**DAY 1**

**What is tokenization   
what is tiktoken  
what is transformer**

**What is vector embeddings**

**What is positional Encodings**

**What is Self Attention & multi-head Attention**

**What is RAG  
What is Synthetic data  
What AI Jargons**

**1. Tokenization**

Tokenization is the process of converting a sequence of text (like a sentence or a paragraph) into smaller units, called tokens. These tokens are typically words, subwords, or even characters. In NLP (Natural Language Processing), tokenization is a crucial step because machine learning models like transformers require input to be in tokenized form.

* **Word-level tokenization**: Splitting the text into individual words.
* **Subword-level tokenization**: Breaking down words into smaller units (subwords). This is especially useful for rare words or misspellings.
* **Character-level tokenization**: Breaking down text into individual characters.

For example:

* Text: "ChatGPT is amazing"
* Tokens: ["ChatGPT", "is", "amazing"]

Tokenization allows a model to better process text, understanding the structure and relationships of words.

**2. Tiktoken**

Tiktoken is a Python library designed to efficiently handle tokenization specifically for OpenAI's language models like GPT. It's optimized for encoding and decoding text into tokens in a way that is computationally efficient and memory-friendly.

* **Purpose**: It's a specialized tool used to manage the tokenization process for large-scale language models like GPT-3, GPT-4, etc.
* **Efficiency**: It reduces overhead by providing fast encoding and decoding of text inputs and outputs, making it highly suitable for applications that require real-time or large-scale processing.

**3. Transformer**

A transformer is a type of neural network architecture introduced in the paper "Attention is All You Need" by Vaswani et al. (2017). Transformers have revolutionized the field of NLP and are the foundation of many state-of-the-art models like BERT, GPT, T5, and others.

* **Key Components**:
  + **Self-Attention Mechanism**: Allows the model to weigh the importance of different words in a sequence relative to each other, helping it understand context.
  + **Positional Encodings**: Since transformers don't process input sequentially like RNNs or LSTMs, positional encodings are used to provide information about the position of words in a sequence.

Transformers have significantly improved performance on NLP tasks because they can capture long-range dependencies in text better than traditional models.

**4. Vector Embeddings**

Vector embeddings are dense representations of text, words, or sentences in continuous vector space. These vectors capture the semantic meaning of the text they represent, allowing models to understand relationships between words and phrases based on their proximity in vector space.

* **Example**: In a word embedding space, the words "king" and "queen" would be represented as vectors that are closer together than "king" and "car".
* **Use**: Embeddings are used in various NLP tasks like machine translation, text classification, sentiment analysis, etc.

Popular embedding techniques include:

* **Word2Vec**
* **GloVe**
* **BERT Embeddings**

**5. Positional Encodings**

Since transformers don’t have an inherent sense of order (unlike RNNs and LSTMs), positional encodings are added to give the model information about the position of words in a sequence.

* **Why needed**: Unlike RNNs, which process text sequentially and inherently have access to word order, transformers treat all words in a sequence at the same time. To account for the order, positional encodings are added to each token's embedding to tell the model its position in the sequence.

Positional encodings are typically vectors added to token embeddings, and they can be learned or fixed. Sinusoidal positional encodings are often used, which use a combination of sine and cosine functions.

**6. Self-Attention & Multi-Head Attention**

**Self-Attention** is a mechanism in which each word (or token) in a sentence is compared to every other word in the sentence to determine their relationship or relevance to each other.

* **Self-Attention**: Helps a model to focus on relevant parts of the input sequence when making predictions, regardless of their distance in the sequence. For instance, in the sentence "The cat sat on the mat", the word "cat" should pay more attention to the word "sat" than to the word "mat".

**Multi-Head Attention** extends self-attention by running multiple self-attention processes in parallel. Each "head" learns a different set of relationships, allowing the model to capture multiple types of relationships and patterns in the data.

* **Benefit**: It helps capture different aspects of the sentence simultaneously, improving the model’s ability to learn diverse information from the context.

**7. RAG (Retrieval-Augmented Generation)**

RAG is a hybrid model architecture that combines the power of retrieval-based methods (retrieving relevant documents from a database) with generation-based models (like transformers).

* **How it works**: RAG first retrieves relevant documents or passages from a large corpus based on the query. Then, it uses a generative model (like GPT) to synthesize an answer using the retrieved information. This allows RAG models to provide more accurate and contextually relevant responses, especially in knowledge-intensive tasks.
* **Use cases**: It’s useful in tasks where the model needs to answer questions based on external knowledge, like question answering or summarization, without having to store all knowledge in the model’s parameters.

**8. Synthetic Data**

Synthetic data refers to artificially generated data that is used to train machine learning models. Instead of using real-world data, synthetic data is created by algorithms to mimic real-world data.

* **Why use synthetic data**: It helps overcome data scarcity or privacy concerns, particularly in areas where obtaining real-world data is expensive, sensitive, or time-consuming (e.g., medical or financial datasets).
* **Applications**: It's used in training models for image recognition, NLP, autonomous vehicles, and more.

Synthetic data is generated using techniques like:

* **Simulation-based generation**
* **Generative models like GANs (Generative Adversarial Networks)**

**9. AI Jargon**

AI jargon refers to the specialized terminology used in the field of artificial intelligence and machine learning. Some common examples include:

* **Model**: The algorithm or system used to process input and make predictions or decisions.
* **Training**: The process of feeding data to a model to help it learn patterns.
* **Inference**: The process of using a trained model to make predictions on new, unseen data.
* **Loss Function**: A mathematical function used to evaluate the performance of a model.
* **Overfitting**: When a model learns too much from the training data and performs poorly on new data.
* **Underfitting**: When a model fails to capture the underlying patterns of the training data and performs poorly.

These terms are fundamental to understanding how machine learning models are built, trained, and deployed.

**DAY 2**

**What is Prompts?**

**What is Alpaca Prompts?**

**What is LLAMA Prompting?**

**What is System Prompt?**

**What is Zero and Few Shot Prompting?**

**What is Self-Consistency Prompting?**

**What is Persona Based Prompting?**

**What is Role-Play Prompting?**

**What is Conceptual Prompting?**

**What is Multimodal Prompting?**

**🧠 1. What is a Prompt?**

A **prompt** is the input or instruction you give to a language model (like GPT, Gemini, etc.) to get a desired output.

* It can be a **question**, a **command**, or a **structured format**.
* Think of it as a conversation starter or a set of instructions.

**Example:**  
Prompt → *"Write a poem about the ocean."*  
Output → A poem about the ocean 🌊

**🐑 2. What is Alpaca Prompting?**

**Alpaca Prompts** come from **Stanford’s Alpaca**, a fine-tuned version of LLaMA using **instruction-following** data.

* Alpaca prompts are **instruction-tuned** prompts like:  
  *“Explain how photosynthesis works.”*  
  *“Write a Python script to calculate factorial.”*

**Goal:**  
Make the model follow specific, **human-style instructions** better (like ChatGPT).

**🦙 3. What is LLaMA Prompting?**

**LLaMA (Large Language Model Meta AI)** is Meta's open-source LLM.

**LLaMA Prompting** refers to the **style of prompting** used for LLaMA models, often requiring:

* Precise input formatting
* Special tokens like <s>, </s> in older versions
* Instructions tailored for **factual, step-by-step** responses

LLaMA + fine-tuning = base for models like Alpaca, Vicuna, etc.

**⚙️ 4. What is a System Prompt?**

A **system prompt** is a **hidden or initial prompt** that sets the behavior of the AI for the session.

* It's often used **under the hood** (like in ChatGPT).
* It defines how the model should behave — tone, style, goals.

**Example:**

*"You are a helpful assistant that explains things in simple terms."*

All your responses will then follow this guideline.

**🎯 5. What is Zero-shot and Few-shot Prompting?**

**🟢 Zero-shot prompting:**

* **No examples** are given.
* Just the instruction.

**Example:**

*"Translate 'Hello' to French."*  
Model → "Bonjour"

**🟡 Few-shot prompting:**

* A few **examples** are provided to help the model understand the task.

**Example:**

diff

CopyEdit

Translate English to French:

- Hello → Bonjour

- Good morning → Bonjour

- Thank you → Merci

Now translate: How are you?

Model → Comment ça va?

**🔁 6. What is Self-Consistency Prompting?**

Instead of just **one output**, the model generates **multiple responses** and selects the **most consistent** answer.

* Increases **accuracy** and **reliability**.
* Often used in **reasoning tasks** or **math problems**.

**Example:**  
Ask the model to solve a math problem 5 times → Choose the most common correct answer.

**👤 7. What is Persona-based Prompting?**

You tell the model to **take on a specific persona** or identity.

**Examples:**

* "Act like Elon Musk and explain Mars colonization."
* "You are a fitness trainer. Give me a home workout."

It influences **tone, language, and personality** in the response.

**🎭 8. What is Role-Play Prompting?**

Similar to persona prompting, but more **interactive** and **scenario-driven**.

**Example:**

You are a doctor. I’m a patient with a headache. Let’s have a consultation.

Model replies like an actual doctor. Used in:

* Chatbots
* Simulations
* Education

**🧠💡 9. What is Conceptual Prompting?**

This is about **abstract thinking** and helping models reason through **ideas, logic, and mental models**.

**Example:**

"Explain democracy as if I’m a 5-year-old."

The model simplifies complex **concepts** into digestible ideas. Great for:

* Teaching
* Analogies
* Simplifying knowledge

**🖼️🔤 10. What is Multimodal Prompting?**

Using **multiple modes** of input (e.g., text, image, audio) to ask a question or get an answer.

**Example:**

[Image of a plant] + “What kind of plant is this?”

Used in:

* Gemini
* GPT-4 with vision
* CLIP models

**Real-world Uses:** AI tutors, visual question answering, image captioning.

**✨ Summary Table:**

| **Prompt Type** | **Description** |
| --- | --- |
| **Prompt** | Instruction to guide LLMs |
| **Alpaca Prompting** | Instruction-following for LLaMA-style models |
| **LLaMA Prompting** | Prompt format and style used for Meta’s LLaMA |
| **System Prompt** | Initial instruction to set AI behavior |
| **Zero-shot Prompting** | No examples, just the task |
| **Few-shot Prompting** | Give a few examples for better context |
| **Self-Consistency** | Generate multiple outputs and pick the best |
| **Persona-based Prompting** | Model acts like a specific character |
| **Role-play Prompting** | Interactive scenario-driven conversations |
| **Conceptual Prompting** | Explaining abstract or deep concepts clearly |
| **Multimodal Prompting** | Use of text + image/audio/video as input |

**DAY 3**

**What is Cursor?  
What is OLLAMA?**

**What is Fine-Tunning?**

**1. What is Cursor?**

**Cursor** is an **AI-powered code editor**, built to make developers faster.  
It’s like **Visual Studio Code (VS Code)** but **deeply integrated with AI models** like GPT-4, etc.

✅ **Features of Cursor:**

* **Autocomplete:** Writes code as you type, like Copilot but better.
* **Explain Code:** Select code → Ask "what does this do?" → It explains.
* **Edit Code with AI:** Highlight code → Say "make it faster" → It rewrites it.
* **Generate Tests:** Automatically create unit tests for your code.
* **Inline Chat:** Chat with AI inside your project.
* **Run local models:** You can connect it to models like **Ollama** and run AI locally.

**Simple Example:**  
Imagine you're writing a React component, and you're stuck.  
In Cursor, you highlight your half-written code and type:

"Make this responsive and add animations."

It will **automatically** modify your code based on your request.

**2. What is Ollama?**

**Ollama** is a tool that lets you **run Large Language Models (LLMs)** **locally** on your computer — without sending data to the cloud.

✅ **Features of Ollama:**

* Run models like **Llama 3**, **Mistral**, **Codellama**, etc. on your laptop.
* **Private:** No data leaves your machine.
* **Easy:** One command to start models (ollama run llama3).
* **Custom Models:** You can fine-tune and create your own model (using Modelfiles).

Think of it like this:

Ollama = "ChatGPT on your laptop."

**Simple Example:**  
You can open your terminal and type:

ollama run llama3

and start chatting with an AI model locally without an internet connection!

**3. What is Fine-Tuning?**

**Fine-tuning** is the process of **training a pre-existing AI model** on your **specific data** to make it better at your tasks.

✅ **Why Fine-Tuning?**

* To make a model **understand specific language** (ex: medical terms, legal language).
* To make a model behave **differently** (like being more formal, funny, or detailed).
* To teach a model about **new knowledge** it didn't have.

✅ **How Fine-Tuning Works:**

* You start with a **base model** (like Llama 3, GPT-3, etc.).
* You give it **lots of examples** (called a **dataset**).
* The model **learns patterns** in your examples.
* After fine-tuning, it behaves **more like what you want**.

**Simple Example:**

Imagine you have a base model that knows English generally.  
But you want a model **only for cooking recipes**.

You fine-tune it by giving **thousands of examples** like:

{"prompt": "How to make chocolate cake?", "completion": "1. Preheat oven... 2. Mix ingredients..."}

{"prompt": "How to cook pasta?", "completion": "1. Boil water... 2. Add pasta..."}

After fine-tuning:

* If you ask: "How to make biryani?" → It gives you **step-by-step cooking instructions** instead of a random essay.

**🔥 Summary Table**

| **Term** | **Meaning** | **Example** |
| --- | --- | --- |
| **Cursor** | AI-first code editor | Autocomplete and explain your code |
| **Ollama** | Run AI models locally | ollama run llama3 |
| **Fine-Tuning** | Training a model for specific behavior | Fine-tune a model to specialize in cooking recipes |

**DAY 4**

**What is Base Model?**

**What are types of Fine-Tunning?**

**What is Full Parameter Fine-Tunning?**

**What is Lo-RA Parameter Fine-Tunning?**

**Explain the Process of Fine-Tunning?**

**When to use RAG and Fine-Tunning?**

**1. What is a Base Model?**

A **Base Model** (also called a **Pre-trained Model**) is an AI model that has already been trained on **huge amounts of general data** — like books, websites, and Wikipedia — but it **hasn’t been customized** for any specific task yet.

✅ **Think of it like:**

* A general-purpose brain 🧠
* It knows **a little bit about everything**, but **isn't an expert** in anything yet.

**Examples of Base Models:**

| **Model Name** | **Base Model Purpose** |
| --- | --- |
| Llama 3 | Chatting, general knowledge |
| GPT-3 | General-purpose understanding |
| Mistral | Fast, efficient general understanding |
| Codellama | Base model focused on coding |

**2. What are the Types of Fine-Tuning?**

There are different ways to fine-tune a model depending on your needs and your resources (money, compute).

✅ **Main types:**

| **Fine-Tuning Type** | **Description** | **When to Use** |
| --- | --- | --- |
| **Full-Parameter Fine-Tuning** | Update **all** parameters of the model | When you have lots of data + GPU power |
| **LoRA Fine-Tuning** | Update only **small parts** (adapters) of the model | When you want faster, cheaper fine-tuning |
| **Prefix-Tuning** | Add **learnable prompts** without touching model weights | For small, quick task adaptations |
| **QLoRA** | Quantized LoRA (even cheaper memory-wise) | When running on limited RAM (ex: 24GB VRAM) |

The two you specifically asked for are **Full-Parameter Fine-Tuning** and **LoRA**, so let’s dive deep:

**3. What is Full-Parameter Fine-Tuning?**

✅ **Definition:**  
You take the **entire base model** — all billions of parameters — and **update all of them** based on your training data.

**Example:**

* Suppose Llama 3 has **70 billion parameters** (weights).
* During fine-tuning, all **70 billion** weights are **modified** to better match your data.

✅ **Advantages:**

* Very powerful.
* Can deeply adapt a model to a new domain.

✅ **Disadvantages:**

* Requires **massive compute** (GPUs like A100/H100).
* Expensive.
* Time-consuming.

**4. What is LoRA (Low-Rank Adaptation) Fine-Tuning?**

✅ **Definition:**  
Instead of updating all model parameters, **LoRA** freezes the original model and **inserts small trainable matrices** (called adapters) at specific points.

✅ **Idea:**

* Original model = **Frozen** (unchanged)
* Add **small new layers** = **Train only these**

**Example:**  
Instead of modifying 70 billion parameters, you **only train a few million parameters**.

✅ **Advantages:**

* Much **cheaper** and **faster**.
* Can fine-tune a huge model **on a normal laptop** or **small GPU**.
* Easy to **share** (because only small adapters are saved).

✅ **Disadvantages:**

* Slightly less powerful than full fine-tuning for very big changes.
* But in 90% of use cases, **LoRA is enough**.

**5. Explain the Process of Fine-Tuning (Step-by-Step)**

Here’s how Fine-tuning works:

| **Step** | **Action** | **Example** |
| --- | --- | --- |
| 1 | **Choose a Base Model** | Llama 3, Mistral, GPT-2 |
| 2 | **Prepare your Dataset** | JSONL, or text files (prompt + output) |
| 3 | **Preprocess Data** | Format into prompt/completion pairs |
| 4 | **Configure Training** | Pick full or LoRA, batch size, learning rate |
| 5 | **Train the Model** | Model reads data and adjusts weights |
| 6 | **Save the Model** | Save final fine-tuned model or LoRA adapters |
| 7 | **Deploy the Model** | Use it in apps, APIs, or Chatbots |

**6. When to use RAG vs Fine-Tuning?**

First, what is **RAG**?

✅ **RAG** (Retrieval-Augmented Generation) means:

* The model **retrieves** information from an external database (knowledge base) during answering.
* Instead of memorizing everything, it **searches** when needed.

| **Use** | **RAG** | **Fine-Tuning** |
| --- | --- | --- |
| When you have **frequent updates** | ✅ (RAG is better, no need to retrain) | ❌ (Fine-tuning would be outdated) |
| When you need **model behavior change** | ❌ (RAG cannot change style/behavior) | ✅ (Fine-tuning needed) |
| When you have **small private documents** | ✅ (RAG can fetch them) | ❌ (Fine-tuning not needed) |
| When you have **domain-specific answers** | ❌ (RAG can miss context) | ✅ (Fine-tuning makes model smarter) |
| Example | Chatbot for latest news | Medical model that gives legal-safe answers |

✅ **Simple Rule:**

* If you just want **up-to-date information** → **Use RAG**.
* If you want **to change the model's brain** → **Do Fine-Tuning**.

**📚 Simple Example to Connect Everything:**

Imagine you want to build a **Medical Assistant AI**:

* Base model: Llama 3
* Dataset: 10,000 medical Q&A pairs

→ If you **fine-tune** the model:

* It starts answering **like a medical professional**.
* Knows context even without documents.

→ If you **use RAG**:

* Model is general.
* When asked about a disease, it **fetches** from a database like "MedInfo DB".

✅ Best projects usually **combine both**:

* Fine-tune a base for behavior
* Use RAG for **real-time updated info**!

**🔥 QUICK DIAGRAM (if you want)**

Base Model --> Fine-Tune --> Specialized Brain

\

-> RAG Engine --> Fetch updated documents

**FINAL QUICK SUMMARY**

| **Term** | **Meaning** |
| --- | --- |
| Base Model | Original pretrained model |
| Full Fine-Tuning | Update **all** model weights |
| LoRA Fine-Tuning | Update **small lightweight adapters** |
| Fine-Tuning Process | Dataset -> Train -> Save Model |
| RAG | Retrieval from external sources during generation |
| When to use RAG | When **up-to-date info** needed |
| When to use Fine-Tuning | When **behavior/knowledge change** is needed |

**DAY 4**

**What is Knowledge-cutoff?**

**What is RAG?**

**What is Context Window?**

**What are vector Embeddings?**

**What is Vector Databases?**

**What is Chunking in RAG?**

**What is Indexing in RAG?**

**Explain RAG Work-Flow?**

**What is Lang-Chain and how to use it?**

**What is RAG-Chain?**

**🔍 In-Depth Guide to RAG and Its Ecosystem**

This comprehensive guide covers everything from foundational concepts like embeddings and vector databases to complete RAG workflows and toolkits like LangChain and RAG-Chains, with detailed examples.

**📅 1. What is a Knowledge Cutoff?**

**Definition:**

The knowledge cutoff is the date until which an AI model (like GPT) has been trained on data. The model has no knowledge of any events, facts, or updates beyond this date.

**Why It's Important:**

If you're building a question-answering system, relying solely on the model's internal knowledge might lead to outdated responses.

**Example:**

If GPT-4 has a knowledge cutoff in **June 2024**, and someone asks, "Who won the 2025 IPL?" — it won’t know unless you add retrieval from an external source like a website or vector database.

**🧠 2. What is RAG (Retrieval-Augmented Generation)?**

**Definition:**

RAG combines traditional language model generation with external document retrieval. Instead of answering from memory, the model retrieves relevant documents and generates a response based on them.

**Key Parts:**

* **Retriever**: Finds documents relevant to the query
* **Generator**: Produces answers using those documents

**Real-World Analogy:**

Imagine a student (LLM) answering a question. Without notes (retrieval), their answer is from memory (sometimes wrong). With notes, they refer to accurate documents before replying — that’s RAG.

**Benefits:**

* Fresh and up-to-date responses
* Reduces hallucinations
* Smaller models can perform better with strong retrieval

**🔄 3. RAG Workflow (Step-by-Step with Example)**

Let’s build a full example: "What are the side effects of paracetamol?"

**Step 1: User Input**

User asks: **"What are the side effects of paracetamol?"**

**Step 2: Embed the Query**

Use a model (like OpenAIEmbeddings or SentenceTransformers) to convert the query into a vector.

from sentence\_transformers import SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-v2')

query\_vector = model.encode("What are the side effects of paracetamol?")

**Step 3: Vector Search**

Search the vector database (like FAISS or Pinecone) for top 5 semantically similar documents.

retrieved\_chunks = vector\_db.search(query\_vector, top\_k=5)

**Step 4: Construct Prompt**

Feed the retrieved content into the LLM along with the question:

Context:

1. Paracetamol can cause liver damage in high doses.

2. Common side effects include nausea, skin rash, headache.

3. Alcohol should be avoided while using paracetamol.

Question:

What are the side effects of paracetamol?

**Step 5: Generate Answer**

answer = openai.ChatCompletion.create(

model="gpt-4",

messages=[

{"role": "system", "content": "You are a medical assistant."},

{"role": "user", "content": prompt}

]

)

**Output:**

"Paracetamol commonly causes nausea, headache, and rash. In large doses, it can damage the liver. Avoid alcohol."

**🧱 4. What are Embeddings?**

**Definition:**

Embeddings are vector representations of text in high-dimensional space. Similar meanings have similar vectors.

**Example:**

* Text: "Car"
* Vector: [0.122, -0.983, 0.433, ..., 0.991]

"Car" and "Automobile" will have close vectors.

**Tools:**

* OpenAIEmbeddings
* HuggingFace Sentence Transformers
* Cohere

**🗃️ 5. What is a Vector Database?**

**Definition:**

A vector database stores and indexes embeddings. It allows fast similarity searches.

**Common DBs:**

* FAISS (open-source)
* Pinecone
* ChromaDB
* Weaviate

**Example:**

You upload 10 PDFs → chunk and embed them → store in vector DB. When a query comes in, search for relevant chunks based on vector similarity.

**🧩 6. What is Chunking in RAG?**

**Definition:**

Breaking large documents into smaller text units (chunks) for easier processing and retrieval.

**Example:**

A 10,000-word article is split into 100 chunks of ~100 words.

**Why It’s Needed:**

LLMs have a limited **context window**. You can’t pass an entire book — just relevant parts.

**Chunking Strategies:**

* Fixed size: 500 tokens
* Semantic: Break by paragraph or sentence boundary

**🗂️ 7. What is Indexing in RAG?**

**Definition:**

Indexing is the process of converting chunks into vectors and storing them with metadata.

**Metadata Includes:**

* Source (filename, URL)
* Chunk number
* Tags (e.g., "medical", "legal")

**Example:**

{

"vector": [0.22, 0.88, ...],

"metadata": {

"source": "drug\_info.pdf",

"chunk": 4,

"category": "medicine"

}

}

**🧠 8. What is Context Window?**

**Definition:**

The context window is the maximum number of tokens (words, punctuation) an LLM can process in one go.

**Examples:**

* GPT-3.5: 4,096 tokens (~3,000 words)
* GPT-4: up to 128,000 tokens (~96,000 words)

If input exceeds this, the model will truncate or reject it.

**🧱 9. What is LangChain?**

**Definition:**

LangChain is a Python framework for building LLM-driven applications using tools like memory, chaining, retrieval, agents, etc.

**Features:**

* Easy chaining of tools and steps
* Integration with Pinecone, FAISS, OpenAI, Hugging Face
* Built-in templates for RAG, chatbots, agents

**Example:**

from langchain.chains import RetrievalQA

from langchain.vectorstores import FAISS

vector\_store = FAISS.load\_local("medical\_index")

qa = RetrievalQA.from\_chain\_type(llm=OpenAI(), retriever=vector\_store.as\_retriever())

result = qa.run("What are paracetamol side effects?")

**🔗 10. What is RAG-Chain?**

**Definition:**

A RAG-Chain is a specialized pipeline that implements Retrieval-Augmented Generation as a series of connected processing steps or chains.

**Components:**

1. **Retriever Chain**: Retrieves top K documents
2. **Reranker Chain**: Reorders based on quality/relevance
3. **LLM Chain**: Generates answer
4. **Post-processing Chain**: Formats or cites sources

**Example:**

In LangChain or RAGas, you can define this as a workflow:

retriever\_chain = Retriever(k=5)

reranker\_chain = ReRanker()

llm\_chain = PromptLLM(prompt\_template)

rag\_chain = retriever\_chain | reranker\_chain | llm\_chain

**🧭 Final Thoughts:**

* RAG is essential for production-grade LLM apps
* Vector databases and chunking help manage large corpora
* LangChain simplifies the pipeline
* RAG-Chains help structure complex flows

Would you like a visual diagram of this process or a working codebase setup

**Day 5**

**What is Advance RAG?**

**What is Query Transformation?**

**What is Query Translation?   
What is Routing?**

**What is Query Construction?**

**Abstractions in Query and its types which is better?**

**What is RAG Fusion and Multi-Query?**

**What is Parallel Query or Fan-Out Retrieval?**

**What is Reciprocal Rank Formula?**

**What is Query decomposition?**

**What is Less Abstraction and More Abstraction in RAG?**

**What is Step back prompting where it performs better?**

**What is HyDE?**

**✅ 1. Query Transformation**

**🧠 What It Is (In-Depth):**

Query Transformation refers to **rewriting or rephrasing the user’s original question** so it aligns better with the way information is stored in your documents or knowledge base. The goal is to **optimize the query for retrieval**, not just generation.

**Why it's important:**

* Users often type vague, short, or ambiguous queries.
* Documents in your knowledge base may use more formal, technical, or different wording.

Query transformation bridges this gap.

**🔍 Use Cases:**

* Enhancing search relevance
* Dealing with natural language variations
* Disambiguating unclear questions

**📘 Example:**

**User Query:**

"What does Apple do?"

This can mean:

* Apple Inc. (the tech company)
* Apple (the fruit)

Let’s assume you’re building a system that serves **business users**, so you want the company context.

**Query Transformation Output:**

“What are the business operations and revenue streams of Apple Inc.?”

**How This Helps:**

* Boosts retrieval accuracy by matching document vocabulary like “revenue”, “products”, “services”.

**✅ 2. Query Translation**

**🧠 What It Is (In-Depth):**

Query Translation changes the **language** or **modality** of a query so that it’s understandable by the target system or data source.

**Types:**

1. **Language Translation:** For multilingual users or documents.
2. **Modality Translation:** Turn natural language into structured formats (e.g., SQL, JSON, GraphQL).

**Why it matters:**  
You might have users querying in Hindi, but your documents are all in English — or users asking questions that must be translated into code/API.

**📘 Example:**

**User Query (French):**

"Quels sont les effets secondaires de l'aspirine ?"

**Translated Query (English):**

“What are the side effects of aspirin?”

→ Now this English query can be used to search an English-only medical database like PubMed.

**Another example – Modality Translation:**

**User Query:**

"Show me vegan restaurants within 5 km"

**Translated SQL Query:**

SELECT \* FROM restaurants WHERE category = 'vegan' AND distance < 5;

**✅ 3. Routing**

**🧠 What It Is (In-Depth):**

Routing in RAG means deciding **which retriever or data source** is best suited for a particular query. You don’t want every query to hit your slow but detailed vector DB.

Instead, classify the query into types:

* Static Knowledge → Vector DB
* Real-Time Info → Web API
* Structured Data → SQL DB

**Why?**  
Improves speed, reduces cost, increases precision.

**📘 Example:**

**User Query:**

“Who won the cricket World Cup 2023?”

This is **time-sensitive**, so route to:

* ✅ Web Search Retriever (not vector DB)

Another Query:

“Explain reverse swing in cricket”

This is **timeless**, so route to:

* ✅ Vector Database of cricket articles

**✅ 4. Query Construction**

**🧠 What It Is (In-Depth):**

It involves **creating multiple well-formed, focused sub-queries** from a general or vague user query. This improves **retrieval coverage** by targeting specific subtopics.

**Why important:**

* Many queries are **compound** in nature.
* Better to answer multiple **atomic** sub-questions, then fuse answers.

**📘 Example:**

**User Query:**

“Tell me about the Mars Mission”

**Constructed Sub-Queries:**

1. “When did the Mars mission start?”
2. “What spacecraft is used in the Mars mission?”
3. “What are the objectives of the Mars mission?”

→ You retrieve more **relevant documents** per sub-topic, reducing chances of missing key info.

**✅ 5. Query Abstraction**

**🧠 What It Is (In-Depth):**

Query abstraction generalizes or specializes a query to **change its scope or specificity** for better retrieval.

**Two types:**

* **Generalization (Abstracting):** From specific to broad
* **Specialization:** From broad to specific

This helps **adjust the granularity** of the answer based on user level.

**📘 Example:**

**Original Query:**

“How to use zip() function in Python?”

**Abstracted Version:**

“Working with iterators in Python”

**Why abstract?**

* Broader articles might explain zip along with other functions like map, filter, helping user understand **context**.

**Or specialized:**

“How does zip() handle uneven iterables in Python?”

**✅ 6. Multi-Query & RAG Fusion**

**🧠 What It Is (In-Depth):**

You take a query and generate **several semantically similar versions**, then retrieve documents using each one. Finally, you **fuse** the results into a coherent answer.

Fusion can be done using scoring methods like **Reciprocal Rank Fusion (RRF)**.

**📘 Example:**

**Original Query:**

“How does quantum computing work?”

**Multi-Queries:**

* “Explain quantum principles in computing”
* “What are qubits?”
* “How does entanglement aid quantum computation?”

**Fusion:**

* Get top documents for each, merge them.
* Use RRF to rank documents and avoid duplicates.

**✅ 7. Parallel Query (Fan-Out Retrieval)**

**🧠 What It Is (In-Depth):**

You send the **same query to multiple retrievers at once**, often connected to different types of data (e.g., public web, vector DB, PDFs, SQL).

**Goal:** Maximize **coverage**, **robustness**, and **diversity** of knowledge.

**📘 Example:**

**Query:**

“Side effects of caffeine”

Sent to:

* ✅ Vector DB of research papers
* ✅ Healthline articles (public web)
* ✅ Reddit discussions (social insights)

→ Merge all retrieved documents for final context.

**✅ 8. Reciprocal Rank Formula (RRF)**

**🧠 What It Is (In-Depth):**

A way to **rank documents retrieved from different sources or queries**. It assigns higher scores to documents that appear near the top across multiple result lists.

**Formula:**

Score = 1 / (k + rank)

Lower ranks → higher score. k is a small constant (e.g., 60).

**📘 Example:**

**Doc A:**

* Appears in Rank 1 of Query 1 → Score = 1 / (60 + 1) = 0.016
* Appears in Rank 2 of Query 2 → Score = 1 / (60 + 2) = 0.0161
* Total Score = 0.0321

This score is higher than a doc that only appears once, lower ranked.

**✅ 9. Query Decomposition**

**🧠 What It Is (In-Depth):**

When the original query includes **multiple topics or tasks**, break it into smaller pieces. Each one is retrieved separately and answered independently, then combined.

This is especially important for **multi-hop reasoning**.

**📘 Example:**

**User Query:**

“How did World War 2 begin, and how did it impact the UN?”

Decomposed into:

1. “What were the causes of World War 2?”
2. “What was the outcome of World War 2?”
3. “How did World War 2 lead to the creation of the UN?”

Each question focuses on a **different aspect** and allows the retriever to target **more precise documents**.

**1. Less Abstraction vs More Abstraction in RAG**

**Definition:**

In RAG, **abstraction** refers to the level of generalization or specificity in the **user's query** or the **representation of information**. It impacts how the retriever fetches documents and how the generator responds.

**🔹 Less Abstraction (Low Abstraction):**

* **Definition**: The query or task is **very specific**, often closely tied to surface-level details or entities.
* **Use case**: When precise, factual information is required (e.g., names, dates, stats).
* **Behavior**: Focuses on retrieving **highly relevant, narrow documents**.

**🧠 Example:**

Query: *“What is the birth date of Albert Einstein?”*

* This is a **low abstraction query** because it asks for a **concrete fact**.
* The retriever should find exact match documents like Wikipedia or a biography page with the birth date.

**📘 Retrieved Doc:** “Albert Einstein was born on March 14, 1879, in Ulm, Germany.”

**🔹 More Abstraction (High Abstraction):**

* **Definition**: The query is **more general, conceptual, or high-level**, often needing reasoning or synthesis.
* **Use case**: Ideal for open-ended or **opinion-based questions**, explanations, or summarization.
* **Behavior**: Retrieves **diverse documents** that may cover concepts broadly.

**🧠 Example:**

Query: *“What factors contributed to Einstein’s success as a scientist?”*

* This requires synthesizing ideas about personality traits, education, historical context.
* Retriever might collect documents on Einstein’s life, theory of relativity, his habits, etc.

**📘 Retrieved Docs:**

* “Einstein was known for his deep curiosity and persistence…”
* “His time at the Swiss Patent Office gave him mental space to develop theories…”

**✅ Summary Table:**

| **Aspect** | **Less Abstraction** | **More Abstraction** |
| --- | --- | --- |
| Focus | Specific fact/entity | General concept/idea |
| Retrieval | Narrow, targeted | Broad, diverse |
| Task Example | Fact retrieval | Explanation, summarization |
| RAG Behavior | Precise grounding | Creative synthesis + broad evidence |

**2. Step-Back Prompting**

**Definition:**

**Step-back prompting** is a technique where, **instead of directly answering the main question**, the system is **prompted to first answer a broader or simpler supporting question**, and then use that to answer the main question.

It’s a **reasoning strategy** used in generation and sometimes retrieval.

**When It Works Better:**

* When the **original query is too complex** or **ill-posed**.
* When multi-step reasoning is needed.
* When there's **a need to abstract or generalize** before answering.

**🧠 Example:**

Original Query: *“How did the invention of the printing press contribute to the French Revolution?”*

This is **too specific** and might miss useful documents if used directly.

**🔁 Step-Back Prompting Flow:**

1. **Step Back Prompt**:  
   *“What were the broad societal effects of the printing press?”*
2. **Get answer**:  
   “The printing press enabled mass communication, spread of political ideas, increased literacy, and public discourse.”
3. **Use that answer to reason about** the original question.
4. **Final Answer**:  
   “The printing press facilitated the dissemination of Enlightenment ideas, which fueled revolutionary sentiment in France.”

**✅ Why it works:**

* Helps the model avoid getting “stuck” on overly narrow paths.
* Allows better retrieval grounding by expanding context.
* Encourages **multi-hop reasoning** in the generation phase.

**3. HyDE (Hypothetical Document Embeddings)**

**Definition:**

**HyDE** is a RAG technique where **instead of directly retrieving documents for a query**, we **first generate a hypothetical answer (or document)** and **embed that** to perform retrieval.

HyDE = **Hypothetical Document Embedding**

**🔍 Motivation:**

* Queries are sometimes **too short or ambiguous**.
* Traditional dense retrieval using the query embedding may not work well.
* But if we **generate a richer hypothetical doc**, we get a better semantic embedding for retrieval.

**Step-by-Step HyDE Workflow:**

1. **Input query**:  
   *“How does a black hole form?”*
2. **Generate a hypothetical answer**:  
   (Using an LLM like GPT)  
   *“A black hole forms when a massive star collapses under its own gravity after exhausting its nuclear fuel…”*
3. **Embed the generated text** using a vector embedding model.
4. **Use that embedding** to perform vector search in your document database (e.g., via Qdrant, FAISS, Pinecone).
5. **Retrieve** the real documents **similar to the hypothetical answer**.
6. **Use retrieved docs + original question** → final answer generation.

**🔁 Why it works:**

* Richer context and semantics than short queries.
* Works especially well for **scientific or abstract questions** where retrieval from the raw query may fail.
* Bridges gap between sparse queries and dense document embeddings.

**🧠 Example:**

Query: *“Effects of climate change on marine life”*

* The query is short and abstract.
* HyDE generates:  
  *“Climate change leads to ocean warming, acidification, and deoxygenation, which negatively affect marine ecosystems such as coral reefs and fisheries…”*
* This richer context leads to better document matches in the vector DB.

**✅ Summary of All Concepts:**

| **Concept** | **Key Idea** | **When to Use** |
| --- | --- | --- |
| Less Abstraction | Specific queries, factual, low-level detail | Fact-based answers, entity search |
| More Abstraction | General, conceptual queries, higher-level synthesis | Open-ended questions, explanations |
| Step-Back Prompting | Reformulate question into a broader or simpler one first | Complex/multi-hop questions, hard retrieval |
| HyDE | Generate a hypothetical answer first, then embed and retrieve based on it | Short, ambiguous, or abstract queries |

**DAY 6**

**What is Query Routing?  
Explain Different types of Query Routing?**

**How Router works?**

**What is Logical Routing?**

**How Logical Routing Works?**

**What is Semantic Routing?**

**How Semantic Routing Works?**

**What is LLM as Judge?**

**What is Lang-Graph?**

**What is Graph Databases?**

**What is Vector Databases?**

**Difference between Graph and Vector Database?**

**Example of Graph and Vector Database?**

**Which is better Graph database or Vector Database? How?**

**How can use Both?**

**🔁 1. What is Query Routing?**

**Query Routing** is the process of analyzing an input query and determining **which retriever, model, or tool** should handle it. In RAG (Retrieval-Augmented Generation), this helps in selecting the **best source of information** for the user's intent.

**💡 Why do we need Query Routing?**

Imagine you have:

* A legal document retriever
* A medical retriever
* A financial document retriever

If a user asks *“What are the symptoms of diabetes?”*, sending it to the legal retriever would be useless. **Query Routing** directs it to the **medical retriever**.

**🔍 Real-World Analogy:**

Think of a **smart receptionist at a hospital**:

* You ask about a stomach ache → routed to the Gastroenterologist
* You ask about a bill → routed to the Billing Department
* You ask about appointment timings → routed to the Reception

Same applies to RAG: **correct route = accurate answer**.

**🔢 2. Types of Query Routing**

There are **two primary approaches**:

**📘 A. Logical Routing**

Logical routing uses **hardcoded rules or keywords** to decide how to route a query.

**✅ Characteristics:**

* Fast
* Transparent
* Rule-based

**❌ Limitations:**

* Brittle (doesn’t handle paraphrased or vague queries)
* Requires frequent updates if query patterns change

**🔧 Example (Pseudo-code):**

def route\_query(query):

if "define" in query.lower():

return "dictionary\_retriever"

elif "symptom" in query or "treatment" in query:

return "medical\_retriever"

elif "law" in query or "section" in query:

return "legal\_retriever"

else:

return "general\_retriever"

**🧪 Output:**

* Input: *“Define inflation”* → Routed to: dictionary retriever
* Input: *“Section 144 explanation”* → Routed to: legal retriever

**🧠 B. Semantic Routing**

**Semantic routing** analyzes the **meaning** or **intent** of the query using embeddings or LLMs. It’s more **intelligent** and **context-aware**.

**✅ Characteristics:**

* Understands paraphrased, ambiguous, or vague queries
* Adapts to user intent

**❌ Limitations:**

* Requires more compute (LLMs/Embeddings)
* Needs training or fine-tuning if done on-prem

**🧪 Example with Embeddings:**

from langchain.embeddings import OpenAIEmbeddings

from sklearn.metrics.pairwise import cosine\_similarity

query\_embedding = embed("What are COVID symptoms?")

retriever\_embeddings = {

"medical\_retriever": embed("Medical queries"),

"legal\_retriever": embed("Legal queries"),

"finance\_retriever": embed("Finance queries"),

}

# Choose the retriever with the highest similarity

best\_match = max(retriever\_embeddings.items(), key=lambda x: cosine\_similarity([query\_embedding], [x[1]]))

**🧠 Output:**

Even if query is: *“Tell me how to know if someone has diabetes?”*, it correctly matches **medical retriever** due to semantic similarity.

**🧮 3. How Routers Work**

Routers are central decision-makers in the RAG pipeline. They evaluate the query, apply logic or embedding matching, and forward the query to the selected tool.

**🔄 Components in LangChain:**

* **Router Chain**: Uses an LLM to classify input
* **MultiRetrievalQAChain**: Supports multiple retrievers with LLM routing
* **ToolRouterChain**: For agent-based workflows

**🔄 LangChain LLM Router Example:**

from langchain.chains.router import MultiRetrievalQAChain

router\_chain = MultiRetrievalQAChain.from\_retrievers(

llm=OpenAI(),

retriever\_names=["medical", "legal", "finance"],

retrievers=[medical\_retriever, legal\_retriever, finance\_retriever]

)

**📐 4. What is Logical Routing?**

Logical routing = **rule-based conditional routing** using keyword patterns, flags, or regex.

**⚙️ Internals:**

* Keyword matching
* Regex
* If-else conditions
* Domain-specific rules

**🧪 Example:**

If query contains:

* “how many”, “when”, or “where” → factual retriever
* “define”, “what is” → definition-based retriever

def logical\_router(query):

if "define" in query or "what is" in query:

return "DefinitionRetriever"

elif "steps" in query:

return "ProcedureRetriever"

return "DefaultRetriever"

**💬 5. What is Semantic Routing?**

Semantic Routing uses **natural language understanding**, embeddings, or LLMs to determine the most relevant retriever or tool.

Unlike Logical Routing, it can handle:

* Complex syntax
* Paraphrasing
* Multi-intent queries

**🔎 Example:**

**Query**: *“Tell me how to get rid of sore throat naturally”*

Even though it has no keywords like “symptom” or “disease”, an embedding-based router will understand it belongs to the **medical** domain.

**⚖️ 6. What is LLM as Judge?**

LLM-as-Judge is when the **LLM decides**:

* Which retriever/tool is most suitable
* Which output (from multiple candidates) is best

**🧠 Example: RAG Fusion with LLM-as-Judge**

Let’s say you retrieved 3 responses from 3 sources. Ask the LLM:

Query: "What causes inflation?"

Candidate 1: Explanation from Wikipedia

Candidate 2: IMF article

Candidate 3: Textbook definition

Prompt: Which of these gives the most accurate, concise answer?

LLM will output:

"Candidate 2 is the best answer due to its economic clarity and updated info."

This is **LLM-as-a-judge**.

**🧭 7. What is LangGraph?**

LangGraph is a **graph-based execution framework** for chaining LLM agents, retrievers, and tools.

Think of it like building a **flowchart** of how your AI system should behave. Each **node** can be:

* An LLM call
* A retriever
* A tool (like calculator, search)

Each **edge** defines how data flows.

**📘 Example:**

graph TD

Start --> Router

Router -->|Medical| MedicalRetriever

Router -->|Legal| LegalRetriever

MedicalRetriever --> Answer

LegalRetriever --> Answer

This is **LangGraph in visual form**. Each node is a step, and you can control flow using conditions or results.

**🧠 8. What is a Graph Database?**

A **Graph Database** represents data as **nodes (entities)** and **edges (relationships)**.

**🧩 Example:**

(Alice)-[:FRIEND\_WITH]->(Bob)

(Bob)-[:WORKS\_AT]->(Google)

Great for:

* Social networks
* Knowledge graphs
* Recommendation engines

**🔧 Example using Neo4j:**

MATCH (p:Person)-[:FRIEND\_WITH]->(f:Person)

WHERE p.name = "Alice"

RETURN f.name

**🧠 9. What is a Vector Database?**

A **Vector Database** stores **high-dimensional numerical vectors** representing text, images, or audio.

Used for **semantic similarity search**.

**🔍 Example:**

query = embed("best places to visit in Italy")

results = qdrant.search(vector=query, top\_k=5)

Vector DB returns the most **similar documents**, regardless of keywords.

**🆚 10. Graph vs. Vector Database**

| **Feature** | **Graph DB** | **Vector DB** |
| --- | --- | --- |
| Data Structure | Nodes and Edges | Embedding vectors |
| Purpose | Relationship querying | Semantic similarity |
| Query Language | Cypher, Gremlin | k-NN, cosine similarity |
| Example | Neo4j, TigerGraph | Qdrant, Pinecone, FAISS |
| Use Case | Multi-hop reasoning | RAG, semantic search |

**🔁 11. Can You Use Both Together?**

Yes — using **Graph for relationships** and **Vector DB for semantic similarity** is a **powerful combination**.

**🧪 Example in a BioMed RAG System:**

1. Use **Qdrant** to retrieve documents about a disease.
2. Extract entities like proteins, genes, diseases.
3. Store them in **Neo4j** as nodes and relationships.
4. Use Graph DB for multi-hop: *“What gene connects this disease to another?”*

This combo allows:

* Retrieval by **meaning** (Vector DB)
* Reasoning via **relations** (Graph DB)

**🚀 Conclusion**

| **Concept** | **Use for** |
| --- | --- |
| Logical Routing | Fast, rule-based task separation |
| Semantic Routing | Robust intent understanding |
| LLM-as-Judge | Quality filtering, scoring, decision-making |
| LangGraph | Multi-step graph-based workflows |
| Vector DB | Semantic retrieval |
| Graph DB | Reasoning, multi-hop inference |
| Both Combined | Semantic + structural AI |

**DAY 7**

**What is Knowledge Graph?**

**How Knowledge Graph is used in LLM’s and RAG?**

**What type of Database it is?**

**How to represent data in Knowledge Graph?**

**What is better Vector Embeddings or Knowledge Graph?**

**What is neo4j?**

**How to use explain everything about neo4j?**

**What is CypherQL?**

**Explain me Neo4j with example in detail in depth with text example along with its explain and along with query example?**

**MERGE Vs CREATE in KG?**

**How to convert PDF document into Knowledge Graph? Explain the entire process?**

**What is mem0?**

**How I can use mem0 to make ai memory?**

**Use cases of Knowledge Graph?**

**🔍 1. What is a Knowledge Graph?**

A **Knowledge Graph (KG)** is not just a data structure — it is a **semantic network** designed to represent real-world entities and the **rich, interrelated context** between them. Think of it as a **machine-readable scaffold of knowledge**, which imitates how humans understand the world — not through isolated facts, but through **interconnected ideas**.

At its core, a Knowledge Graph consists of **entities** (such as people, places, concepts, events) and **relations** between them. But it’s not limited to this — entities can also have **attributes** (properties or metadata), and the **relations themselves** can carry properties, weights, and confidence scores.

Unlike traditional databases that store data in **tables** or even documents, KGs are built upon **graph theory**, and follow a **triple-based structure**:  
<Subject> — <Predicate> — <Object>  
This forms a **directed edge** in a graph from one node (entity) to another, via a labeled relationship.

**📚 Conceptual Expansion:**

Imagine how your brain recalls everything related to "Albert Einstein" — not just the year he was born, but also that he developed the theory of relativity, was a physicist, won the Nobel Prize, and inspired modern quantum theory. This web of facts is exactly how a **KG is structured**: information is no longer flat but forms a **semantic web** that can be navigated, queried, and reasoned upon.

**🧠 2. How is a Knowledge Graph Used in LLMs and RAG?**

**In LLMs:**

Modern LLMs, while powerful, operate as **probabilistic pattern generators**. They lack a **persistent memory** of verified facts or consistent knowledge reasoning. This is where Knowledge Graphs become invaluable.

**🧬 Role of KG in LLMs:**

1. **Factual Grounding**: KGs help LLMs **anchor generations in reality**. Instead of hallucinating facts, they can pull verified relationships (e.g., "Marie Curie won 2 Nobel Prizes" → validated in KG).
2. **Entity Resolution and Linking**: KGs help models understand **which “Apple” you mean** — the company or the fruit? Based on your query’s context and KG’s graph traversal, it connects the input to the correct entity.
3. **Prompt Augmentation**: KGs can dynamically augment prompts with relevant relationships — giving LLMs **semantic context** that goes beyond the input.

**In RAG (Retrieval-Augmented Generation):**

**What is RAG?**

RAG is a framework where you **retrieve relevant external information** (usually via vector search), and then **generate an answer using an LLM** over this context.

**Where do KGs fit?**

While vector search retrieves semantically similar text (fuzzy matching), KGs excel at **logical, symbolic, and deterministic retrieval**. Here's how they complement RAG:

* **Pre-Retrieval Filtering**: Use the KG to find a precise topic (e.g., all people connected to "machine learning" via "researches"), and then fetch documents.
* **Post-Retrieval Re-ranking**: You can filter or re-rank semantic results using entity types or relationships defined in the KG.
* **Context Graphs**: Inject dynamic knowledge into LLMs at runtime.

**Example Hybrid Use Case:**

* Question: “Who are the living Nobel laureates in Physics?”
* KG: Filters laureates who are alive and in Physics.
* Vector search: Fetches articles or biographies of these people.
* LLM: Generates final natural-language answer using both sources.

**🗃️ 3. What Type of Database is a Knowledge Graph?**

A Knowledge Graph is typically implemented on top of a **Graph Database**, which is a type of **NoSQL database** optimized to store and query data using **graph structures** — **nodes**, **edges**, and **properties**.

These databases are designed to **natively traverse relationships**, which is extremely expensive in relational databases (JOIN hell).

**Characteristics:**

* Schema-optional: You can add new types of nodes/relationships on the fly.
* ACID-compliant: Especially Neo4j, suitable for transactional applications.
* Optimized for graph algorithms: shortest path, centrality, etc.

**🧾 4. How to Represent Data in a Knowledge Graph?**

**Core Model: Triples**

The foundation of KGs is RDF-style triples:

<Albert Einstein> — <fieldOfStudy> — <Physics>

<Physics> — <partOf> — <Science>

But beyond RDF, modern knowledge graphs allow **rich modeling**, including:

* **Typed Entities**: (:Person {name: "Einstein"})
* **Typed Relations**: [:DISCOVERED {year: 1915}]
* **Nested Graphs**: Some graphs support **meta-relationships** (e.g., "The relationship X was mentioned in document Y").
* **Properties**: Attributes attached to both nodes and edges

This makes it possible to query **why** a connection exists, not just that it does.

**⚔️ 5. What is Better: Vector Embeddings or Knowledge Graph?**

This is not a competition; it’s a **complementarity**.

**Vector Embeddings:**

* Represent text/images/code as high-dimensional points
* Enable **semantic retrieval**, like “find me something similar to this”
* Excellent for **fuzzy, soft** similarity matching
* Examples: FAISS, Qdrant, Pinecone

**Knowledge Graphs:**

* Represent **hard relationships**, explainable and verifiable
* Support **logic, inference**, and **reasoning**
* Better for **structured data** and symbolic AI

**Hybrid Systems Combine:**

* Use embeddings to find **contextually similar** content
* Use KGs to **filter**, **validate**, or **extend** the search space

**🧰 6. What is Neo4j?**

**Neo4j** is the most widely-used **native graph database**, designed specifically for storing large-scale, complex graphs.

What makes Neo4j powerful is that it:

* Stores data as **property graphs**
* Uses a declarative, powerful language called **Cypher**
* Supports **traversals** and **graph algorithms** natively
* Is used by NASA, eBay, Microsoft, and more

You can use Neo4j to:

* Build knowledge graphs
* Model complex networks (e.g., telecom, logistics)
* Detect fraud (through transaction link analysis)
* Implement recommendations (via mutual relationships)

**📜 7. What is CypherQL?**

**Cypher** is to Neo4j what **SQL** is to MySQL — a language for **querying and mutating graph data**.

**Syntax:**

* Looks like ASCII art for graphs:

(MATCH)-[:RELATION]->(TO)

* Declarative: you specify **what** you want, not how to fetch it
* Supports MATCH, CREATE, MERGE, SET, RETURN, DELETE, etc.

**🧪 8. Neo4j Deep Example: Complete Walkthrough**

Let’s construct a rich domain: **Scientific Discovery Knowledge Graph**

**👨‍🔬 Entities:**

* Person: Einstein, Newton
* Theories: Relativity, Gravity
* Awards: Nobel Prize
* Institutions: Princeton, Cambridge

**1. Create Nodes:**

CREATE (einstein:Scientist {name: "Albert Einstein", born: 1879}),

(newton:Scientist {name: "Isaac Newton", born: 1643}),

(relativity:Theory {name: "Relativity"}),

(gravity:Theory {name: "Gravity"}),

(nobel:Award {name: "Nobel Prize"}),

(princeton:Institution {name: "Princeton"}),

(cambridge:Institution {name: "Cambridge"})

**2. Create Relationships:**

MATCH (einstein:Scientist {name: "Albert Einstein"}),

(newton:Scientist {name: "Isaac Newton"}),

(relativity:Theory {name: "Relativity"}),

(gravity:Theory {name: "Gravity"}),

(nobel:Award {name: "Nobel Prize"}),

(princeton:Institution {name: "Princeton"}),

(cambridge:Institution {name: "Cambridge"})

CREATE (einstein)-[:DEVELOPED]->(relativity),

(newton)-[:DISCOVERED]->(gravity),

(einstein)-[:WON]->(nobel),

(einstein)-[:AFFILIATED\_WITH]->(princeton),

(newton)-[:AFFILIATED\_WITH]->(cambridge)

**3. Complex Query: Who developed a theory and won a Nobel Prize?**

MATCH (s:Scientist)-[:DEVELOPED|DISCOVERED]->(t:Theory),

(s)-[:WON]->(a:Award)

RETURN s.name AS scientist, t.name AS theory, a.name AS award

**Output:**

Albert Einstein | Relativity | Nobel Prize

**🧪 Three In-Depth Real-World Examples**

**📖 Example 1: Movie Recommendation KG**

**Entities**: Actor, Movie, Genre, Director  
**Relationships**: ACTED\_IN, DIRECTED\_BY, BELONGS\_TO

CREATE (leo:Actor {name: "Leonardo DiCaprio"}),

(inception:Movie {title: "Inception"}),

(nolan:Director {name: "Christopher Nolan"}),

(sciFi:Genre {name: "Science Fiction"})

CREATE (leo)-[:ACTED\_IN]->(inception),

(nolan)-[:DIRECTED]->(inception),

(inception)-[:BELONGS\_TO]->(sciFi)

**Query:**

Find all sci-fi movies Leonardo DiCaprio acted in:

MATCH (a:Actor {name: "Leonardo DiCaprio"})-[:ACTED\_IN]->(m:Movie)-[:BELONGS\_TO]->(:Genre {name: "Science Fiction"})

RETURN m.title

**🧬 Example 2: Biomedical KG**

**Entities**: Disease, Gene, Drug, Symptom  
**Relationships**: ASSOCIATED\_WITH, TREATS, CAUSES

CREATE (covid:Disease {name: "COVID-19"}),

(fever:Symptom {name: "Fever"}),

(remdesivir:Drug {name: "Remdesivir"}),

(ACE2:Gene {name: "ACE2"})

CREATE (covid)-[:HAS\_SYMPTOM]->(fever),

(remdesivir)-[:TREATS]->(covid),

(ACE2)-[:ASSOCIATED\_WITH]->(covid)

**🧭 Example 3: Travel Route Planning KG**

**Entities**: City, Country, Airline  
**Relationships**: FLIES\_TO, LOCATED\_IN

CREATE (mumbai:City {name: "Mumbai"}),

(dubai:City {name: "Dubai"}),

(uae:Country {name: "UAE"}),

(emirates:Airline {name: "Emirates"})

CREATE (mumbai)-[:FLIES\_TO]->(dubai),

(dubai)-[:LOCATED\_IN]->(uae),

(emirates)-[:OPERATES]->(mumbai),

(emirates)-[:OPERATES]->(dubai)

**1. MERGE vs CREATE in Knowledge Graphs (Neo4j context)**

In Neo4j, which is one of the most popular graph databases used to store and query knowledge graphs, two primary commands are used to create nodes and relationships: CREATE and MERGE. While they seem similar at first glance, they behave very differently, and understanding these differences is critical when working with data in a knowledge graph.

**CREATE**

* The CREATE clause is used to **always create a new node or relationship**, regardless of whether something identical already exists.
* It does **not check for duplication**. Every time you run CREATE, a new entity is added.

**Example:**

CREATE (p:Person {name: "Alice", age: 30})

If this query is run twice, two nodes with the same name and age will be created.

**MERGE**

* MERGE is **like a combination of MATCH and CREATE**.
* It first **tries to find** a pattern in the graph.
* If it exists, it returns it.
* If it doesn’t exist, it **creates it**.
* This is useful when you want to avoid duplication and maintain **idempotency** (running the same command multiple times won’t change the graph after the first run).

**Example:**

MERGE (p:Person {name: "Alice"})

ON CREATE SET p.age = 30

If a node with the name "Alice" already exists, nothing is changed. If not, it creates one and sets age.

**Deeper Understanding:**

* Think of CREATE as “force insert” and MERGE as “find-or-insert.”
* MERGE is costlier in performance due to the lookup phase but is more semantically correct for ensuring data integrity.

**Real-world Analogy:**

* Imagine a university database. MERGE is like checking if a student already exists before enrolling them. CREATE is like blindly enrolling someone regardless of whether they’re already on the list.

**2. How to Convert a PDF Document into a Knowledge Graph**

This is a complex, multi-stage process that involves various fields of NLP, information extraction, and graph database modeling.

**Step-by-Step Process:**

**Step 1: Extract Text from PDF**

* Use libraries like PyMuPDF, pdfplumber, or PDFMiner in Python to extract text.
* Handle multi-column layouts, headers, footers, and images if needed.

import fitz # PyMuPDF

doc = fitz.open("example.pdf")

text = ""

for page in doc:

text += page.get\_text()

**Step 2: Perform Information Extraction (NER and Relation Extraction)**

* Use NLP libraries like **spaCy**, **transformers**, **Haystack**, or **LLMs**.
* Identify **entities** (like people, organizations, places) using **Named Entity Recognition**.
* Then use **Relation Extraction** or **OpenIE** models to understand how these entities relate.

# Example: Using spaCy

import spacy

nlp = spacy.load("en\_core\_web\_sm")

doc = nlp(text)

for ent in doc.ents:

print(ent.text, ent.label\_)

**Step 3: Normalize and Structure the Data**

* Convert raw entities and relations into triples of the form:

(Entity1) -[RELATION]-> (Entity2)

e.g., ("Alan Turing") -[FOUNDED]-> ("Computer Science")

**Step 4: Store in a Graph Database (Neo4j)**

* Use py2neo, Neo4j Python driver, or Cypher queries to insert the nodes and relationships.

from py2neo import Graph, Node, Relationship

graph = Graph("bolt://localhost:7687", auth=("neo4j", "password"))

person = Node("Person", name="Alan Turing")

field = Node("Field", name="Computer Science")

relation = Relationship(person, "FOUNDED", field)

graph.create(relation)

**Step 5: Query and Visualize**

* Use Neo4j Browser or Bloom to visualize and query relationships in natural ways.

**Real-World Example 1:**

**Legal Document** → Extract sections, cases, legal terms, precedent relationships.

**Real-World Example 2:**

**Research Paper** → Extract authors, institutions, keywords, methods, and their interactions.

**Real-World Example 3:**

**Product Manual** → Extract part names, dependencies, assembly instructions, and troubleshooting steps.

**3. What is mem0?**

mem0 (Memory Zero) is an experimental and research-based **AI memory framework** designed to allow LLMs and AI agents to **retain and recall context over long timeframes**. It’s still in development or academic stages in most cases, but the **general idea** is to make AI capable of long-term memory, unlike the usual session-limited LLMs.

**Key Concepts:**

* **Persistent memory**: Stores facts, history, and interactions.
* **Embeddings + Retrieval**: Uses a vector store to find past relevant memories.
* **Symbolic memory**: Stores structured knowledge or KG-based memory.

**Using mem0 (or similar frameworks):**

* Usually integrated with frameworks like **LangChain**, **OpenAgents**, or **MemGPT**.
* Memories are stored either in **vector databases (Qdrant, Pinecone)** or **graph stores (Neo4j, Memgraph)**.
* These memories are then **retrieved contextually** and injected into prompts to simulate continuity.

**Use Case Example:**

Imagine a personal AI assistant that remembers your dietary preferences, vacation history, and preferred learning style — this memory is powered by mem0-style architectures.

**4. Use Cases of Knowledge Graphs (in Depth)**

**A. Search and Recommendation Engines**

* **Google Knowledge Graph**: Enhances search results with structured information panels.
* Recommends related products, movies, music by traversing connections between entities.

**Example**: Netflix uses a graph of movies, genres, actors, and user behavior to recommend content.

**B. Healthcare and Biomedical Research**

* Connects drugs, genes, diseases, symptoms, and treatments.
* Used in **drug discovery**, side effect analysis, clinical trial mapping.

**Example**: OpenBioLink or Hetionet creates knowledge graphs connecting over 50 different biomedical datasets.

**C. Enterprise Knowledge Management**

* Unifies documents, wikis, email conversations, databases into a semantic graph.
* Helps in onboarding, troubleshooting, compliance.

**Example**: Airbus or Siemens uses KGs to map parts, manuals, suppliers, engineers, and incidents for rapid diagnostics.

**D. Fraud Detection and Risk Analytics**

* Detect suspicious activities by analyzing graph-based connections between accounts, transactions, devices, and IPs.

**Example**: A bank notices a ring of accounts connected to the same IP or device with strange transaction patterns.

**E. AI Assistants and Memory (like memGPT)**

* Provide long-term, contextual memory using a hybrid of vector and symbolic (graph) memory.

**Summary**

You can see that Knowledge Graphs aren't just data structures — they're a **semantic layer for intelligence**. Whether you're building an intelligent search engine, a question-answering system, or a long-term memory for AI, KGs help make data interconnected and machine-understandable.

**DAY 8**

**What is Tracing?**

**What is Tracing AI Apps?**

**What is Grfana, Prometheus, Loki?**

**What is LangChain and LangGraph?**

**What is Langsmith?**

**How to monitor Ai Apps with Langsmith?**

**What us Langfuse?**

**🔍 What is Tracing?**

**Tracing** refers to the process of following a request or task as it flows through various components in a system—especially useful in **complex applications** where many operations or services are involved. It helps developers **see everything that happened** during a specific interaction, like a magnifying glass for your backend or LLM workflow.

Imagine you're running a food delivery app. A customer places an order:

* That request goes from the frontend to the backend.
* The backend checks the restaurant’s availability.
* Then calculates delivery ETA.
* Then routes the task to a delivery agent.  
  If there’s a failure or delay, **tracing** helps you identify exactly where the breakdown occurred. The same concept applies to **AI apps**.

When applied to AI, especially LLM applications, tracing is even more vital. Why? Because these systems aren't deterministic. You can’t always guarantee the same output for the same input. By tracing:

* You can log what the model received as input.
* What retrieval outputs were fed into it (in RAG).
* How long each step took.
* What each intermediate result looked like.

**📘 Example 1: Tracing in a RAG system**

Suppose a user asks your AI app: *“What is contrastive learning?”*

A trace might look like this:

1. You log the original question.
2. You generate embeddings using text-embedding-ada-002.
3. You retrieve top 5 relevant chunks from Qdrant.
4. You format a prompt with those chunks and send to GPT-4.
5. GPT-4 returns the answer in 1.4 seconds.
6. You log the final answer, the token usage, and cost.

Now, if the output is wrong or hallucinated, you can look into this trace and say, “Oh! The retrieved docs were about *transfer learning*, not *contrastive learning*. That’s why the model failed.” Without tracing, you would be flying blind.

**📘 Example 2: Tracing performance**

Your chatbot suddenly slows down. You run a trace on a few queries and realize:

* Token generation by GPT-4 has a consistent latency of 2s.
* But retrieval from Pinecone is taking 4s instead of 200ms (as it usually does).  
  Now you know that your vector store is under heavy load, and that’s your bottleneck—not the LLM.

**🤖 What is Tracing AI Apps?**

While “tracing” is a general concept in distributed systems, **tracing AI apps** specifically refers to logging and analyzing **LLM workflows**, **prompt chains**, **tool usage**, and **reasoning steps** involved in a single inference or conversation.

AI apps, especially those using LangChain, Hugging Face, or custom prompt chains, are not monolithic. A simple query might go through:

1. Input sanitation
2. Embedding generation
3. Document retrieval (RAG)
4. Prompt composition
5. LLM inference
6. Post-processing
7. Output delivery

Each of these steps might interact with external APIs, internal databases, or run custom logic.

**📘 Example 1: Debugging a hallucination in a legal AI assistant**

A user asks: *“Can I terminate my rental contract early?”*

You trace the request and see:

* The embedding step ran fine.
* The retrieval step returned unrelated laws from a different country.
* The prompt was constructed with these irrelevant documents.
* The LLM answered incorrectly based on bad info.

The trace lets you identify where the problem happened (bad retrieval), and now you can fix the chunking strategy or improve metadata filtering.

**📘 Example 2: Evaluating Tool Use in an Agent**

You build a LangChain agent that can call a calculator, Wolfram Alpha, or search API based on the question. For the question *“What’s the square root of 676?”*, the agent unnecessarily calls Google Search instead of the calculator.

By tracing, you find:

* The reasoning step misclassified the query.
* The tool routing logic is biased toward search tools.

Now you can fine-tune your router or add explicit instructions.

**📊 What is Grafana, Prometheus, and Loki?**

These three tools form the **open-source observability stack** used for monitoring modern applications. While they’re traditionally used for system monitoring (servers, microservices, etc.), they are now commonly applied to AI applications for performance, cost, and stability metrics.

**🧠 Prometheus**

Prometheus is a **metrics collection and time-series database system**. It scrapes numeric data (like request latency, memory usage, or API call duration) at regular intervals and stores it with timestamps. These metrics can be queried using a powerful language called PromQL.

For AI apps, you can expose metrics like:

* Token count per request
* Embedding model latency
* API call failures
* Memory usage during vector search

**📚 Loki**

Loki is like Prometheus but for **logs**. Instead of metrics, Loki aggregates **unstructured logs** — the console prints and error messages your app generates.

You can query logs like:

* All errors from retrieval step
* Logs with “Rate limit exceeded” from OpenAI
* Last 100 logs containing “Invalid JSON”

**📈 Grafana**

Grafana is a **dashboarding and visualization tool**. It connects to Prometheus, Loki, and many others to show:

* Time series graphs
* Status panels
* Logs
* Alert triggers

**📘 Example 1: Monitoring LangChain-based app**

You build a LangChain app for customer service.

* Prometheus tracks:
  + Total queries served
  + Average LLM latency
  + Token cost per day
* Loki stores logs like:
  + “Invoking tool: search”
  + “Prompt construction failed”
  + “LangChainAgent threw exception”
* Grafana visualizes:
  + A live graph of requests per minute
  + Pie chart of which tools are used the most
  + Alert if latency > 2 seconds

**📘 Example 2: Monitoring Qdrant retrieval**

You have a vector DB-backed RAG app.

* Prometheus tracks Qdrant search duration.
* Loki logs failed queries.
* Grafana alerts you when retrieval returns 0 documents for 3 queries in a row, indicating bad embeddings.

**🧠 What is LangChain?**

LangChain is a powerful **framework** for building AI-powered applications using **large language models (LLMs)**. It abstracts away complex interactions between LLMs, prompts, memory, tools, and vector stores.

Instead of hardcoding everything manually, LangChain lets you:

* Create **Chains** of LLM calls and functions.
* Use **Agents** that dynamically choose which tools to use.
* Add **Memory** to track conversation.
* Integrate **Retrievers** to do RAG-style answering.

**📘 Example 1: Basic RAG Chain**

A user asks: *“How does backpropagation work?”*

Your LangChain flow:

1. Embed query with OpenAI.
2. Retrieve docs from Qdrant.
3. Format prompt like:

“Answer the following question using these docs: {retrieved\_docs}”

1. Send to LLM.
2. Return the response.

This is easier and safer than doing everything manually, and LangChain lets you trace/log/debug this entire flow.

**📘 Example 2: LangChain Agent with Tools**

A user says: *“What’s the weather in Mumbai and convert the temperature to Fahrenheit.”*

LangChain agent:

1. Decides to use the weather API tool.
2. Gets temperature: 32°C.
3. Decides to use the calculator tool to do (32 \* 9/5) + 32.
4. Returns: “It’s 89.6°F in Mumbai.”

LangChain coordinates the reasoning and tool selection with just a few lines of configuration.

**🧭 What is LangGraph?**

LangGraph is like the **LangChain for state machines**. Instead of linear chains or dynamic agents, LangGraph lets you **build LLM workflows as a graph**, with:

* Nodes (steps): e.g., classify, search, respond
* Edges (transitions): logic flows
* Memory: persistent state between turns

This is extremely powerful when you need complex behavior like:

* Looping through clarifications
* Conditional paths
* Multi-agent conversations

**📘 Example 1: Multi-turn Conversational Agent**

You want an assistant that:

* Classifies the user’s intent
* If vague, asks a clarifying question
* If clear, proceeds to respond
* If user confirms, ends

LangGraph lets you build this as a graph:

Intent → [Clarify] → Intent → [Respond] → [End]

Each step is a node running LLM + logic.

**📘 Example 2: Multi-retriever system**

You have 3 domains: finance, healthcare, sports.

* First node classifies topic.
* Then routes to appropriate retriever.
* Then merges retrieved results → LLM → Response.

LangGraph supports parallel paths and merging.

**🔬 What is LangSmith?**

LangSmith is a **full observability and experimentation suite** for LLM applications built with LangChain or similar tools. It lets you:

* Trace every execution
* Compare prompts
* Monitor costs and token usage
* Create test cases and evaluations
* Collaborate in teams

It’s like having a microscope for every LLM query your app handles.

**📘 Example 1: Prompt evaluation**

You’re iterating on a prompt template for your legal advisor bot. You log 100 sample questions through prompt v1 and v2. LangSmith shows:

* v1 has more hallucinations.
* v2 uses fewer tokens and is faster.  
  You switch to v2.

**📘 Example 2: Tool usage audit**

You review traces and notice your agent is calling the calculator tool unnecessarily. You revise your tool routing logic and re-run tests to verify the fix.

**🛠️ What is LangFuse?**

LangFuse is a **self-hosted** alternative to LangSmith that offers:

* Tracing
* Logging
* Analytics
* Token usage stats
* Alerts

It supports integration with LangChain, LlamaIndex, OpenAI SDK, and custom codebases.

**📘 Example 1: Self-hosted Observability**

Your company doesn’t want to use cloud observability for privacy. You host LangFuse locally, and now trace every AI query on your own servers.

You monitor:

* Which prompts are slow
* Where most errors happen
* How many tokens are consumed per user

**📘 Example 2: Team Dashboard**

You run a multi-agent RAG app. Each agent is traced separately in LangFuse. Your team members view graphs and logs to debug failures and measure performance.