Rasa vs FAISS for Chatbot

For an end to end project having a expense tracker whatsapp cahtbot. User can input atatements like u spend 30 rupees on cofee etc and it sh be stored in vorrect manner in databavse. User can ask indirect qs too like what kind of siff drinks i drank and bot sh answer it.
I saw 2 approaches. First using faiss
store embedding in vector database and then faiss to retrieve them efficiently.
Secondly using RASA
Your task is tot ell me Which of above 2 is best for my usecase and also if there exists better method tell me. I want to make it best

Reasoned for 4 seconds >

When building a WhatsApp expense tracker chatbot that both logs natural language expense entries and handles indirect queries (e.g., "What kind of soft drinks did I drink?"), you essentially have two key challenges:

1. **Understanding and Structuring User Inputs:** Extracting key details (amount, category, merchant, date, etc.) from varied, natural language inputs.
2. **Retrieving Relevant Data:** Answering potentially fuzzy or indirect queries by finding the relevant expense records.

Let's review the two approaches you mentioned and then look at a potentially better hybrid solution.

---

1. Using Rasa

Pros:

- **Built-in NLU:** Rasa is designed for conversational agents. It provides robust intent classification and entity extraction capabilities, so it can effectively parse statements like "I spent 30 rupees on coffee."
- **Dialogue Management:** It helps manage conversation flow and can handle follow-up questions and context.
- **Customization:** You can train Rasa to understand domain-specific language, which is useful for financial/expense entries.

Cons:

- **Structured Data Dependency:** Rasa excels at extracting structured information from user inputs, but once the data is stored, Rasa isn't inherently optimized for complex semantic retrieval (e.g., fuzzy queries that require understanding semantic similarity).
- **Indirect Query Challenges:** For questions like "What kind of soft drinks did I drink?", you'd need to pre-define a lot of intents or rules, which can be limiting if queries are phrased in unexpected ways.

---

2. Using FAISS (Vector Search)

Pros:

- **Semantic Retrieval:** FAISS is excellent for storing high-dimensional embeddings and performing fast nearest-neighbor searches. This means you can capture the semantic meaning of both your expense entries and your queries.
- **Handling Fuzzy Queries:** For indirect queries (like "soft drinks"), embedding-based similarity search can help match the query against records even if the phrasing doesn't exactly match the stored text.
- **Scalability:** It scales well when you have a large number of records.

Cons:

- **Data Extraction Not Included:** FAISS doesn't do natural language understanding. You'd still need a mechanism to extract structured expense information (amount, category, etc.) from user input before storing the data.
- **Integration Overhead:** You'd have to build or integrate an NLU layer (or use an LLM/other tool) to convert text into embeddings, and then manage the hybrid process of extraction + storage + retrieval.

A Hybrid (and Often Superior) Approach

Given your requirements, neither Rasa nor FAISS alone covers all aspects optimally. Instead, a **hybrid solution** can leverage the strengths of both:

1. **NLU for Extraction:**
    - **Use Rasa (or another NLU framework):** Employ Rasa to parse incoming messages, extracting structured information (e.g., amount, category, merchant, date). This structured data can then be stored in a conventional relational or NoSQL database.
2. **Semantic Retrieval for Indirect Queries:**
    - **Complement with FAISS (or similar vector databases):** For queries that are indirect or fuzzy, convert both the query and the stored expense records (or a summary of them) into embeddings. Use FAISS to perform similarity search and retrieve relevant records. For example, if a user asks about "soft drinks," the embeddings can help match records that were tagged or described similarly.
3. **Alternative Consideration – LLMs and Retrieval-Augmented Generation (RAG):**
    - **LLM Integration:** With advancements in large language models (LLMs), you could also consider using an LLM (via frameworks like LangChain) to interpret and process queries. In a retrieval-augmented generation (RAG) setup, the LLM can take structured queries and use a vector search backend (like FAISS) to gather context before generating a response.
    - **Benefits:** This approach can be highly flexible for natural language queries while still relying on the precision of structured data.

Recommendation

- **For Expense Logging:**
  Use a dedicated NLU system like **Rasa** to reliably extract and structure expense data. This ensures that every expense is accurately parsed and stored.

- **For Query Handling:**
  Use a **vector search mechanism (FAISS or similar)** to handle the semantic similarity aspect of indirect queries. This can be done by generating embeddings for expense entries and for incoming queries, then retrieving relevant results via nearest-neighbor search.
- **For a Future-Proof, “Best” System:**
  Consider a **hybrid approach** that combines:
  - Rasa for conversational flow and structured data extraction.
  - FAISS (or a managed vector database service like Pinecone or Weaviate) for semantic retrieval.
  - Optionally, integrate an LLM in a retrieval-augmented framework to bridge the two, which can handle nuanced, indirect queries even better.

This combination lets you cover both the structured data extraction and the semantic matching aspects of your expense tracker chatbot effectively.

Final Thoughts

- **Rasa Alone:** Great for direct interactions and structured data extraction, but not ideal for fuzzy, indirect semantic searches.
- **FAISS Alone:** Excellent for semantic retrieval but does not handle the parsing of natural language expense entries.
- **Hybrid/LLM-Enhanced Approach:** Likely the best overall solution—using Rasa for structured extraction and a vector search backend (with or without LLM integration) for handling flexible, indirect queries.

In summary, for your use case, a hybrid solution that leverages Rasa for the NLU component (expense extraction) and FAISS (or a similar vector search solution) for semantic querying is the most robust and scalable method.

Also tell me about a RAG approach i thought it's same as faiss explain in detail if it suits here

Reasoned for 5 seconds

Understanding the RAG Approach

What is RAG?

Retrieval-Augmented Generation (RAG) is a technique that combines a retrieval system with a generative language model. In a RAG system, the process typically works in two main steps:

1. **Retrieval:**
   - The system uses a retrieval component (often powered by vector search libraries like FAISS) to search through a large collection of documents or data entries.
   - Given a user query, this component finds the most relevant pieces of information (e.g., expense records or contextual documents) by comparing embeddings.
2. **Augmented Generation:**
   - The retrieved documents or records are then passed, along with the original query, to a generative model (such as an LLM like GPT-3 or GPT-4).

- The LLM uses both the query and the retrieved context to generate a detailed, coherent response.

This approach effectively marries precise retrieval (to fetch relevant data) with the flexibility of generation (to craft human-like, contextually rich answers).

How Does RAG Differ from Just Using FAISS?

- **FAISS Alone:**
  - FAISS is a library for efficient similarity search on high-dimensional embeddings. When you use FAISS by itself, you're essentially performing a nearest-neighbor search to retrieve entries that are semantically similar to a query.
  - It does not generate text or synthesize information; it simply returns matching data points from your database.
- **RAG (with FAISS as a Component):**
  - In a RAG setup, FAISS (or any similar vector search mechanism) is used as the retrieval backend to fetch relevant records.
  - The generative model then takes these retrieved records and the query to produce a nuanced answer. This means that instead of merely returning a list of similar records, the system can integrate the information to answer more complex or indirect queries.

Does RAG Suit Your Expense Tracker Chatbot Use Case?

For an expense tracker chatbot where users might log entries like "I spent 30 rupees on coffee" and later ask indirect questions such as "What kind of soft drinks did I drink?", RAG can be highly beneficial:

1. **Handling Indirect Queries:**
   - A RAG system can use FAISS to retrieve expense entries related to beverages, and then the LLM can generate a coherent answer that summarizes or aggregates the data (e.g., "You had cola and lemonade on your last visits").
   - This is particularly useful when the query is fuzzy or indirect because the generative model can interpret nuances and provide contextually relevant responses.
2. **Bridging Structured Data and Natural Language:**
   - You might first use a tool like Rasa to parse and structure the expense entries (extracting amount, category, etc.) for storage.
   - Then, when a user asks a complex query, the RAG system can combine structured data retrieval with generative capabilities to create a natural language response.
3. **Enhanced User Experience:**
   - Rather than just listing matching records, the RAG approach can provide a summary, explanations, or even suggestions based on the retrieved data.
   - For instance, if the user asks, "What kind of soft drinks did I drink?" the system could respond with, "It looks like you had Pepsi and Sprite over the past month," integrating multiple records into one coherent answer.

Implementation Considerations

- **Data Extraction and Storage:**
  Use an NLU system (like Rasa) to reliably extract expense details from user inputs and store these details in your database.

- **Vectorization and Retrieval:**
  Convert your stored expense records (or their summaries) into embeddings using a suitable model. Use FAISS (or an alternative vector database) to perform fast similarity searches when a query is made.
- **Generative Response:**
  Integrate a generative model (LLM) that takes the retrieved data and the original query to generate the final response. Frameworks like LangChain or existing RAG implementations can help set up this pipeline.
- **Scalability and Accuracy:**
  Ensure that your retrieval step is well-tuned so that the most relevant records are passed to the LLM. The quality of your response will depend on both the retrieval accuracy and the generative model's capabilities.

Summary

- **FAISS** is excellent for semantic similarity search but is limited to retrieving data.
- **RAG** combines retrieval (using FAISS or a similar tool) with the power of generative models to produce rich, contextual responses.
- **For your use case,** a RAG approach is well-suited, especially when paired with a robust NLU system like Rasa for data extraction. This combination allows you to handle both direct logging of expenses and indirect, nuanced queries, providing a better overall user experience.

By implementing a RAG system, you get the best of both worlds: accurate retrieval of expense data and the ability to generate natural, conversational responses based on that data.

or your use case, a RAG approach is well-suited, especially when paired with a robust NLU system like Rasa for data extraction. This combination allows you to handle both direct logging of expenses and indirect, nuanced queries, providing a better overall user experience.

what is used in industry companies for the same usecase. i want a industry working project as it is a task frintenrship

Reasoned for 5 seconds >

When it comes to building production-grade conversational systems—especially ones that combine structured data extraction with the ability to answer nuanced, indirect queries—industry solutions typically don't rely on a single technology. Instead, they use a **modular, hybrid architecture** that combines the strengths of multiple tools. Here's how industry leaders commonly approach such use cases:

---

1. Natural Language Understanding (NLU)

What Companies Use:

- **Rasa:** Widely adopted by startups and enterprises for building custom conversational agents. It offers robust intent classification and entity extraction.

- **Dialogflow or Microsoft LUIS:** Many companies also use these managed platforms for their ease of integration with cloud services and multi-channel support.

How It's Used:

- **Data Extraction:** When a user logs an expense (e.g., "I spent 30 rupees on coffee"), the NLU engine parses the message to extract structured information such as amount, category, merchant, and time.
- **Intent Handling:** The system determines if the input is meant to log an expense or ask a question, routing the conversation accordingly.

---

2. Data Storage and Indexing

What Companies Use:

- **Relational/NoSQL Databases:** For storing structured data (expense records) reliably.
- **Vector Databases or Enhanced Search Engines:**
  - **FAISS:** An open-source library for fast vector similarity search.
  - **Pinecone, Weaviate, or Elasticsearch (with kNN):** These managed or scalable solutions are popular in industry because they offer out-of-the-box support for semantic search and are easier to scale.

How It's Used:

- **Structured Data Storage:** Expense details are stored in a database for record-keeping and further analysis.
- **Semantic Indexing:** Alongside the structured records, companies often maintain embeddings (vector representations) of key text fields. This allows for semantic searches—critical when handling indirect or fuzzy queries like "What kind of soft drinks did I drink?"

---

3. Retrieval-Augmented Generation (RAG) for Response Generation

What Companies Use:

- **RAG Architectures:** Many innovative companies now deploy RAG setups that integrate a retrieval mechanism (often based on FAISS/Pinecone) with a generative language model (e.g., GPT-4, Anthropic's Claude, or other LLMs).
- **LLM Integrations:** Frameworks like LangChain help glue together the retrieval and generation parts of the pipeline.

How It's Used:

- **Retrieval Step:** When a user asks an indirect question, the system uses the vector search component to pull the most relevant expense entries or summaries.
- **Generation Step:** The generative model then takes these retrieved records along with the original query to craft a coherent and contextually appropriate response (for example, "Over the last month, you had Pepsi and Sprite recorded as your soft drinks.").

---

4. Integration with Messaging Platforms

What Companies Use:

- **WhatsApp Business API:** For companies building chatbots on WhatsApp, integrating with the official API is common.
- **Middleware/Orchestration Layers:** Custom backend services (often built with Node.js, Python, or Java) act as the glue between the messaging platform, NLU engine, data storage, and retrieval/generation systems.

How It's Used:

- **Seamless User Experience:** The end user interacts via WhatsApp, while the backend seamlessly processes and routes messages through the various components of the system.

Industry-Grade Example Architecture

1. **User Input:**
   - A user sends "I spent 30 rupees on coffee" via WhatsApp.
2. **NLU Processing:**
   - **Rasa** (or Dialogflow/LUIS) extracts entities like `` `amount: 30` ``, `` `category: beverage/coffee` ``, etc.
   - The data is stored in a relational/NoSQL database.
3. **Semantic Indexing:**
   - The text or a summary of the entry is converted into an embedding using a pre-trained language model.
   - This embedding is stored in a vector database (FAISS, Pinecone, or similar).
4. **Query Handling (Indirect Queries):**
   - When the user asks, "What kind of soft drinks did I drink?", the system first uses the vector database to retrieve semantically similar records.
   - These records, along with the query, are passed to an LLM in a RAG setup to generate a natural, contextual answer.
5. **Response Delivery:**
   - The final answer is sent back to the user via the WhatsApp Business API.

Real-World Industry Adoption

- **Fintech & Banking:** Many financial institutions and fintech startups use similar architectures. They often deploy Rasa (or comparable NLU engines) for conversation management, use scalable vector search systems for semantic queries, and integrate with advanced generative models to handle complex customer queries.
- **Enterprise Chatbots:** Large companies and customer service platforms combine traditional search (Elasticsearch with kNN) with RAG systems for dynamic and context-aware responses.
- **Managed Cloud Services:** Providers like Microsoft (Azure Cognitive Services), Google (Vertex AI), and AWS (Lex combined with Kendra for search) offer integrated solutions that mirror this architecture, making it easier for enterprises to adopt a modular approach.

Final Thoughts for Your Internship Project

For an industry-grade internship project, consider building a **hybrid system** that:

- **Uses Rasa (or another robust NLU platform)** for accurately extracting and structuring expense data.
- **Integrates a vector search system** (using FAISS for a prototype or managed solutions like Pinecone/Elasticsearch for scalability) to enable semantic matching.
- **Adopts a RAG framework** by combining your retrieval system with a generative model (via LangChain or a similar framework) to handle indirect queries elegantly.
- **Connects with WhatsApp** using the official WhatsApp Business API to demonstrate end-to-end functionality.

This modular, industry-inspired approach not only meets your use case requirements but also aligns with best practices seen in production systems across fintech and customer service domains. It will be a strong demonstration of both your technical skills and your ability to design scalable, robust solutions for real-world applications.

Industry-Grade Example Architecture
User Input:

A user sends "I spent 30 rupees on coffee" via WhatsApp.
NLU Processing:

Rasa (or Dialogflow/LUIS) extracts entities like amount: 30, category: beverage/coffee, etc.
The data is stored in a relational/NoSQL database.
Semantic Indexing:

The text or a summary of the entry is converted into an embedding using a pre-trained language model.
This embedding is stored in a vector database (FAISS, Pinecone, or similar).
Query Handling (Indirect Queries):

When the user asks, "What kind of soft drinks did I drink?", the system first uses the vector database to retrieve semantically similar records.
These records, along with the query, are passed to an LLM in a RAG setup to generate a natural, contextual answer.
Response Delivery:

The final answer is sent back to the user via the WhatsApp Business API.

Carefully note the above architecture and gve me a complete steps and code full proof plan of workflow to complete below project assignment. Tell me about all baby steps as im a newbie as to how i can use this suggested NLU plus RAG approach. make sure to answer what databse to use and what other things to use. I want to use free or near free stuff only so answer likewise only but make it industry working

[
WhatsApp Expense Tracker Bot

Objective
Build a WhatsApp bot that helps users track and query their daily expenses using simple chat

messages. The bot should understand natural language, automatically categorize expenses, and provide expense-related answers based on user queries.

Key Features
Send & Receive Messages on WhatsApp
Connect the bot with the WhatsApp API.
Allow users to interact with the bot through chat messages.
Understanding Natural Language Messages
The bot should understand natural language.
Identify whether the user wants to add an expense or ask a question about their expenses.
Adding Expenses
Users can send expenses in natural language, e.g., "Spent 500Rs on a latte at Starbucks." to the bot.
Storing Data
The bot should save all expense details in a database.
Querying Expenses
Users can ask about their spending in natural language to the bot, e.g.:
"How much did I spend on coffee this month?"
"Show me my biggest expenses this week."
"List all my food expenses in January."

Feel free to play around and add more useful features 🚀!
]

Reasoned for 9 seconds >

Below is a step-by-step, "baby-step" plan that you can follow to build an industry-grade WhatsApp expense tracker bot using a free/near-free stack. In this example, you'll combine:

- **Rasa** for natural language understanding (NLU) and dialogue management,
- **SQLite** as your relational database (free and lightweight),
- **Sentence Transformers + FAISS** for semantic indexing and retrieval, and
- **A generative model (via a Hugging Face transformer or an API) in a RAG setup** to "augment" responses based on retrieved data.
- **Flask + Twilio** to expose a webhook that receives and sends WhatsApp messages.

Below is a complete plan and sample code snippets so you (even as a newbie) can get an industry-style working project.

Overview of the Architecture

1. **User Input:**
   A user sends a WhatsApp message (e.g., "I spent 30 rupees on coffee").
2. **NLU Processing:**
   Rasa extracts structured data (e.g., `amount: 30`, `category: coffee`) and determines the intent (add expense vs. query).
3. **Storing Data:**
   The extracted expense data is stored in an SQLite database.
4. **Semantic Indexing:**
   The raw text (or a summary) is embedded using a Sentence Transformer model, and the

embedding is added to a FAISS index.
(A mapping is maintained to know which FAISS vector corresponds to which expense record.)

5. **Query Handling (Indirect Queries):**
   When the user asks an indirect query (e.g., "What kind of soft drinks did I drink?"), the system:
   - Converts the query into an embedding,
   - Retrieves similar records via FAISS,
   - Feeds the retrieved records along with the query into a generative model (RAG approach) to generate a contextual answer.
6. **Response Delivery:**
   The answer is sent back to the user over WhatsApp using the Twilio Business API.

Step 1: Set Up Your Development Environment

Create a virtual environment and install the required libraries.

```bash
# Create and activate a virtual environment
python -m venv env
source env/bin/activate  # On Windows: env\Scripts\activate

# Install required packages
pip install rasa==3.0.0  # or the latest stable version
pip install flask twilio sentence-transformers faiss-cpu
```

> **Note:**
> • Rasa is used for NLU and dialogue management.
> • Flask will host our webhook endpoint.
> • The Twilio Python SDK will help us send/receive WhatsApp messages.
> • Sentence-Transformers (from Hugging Face) and FAISS are used for embeddings and vector indexing.

Step 2: Initialize and Train Rasa

1. **Initialize a Rasa Project:**
   In your project folder, run:

   ```bash
   rasa init --no-prompt
   ```

   This creates a basic project structure.

2. **Define Intents & Entities:**
   Update your `data/nlu.yml` file with intents such as `add_expense` and `query_expense`. For example:

```yaml
version: "3.0"
nlu:
- intent: add_expense
  examples: |
    - I spent 500Rs on a latte at Starbucks.
    - Spent 250 on lunch.
    - I paid 1000 rupees for groceries.
- intent: query_expense
  examples: |
    - How much did I spend on coffee this month?
    - Show me my biggest expenses this week.
    - List all my food expenses in January.
```

3. **Define Entities & Slots:**
   In your `domain.yml`, add entities like `amount`, `category`, and a custom slot (if needed):

```yaml
version: "3.0"
intents:
  - add_expense
  - query_expense
entities:
  - amount
  - category
slots:
  amount:
    type: float
  category:
    type: text
```

4. **Custom Actions for Expense Logging:**
   Create a custom action to write expense data to the database. In your `actions.py`, add:

```python
# actions.py
from typing import Any, Text, Dict, List
from rasa_sdk import Action, Tracker
from rasa_sdk.executor import CollectingDispatcher
import sqlite3

class ActionAddExpense(Action):
    def name(self) -> Text:
        return "action_add_expense"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        # Extract entities (you may need to enhance extraction as per your data)
        amount = tracker.get_slot("amount")
        category = tracker.get_slot("category")
        raw_text = tracker.latest_message.get('text')

        # Save to database (assumes a database helper function exists)
        conn = sqlite3.connect("expenses.db")
        cursor = conn.cursor()
        cursor.execute(
            "INSERT INTO expenses (user_id, amount, category, raw_text) VALUES (?, ?, ?, ?)",
            ("default_user", amount, category, raw_text)
        )
        conn.commit()
        conn.close()
```

```
        dispatcher.utter_message(text="Expense recorded!")
        return []
```

5. **Update domain and stories:**
   In `domain.yml`, register the custom action:

```
yaml

actions:
  - action_add_expense
```

   Create simple stories in `data/stories.yml` to connect the user's expense message with your custom action.

6. **Run the Rasa Action Server & Train the Model:**

```
bash

rasa train
rasa run actions &
rasa shell
```

Step 3: Set Up Your SQLite Database

Create an SQLite database and a table for expenses. You can create a file named `init_db.py`:

```
python

# init_db.py
import sqlite3

conn = sqlite3.connect("expenses.db")
cursor = conn.cursor()

cursor.execute("""
CREATE TABLE IF NOT EXISTS expenses (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id TEXT,
    amount REAL,
    category TEXT,
    description TEXT,
    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP,
    raw_text TEXT
)
""")
conn.commit()
conn.close()

print("Database and table created successfully!")
```

Run it once to create your database:

```
bash

python init_db.py
```

Step 4: Build Semantic Indexing with FAISS

1. **Create a FAISS Manager Module:**
   Create a file named `faiss_index.py` that loads a Sentence Transformer model, builds (or loads) a FAISS index, and provides helper functions.

```python
# faiss_index.py
import faiss
import numpy as np
from sentence_transformers import SentenceTransformer
import sqlite3

# Load your embedding model (using a free model such as 'all-MiniLM-L6-v2')
model = SentenceTransformer('all-MiniLM-L6-v2')
embedding_dim = model.get_sentence_embedding_dimension()

# Create or load FAISS index (we use IndexFlatL2 for simplicity)
index = faiss.IndexFlatL2(embedding_dim)

# A simple mapping dictionary {vector_id: expense_id} for demo purposes.
mapping = {}

def add_expense_to_index(expense_id: int, text: str):
    """Generate embedding for the text and add it to the FAISS index."""
    embedding = model.encode([text])
    vector = np.array(embedding).astype('float32')
    current_index = index.ntotal  # get current index length
    index.add(vector)
    mapping[current_index] = expense_id

def build_index_from_db():
    """(Optional) Load all expense records from the DB and build the index."""
    conn = sqlite3.connect("expenses.db")
    cursor = conn.cursor()
    cursor.execute("SELECT id, raw_text FROM expenses")
    records = cursor.fetchall()
    conn.close()

    texts = []
    expense_ids = []
    for rec in records:
        expense_ids.append(rec[0])
        texts.append(rec[1])

    if texts:
        embeddings = model.encode(texts)
        vectors = np.array(embeddings).astype('float32')
        global index
        index = faiss.IndexFlatL2(embedding_dim)
        index.add(vectors)
        # Update mapping: assume order is preserved
        for idx, exp_id in enumerate(expense_ids):
            mapping[idx] = exp_id

def query_index(query: str, top_k: int = 3):
    """Return the top_k expense_ids that are semantically similar to the query."""
    embedding = model.encode([query])
    vector = np.array(embedding).astype('float32')
    D, I = index.search(vector, top_k)
    # Map the indices back to expense_ids
    result_ids = [mapping.get(idx) for idx in I[0]]
    return result_ids
```

2. **When Adding an Expense:**
   In your Rasa custom action (or after you write to the database), call:

```python
from faiss_index import add_expense_to_index
# After inserting into DB, get the expense_id (e.g., via cursor.lastrowid)
add_expense_to_index(expense_id, raw_text)
```

Step 5: Set Up Retrieval-Augmented Generation (RAG)

1. **Create an LLM Inference Module:**
   For a free/near-free solution, you can use a smaller Hugging Face model (e.g., GPT-2 or distilGPT2) for demonstration. In production you might use an API model like OpenAI's GPT-3. Create a file called `` `llm_inference.py` ``:

```python
# llm_inference.py
from transformers import pipeline

# For demonstration, we use distilGPT2; note that the results may be basic.
generator = pipeline('text-generation', model='distilgpt2')

def generate_response(query: str, expense_records: list):
    """
    Generate a contextual answer given the user query and a list of expense records.
    """
    # Combine the query and retrieved expense details into a prompt.
    context = "\n".join([f"- {record}" for record in expense_records])
    prompt = f"User query: {query}\nBased on the following expense records:\n{context}\nAnswer

    generated = generator(prompt, max_length=150, do_sample=True, temperature=0.7)
    return generated[0]['generated_text']
```

2. **Retrieving Expense Details:**
   Create a helper function to fetch expense text from the database by ID. For example, in a module called `` `database.py` ``:

```python
# database.py
import sqlite3

def get_expense_by_id(expense_id: int) -> str:
    conn = sqlite3.connect("expenses.db")
    cursor = conn.cursor()
    cursor.execute("SELECT raw_text FROM expenses WHERE id = ?", (expense_id,))
    record = cursor.fetchone()
    conn.close()
    return record[0] if record else ""
```

3. **Integrate RAG Workflow:**
   When handling a query intent, do the following:
   - Generate the query embedding and retrieve similar expense IDs via `` `faiss_index.query_index(query)` ``.
   - Fetch the corresponding expense texts.
   - Pass the query and texts to the generative model via `` `llm_inference.generate_response(...)` ``.
   - Return the generated answer.

Step 6: Create the Flask Webhook to Integrate with WhatsApp

You'll use Flask to create an endpoint that receives messages from Twilio and sends responses. Create a file called `app.py`:

```python
# app.py
from flask import Flask, request, Response
from twilio.twiml.messaging_response import MessagingResponse
import requests
import json
from faiss_index import query_index
from llm_inference import generate_response
from database import get_expense_by_id

app = Flask(__name__)

# A helper function to call your Rasa NLU server (if running separately)
def call_rasa_nlu(message: str):
    url = "http://localhost:5005/model/parse"  # default Rasa endpoint
    payload = {"text": message}
    response = requests.post(url, json=payload)
    return response.json()

@app.route("/webhook", methods=["POST"])
def webhook():
    # Extract incoming WhatsApp message from Twilio payload
    incoming_msg = request.values.get('Body', '').strip()
    sender = request.values.get('From', '')

    # Call Rasa to parse the message
    nlu_data = call_rasa_nlu(incoming_msg)
    intent = nlu_data.get("intent", {}).get("name")

    resp = MessagingResponse()

    if intent == "add_expense":
        # In production, Rasa would call your custom action to store the expense.
        # Here, for demonstration, we simply acknowledge the expense.
        response_text = "Expense recorded! (In a full implementation, the expense would be saved.)"
        # (The custom action you built in Rasa would handle DB insertion and FAISS indexing.)

    elif intent == "query_expense":
        # Use FAISS to retrieve expense IDs similar to the query
        expense_ids = query_index(incoming_msg, top_k=3)
        # Fetch expense texts from the database
        expense_records = [get_expense_by_id(eid) for eid in expense_ids if eid is not None]
        # Generate a response using our generative model
        response_text = generate_response(incoming_msg, expense_records)
    else:
        response_text = "I'm sorry, I didn't understand that. Please try again."

    # Build the Twilio response
    msg = resp.message(response_text)
    return Response(str(resp), mimetype="application/xml")

if __name__ == "__main__":
    app.run(port=5000, debug=True)
```

Important:
• Make sure Rasa is running (e.g., on port 5005) so that the `/model/parse` endpoint is available.
• For testing WhatsApp integration with Twilio, you can use Twilio's sandbox for WhatsApp.
• Update the webhook URL in your Twilio console to point to your publicly accessible server (you might use ngrok during development).

Step 7: Bringing It All Together

1. **Run Rasa and the Action Server:**
   In one terminal:

   ```bash
   rasa run actions
   ```

   In another terminal:

   ```bash
   rasa run --enable-api
   ```

2. **Start Your Flask Server:**

   ```bash
   python app.py
   ```

3. **Expose Your Flask Server:**
   Use ngrok (or a similar tool) to expose your local Flask server:

   ```bash
   ngrok http 5000
   ```

   Use the provided public URL as your webhook endpoint in the Twilio console.

4. **Test the Workflow:**
   - **Expense Logging:** Send a WhatsApp message like "I spent 500Rs on a latte at Starbucks."
     → Rasa should parse it and trigger the custom action (which saves the expense to SQLite and adds to FAISS).
   - **Expense Query:** Send a WhatsApp message like "How much did I spend on coffee this month?"
     → Your Flask webhook calls Rasa to get the intent, then uses FAISS to retrieve similar expenses and the generative model to produce a reply.
     → The final answer is sent back to WhatsApp via Twilio.

Summary

- **NLU & Dialogue:** Rasa (open source and free) for parsing and intent handling.
- **Database:** SQLite is used for ease of development and is free.
- **Semantic Indexing:** Sentence Transformers and FAISS to embed and index expense entries.
- **RAG Setup:** A lightweight generative model (via Hugging Face transformers) combines with retrieval results to answer indirect queries.
- **Messaging Integration:** Flask plus Twilio's WhatsApp sandbox/API for sending and receiving messages.

By following these baby steps and adapting the provided code snippets, you'll have an industry-inspired, fully working WhatsApp expense tracker bot using free/near-free tools. Happy coding, and best of luck with your internship project!