Problem Statement 1:

1. API Data Retrieval and Storage:

Assumption:

I am assuming that the API returns JSON in this format:

```
[
        {
                "title" : "Book Title",
                "author": "Author Name",
                "publication year": 2021
        }
]
```

The program will:
Fetch data from REST API -> store data in local SQLite Database -> Retrieve and display the stored data.

Program:

```
import requests             # For API calls
import sqlite3              # Local SQLite database

API_URL = "https://www.exampleapi.com/api/books_list/"        # Example API
DB_NAME = "book_list.db"                    # Database name

# - - - 1. Fetch Data from API - - -

def fetch_books():
        response = requests.get(API_URL)
        response.raise_for_status()         # Raises error for failed request
        return response.json()

# - - - 2. Initialize SQLite Database - - -

def init_db():
        conn = sqlite3.connect(DB_NAME)
        cursor = conn.cursor()              # Control structure to interact with db
        cursor.execute("""
                CREATE TABLE IF NOT EXISTS books (
                        id INTEGER PRIMARY KEY AUTOINCREMENT,
                        title TEXT NOT NULL,
                        author TEXT NOT NULL,
                        year INTEGER
                )
                        “””)
        conn.commit()           # Permanently saves changes
        conn.close()            # Terminate database connection

# - - - Store Data in DB - - -

def store_books(books):
        conn = sqlite3.connect(DB_NAME)
        cursor = conn.cursor()

        for book in books:
                cursor.execute(“””
                        INSERT INTO books (title, author, year)
                        VALUES (?, ?, ?)”””,
                        (book[“title”], book[“author”], book[“year”])
                        )

        conn.commit()
        conn.close()

# - - - 3. Retrieve and Display Data - - -

def display_books():
        conn = sqlite3.connect(DB_NAME)
        cursor = conn.cursor()

        cursor.execute(“ SELECT title, author, year FROM books”)
        rows = cursor.fetchall()                # Retrieve all rows from db

        print(“\n Stored Books:”)
        for title, author, year in rows:
                print(“f - {title} by {author} ({year})”)

        conn.close()            # Not using commit as no changes made to database

if __name__ == “__main__”:
        init_db()
        books = fetch_books()
```

```
store_books(books)
display_books()
```

Functions used:

1. **fetch_books()**:- Gets data from API using “requests” library, raises status code error, returns in JSON format.
2. **init_db()**:-
    a. Initializes database “books.db”.
    b. “.connect()” creates a connection to the database, “.cursor() helps us to interact with the database.
    c. SQL AUTOINCREMENT is a feature used to automatically generate unique, sequential values for a column in a database table, ideal for creating primary keys, like “Id” in this case.
    d. “.execute() executes the command. Triple quotes are used as there are multiple lines.
    e. “.commit()” saves changes permanently and “.close()” terminates the connection.
3. **store_books(books):-**
    a. Inserts book “title”, “author” and “publication year” into the table.
    b. Code and data is separated to avoid SQL injection.
4. **display_books():-**
    a. Displays all rows using fetchall() and “for loop”.

2. Data Processing and Visualization:

Assumption:

I am assuming that the API returns JSON in this format:

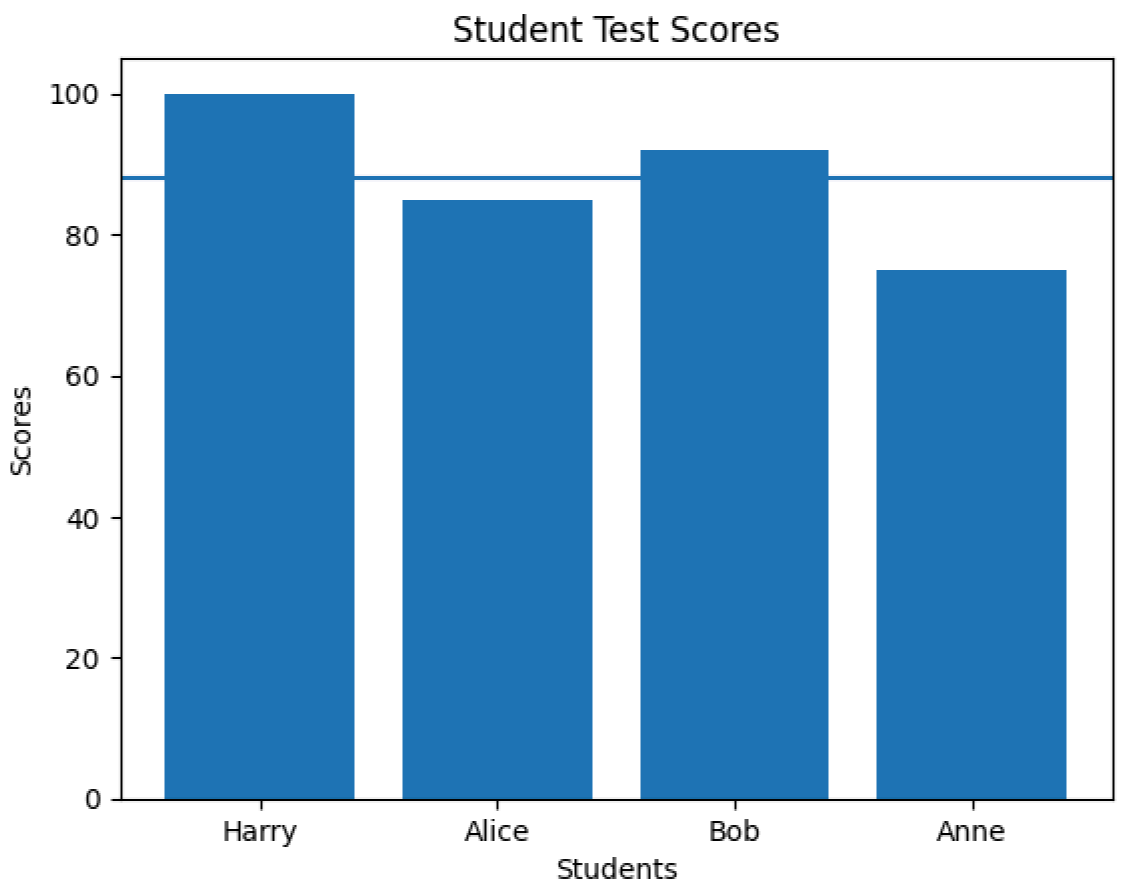
```
[
        {“name”: “Harry”, “score”: 100},
        {“name”: “Alice”, “score”: 85},
        {“name”: “Bob”, “score”: 92},
        {“name”: “Anne”, “score”: 75}
]
```

The program will:
Fetch data from API -> Calculate average score -> Create bar chart to visualize data.

Program:

```
import requests
import pandas as pd
import matplotlib.pyplot as plt


API_URL = "https://example.com/api/students_list"

# - - - Fetch data from API - - -

def fetch_data():
    response = requests.get(API_URL)
    response.raise_for_status()
    return response.json()


# - - - Main Logic - - -

def main():
    # Example API response (We shall use fetch_data() in real case)
    data = [
        {"name": "Harry", "score": 100},
        {"name": "Alice", "score": 85},
        {"name": "Bob", "score": 92},
        {"name": "Anne", "score": 75}
    ]

    # Convert JSON to DataFrame
    df = pd.DataFrame(data)

    # Calculate average score
    average_score = df["score"].mean()
    print(f"Average Score: {average_score:.2f}")            # Average Score: 88.00

    # Create bar chart
    plt.figure()
    plt.bar(df["name"], df["score"])
    plt.axhline(average_score)                # Avg. score reference line
    plt.xlabel("Students")
    plt.ylabel("Scores")
    plt.title("Student Test Scores")
    plt.show()

if __name__ == "__main__":
    main()
```

Plot:



Function:

1. Since we do not have an actual API, I created a sample JSON output ‘data’ and converted it into a dataframe ‘df’ using Pandas.
2. I calculated the average score using the '.mean()’ function.
3. I used matplotlib.pyplot to create the above plot in a Google Colab notebook.

3. CSV Data Import to a Database:

Assumption:

I am assuming that the CSV file has the columns ‘name’ and ‘email’.

Program:

```
import csv
import sqlite3

# - - - Configuration - - -
CSV_FILE = "users.csv"
DB_FILE = "users.db"

# - - - Initialize database - - -
def init_db():
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()

    cursor.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            email TEXT NOT NULL
        )
    """)

    conn.commit()
    conn.close()


# - - - Insert CSV data into database - - -

def insert_users_from_csv():
    conn = sqlite3.connect(DB_FILE)
    cursor = conn.cursor()

    with open(CSV_FILE, newline="", encoding="utf-8") as file:
        reader = csv.DictReader(file)
        for row in reader:
            cursor.execute(
                "INSERT INTO users (name, email) VALUES (?, ?)",
                (row["name"], row["email"])
            )

    conn.commit()
    conn.close()

# - - - Main execution - - -
if __name__ == "__main__":
    init_db()
    insert_users_from_csv()
    print("CSV data successfully inserted into SQLite database.")
```

Functions used:

1. **init_db():-** Initializes the database “users.db”, creates a connection, executes “CREATE TABLE” to create a table called “users” with “id, name and email”, saves changes and terminates the connection.
2. **insert_users_from_csv():-** “csv.DictReader” reads the csv file into a dictionary, inserts the csv data values into the table and saves changes before terminating the connection.

4. Complex Python Code Link:

https://www.kaggle.com/code/jordandsouza/smart-tutor-capstone-project
I created this project as a part of a Kaggle Capstone.

5. Complex Database Code Link:

https://github.com/jordan-dsouza/Projects/blob/main/JordanDsouza_SQL_Code.pdf

Problem Statement 2:

1. **Where would you rate yourself on (LLM, Deep Learning, AI, ML)?**

   B [Can code under supervision]. I am confident in my Deep Learning and ML skills. However I have recently started with LLMs.

2. **What are the key architectural components to create a chatbot based on LLM?**

   a. User Interface (UI):
   User interface is used by the user to interact with the chatbot, to display responses and handle session IDs and user metadata.
   Eg. Mobile apps, Chat apps, Web UI

   b. API/Backend Service Layer:
   Receives user requests from the UI, sends requests to LLM pipeline, authenticates users, returns LLM responses to the UI, accesses real time data, etc.
   Eg. FastAPI.

   c. Orchestration Layer:
   Routes queries, manages flow, decides when to use LLM or external tools.
   Eg. LangChain, LangGraph.

   d. Context and Memory Management:
   Short term memory - Recent messages in sessions.
   Long term memory - Stored user facts or past conversations.
   Summarizes long conversations, stores and retrieves relevant context, maintains conversational continuity.
   Tracks conversational history and state.
   Eg. Databases (MongoDB, Postgres), Vector Databases (FAISS, Pinecone).

   e. Large Language Model (LLM):
   The core generative language model for reading and understanding requests and generating human-like responses, calling functions.
   Eg. Hosted LLM APIs (Gemini, GPT), Self - Hosted LLMs (Mistral).

   f. Knowledge Retrieval (Retrieval Augmented Generation - RAG):
   A vector database (offline) and retrieval system that fetches contextual data/documents thus preventing hallucinations and grounding answers.
   Data sources can be APIs, Databases, Documents.

g. Tool/Function Layer:
   Enables actions beyond text generation. LLM decides when to call a tool, backend executes the tool, tool output is fed to the LLM and final output is generated.
   Eg. Web search, file operations, database querying.

h. Safety, Validation and Guardrails:
   Prevents hallucinations, detects sensitive content and enforces policies to ensure safe and reliable outputs.
   Eg. Input validation, output filtering, moderation models (OpenAI's moderation API), prompt constraints, schema validation for JSON outputs, etc.

i. Monitoring Layer:
   Monitors token usage, latency (delay between request and response), cost, user satisfaction, LLM error rate, etc.
   Eg. Version tracking, feedback loops, logs.

j. Infrastructure Deployment:
   Makes the chatbot scalable and reliable.
   Eg. Cloud hosting (AWS, GCP), Caching (Redis), Secret management (Google Cloud Secret Management), etc.

   Flow:
   1. The user sends a request in the UI.
   2. The backend receives the request.
   3. Context and memory is retrieved.
   4. The prompt is assembled.
   5. LLM generates a response or calls necessary tools.
   6. The output is validated for safety.
   7. The generated and validated response is returned to the user.
   8. The session logs and metrics are stored.

3. **Please explain vector databases. If you were to select a vector database for a hypothetical problem (you may define the problem) which one will you choose, and why?**
   a. Definition:
   A vector database is a specialized database that stores, indexes and searches high-dimensional vector embeddings efficiently. It enables similarity searches unlike traditional databases that rely on exact matches.

   b. Vectors:
   - Modern AI models convert data into vector embeddings which are an ideal data structure for machine learning algorithms as modern CPUs and GPUs are optimized to perform the mathematical operations needed to process them.
   - Vector embeddings are numerical representations (lists of numbers or vectors) that capture the meaning and relationships of complex data like words, images, or audio, allowing AI to process and understand them.
   - Vectors allow us to find semantic similarity and perform approximate nearest neighbour (ANN) search.
   - They are created via algorithms and models like "Word2Vec".

   c. Problem Context (Research Paper Q&A):
   - I have created a "Research Paper Q&A" bot that works across multiple PDFs to give context grounded answers on research papers.

   d. Database Used - Pinecone:
   - A vector database is needed as research papers are long and unstructured, which are not suitable to be passed into the LLM prompt directly as there are context window limits and no built-in knowledge of the PDFs.
   - So instead of parsing the entire paper repeatedly, we index the paper semantically and retrieve only relevant sections in order to ground the LLM.

   e. Why Pinecone:
   - Latency is about 100ms which is great as users want low latency responses.
   - It combines vector search with metadata filtering for relevant results.
   - User-friendly API for Python.