Problem Statement 1

1. **API Data Retrieval and Storage**

   In this task, I worked with an external REST API that provides book information in JSON format, including details such as title, author, and publication year. I fetched the data using Python, parsed the JSON response, and stored the relevant fields in a **local SQLite database**. A database table was created to properly organize the book records. After storing the data, I retrieved it from the database and displayed it to verify that the information was saved correctly.
   This task demonstrates my understanding of API integration, JSON data handling, database creation, and persistent data storage.

   Code Link:- API Data Retrieval and Storage

2. **Data Processing and Visualization**

   For this task, I fetched a dataset containing students' test scores from an API. After retrieving the data, I extracted individual student scores and calculated the average score to analyze overall performance. To make the data easier to understand, I created a bar chart that visually compares the scores of different students. The chart includes proper labels and titles for clarity.

   This task highlights my ability to process raw data, perform basic statistical analysis, and present insights visually using Python.

   Code Link:- Data Processing and Visualization

3. **CSV Data Import to a Database**

   In this task, I worked with a CSV file containing user information such as names and email addresses. I read the CSV file row by row using Python and inserted each record into a **SQLite database**. A database table was designed to match the structure of the CSV file. After inserting the data, I verified the database entries to ensure all records were stored correctly.

   This task reflects real-world data migration scenarios where CSV files are commonly used to transfer data into databases.

   Code Link:- CSV Data Import to a Database

4. The most complex Python code I have written is for a **multipurpose Crop Recommendation System**. The project includes a machine learning model that suggests the most suitable crops for a given location based on environmental conditions. It also features a web application that fetches real-time weather data such as rainfall and humidity using the **OpenWeather API**, and incorporates **companion crop recommendations** to improve yield and sustainability.

   GitHub Link:- Crop Recommendation System

5. The most complex database code I have written is part of **PodClipper**, an AI-driven platform for converting podcasts into viral short-form videos. In this project, I designed and worked with a **PostgreSQL** database to handle user authentication, credit-based usage (via Stripe), job tracking, and metadata for generated clips. The backend integrates database operations with a serverless FastAPI stack, AWS S3 for media storage, and Inngest for background workflows, ensuring scalability and reliability.

   Github Link:- PodClipper

Problem Statement 2

1. I would rate myself as **B** in **Machine Learning, AI, Deep Learning and LLM**. I have a strong understanding of core concepts and have implemented real-world projects such as CNN-based crop health detection, ML-based recommendation systems, and LLM-powered applications. I can code and build solutions independently for moderate problems and work effectively under supervision for more advanced or large-scale systems.

2. **Key Architectural Components of an LLM-Based Chatbot**
   Building a real, production-level LLM chatbot is more than just calling an API. To make sure the chatbot gives accurate answers, remembers past conversations, and stays safe, it needs a modular setup with multiple layers working together.

1. User Interface (UI):- This is where users interact with the chatbot. It could be a web app, mobile app, or even a chat platform like Slack or Discord. Its job is to take user input and show responses in real time.

2. Orchestration Layer (Main Logic):- This is the brain of the system. It controls the flow of the chatbot—deciding when to call the LLM, when to search documents, and how to structure the final response. Tools like LangChain or LlamaIndex are often used here.

3. Large Language Model (LLM):- This is the core model (such as GPT or Llama) that understands the question and generates the answer. It follows predefined prompts so it behaves in the right way.

4. Retrieval-Augmented Generation (RAG) & Vector Database:- LLMs don't always know up-to-date or domain-specific information. To fix this, the chatbot can search its own documents before answering. These documents are stored as embeddings in a vector database, which allows searching by meaning instead of keywords.

5. Memory & Session Management:- Since LLMs don't remember past messages on their own, this layer stores conversation history. Important parts of previous chats are added back into the prompt so the chatbot can respond with proper context.

6. Safety & Guardrails:- This layer ensures the chatbot doesn't produce harmful, irrelevant, or sensitive content. It also helps keep responses on topic and safe for users.

High-Level Flow of a Chatbot Response

1. The user sends a question.
2. The system searches relevant documents if needed.
3. Past conversation and instructions are combined with the user's question.
4. This combined context is sent to the LLM.
5. The response is checked for safety and formatting.
6. The final answer is shown to the user.

This structure keeps the chatbot accurate, reliable, and scalable, making it suitable for real-world use.

3. Vector Databases

A vector database is used to store data in a way that lets you search by meaning, not just exact words. Text, images, or audio are first converted into numbers called vectors. When a user asks a question, that question is also converted into a vector, and the

database finds the most similar ones. This makes it very useful for AI applications like chatbots, recommendations, and semantic search.

Hypothetical Problem & Database Choice

Suppose I am building a chatbot that answers questions using internal company documents like PDFs, policies, and FAQs.

For this problem, I would choose **Pinecone** as the vector database.

Why Pinecone:

- It is fast and works well with large amounts of data
- It scales easily without much setup
- It integrates smoothly with LLM-based applications
- It is reliable for production use

Using Pinecone, the chatbot can quickly find the most relevant documents and use them to generate accurate answers, instead of guessing.