Problem statement- assignment 2

Q1. Where would you rate yourself on (LLM, Deep Learning, AI, ML). A, B, C [A = can code independently; B = can code under supervision; C = have little or no understanding]

Area	**Rating**	**Explanation**
LLM (Large Language Models)	**B**	I understand how LLMs work at a high level, including prompt engineering, embeddings, APIs (OpenAI/Gemini) and chatbot pipelines. I can build LLM-based applications under guidance and documentation.
Deep Learning	**B**	I understand neural networks, CNNs, RNNs and Transformers. I can implement models using frameworks like TensorFlow and PyTorch with reference and supervision.
AI (Artificial Intelligence)	**B**	I understand intelligent agents, search strategies, knowledge representation, and real-world AI applications.
ML (Machine Learning)	**B**	I understand supervised and unsupervised learning, feature engineering, and model evaluation. I can implement ML models using Python libraries such as scikit-learn.

Q2.What are the key architectural components to create a chatbot based on LLM? Please explain the approach on a high-level.

Key Architectural Components

A. The Orchestration Layer (The "Controller")

This is the application logic that glues everything together. It doesn't "think", but it directs traffic.

- **Frameworks:** Tools like **LangChain**.
- **Role:** It manages the flow: receiving the user query, deciding if it needs to look up information, formatting the prompt, sending it to the LLM, and handling the output. It also manages the "Chain of Thought" if complex reasoning is required.

B. The Cognitive Layer (The LLM)

The core intelligence engine.

- Proprietary models (GPT-4, Claude 3.5, Gemini 1.5) or Open-Source models (Llama 3, Mistral) hosted locally or on clouds like AWS Bedrock/Azure OpenAI.
- **Role:** It performs the actual reasoning, summarization, and language generation. It takes the "enriched prompt" (user query + retrieved context) and generates the natural language response.

C. The Knowledge Layer (RAG & Vector Database)

LLMs are frozen in time (trained on past data). To make a chatbot useful for a specific business, you need to give it access to private, up-to-date data.

- **Vector Database:** Stores data not as text, but as **vectors** that represent semantic meaning.
- **Embedding Model:** A small AI model that converts text documents into these vectors.
- **Role:** When a user asks "What is our refund policy?",the system searches the Vector DB for the most mathematically similar documents, retrieves the text of the refund policy, and feeds it to the LLM.

D. The Memory Component (State Management)

LLMs are stateless; they treat every request as a brand new interaction.

- **Short-term Memory:** Stores the last N turns of the conversation (e.g. In Redis or Postgres).

- **Role:** When a user says "Change *it* to blue," the memory component provides the context that "*it*" refers to the "logo" discussed in the previous message.

E. The Guardrails & Safety Layer

- **Input/Output Guardrails:** Systems (like NeMo Guardrails) that intercept messages.
- **Role:** It prevents the bot from answering off-topic questions (e.g. "Write a poem about competitors") or generating toxic content. It also masks PII (Personally Identifiable Information) before it is sent to the LLM provider.

F. The Action Layer (Tools & Agents)

- **Function Calling:** The ability to connect to external APIs (SQL databases, CRMs like Salesforce, Calendar APIs).
- **Role:** If the user asks "Book a meeting," the LLM generates a structured JSON output (e.g., {"action": "book_meeting", "time": "2pm"}) which the Orchestrator uses to actually execute the API call.

High-Level System Architecture

User Interface

|

Backend API Server

|

Query Preprocessing & Prompt Engineering

|

Embedding Model

|

Vector Database

|

Context Retrieval

|

LLM Engine

|

Response Formatter

|

User

Detailed Step-by-Step Workflow

1 User Interface (Frontend Layer)

The chatbot starts with a user interface such as:

- Web application
- Mobile app
- Chat interface (like WhatsApp bot, Slack bot)

Responsibilities:

- Accept user input (query/message)
- Display chatbot responses
- Maintain chat session

2 Backend API Server (Application Layer)

The backend server acts as the **central controller**.

Built using:

- Flask / FastAPI

Responsibilities:

- Receive user request
- Authenticate user
- Manage sessions
- Route query through AI pipeline

3. Query Preprocessing & Prompt Engineering Layer

This layer prepares the user query for AI processing.

Tasks:

- Clean user input
- Detect intent
- Add system instructions
- Attach conversation history

4. Embedding Model (Semantic Understanding)

The user query is converted into a **numerical vector (embedding)** using an embedding model.

Examples:

- OpenAI Embeddings
- Sentence Transformers
- Gemini Embeddings

5 Vector Database (Knowledge Store)

The vector database stores:

- University documents
- Policies
- FAQs
- Notices
- Study material

Each document is already converted into embeddings and stored.

Popular Vector DBs:

- FAISS
- Pinecone
- Weaviate
- Milvus
- Chroma

6.Context Retrieval (Semantic Search)

- The user query embedding is compared with stored embeddings using similarity search.

User Query Vector → Vector DB → Top K similar documents

7. LLM Engine (Reasoning & Generation)

The LLM receives:

- User question
- Retrieved documents
- System instructions

LLM then:

- Understands the question
- Reads the context
- Generates an accurate response

8.Memory & Knowledge Store

Conversation memory is stored using:

- Vector database (long-term memory)
- Redis / SQL (short-term memory)

This allows:

- Context continuity
- Follow-up questions
- Personalization

9. Response Formatter

The raw LLM output is processed:

- Clean formatting
- Add bullet points
- Add markdown
- Add references

Q3. 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?

What is a Vector Database?

A vector database is a special type of database that stores **embeddings** which are numerical representations of text, images, audio or other data. These embeddings allow the database to quickly find items that are similar to a given query using a method called **nearest neighbor search**.

Unlike traditional databases, vector databases are built for similarity search. They also support useful features such as:

- Storing and updating data (insert, update, delete)
- Filtering based on metadata (for example: language, product type, user category)
- Scaling across multiple machines
- Fast searching even with millions or billions of vectors
- Hybrid search (vector + keyword search)
- Compression to reduce storage cost

Why use a vector database?

Modern AI systems like Large Language Models (LLMs) and multimodal models generate embeddings. These embeddings are used in applications such as:

- Retrieval-Augmented Generation (RAG)
- Semantic search
- Recommendation systems
- Image and audio search
- Chat memory systems

A vector database turns these raw embeddings into a production ready system that is fast, scalable and reliable.

Important Technical Components

1. Indexing and Search Algorithms
Algorithms like HNSW, IVF and PQ are used to search large datasets efficiently. They trade a small amount of accuracy for much faster search speeds.

2. Compression and Quantization
These techniques reduce memory usage and storage cost, which is especially important when dealing with millions or billions of vectors.

3. Metadata and Filtering
This allows you to combine vector similarity search with traditional filtering, such as searching only documents from a specific customer or region.

4. Real-time Updates and Scaling
Some vector databases allow real time insertion and updates, while others are designed more for batch processing.

5. Managed vs Self-Hosted Options
You can choose a fully managed cloud service (like Pinecone or Qdrant Cloud) or run an open-source version yourself (like Milvus, Qdrant, or Weaviate).

Comparison of Major Players (2026)

Database	**Best For**	**Key Strength**
Pinecone	Startups & Enterprise RAG	Fully managed, "Zero-Ops" Serverless.
Weaviate	Knowledge Graphs	Modular, built-in AI models, great "Hybrid Search."
Milvus	Massive Scale	Extremely scalable (billions of vectors), handles heavy throughput.
Qdrant	Performance & Filtering	Written in Rust; blazing fast with complex metadata filters.
Chroma	Prototyping	Open-source, lightweight and very easy to set up locally.

Hypothetical Problem: "Project Flash-Fashion"

The Problem:

We are building a Global Visual Discovery Engine for a fashion marketplace.

- **The User Experience:** A user takes a photo of someone's sneakers on the street and uploads it. The app must instantly find the exact sneakers or "visually similar" alternatives from a catalog of 50 million items.
- **Requirement 1 (Multi-modal):** The search must work with images (from the user) against both images and text descriptions in the database.
- **Requirement 2 (Geo-Filtering):** We only want to show items available for shipping to the user's specific country (e.g., "Find similar shoes *but only* if they ship to India").
- **Requirement 3 (Latency):** The search must take less than 200ms to keep the "shopping flow" smooth.

My Choice: Qdrant

Why Qdrant?

For this specific fashion discovery problem, I would choose Qdrant over the others for three reasons:

1.High-Performance Payload Filtering:

In fashion, you aren't just searching for "similar shoes." You are searching for "similar shoes AND size 10 AND in stock AND ships to India." Qdrant is optimized for Payload Filtering.

While some databases perform "Post Filtering" (which is slow), Qdrant uses "Filtered HNSW," which ignores irrelevant items during the search process, ensuring the 200ms latency goal is met even with complex business rules.

2.Rust-Based Efficiency:

Because it is written in Rust, Qdrant offers extremely high throughput and low resource consumption. For a "Flash-Fashion" app that might experience huge traffic spikes during sales, the hardware efficiency of Rust translates to lower cloud costs compared to Java-based alternatives.

3.Advanced Distance Metrics:

Fashion embeddings often benefit from specific mathematical distance formulas. Qdrant supports a wide variety of metrics (Cosine, Dot Product, Manhattan, etc.) out of the box, allowing our data scientists to experiment with which "similarity" feels most natural to a human eye.