Author	Reviewer	Approver
BaoHT5		

Workshop 4: Building Chatbot RAG Systems with Vector Store, Langchain & Function Calling

Workshop Objective

Guide teams to collaboratively design and develop Retrieval-Augmented Generation (RAG) chatbot systems using **FAISS or PineCone** for fast vector search, **Langchain** for prompt and chain management, and **function calling** to extend chatbot capabilities dynamically. The chatbot should solve real-life or business problems by assisting users and reducing operational costs. Mock data generation is encouraged to simulate realistic scenarios.

Open-ended Question

Teams will build intelligent chatbots that combine external knowledge retrieval with language generation to provide accurate, context-aware responses. Choose any problem domain relevant to everyday life or company operations (e.g., customer support automation, HR assistant, IT troubleshooting bot). Use generated mock data to build and test your system.

Applications (Team Choice Examples)

- Customer Support FAQ Bot
- Employee Onboarding Assistant
- IT Helpdesk Troubleshooting Chatbot
- Sales & Product Information Assistant
- Personal Finance or Expense Management Bot

AI Tools & Technologies Usage Guidelines

Tool / Tech	Usage
FAISS	Local vector database for fast similarity search
Langchain	Building prompt chains, managing conversational flows, retrieval integration, function calling
OpenAI or Langchain Function Calling	Enhance chatbot by invoking functions for dynamic/external data responses

Tool / Tech	Usage
Mock Data Generation	Create synthetic datasets simulating realistic business/user data

Team Composition & Collaboration

- Teams of 4–5 participants collaboratively develop all aspects together.
- No fixed roles—members share ideation, coding, prompt engineering, and testing.
- Focus on end-to-end RAG chatbot pipeline: from data embedding → retrieval → generation + function calling.

Agenda

1. **Define Real-World Problem & Data Needs**
   Select a business or personal assistant use case. Identify key information needs and data types. Design mock datasets accordingly.
2. **Build Document Store with FAISS**
   Generate embeddings for mock data and set up FAISS or PineCone index for vector search.
3. **Design Retrieval-Generation Workflow with Langchain**
   Configure chains integrating retrieval and Azure OpenAI generation. Prepare prompt templates.
4. **Implement Azure OpenAI Function Calling**
   Define functions (e.g., database queries, calculations) callable from the chatbot to extend capabilities.
5. **Develop Chat Interface or Notebook Prototype**
   Create a simple frontend or Jupyter notebook demo for chatbot interaction.
6. **Test, Refine & Demo**
   Validate retrieval accuracy, generation relevance, and function calling correctness. Demo solutions and share learnings.

Deliverables by End of Workshop

- Clearly defined problem statement and mock data schema.
- FAISS, PineCone or any vector store populated with embeddings.
- Langchain chain configuration combining retrieval and generation.
- Azure OpenAI function call integration extending chatbot abilities.
- Fully functional chatbot prototype with UI or notebook.
- Test cases and conversation examples showcasing solution effectiveness.
- Team presentation with demo and lessons learned.

Example Project

Example Topic: IT Helpdesk Troubleshooting Bot

Project Brief:
Build a chatbot that helps employees diagnose common IT issues by retrieving solutions from a knowledge base and dynamically running troubleshooting functions (e.g., checking system status).

Example Features:

- Search IT FAQs and troubleshooting docs using FAISS vector search
- Use Langchain to combine retrieved docs with ChatGPT response generation
- Function calls to mock system status checks or ticket creation
- Simple chat UI or notebook interface

Example Mock Data (IT Helpdesk FAQ)

```
mock_docs = [
   {
      "page_content": "How to reset my password? Visit the password reset page and follow the emailed instructions.",
      "metadata": {"source": "FAQ - Password Reset"}
   },
   {
      "page_content": "My computer is slow. Restart it, close unused apps, and run antivirus scans.",
      "metadata": {"source": "FAQ - Performance Issues"}
   },
   {
      "page_content": "To connect to VPN, install the client from IT portal and login with your credentials.",
      "metadata": {"source": "FAQ - VPN Setup"}
   },
   {
      "page_content": "Printer not working? Ensure it’s powered on, connected, and has ink and paper.",
      "metadata": {"source": "FAQ - Printer Troubleshooting"}
   },
]
```

Example Code: Langchain + FAISS + OpenAI Function Calling

```
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import AzureOpenAIEmbeddings
from langchain.chat_models import AzureChatOpenAI
from langchain.chains import ConversationalRetrievalChain
from langchain.schema import HumanMessage, AIMessage
import openai
import json

# Mock function simulating system status check
def check_system_status(device_id: str) -> str:
    status_map = {
        "printer01": "Online and functioning normally.",
```

```
        "router23": "Offline - requires restart.",
        "server07": "Online but high CPU usage.",
    }
    return status_map.get(device_id, "Device not found.")

# Prepare mock documents
mock_docs = [
    "How to reset my password? Visit the password reset page and follow
instructions.",
    "My computer is slow. Restart, close apps, run antivirus scan.",
    "Connect to VPN by installing client from IT portal and login.",
    "Printer issues: check power, connection, ink and paper.",
]

# Step 1: Generate embeddings
embeddings = AzureOpenAIEmbeddings(
 model="text-embedding-3-large",
 # azure_endpoint="https://<your-endpoint>.openai.azure.com/", If not
provided, will read env variable AZURE_OPENAI_ENDPOINT
 # api_key=... # Can provide an API key directly. If missing read env
variable AZURE_OPENAI_API_KEY
 # openai_api_version=..., # If not provided, will read env variable
AZURE_OPENAI_API_VERSION
)

# Step 2: Create FAISS index from mock docs embeddings
vectorstore = FAISS.from_texts(mock_docs, embedding=embeddings)

# Step 3: Initialize AzureChatOpenAImodel
chat = AzureChatOpenAI(
    azure_deployment="gpt-4o-mini",
    azure_endpoint= os.getenv("AZURE_OPENAI_ENDPOINT"),  # or your deployment
    api_version="2024-07-01-preview",  # or your api version
    api_key= os.getenv("AZURE_OPENAI_API_KEY"),
    # other params...
)


# Step 4: Setup Conversational Retrieval Chain
retrieval_chain = ConversationalRetrievalChain.from_llm(
    llm=chat,
    retriever=vectorstore.as_retriever(),
    return_source_documents=True,
)

# Step 5: Define OpenAI functions metadata
functions = [
    {
        "name": "check_system_status",
        "description": "Checks device status by device ID",
        "parameters": {
            "type": "object",
            "properties": {
                "device_id": {
                    "type": "string",
                    "description": "Device unique identifier"
                }
```

```
            },
            "required": ["device_id"],
        },
    }
]

# Step 6: Conversation with function calling
def chat_with_functions(user_input, chat_history):
    messages = [{"role": "system", "content": "You are an IT helpdesk
assistant."}]
    for q, a in chat_history:
        messages.append({"role": "user", "content": q})
        messages.append({"role": "assistant", "content": a})
    messages.append({"role": "user", "content": user_input})

    response = openai.ChatCompletion.create(
        model="gpt-4o-mini",
        messages=messages,
        functions=functions,
        function_call="auto"
    )
    message = response["choices"][0]["message"]

    if message.get("function_call"):
        func_name = message["function_call"]["name"]
        args = json.loads(message["function_call"]["arguments"])
        if func_name == "check_system_status":
            result = check_system_status(args["device_id"])
            chat_history.append((user_input, result))
            return result, chat_history

    reply = message["content"]
    chat_history.append((user_input, reply))
    return reply, chat_history

# Example interactive loop
if __name__ == "__main__":
    chat_history = []
    print("Welcome to IT Helpdesk RAG Chatbot!")
    while True:
        query = input("You: ")
        if query.lower() in ("exit", "quit"):
            break

        # Retrieve relevant docs and generate answer
        rag_result = retrieval_chain({"question": query, "chat_history":
chat_history})
        print(f"RAG Answer: {rag_result['answer']}")

        # Generate answer using function calling if needed
        func_answer, chat_history = chat_with_functions(query, chat_history)
        print(f"Function Call Answer: {func_answer}\n")
```

Useful References

- **FAISS Vector Store**
  https://python.langchain.com/docs/integrations/vectorstores/faiss/
  https://docs.pinecone.io/reference/python-sdk
- **Langchain RAG Chains**
  https://python.langchain.com/docs/tutorials/rag/
  https://learn.deeplearning.ai/courses/langchain/lesson/mv7m1/question-and-answer
  https://python.langchain.com/docs/versions/migrating_chains/retrieval_qa/
  https://python.langchain.com/docs/versions/migrating_chains/conversation_retrieval_chain/
- **Langchain Chat Models Setup**
  https://python.langchain.com/docs/integrations/chat/openai/
  https://python.langchain.com/docs/integrations/chat/azure_chat_openai/