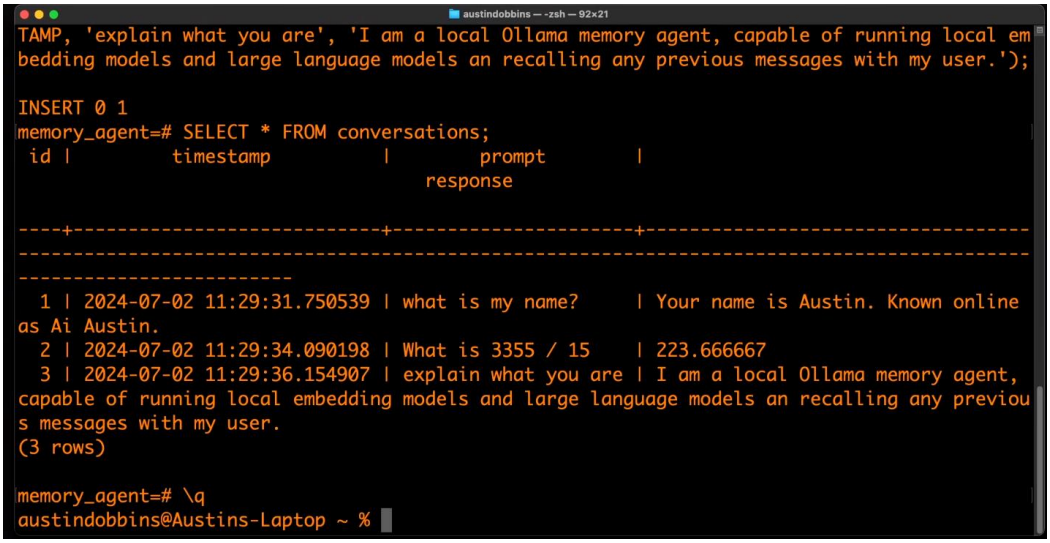
- ▶ Step by Step (linux)
- ▶ Step by Step (windows)

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

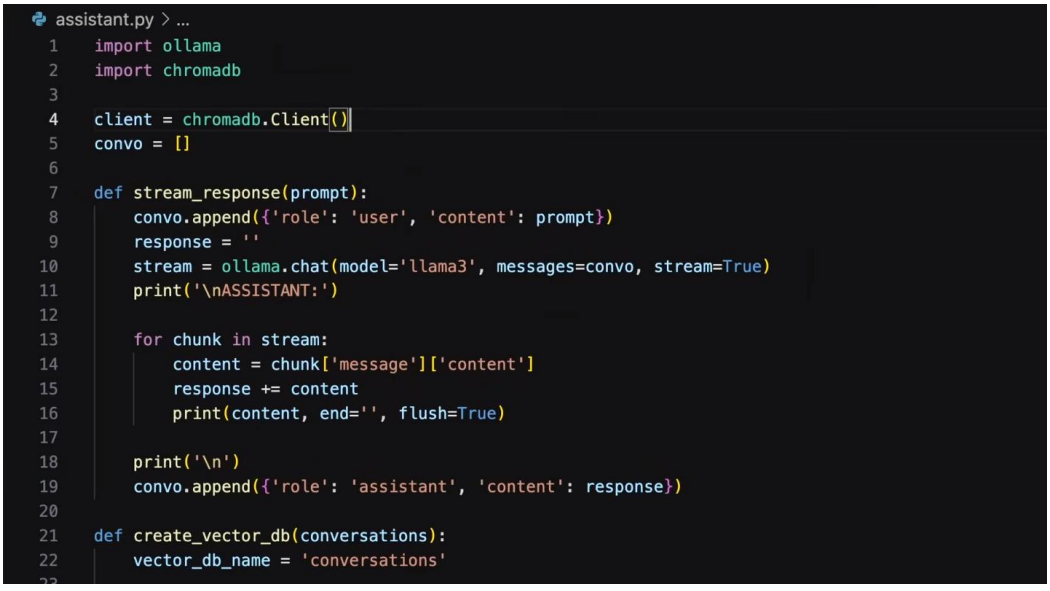
```
austindobbins — psql -U postgres — 92×21
austindobbins@Austins-Laptop ~ % psql -U postgres
psql (16.3)
Type "help" for help.

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(# );
```

```
austindobbins — psql -U postgres — 92×21
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_TIMES
TAMP, 'what is my name?', 'Your name is Austin. Known online as Ai Austin.'
```





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

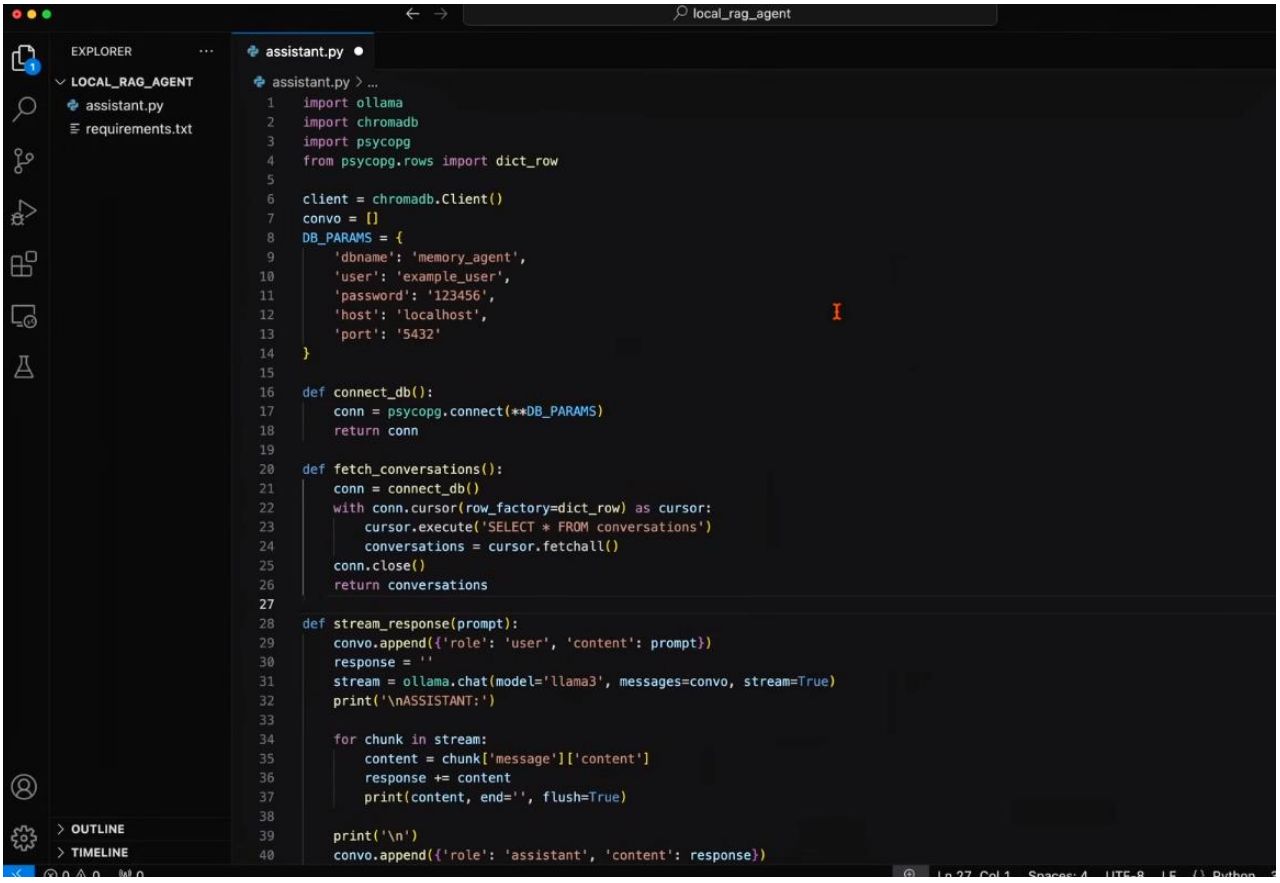


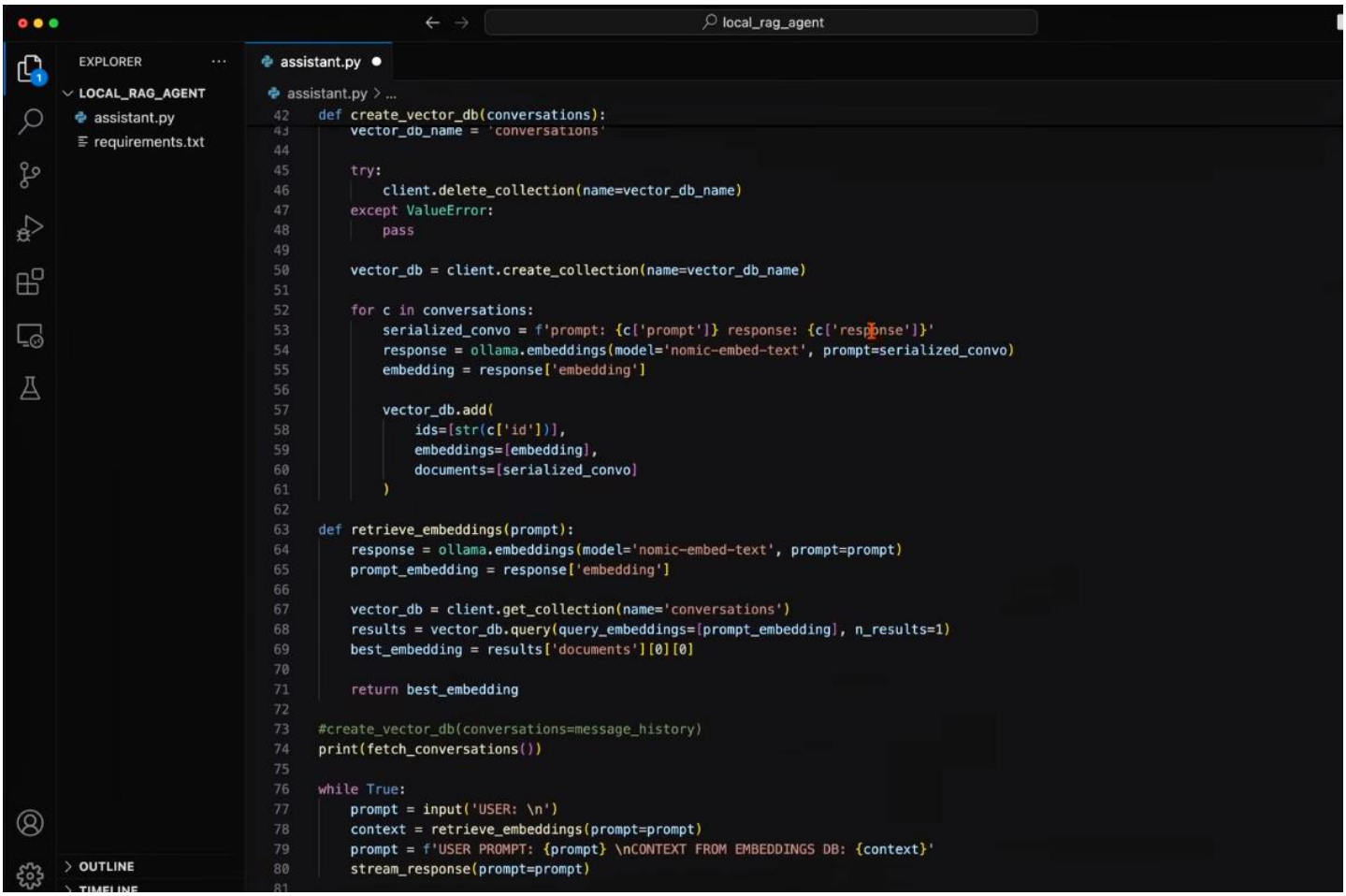
assistant.py > ...

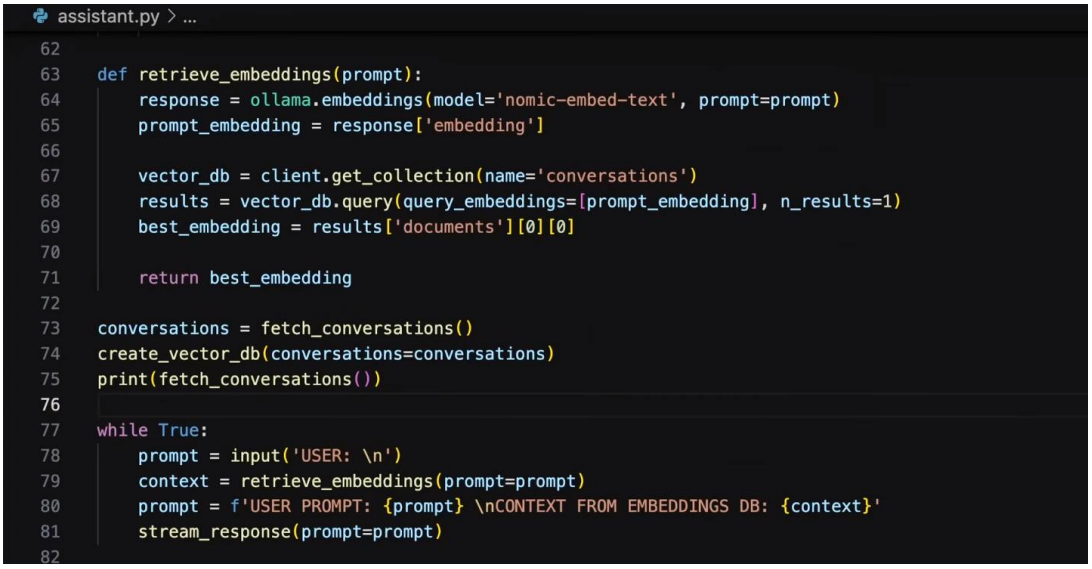
```python
import ollama
import chromadb
import psycopg
from psycopg.rows import dict_row

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

```

assistant.py > ...

```python
def connect_db():
    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 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')
```







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

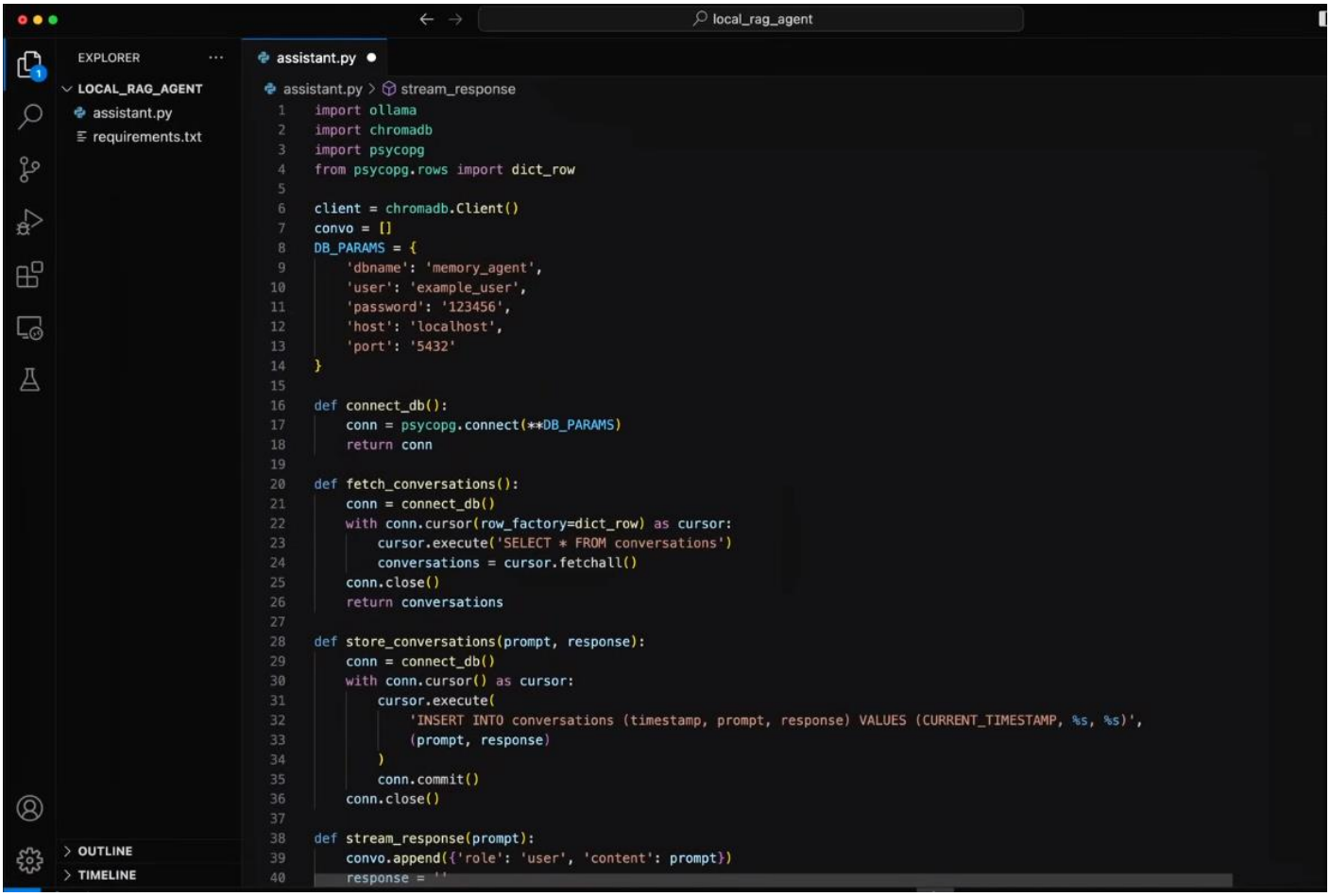
assistant.py > store_conversations

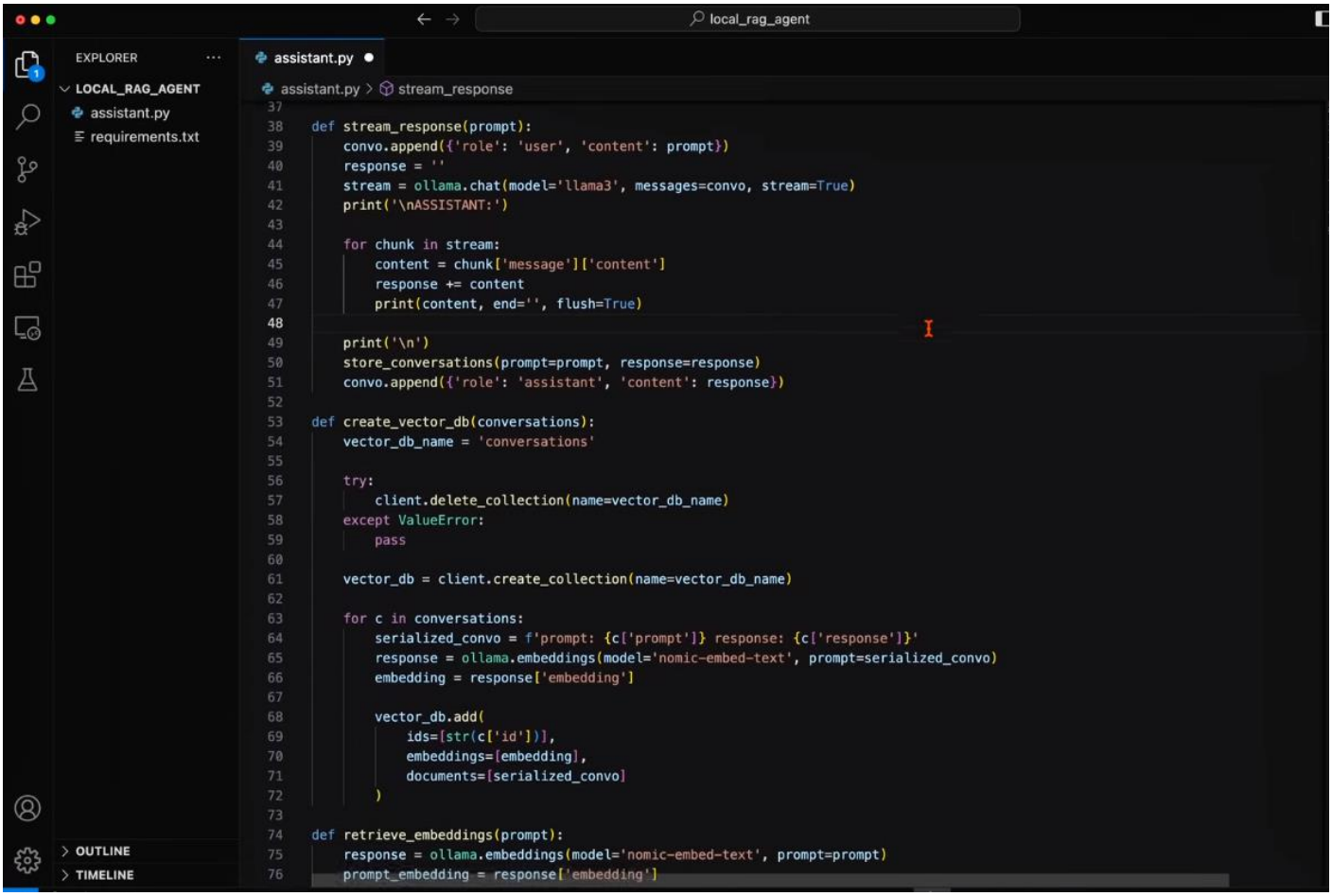
```python
def fetch_conversations():
        conversations = cursor.fetchall()
    conn.close()
    return conversations

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 = ''
    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)
```

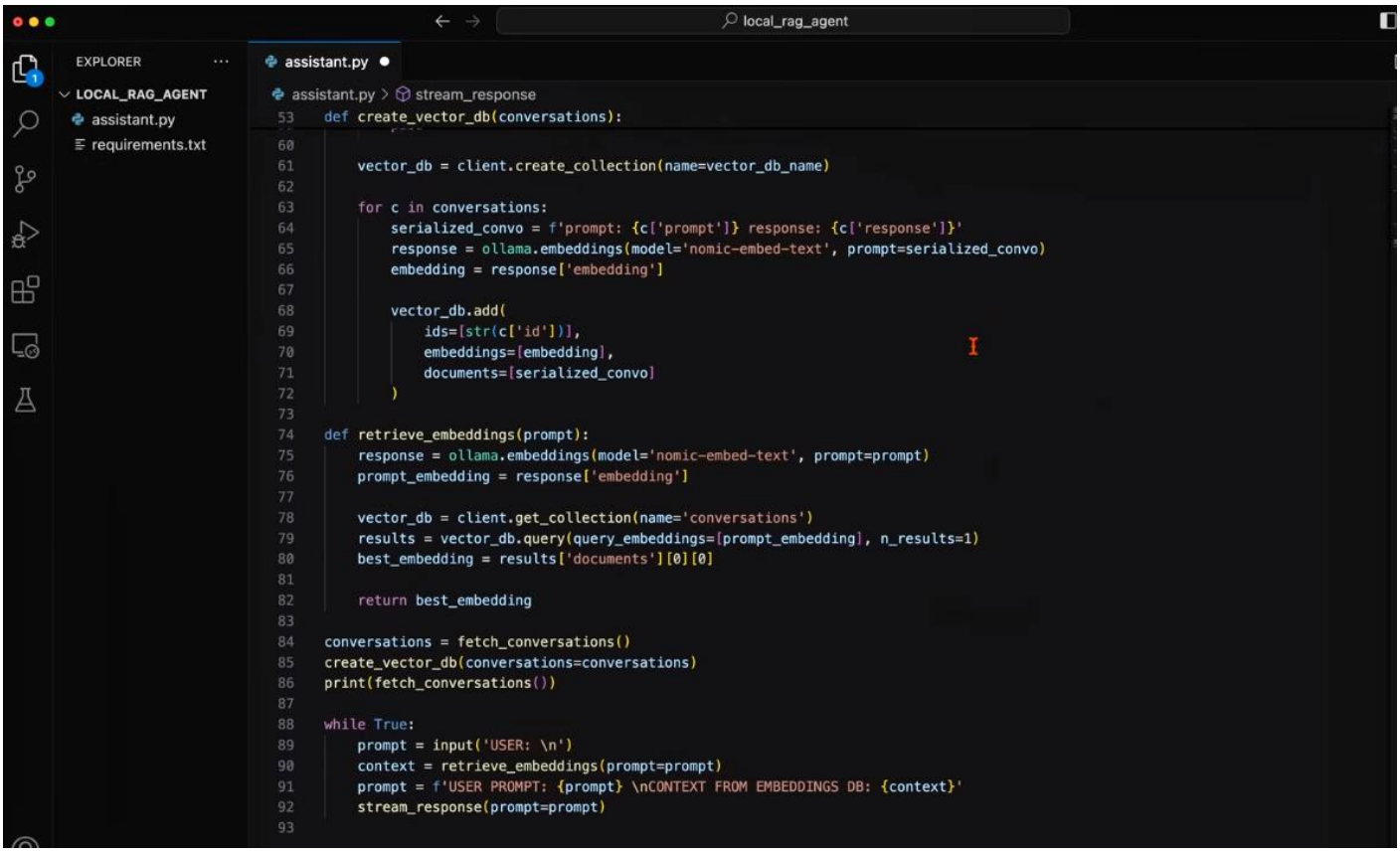
assistant.py > ...

```python
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)
```

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.


local_rag_agent
EXPLORER
LOCAL_RAG_AGENT
assistant.py
requirements.txt
assistant.py
assistant.py > stream_response
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 = ''
OUTLINE
TIMELINE


local_rag_agent
EXPLORER
LOCAL_RAG_AGENT
assistant.py
requirements.txt
assistant.py
assistant.py > stream_response
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(
ids=[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']
OUTLINE
TIMELINE



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

```python
import ollama
import chromadb
import psycopg
from psycopg.rows import dict_row

client = chromadb.Client()

system_prompt = (
    'You are an AI assistant that has memory of every conversation you have ever had with this user. '
    'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
    'If any embedded previous conversations are attached, use them for context to responding to the user, '
    'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
    'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
    'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
)
convo = [{'role': 'system', 'content': system_prompt}]

DB_PARAMS = {
    'dbname': 'memory_agent',
    'user': 'example_user',
    'password': '123456',
    'host': 'localhost',
    'port': '5432'
}

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

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

assistant.py > create_queries

```python
def retrieve_embeddings(prompt):
    vector_db = client.get_collection(name='conversations')
    results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
    best_embedding = results['documents'][0][0]

    return best_embedding

def create_queries(prompt):


conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)

```

ollama_recall — Python • Python assistant.py — 127×35

```
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 — 127×35

```
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

```python
def retrieve_embeddings(prompt):
    vector_db = client.get_collection(name='conversations')
    results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
    best_embedding = results['documents'][0][0]

    return best_embedding

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'
    )

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)
```

assistant.py > ...

```python
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 pursuasive request for them to lowe
        {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "Wha
        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 ins
        {'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)


conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
```

assistant.py > create_queries

```python
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 pursuasive request for them to lowe
        {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "Wha
        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 ins
        {'role': 'assistant', 'content': '["Llama3 voice assistant", "Python voice assistant", "OpenAI TTS", "openai speak"]'},

    ]

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)
```

assistant.py > ...

```python
def create_queries(prompt):
        {'role': 'user', 'content': prompt}
    ]

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

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)
```

assistant.py ● example.py ●

example.py > ...

```python
llm_output = "['query1', 'query2', 'query3']"
we_need = ['query1', 'query2', 'query3']
```

assistant.py > ...

```python
import ollama
import chromadb
import psycopg
import ast
from psycopg.rows import dict_row

client = chromadb.Client()

system_prompt = (
    'You are an AI assistant that has memory of every conversation you have ever had with this user. '
    'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
    'If any embedded previous conversations are attached, use them for context to responding to the user, '
    'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
    'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
    'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
)
convo = [{'role': 'system', 'content': system_prompt}]

DB_PARAMS = {
    'dbname': 'memory_agent',
    'user': 'example_user',
    'password': '123456',
    'host': 'localhost',
    'port': '5432'
}

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

assistant.py > ...

```python
def create_queries(prompt):
        {'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]

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)
```

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

assistant.py > ...

```python
import ollama
import chromadb
import psycopg
import ast
from psycopg.rows import dict_row

client = chromadb.Client()

system_prompt = (
    'You are an AI assistant that has memory of every conversation you have ever had with this user. '
    'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
    'If any embedded previous conversations are attached, use them for context to responding to the user, '
    'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
    'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
    'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
)
convo = [{'role': 'system', 'content': system_prompt}]

DB_PARAMS = {
    'dbname': 'memory_agent',
    'user': 'example_user',
    'password': '123456',
    'host': 'localhost',
    'port': '5432'
}

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

assistant.py > ...

```python
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 = ''
    stream = ollama.chat(model='llama3', messages=convo, stream=True)
    print('\nASSISTANT:')

    for chunk in stream:
```

assistant.py > ...

```python
def stream_response(prompt):
    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(
            ids=[str(c['id'])],
            embeddings=[embedding],
```

assistant.py > ...

```python
def create_vector_db(conversations):
        vector_db.add(
            ids=[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']

    vector_db = client.get_collection(name='conversations')
    results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
    best_embedding = results['documents'][0][0]

    return best_embedding

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 pursuasive request for them to lowe
        {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "Wha
```

assistant.py > ...

```python
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 pursuasive request for them to lowe
        {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "Wha
        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 ins
        {'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]

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())
```

assistant.py > ...

```python
def create_queries(prompt):
        '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 pursuasive request for them to lowe
        {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "Wha
        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 ins
        {'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]

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())
```

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

assistant.py > retrieve_embeddings

```python
def create_vector_db(conversations):
            embeddings=[embedding],
            documents=[serialized_convo]
        )

def retrieve_embeddings(queries, results_per_query=2):
    response = ollama.embeddings(model='nomic-embed-text', prompt=prompt)
    prompt_embedding = response['embedding']

    vector_db = client.get_collection(name='conversations')
    results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
    best_embedding = results['documents'][0][0]

    return best_embedding

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 pursuasive request for them to lowe
        {'role': 'assistant', 'content': '["What is the users name?", "What is the users current auto insurance provider?", "Wha
        {'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 ins
```

assistant.py > retrieve_embeddings

```python
def create_vector_db(conversations):
            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]

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

    vector_db = client.get_collection(name='conversations')
    results = vector_db.query(query_embeddings=[prompt_embedding], n_results=1)
    best_embedding = results['documents'][0][0]

    return best_embedding

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 '
```

assistant.py > ...

```python
def create_queries(prompt):
    except:
        return [prompt]

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)
```

We now can classify the embedding as below

assistant.py > classify_embedding

```python
def create_queries(prompt):
    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 h
        {'role': 'assistant', 'content': 'yes'},
        {'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistan
        {'role': 'assistant', 'content': 'no'},
        {'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
    ]

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
```

assistant.py > classify_embedding

```python
 response['message']['content'])



ntext):

assification AI agent. Your input will be a prompt and one embedded chunk of text. '
 an AI assistant. You only respond "yes" or "no". '
ontext contains data that directly is related to the search query. '
ngly exactly what the search query needs, respond "yes" if it is anything but directly '
o not respond "yes" unless the content is highly relevant to the search query.'


ent': classify_msg},
t': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
ontent': 'yes'},
t': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac OS. The
ontent': 'no'},
t': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}


ons()
onversations)
```

assistant.py > classify_embedding

```python
def create_queries(prompt):
    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
        {'role': 'assistant', 'content': 'yes'},
        {'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistar
        {'role': 'assistant', 'content': 'no'},
        {'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
    ]

    response = ollama.chat(model='llama3', messages=classify_convo)

    return response['message']['content'].strip().lower()

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
```

We can now use the **classify_embedding** function as below

assistant.py > ...

```python

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):
    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 = [
```

```python
assistant.py > ...
def classify_embedding(query, context):
    response = ollama.chat(model='llama3', messages=classify_convo)

    return response['message']['content'].strip().lower()

def recall(prompt):
    queries = create_queries(prompt=prompt)
    embeddings = retrieve_embeddings(queries=queries)
    convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
    print(f'\n{len(embeddings)} message:response embeddings added for context.')

conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)

```

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

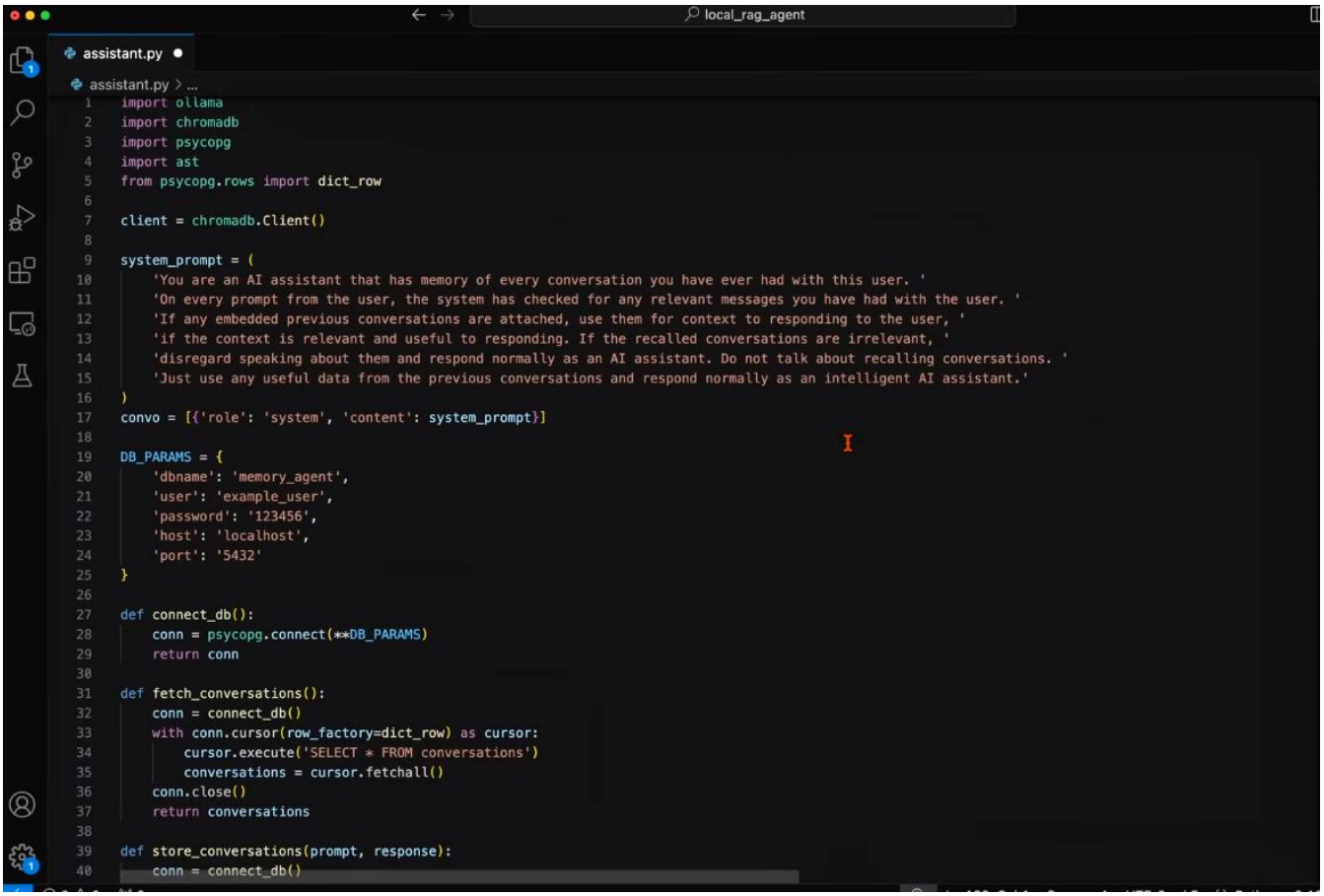
```python
assistant.py > stream_response
def store_conversations(prompt, response):
    conn.close()

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:
```

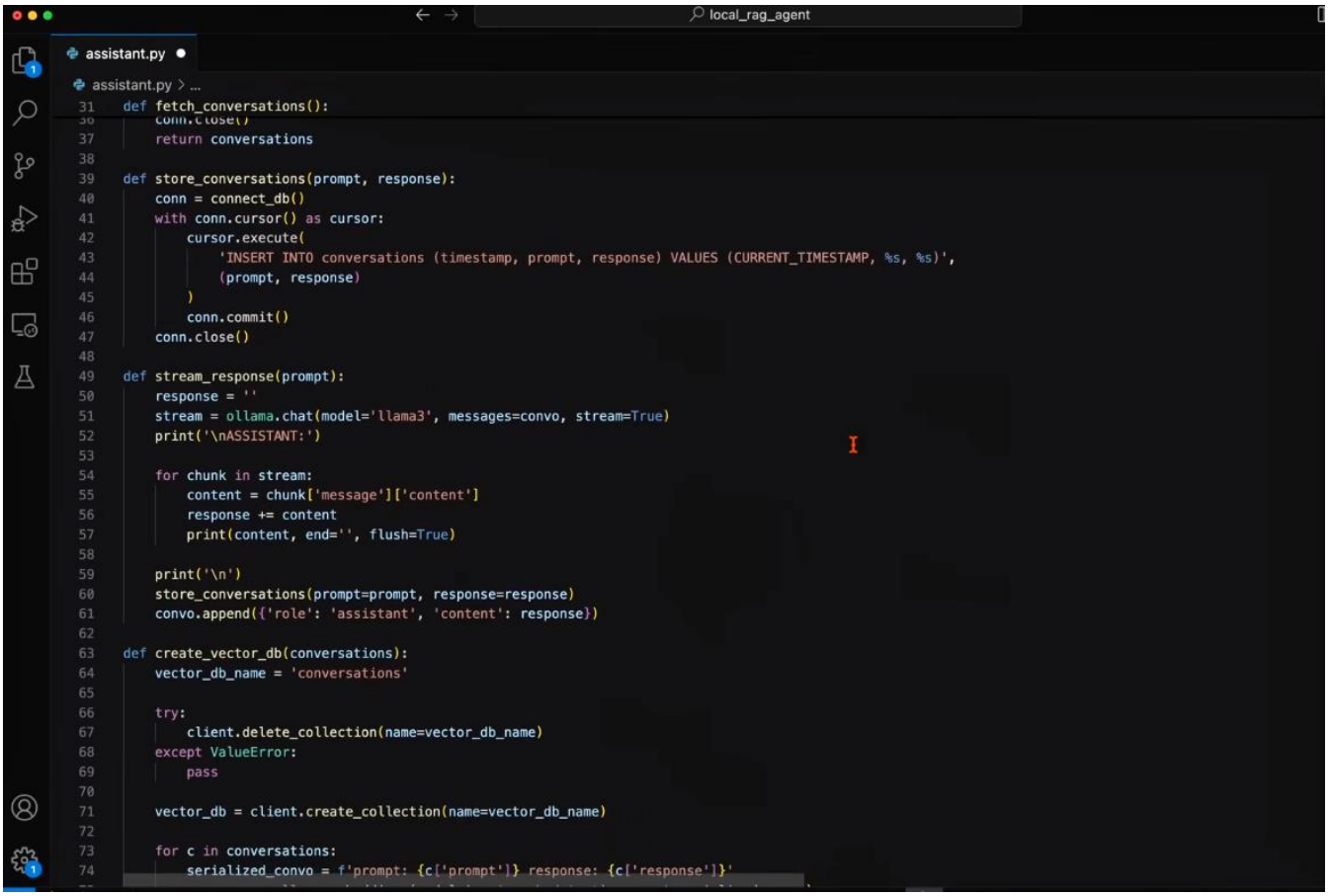
```python
assistant.py > stream_response
def store_conversations(prompt, response):
        )
        conn.commit()
    conn.close()

def stream_response(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)
```

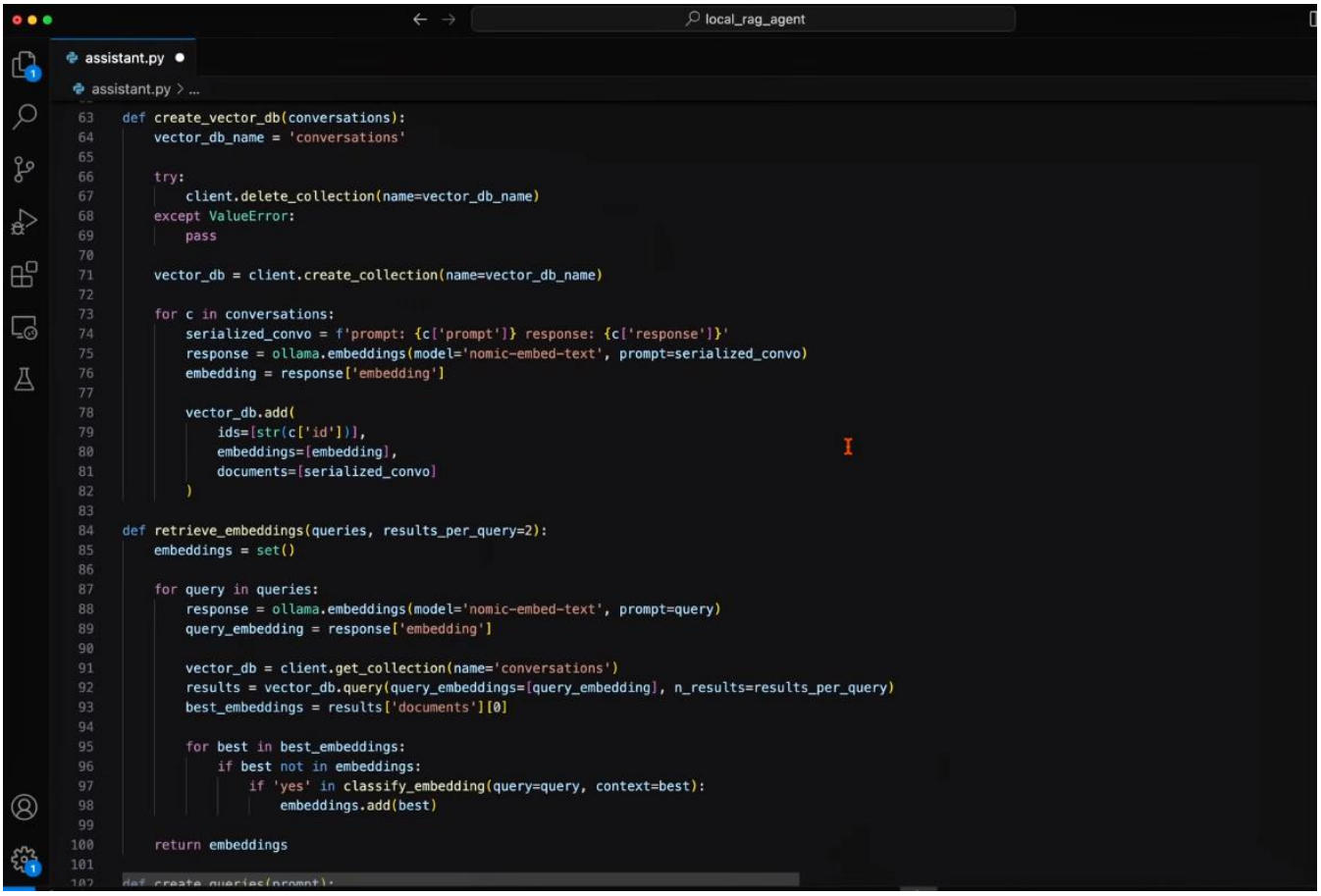
```python
assistant.py > ...
conversations = fetch_conversations()
create_vector_db(conversations=conversations)
print(fetch_conversations())

while True:
    prompt = input('USER: \n')
    context = retrieve_embeddings(prompt=prompt)
    prompt = f'USER PROMPT: {prompt} \nCONTEXT FROM EMBEDDINGS DB: {context}'
    stream_response(prompt=prompt)
```

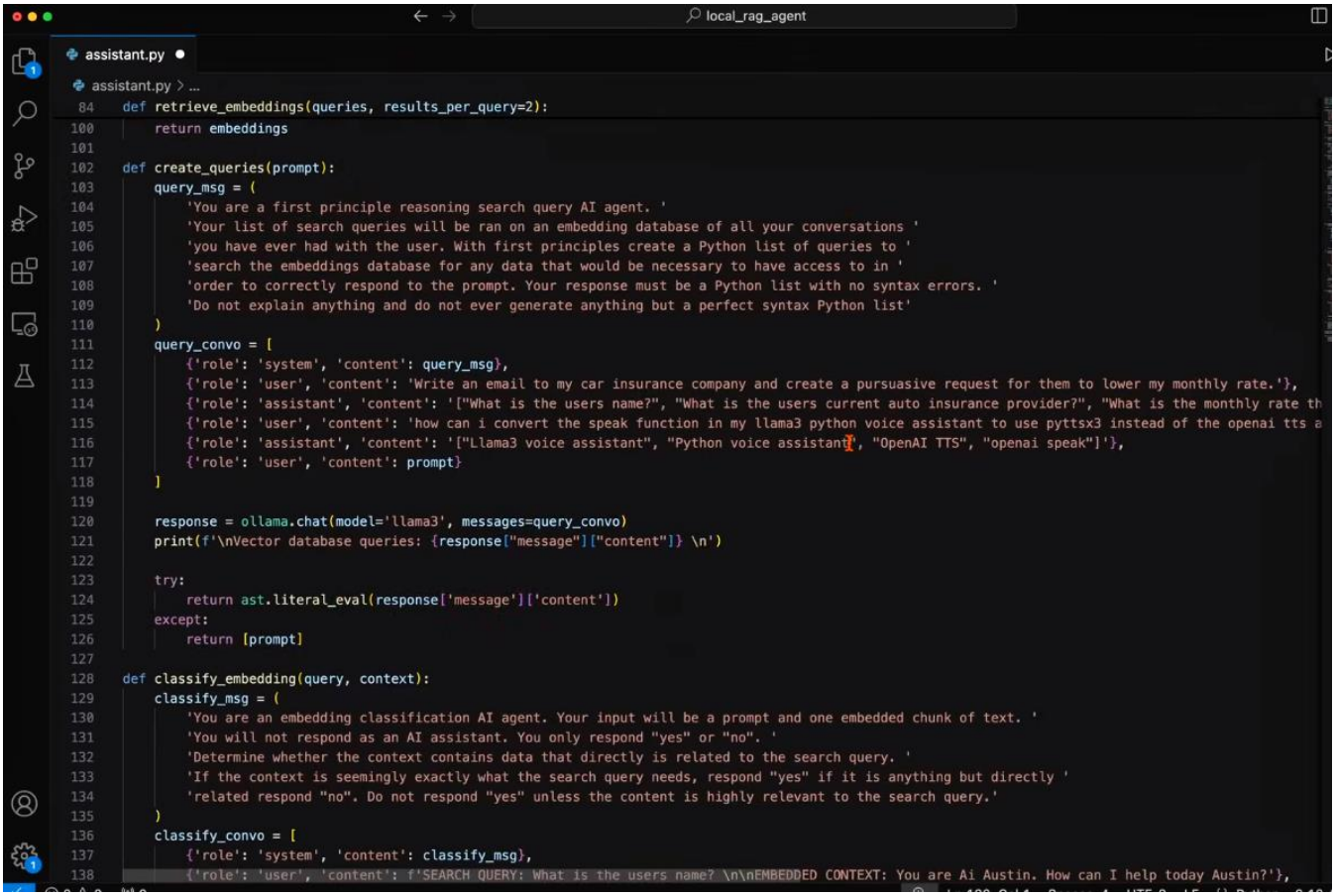
```python
assistant.py > ...
def recall(prompt):
    convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
    print(f'\n{len(embeddings)} message:response embeddings added for context.')

conversations = fetch_conversations()
create_vector_db(conversations=conversations)

while True:
    prompt = input('USER: \n')
    recall(prompt=prompt)
    stream_response(prompt=prompt)
```

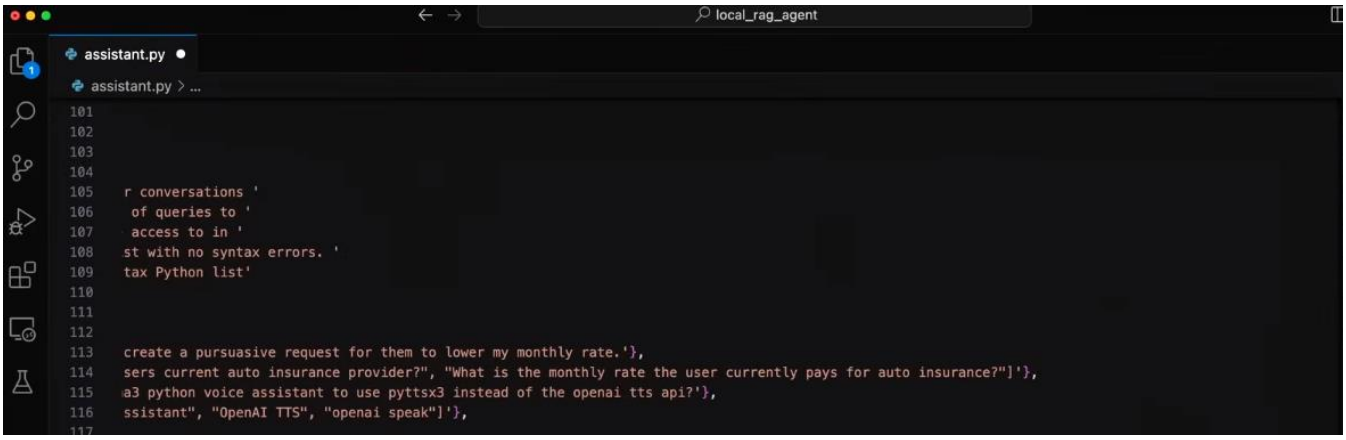
This completes our work on an AI agent that stores its conversations as embeddings in a database.

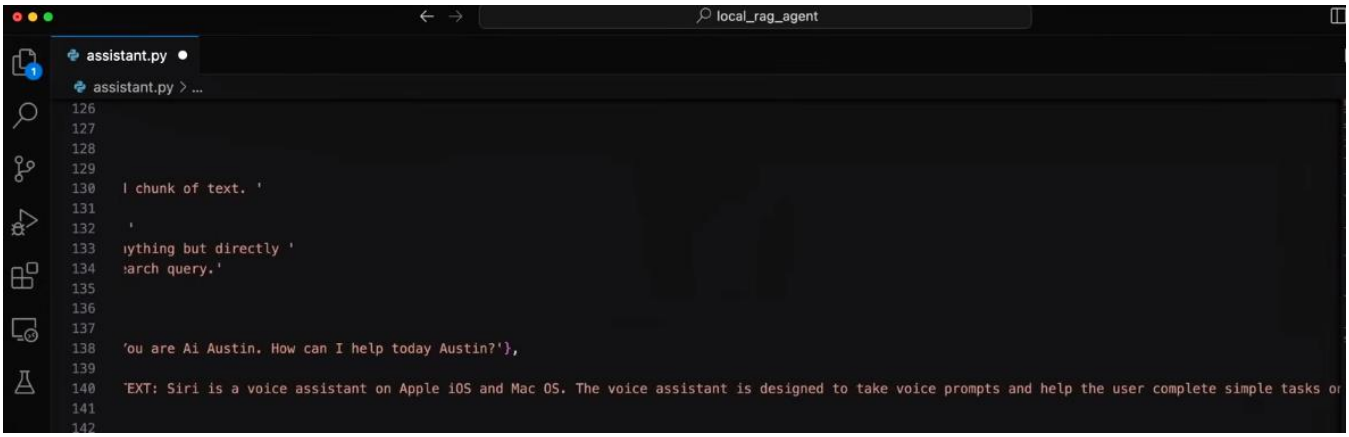

local_rag_agent
assistant.py
assistant.py > ...
import ollama
import chromadb
import psycopg
import ast
from psycopg.rows import dict_row
client = chromadb.Client()
system_prompt = (
'You are an AI assistant that has memory of every conversation you have ever had with this user. '
'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
'If any embedded previous conversations are attached, use them for context to responding to the user, '
'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
)
convo = [{'role': 'system', 'content': system_prompt}]
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()

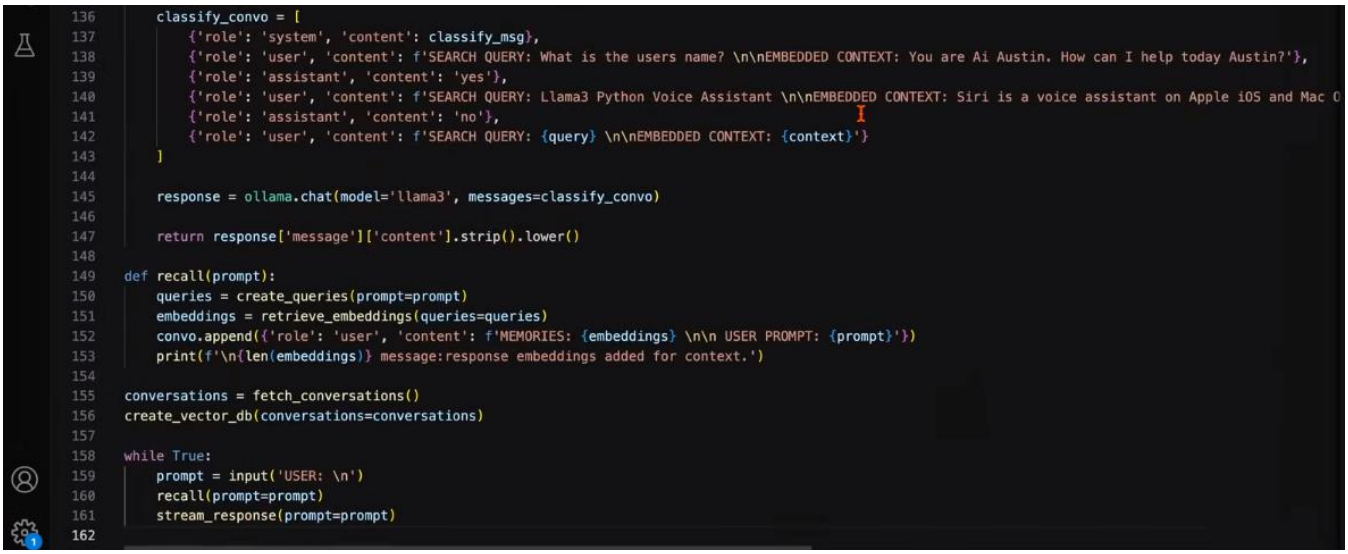

local_rag_agent
assistant.py
assistant.py > ...
def fetch_conversations():
return conversations
def store_conversations(prompt, response):
conn = connect_db()
with conn.cursor() as cursor:
cursor.execute(
'INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, %s, %s)',
(prompt, response)
)
conn.commit()
conn.close()
def stream_response(prompt):
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']}'

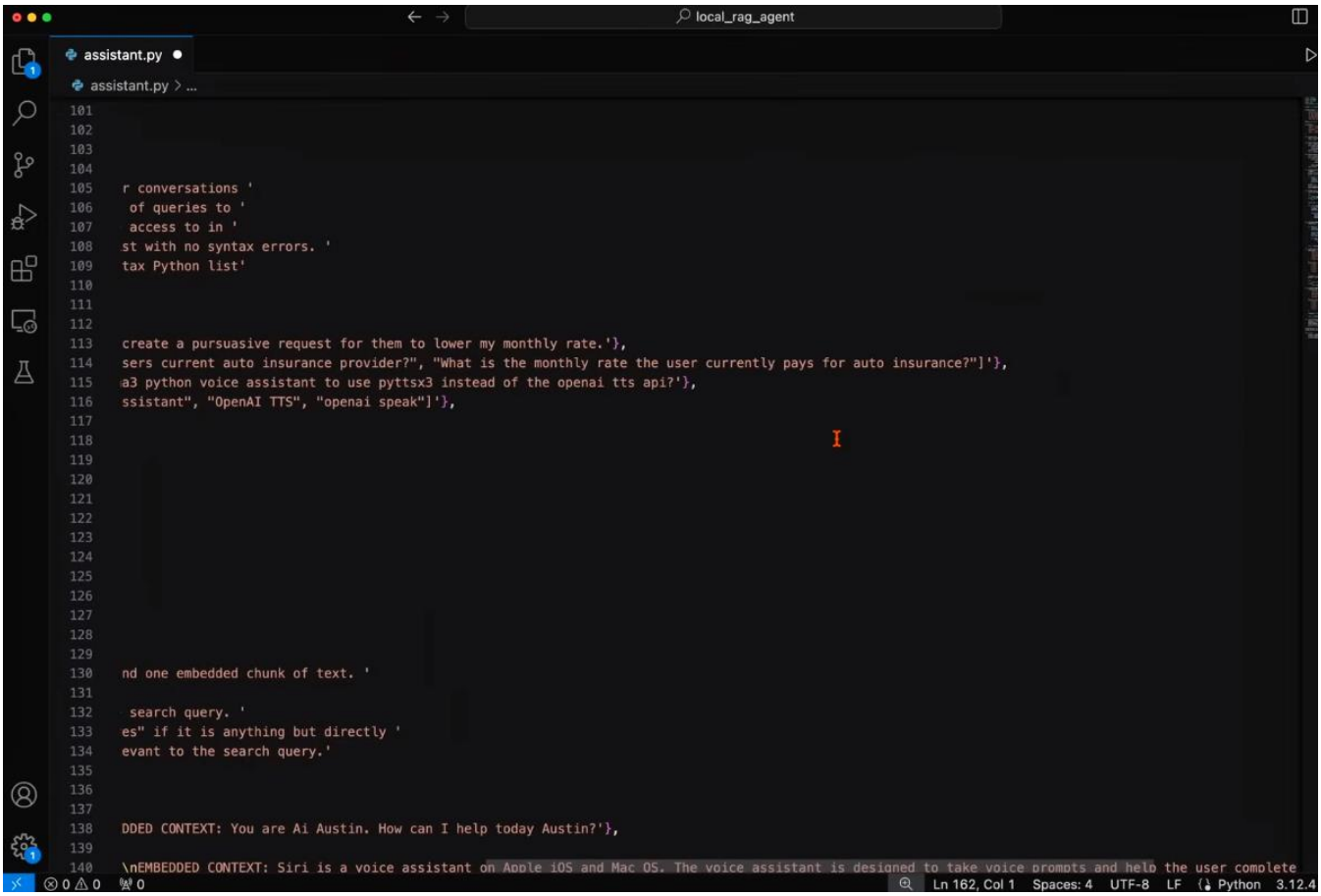

local_rag_agent
assistant.py
assistant.py > ...
def create_vector_db(conversations):
vector_db_name = 'conversations'
try:
client.delete_collection(name=vector_db_name)
except ValueError:
pass
vector_db = client.create_collection(name=vector_db_name)
for c in conversations:
serialized_convo = f'prompt: {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

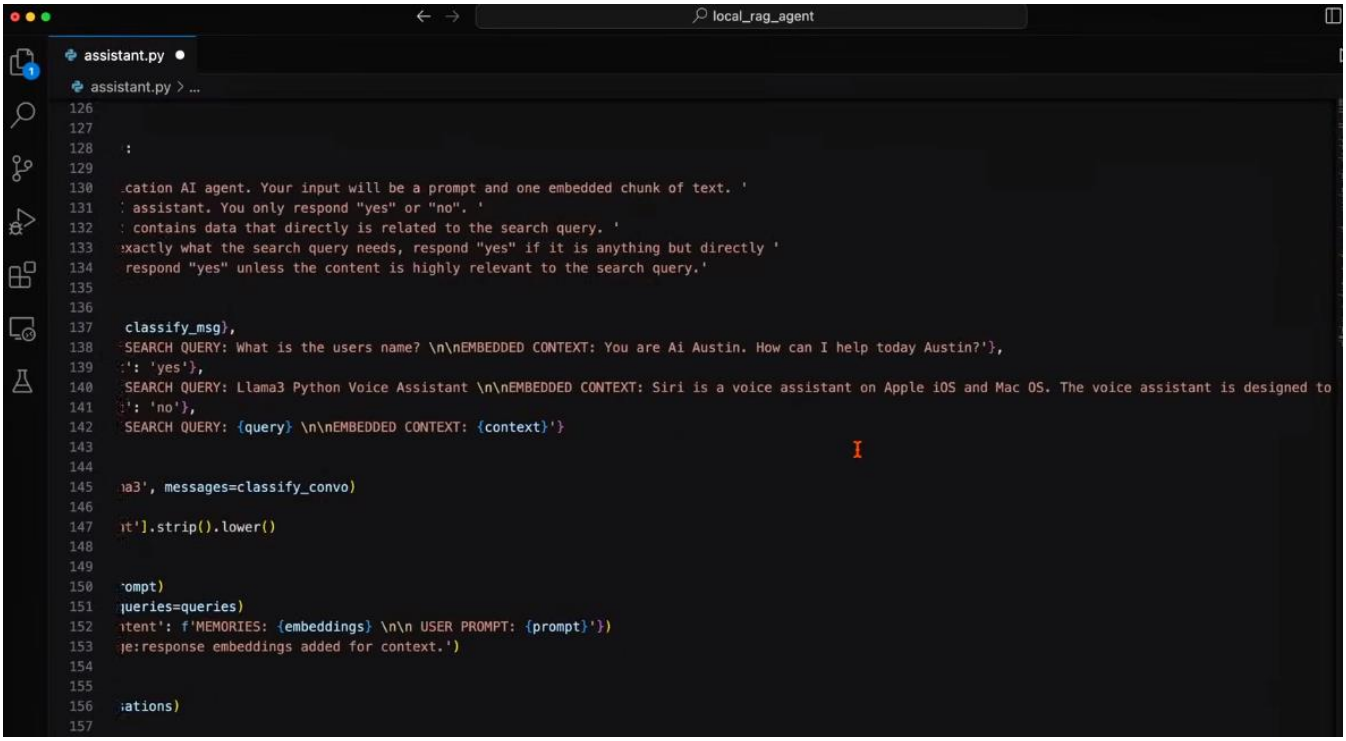

local_rag_agent
assistant.py
assistant.py > ...
def retrieve_embeddings(queries, results_per_query=2):
return embeddings
def create_queries(prompt):
query_msg = (
'You are a first principle reasoning search query AI agent. '
'Your list of search queries will be ran on an embedding database of all your 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 pursuasive 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 th
{'role': 'user', 'content': 'how can i convert the speak function in my llama3 python voice assistant to use pyttsx3 instead of the openai tts a
{'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?'},

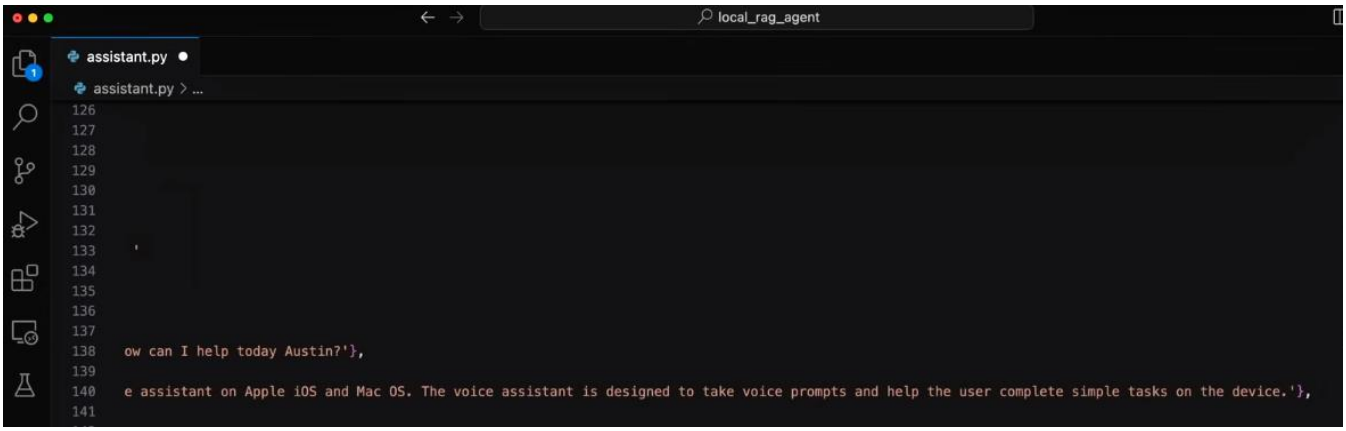

local_rag_agent
assistant.py
assistant.py > ...
r conversations '
of queries to '
access to in '
st with no syntax errors. '
tax Python list'
create a pursuasive request for them to lower my monthly rate.'},
sers current auto insurance provider?", "What is the monthly rate the user currently pays for auto insurance?"]'},
a3 python voice assistant to use pyttsx3 instead of the openai tts api?'},
ssistant", "OpenAI TTS", "openai speak"]'},

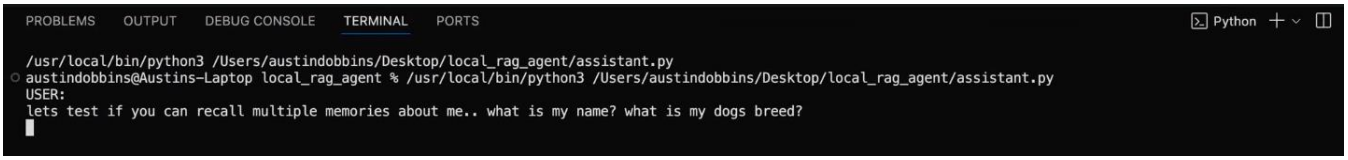

local_rag_agent
assistant.py
assistant.py > ...
l chunk of text. '
'
ything but directly '
arch query.'
ou are Ai Austin. How can I help today Austin?'},
EXT: Siri is a voice assistant on Apple iOS and Mac OS. The voice assistant is designed to take voice prompts and help the user complete simple tasks o

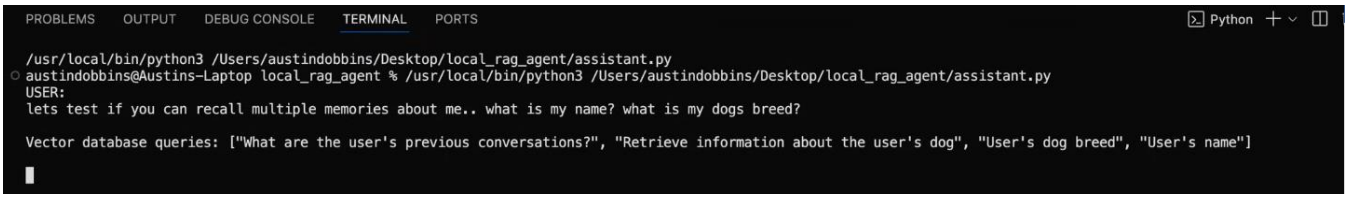

classify_convo = [
{'role': 'system', 'content': classify_msg},
{'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
{'role': 'assistant', 'content': 'yes'},
{'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac O
{'role': 'assistant', 'content': 'no'},
{'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
]
response = ollama.chat(model='llama3', messages=classify_convo)
return response['message']['content'].strip().lower()
def recall(prompt):
queries = create_queries(prompt=prompt)
embeddings = retrieve_embeddings(queries=queries)
convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
print(f'\n{len(embeddings)} message:response embeddings added for context.')
conversations = fetch_conversations()
create_vector_db(conversations=conversations)
while True:
prompt = input('USER: \n')
recall(prompt=prompt)
stream_response(prompt=prompt)


local_rag_agent
assistant.py
assistant.py > ...
r conversations '
of queries to '
access to in '
st with no syntax errors. '
tax Python list'
create a pursuasive request for them to lower my monthly rate.'},
sers current auto insurance provider?", "What is the monthly rate the user currently pays for auto insurance?"]'},
a3 python voice assistant to use pyttsx3 instead of the openai tts api?'},
ssistant", "OpenAI TTS", "openai speak"]'},
nd one embedded chunk of text. '
search query. '
es" if it is anything but directly '
evant to the search query.'
DDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
\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
Ln 162, Col 1
Spaces: 4
UTF-8
LF
Python
3.12.4


local_rag_agent
assistant.py
assistant.py > ...
:
cation AI agent. Your input will be a prompt and one embedded chunk of text. '
assistant. You only respond "yes" or "no". '
contains data that directly is related to the search query. '
xactly what the search query needs, respond "yes" if it is anything but directly '
respond "yes" unless the content is highly relevant to the search query.'
classify_msg},
SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
': 'yes'},
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
': 'no'},
SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
a3', messages=classify_convo)
t'].strip().lower()
ompt)
ueries=queries)
tent': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
e:response embeddings added for context.')
ations)







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

```python
import ollama
import chromadb
import psycopg
import ast
from tqdm import tqdm
from psycopg.rows import dict_row

client = chromadb.Client()

system_prompt = (
    'You are an AI assistant that has memory of every conversation you have ever had with this user
    'On every prompt from the user, the system has checked for any relevant messages you have had w
    'If any embedded previous conversations are attached, use them for context to responding to the
    'if the context is relevant and useful to responding. If the recalled conversations are irrelev
    'disregard speaking about them and respond normally as an AI assistant. Do not talk about recal
    'Just use any useful data from the previous conversations and respond normally as an intelligen
)
convo = [{'role': 'system', 'content': system_prompt}]

DB_PARAMS = {
    'dbname': 'memory_agent',
    'user': 'example_user',
    'password': '123456',
```

```python
        )

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

    for query in tqdm(queries, desc='Processing queries to vector database'):
        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):
    query_msg = (
        'You are a first principle reasoning search query AI agent. '
```

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

```python
conversations = fetch_conversations()
create_vector_db(conversations=conversations)

while True:
    prompt = input('USER: \n')

    if prompt[:7].lower() == '/recall':
        prompt = prompt[8:]
        recall(prompt=prompt)
        stream_response(prompt=prompt)
    else:
        convo.append({'role': 'user', 'content': prompt})
        stream_response(prompt=prompt)

    stream_response(prompt=prompt)
```

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

assistant.py > ...

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

```python
import ollama
import chromadb
import psycopg
import ast
from colorama import Fore
from tqdm import tqdm
from psycopg.rows import dict_row

client = chromadb.Client()

system_prompt = (
    'You are an AI assistant that has memory of every conversation you have ever had with this user
    'On every prompt from the user, the system has checked for any relevant messages you have had w
    'If any embedded previous conversations are attached, use them for context to responding to the
    'if the context is relevant and useful to responding. If the recalled conversations are irrelev
    'disregard speaking about them and respond normally as an AI assistant. Do not talk about recal
    'Just use any useful data from the previous conversations and respond normally as an intelligen
)
convo = [{'role': 'system', 'content': system_prompt}]

DB_PARAMS = {
    'dbname': 'memory_agent',
    'user': 'example_user',
```

assistant.py > stream_response

```python
    conn.close()

def stream_response(prompt):
    response = ''
    stream = ollama.chat(model='llama3', messages=convo, stream=True)
    print(Fore.LIGHTGREEN_EX + '\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
```

assistant.py > create_queries

```python
def create_queries(prompt):
        {'role': 'user', 'content': prompt}
    ]

    response = ollama.chat(model='llama3', messages=query_convo)
    print(Fore.YELLOW + 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
        '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 an
        'related respond "no". Do not respond "yes" unless the content is highly relevant to the se
    )
    classify_convo = [
        {'role': 'system', 'content': classify_msg},
        {'role': 'user', 'content': f'SEARCH QUERY: What is the users name? \n\nEMBEDDED CONTEXT: Y
```

assistant.py > ...

```python
conversations = fetch_conversations()
create_vector_db(conversations=conversations)

while True:
    prompt = input(Fore.WHITE + 'USER: \n')

    if prompt[:7].lower() == '/recall':
        prompt = prompt[8:]
        recall(prompt=prompt)
        stream_response(prompt=prompt)
    elif prompt[:7].lower() == '/forget':
        remove_last_conversation()
        convo = convo[:-2]
        print('\n')
    elif prompt[:9].lower() == '/memorize':
        prompt = prompt[10:]
        store_conversations(prompt=prompt, response='Memory stored.')
        print('\n')
    else:
        convo.append({'role': 'user', 'content': prompt})
```

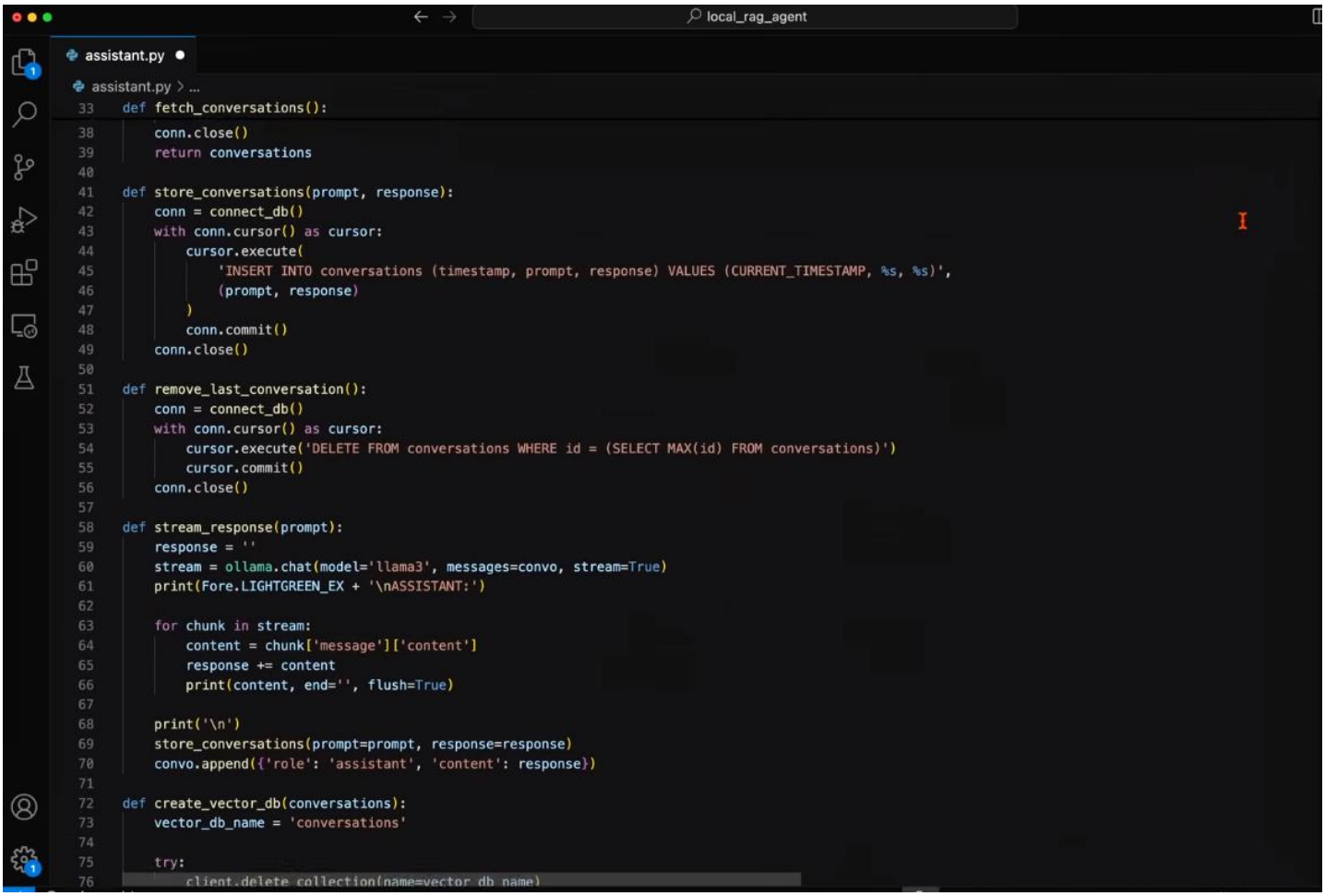
This completes the local AI agent with persistent memory

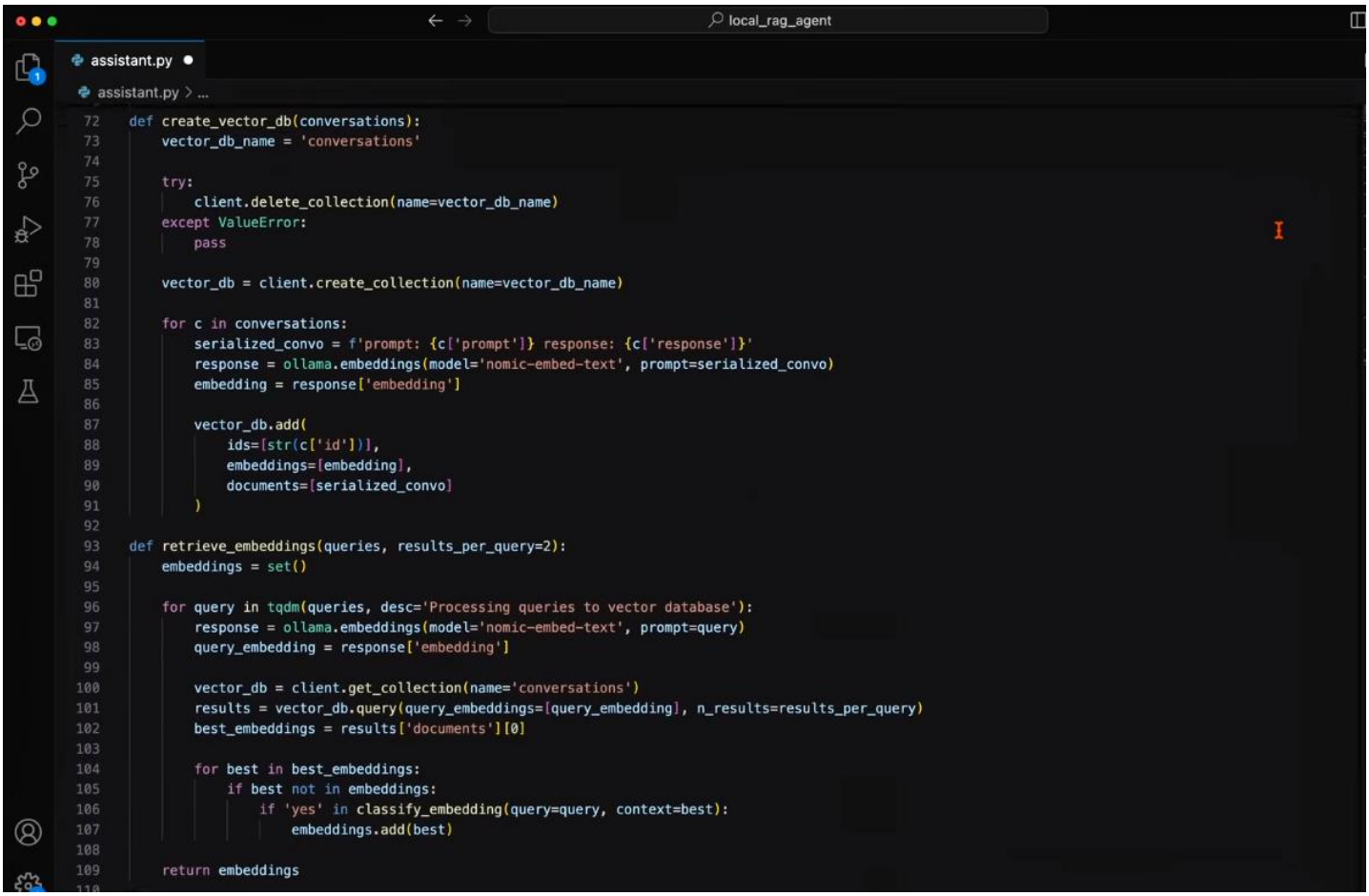
local_rag_agent

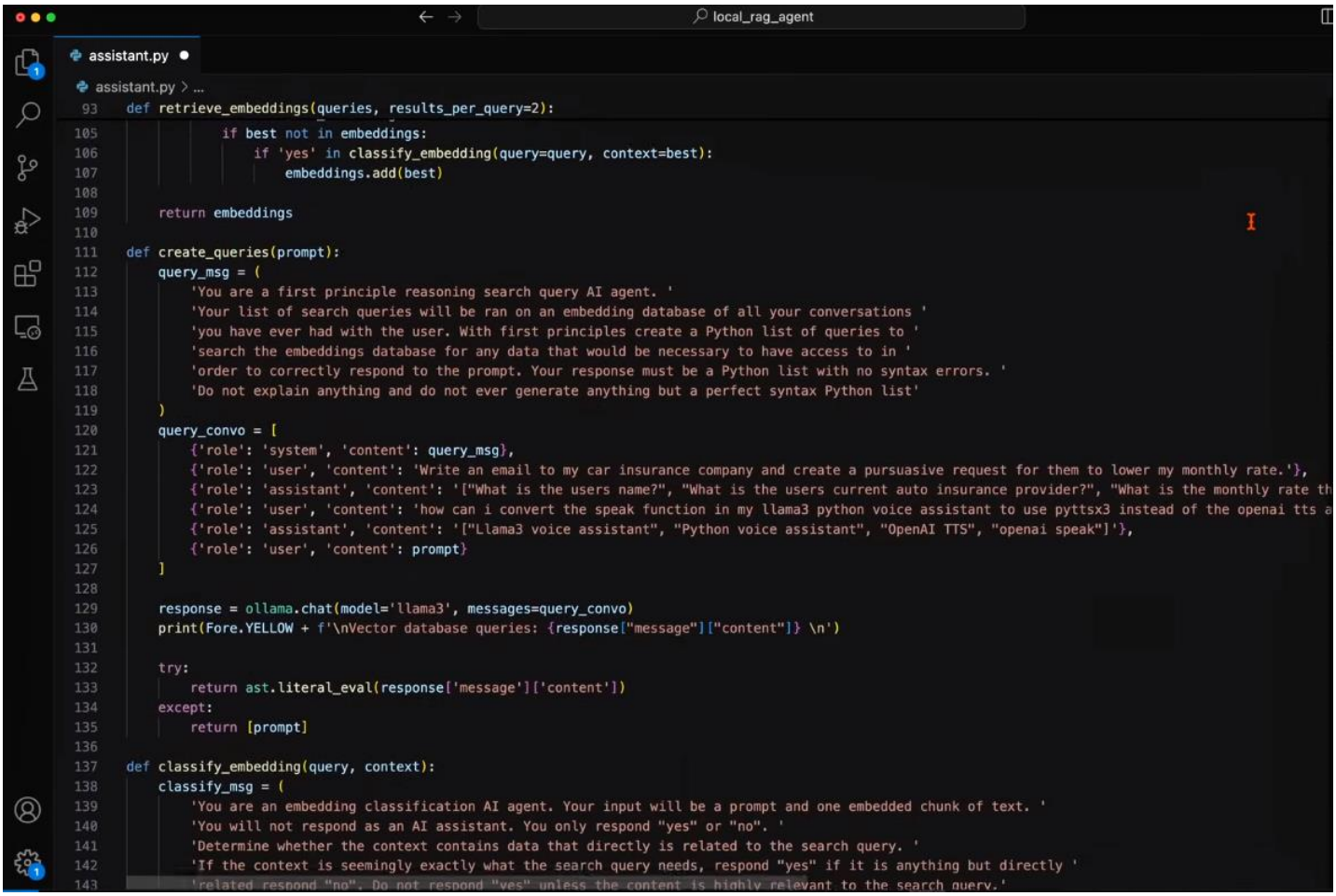
assistant.py

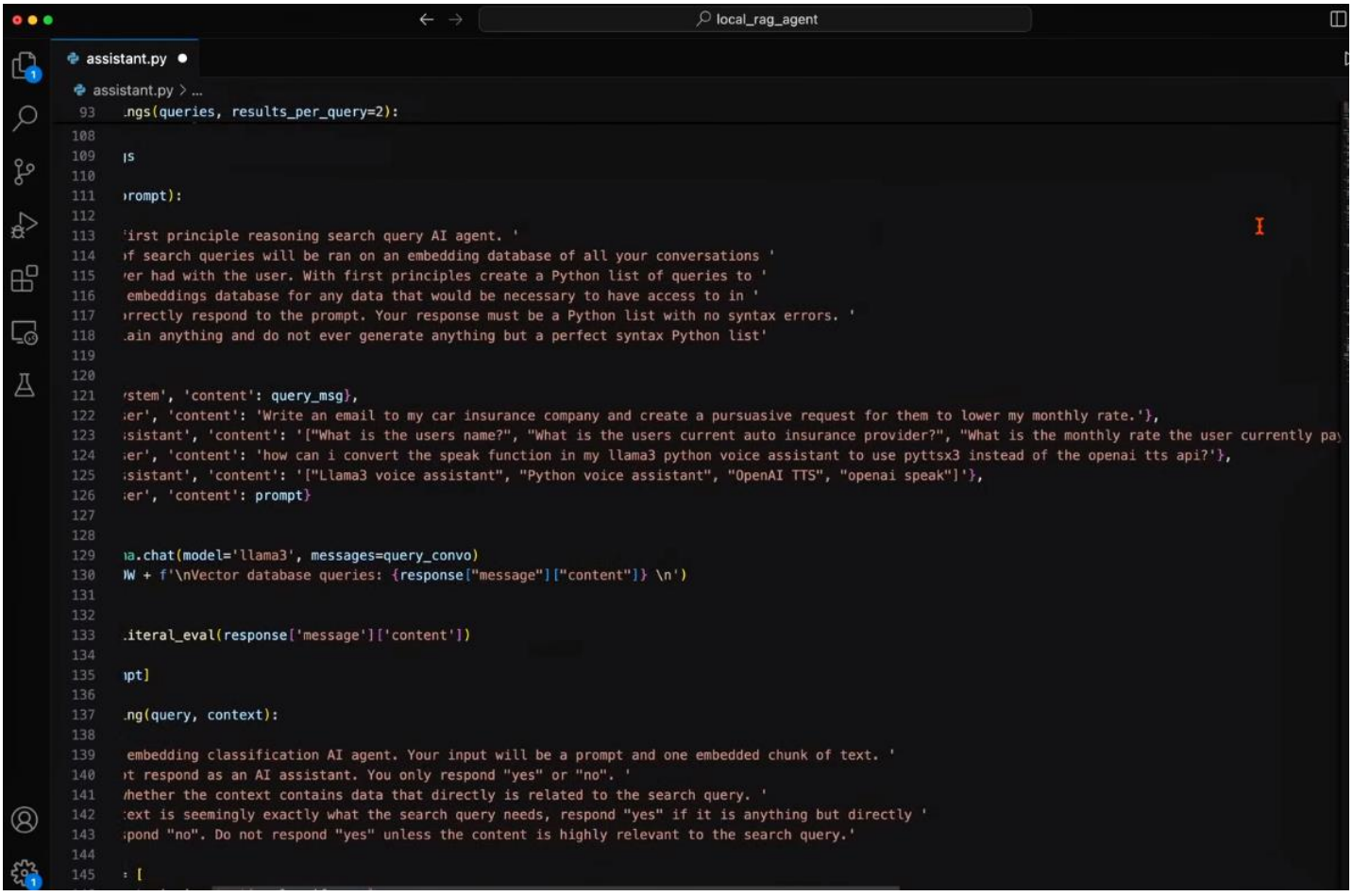
assistant.py > ...

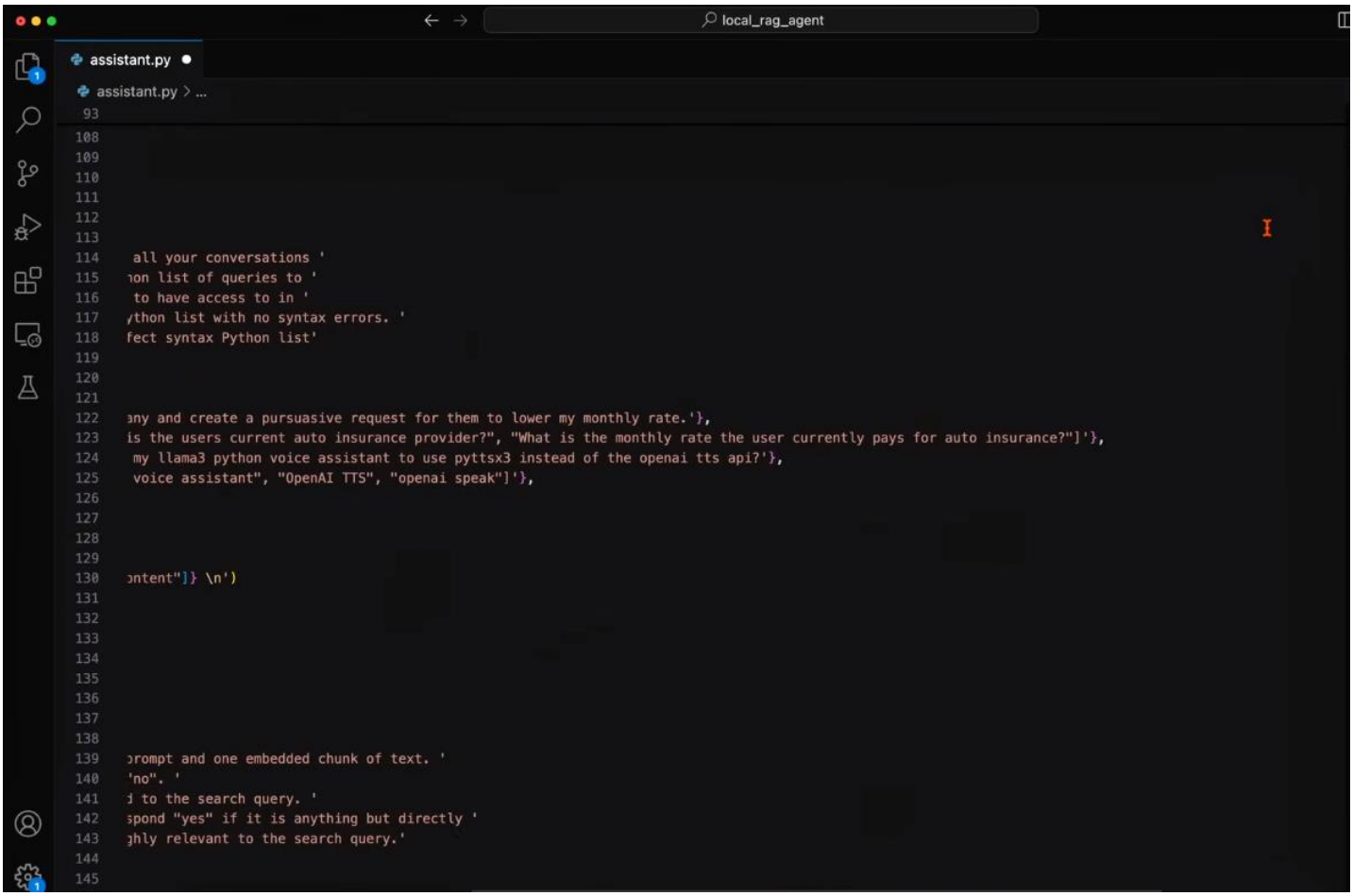
```python
import ollama
import chromadb
import psycopg
import ast
from colorama import Fore
from tqdm import tqdm
from psycopg.rows import dict_row

client = chromadb.Client()

system_prompt = (
    'You are an AI assistant that has memory of every conversation you have ever had with this user. '
    'On every prompt from the user, the system has checked for any relevant messages you have had with the user. '
    'If any embedded previous conversations are attached, use them for context to responding to the user, '
    'if the context is relevant and useful to responding. If the recalled conversations are irrelevant, '
    'disregard speaking about them and respond normally as an AI assistant. Do not talk about recalling conversations. '
    'Just use any useful data from the previous conversations and respond normally as an intelligent AI assistant.'
)
convo = [{'role': 'system', 'content': system_prompt}]

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
```

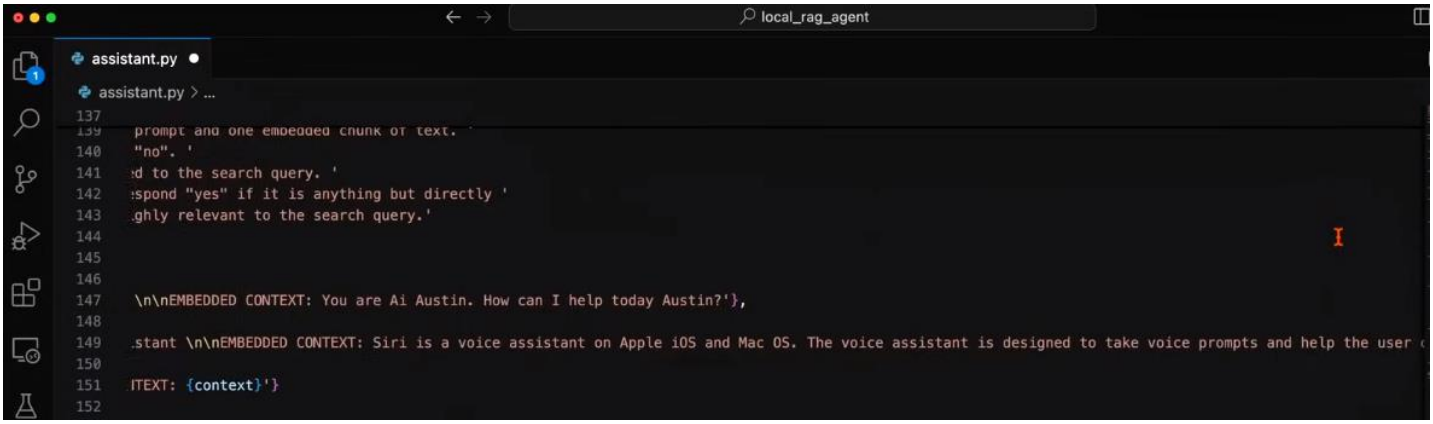

local_rag_agent
assistant.py
assistant.py > ...
def fetch_conversations():
conn.close()
return conversations
def store_conversations(prompt, response):
conn = connect_db()
with conn.cursor() as cursor:
cursor.execute(
'INSERT INTO conversations (timestamp, prompt, response) VALUES (CURRENT_TIMESTAMP, %s, %s)',
(prompt, response)
)
conn.commit()
conn.close()
def remove_last_conversation():
conn = connect_db()
with conn.cursor() as cursor:
cursor.execute('DELETE FROM conversations WHERE id = (SELECT MAX(id) FROM conversations)')
cursor.commit()
conn.close()
def stream_response(prompt):
response = ''
stream = ollama.chat(model='llama3', messages=convo, stream=True)
print(Fore.LIGHTGREEN_EX + '\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)

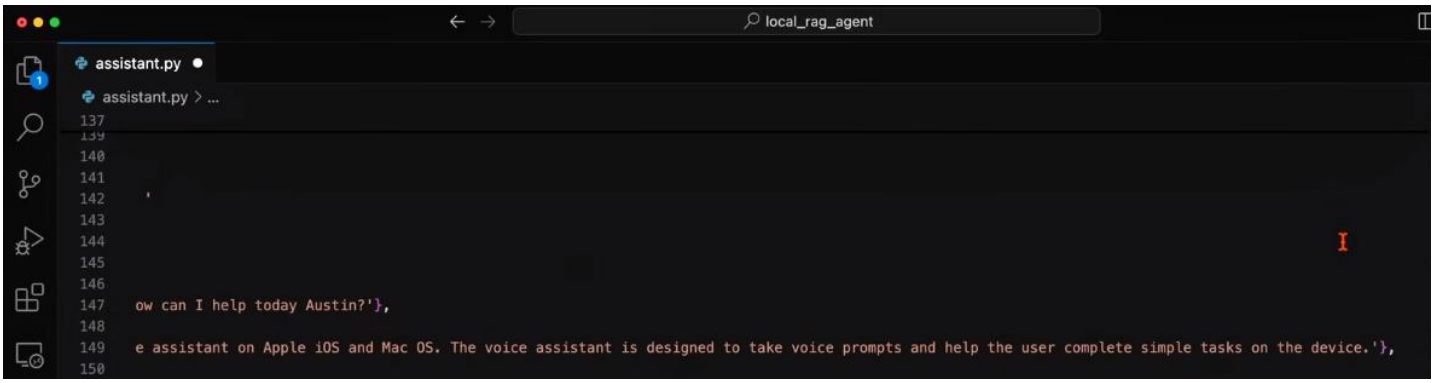

local_rag_agent
assistant.py
assistant.py > ...
def create_vector_db(conversations):
vector_db_name = 'conversations'
try:
client.delete_collection(name=vector_db_name)
except ValueError:
pass
vector_db = client.create_collection(name=vector_db_name)
for c in conversations:
serialized_convo = f'prompt: {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 tqdm(queries, desc='Processing queries to vector database'):
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

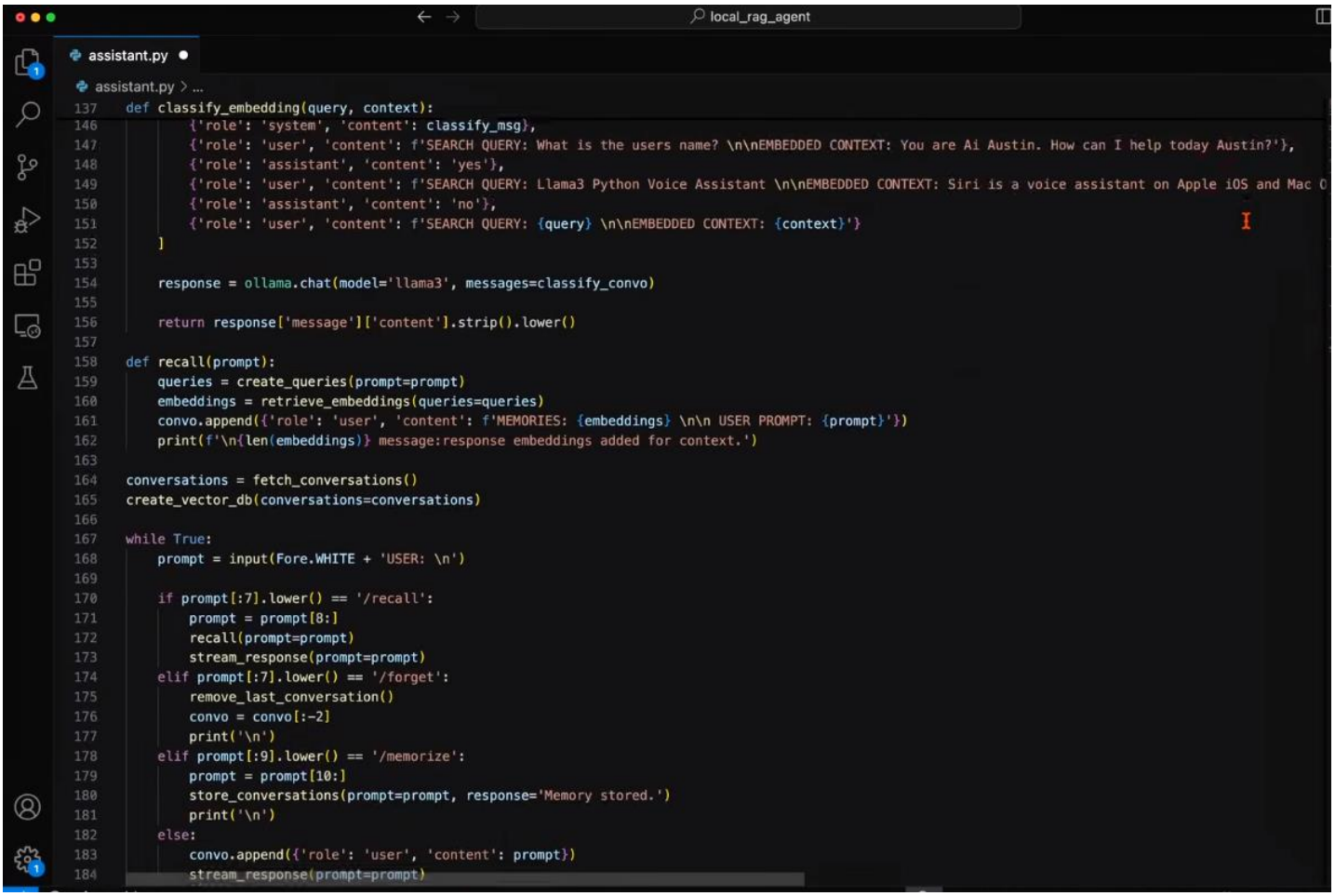

local_rag_agent
assistant.py
assistant.py > ...
93 def retrieve_embeddings(queries, results_per_query=2):
105 if best not in embeddings:
106 if 'yes' in classify_embedding(query=query, context=best):
107 embeddings.add(best)
108
109 return embeddings
110
111 def create_queries(prompt):
112 query_msg = (
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 pursuasive 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 th
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 a
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.'

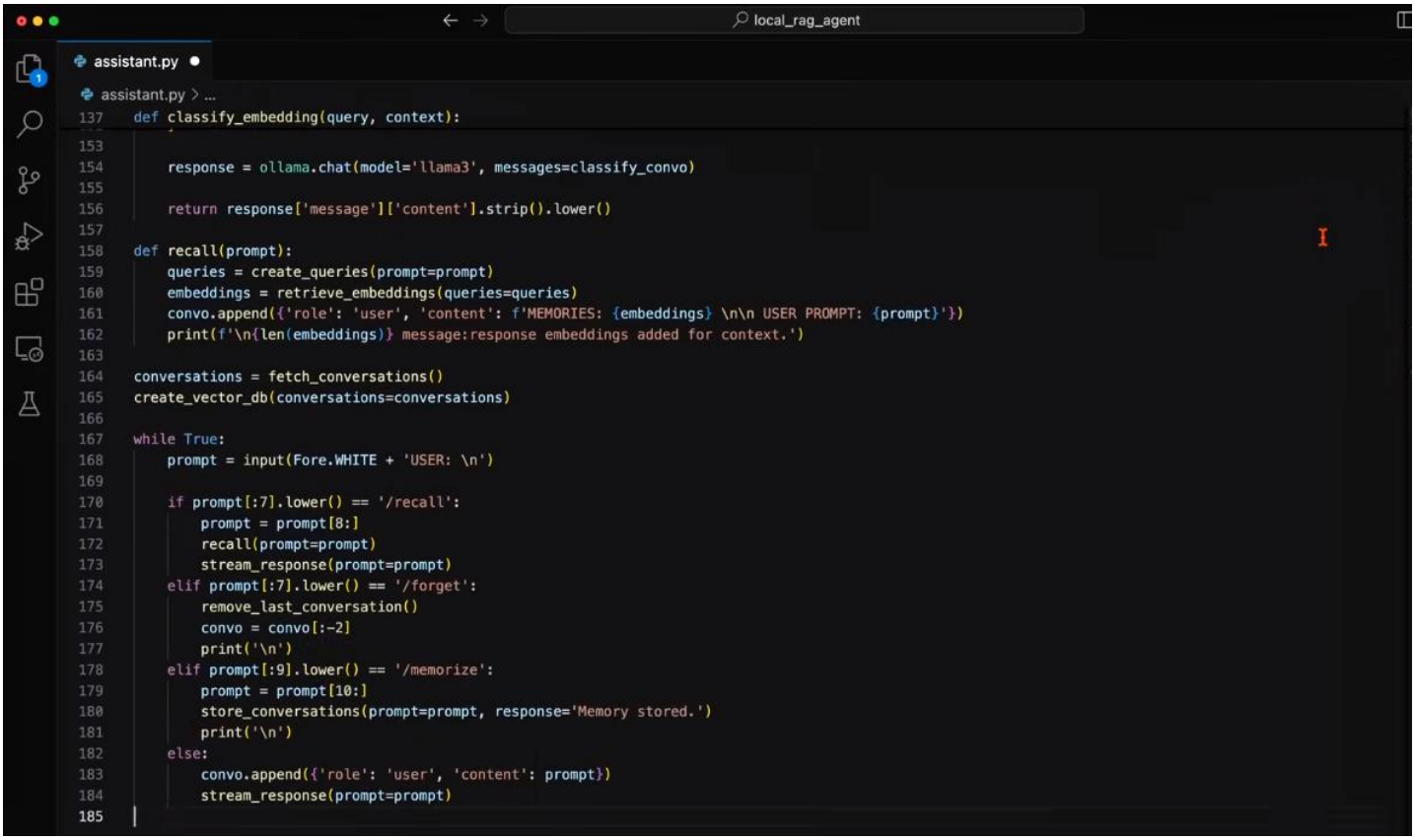

local_rag_agent
assistant.py
assistant.py > ...
93 .ngs(queries, results_per_query=2):
108
109 ıs
110
111 ›rompt):
112
113 'irst principle reasoning search query AI agent. '
114 ›f search queries will be ran on an embedding database of all your conversations '
115 'er had with the user. With first principles create a Python list of queries to '
116 embeddings database for any data that would be necessary to have access to in '
117 ›rrectly 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 pursuasive 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 pay
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 ıa.chat(model='llama3', messages=query_convo)
130 W + f'\nVector database queries: {response["message"]["content"]} \n')
131
132
133 .iteral_eval(response['message']['content'])
134
135 ıpt]
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 'hether the context contains data that directly is related to the search query. '
142 :ext 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 : [


local_rag_agent
assistant.py
assistant.py > ...
all your conversations '
ion list of queries to '
to have access to in '
ython list with no syntax errors. '
fect syntax Python list'
any and create a pursuasive request for them to lower my monthly rate.'},
is the users current auto insurance provider?", "What is the monthly rate the user currently pays for auto insurance?"]'},
my llama3 python voice assistant to use pyttsx3 instead of the openai tts api?'},
voice assistant", "OpenAI TTS", "openai speak"]'},
ontent"]} \n')
prompt and one embedded chunk of text. '
'no". '
d to the search query. '
spond "yes" if it is anything but directly '
ghly relevant to the search query.'


local_rag_agent
assistant.py
assistant.py > ...
prompt and one embedded chunk of text. '
"no". '
d to the search query. '
spond "yes" if it is anything but directly '
ghly relevant to the search query.'
\n\nEMBEDDED CONTEXT: You are Ai Austin. How can I help today Austin?'},
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
ITEXT: {context}'}


local_rag_agent
assistant.py
assistant.py > ...
'
ow can I help today Austin?'},
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.'},


local_rag_agent
assistant.py
assistant.py > ...
def classify_embedding(query, context):
{'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?'},
{'role': 'assistant', 'content': 'yes'},
{'role': 'user', 'content': f'SEARCH QUERY: Llama3 Python Voice Assistant \n\nEMBEDDED CONTEXT: Siri is a voice assistant on Apple iOS and Mac C
{'role': 'assistant', 'content': 'no'},
{'role': 'user', 'content': f'SEARCH QUERY: {query} \n\nEMBEDDED CONTEXT: {context}'}
]
response = ollama.chat(model='llama3', messages=classify_convo)
return response['message']['content'].strip().lower()
def recall(prompt):
queries = create_queries(prompt=prompt)
embeddings = retrieve_embeddings(queries=queries)
convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
print(f'\n{len(embeddings)} message:response embeddings added for context.')
conversations = fetch_conversations()
create_vector_db(conversations=conversations)
while True:
prompt = input(Fore.WHITE + 'USER: \n')
if prompt[:7].lower() == '/recall':
prompt = prompt[8:]
recall(prompt=prompt)
stream_response(prompt=prompt)
elif prompt[:7].lower() == '/forget':
remove_last_conversation()
convo = convo[:-2]
print('\n')
elif prompt[:9].lower() == '/memorize':
prompt = prompt[10:]
store_conversations(prompt=prompt, response='Memory stored.')
print('\n')
else:
convo.append({'role': 'user', 'content': prompt})
stream_response(prompt=prompt)


local_rag_agent
assistant.py
assistant.py > ...
def classify_embedding(query, context):
response = ollama.chat(model='llama3', messages=classify_convo)
return response['message']['content'].strip().lower()
def recall(prompt):
queries = create_queries(prompt=prompt)
embeddings = retrieve_embeddings(queries=queries)
convo.append({'role': 'user', 'content': f'MEMORIES: {embeddings} \n\n USER PROMPT: {prompt}'})
print(f'\n{len(embeddings)} message:response embeddings added for context.')
conversations = fetch_conversations()
create_vector_db(conversations=conversations)
while True:
prompt = input(Fore.WHITE + 'USER: \n')
if prompt[:7].lower() == '/recall':
prompt = prompt[8:]
recall(prompt=prompt)
stream_response(prompt=prompt)
elif prompt[:7].lower() == '/forget':
remove_last_conversation()
convo = convo[:-2]
print('\n')
elif prompt[:9].lower() == '/memorize':
prompt = prompt[10:]
store_conversations(prompt=prompt, response='Memory stored.')
print('\n')
else:
convo.append({'role': 'user', 'content': prompt})
stream_response(prompt=prompt)