Contents

[GENERATIVE AI 3](#_Toc205820494)

[NLP(NATURAL LANGUAGE PROCESSING) 3](#_Toc205820495)

[GENERATIVE AI VS DISCRIMINATIVE(PREDICTIVE) AI 5](#_Toc205820496)

[APPLICATION OF GEN-AI 7](#_Toc205820497)

[GEN-AI LIMITATIONS & CHALLENGES 7](#_Toc205820498)

[LARGE LANGUAGE MODELS(LLMs) 9](#_Toc205820499)

[LLM ARCHITECTURE 10](#_Toc205820500)

[CORE COMPONENTS OF LLMs 10](#_Toc205820501)

[TRANSFORMER ARCHITECTURE 10](#_Toc205820502)

[HOW LLMS WORK 14](#_Toc205820503)

[PROMPTS & TOKENS 16](#_Toc205820504)

[TOTAL TOKENS 17](#_Toc205820505)

[HOW TO USE LLMs 18](#_Toc205820506)

[TEXT SUMMARIZATION 19](#_Toc205820507)

[TYPES OF TEXT SUMMARIZATION 19](#_Toc205820508)

[EXAMPLE 20](#_Toc205820509)

[SENTIMENT ANALYSIS 24](#_Toc205820510)

[PROMT ENGINEERING 25](#_Toc205820511)

[COMPONENTS OF PROMPTS 25](#_Toc205820512)

[PARAMETERS TO CONTROL RESPONSES 25](#_Toc205820513)

[CHAT ML FORMAT 26](#_Toc205820514)

[PromptTemplate vs ChatPromptTemplate 27](#_Toc205820515)

[PROMPTING TECHNIQUES 27](#_Toc205820516)

[ZERO SHOT PROMPTING 27](#_Toc205820517)

[FEW SHOT PROMPTING 28](#_Toc205820518)

[CHAIN OF THOUGHT PROMPTING 31](#_Toc205820519)

[TREE OF THOUGHT(TOT) PROMPTING 33](#_Toc205820520)

[SELF-CONSISTENCY PROMPTING 36](#_Toc205820521)

[META-PROMPTING 36](#_Toc205820522)

[PROMPT CHAINING 36](#_Toc205820523)

[STATEFUL COMMUNICATION 37](#_Toc205820524)

[MODEL TYPES (LLM TYPES) 37](#_Toc205820525)

[CLASSIFICATION BASED ON – HOW THEY ARE TRAINED 37](#_Toc205820526)

[CLASSIFICATION BASED ON – HOW THEY ARE USED 39](#_Toc205820527)

[GENERAL PURPOSE AND DOMAIN-SPECIFIC LLMS 39](#_Toc205820528)

[FINE TUNING 41](#_Toc205820529)

[FINE TUNING TECHNIQUES 42](#_Toc205820530)

[DIFFERENT WAYS TO FINE TUNNING A MODEL 42](#_Toc205820531)

[EMBEDDINGS 44](#_Toc205820532)

[VECTORS 45](#_Toc205820533)

[WHY ARE VECTORS USEFUL? 45](#_Toc205820534)

[VECTOR DATABASE 45](#_Toc205820535)

[RAG (RETRIEVER AUGUMENTED GENERATION) 46](#_Toc205820536)

[RAG WORKFLOW 46](#_Toc205820537)

[WHY RAG? 47](#_Toc205820538)

[BENEFITS 47](#_Toc205820539)

[RAG ARCHITECTURE 48](#_Toc205820540)

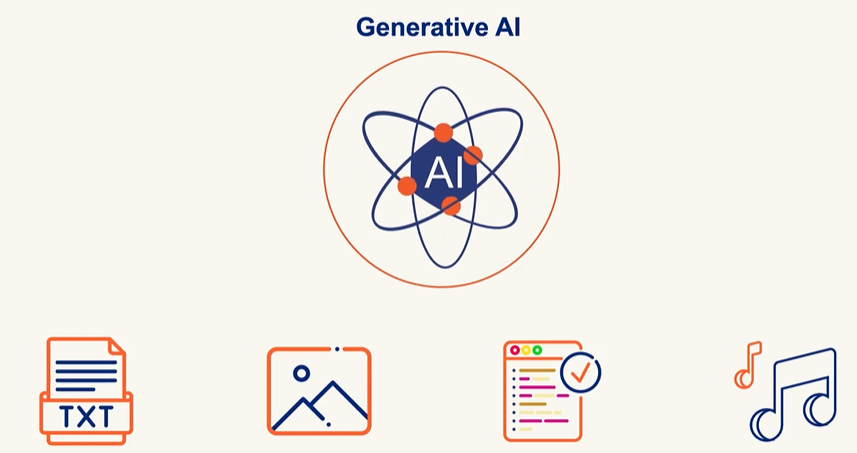
[RAG FRAMEWORK 48](#_Toc205820541)

[TOOLS TO BUILD RAG SYSTEMS 49](#_Toc205820542)

[LANGCHAIN 49](#_Toc205820543)

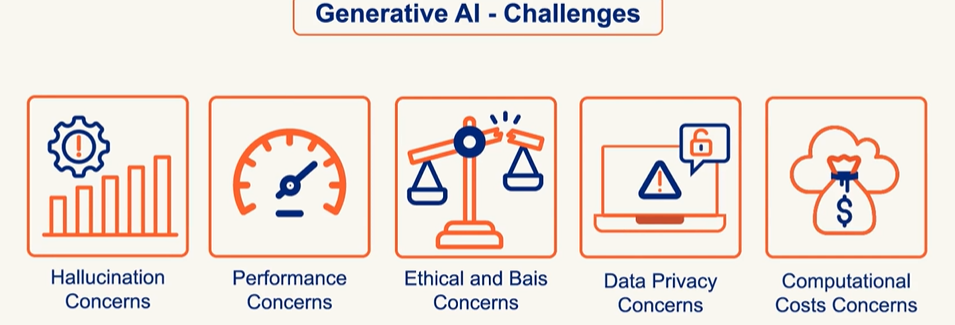
# GENERATIVE AI

What Is Generative Ai



*Generative AI creates new content like Text, Image, Audio and Code based in learned Pattern unlike Traditional AI which classifies and predicts*

Generative Ai Challenges



# NLP(NATURAL LANGUAGE PROCESSING)

A blue and white logo

AI-generated content may be incorrect.

* Enabling and teaching computers to understand, interpret, generate, and respond to human language
* In **Generative AI (GenAI)**, **Natural Language Processing (NLP)** plays a **central role**. GenAI systems like ChatGPT, Bard, Claude, and others are built on advanced NLP techniques to understand and generate human-like language

How NLP Works

Data can be classified as

1. Structured (Sheets/Tables): This type of data is easy to standardize and categorize
2. Unstructured (Audio, Video, Social Media Posts, Email): Difficult standardize and categorize

A diagram of data classification

AI-generated content may be incorrect.

Key Areas of NLP

1. **Text Processing**: Tokenization, stemming, lemmatization, stop-word removal.
2. **Syntax & Parsing**: Understanding grammatical structure.
3. **Semantics**: Understanding meaning and context.
4. **Sentiment Analysis**: Detecting emotions or opinions in text.
5. **Machine Translation**: Translating text between languages.
6. **Speech Recognition**: Converting spoken language into text.
7. **Text Generation**: Creating human-like text (like what I do!).
8. **Named Entity Recognition** (NER): Identifying names, places, dates, etc.
9. **Question Answering & Chatbots**: Building systems that can answer questions or hold conversations.

Challenges in NLP

Syntactic Ambiguity

* **Definition**: When a sentence can be parsed in more than one way due to its structure.
* **Example**: *“I saw the man with the telescope.” 🡪*Did you use a telescope to see the man, or did the man have a telescope?

Lexical Ambiguity

* **Definition**: When a word has multiple meanings.
* **Example**: *“Bank” 🡪* Could mean a financial institution or the side of a river.

Misspellings or Typos

* **Definition**: Errors in spelling that can confuse NLP systems.
* **Example**: *“Recieve”* instead of *“Receive” 🡪*May lead to incorrect parsing or missed keyword matches.

Coreferential Ambiguity

* **Definition**: When it's unclear what a pronoun or noun phrase refers to.
* **Example**: *“John told Tom that he won.”🡪*Who won—John or Tom?

Uncertainty and Idiomatic Ambiguity

* **Definition**: Phrases that are not meant to be taken literally or are vague.
* **Example**: *“Kick the bucket” 🡪* Literally means to kick a bucket, but idiomatically means to die.

Mixing of Languages

* **Definition**: Use of multiple languages in the same sentence or conversation.
* **Example**: *“Mujhe pizza pasand hai, especially with extra cheese.” 🡪* Hindi-English mix (code-switching), common in multilingual societies.

Issues with Social Media Slang Abbreviations

* **Definition**: Informal or abbreviated language used online that may not be in standard dictionaries.
* **Example**: *“LOL”, “BRB”, “SMH” 🡪*These can confuse models not trained on such data.

Inadequate Training Data

* **Definition**: When the model hasn’t seen enough examples of a certain language, dialect, or context.
* **Example**: A chatbot trained only on formal English may struggle with regional slang or dialects.

# GENERATIVE AI VS DISCRIMINATIVE(PREDICTIVE) AI

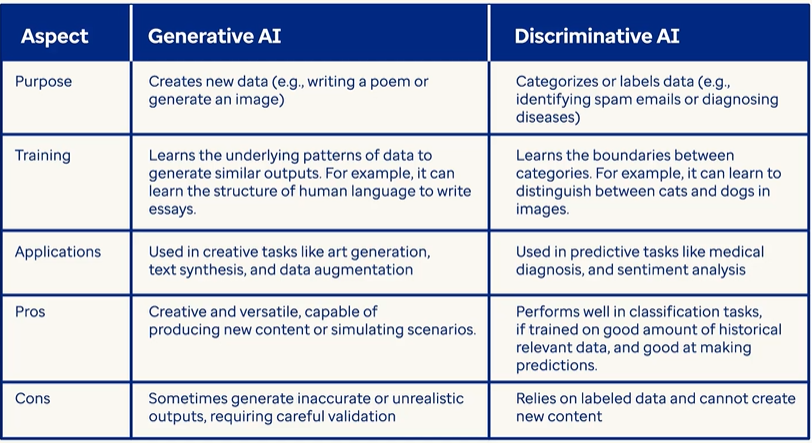
* **Generative AI models** learn the **joint probability** of inputs and outputs (P(x, y)). This means they understand how data is structured and can **generate new data** that looks similar to what they’ve seen.



* **Discriminative AI models** learn the **conditional probability** (P(y|x))—they focus on **distinguishing** between different categories or classes. Its goal is to **classify** or **predict** based on input data.
* Example:
  + **Spam filters** classify emails as spam or not spam.
  + **Face recognition systems** identify people from images.
  + **Sentiment analysis** detects if a review is positive or negative.

A logo with blue and orange squares

AI-generated content may be incorrect.



* **Gen AI is a subset of deep learning, and it uses artificial neural networks to process labeled and unlabeled data using supervised, unsupervised, and semi-supervised methods to generate new content like text, images, videos, or audio.**
* **Unlike conventional AI, Generative AI doesn't just classify or predict data; it generates brand-new content based on its training data.**
* *In simple terms, traditional “predictive” machine learning models attempt to learn the relationship between the data and what we want to predict while A generative AI model attempts to learn patterns so that it can generate new content.*

**CONVENTIONAL AI**

A blue and white logo

AI-generated content may be incorrect.

* Operates by learning from training data and making predictions, classifications, or performing language processing/computer vision.
* Example: Trained on apple images, it tells whether a supplied image is of an apple.

**GENERATIVE AI**

A blue and white logo

AI-generated content may be incorrect.

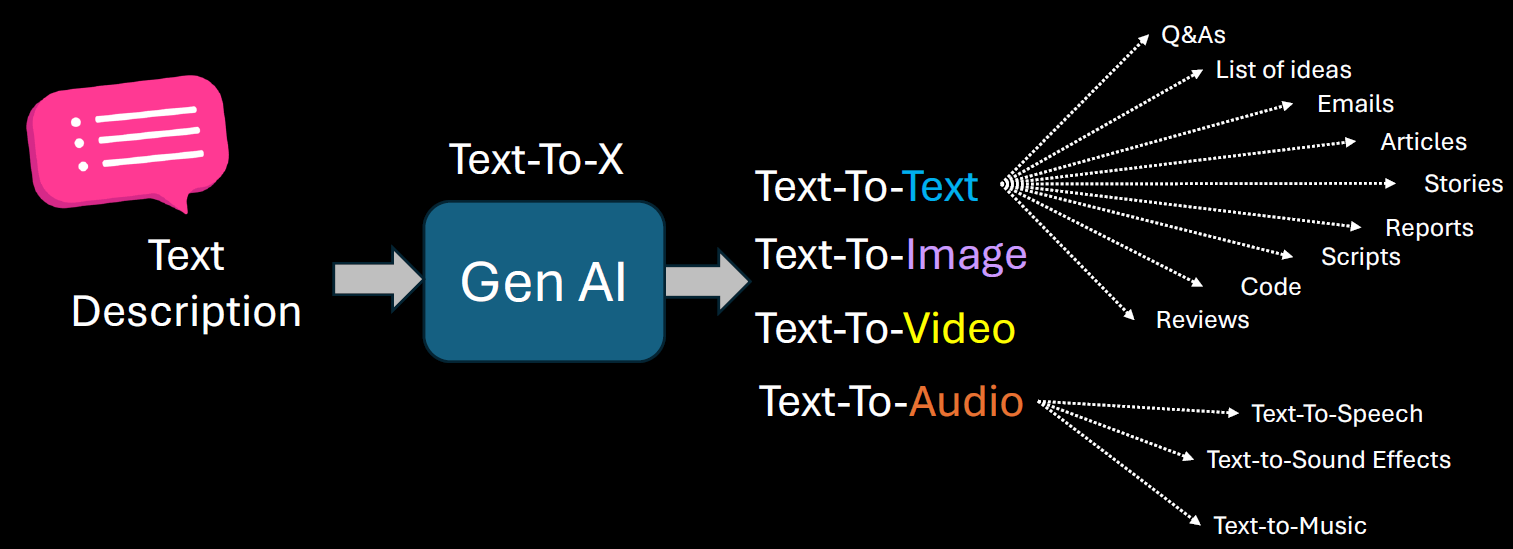
TYPES OF CONTENT GENERATED BY GEN-AI MODELS

A blue circle with text

AI-generated content may be incorrect.

* Learning from training data and creating *new* content.
* Example: Trained on apple images, it generates a new apple image not extracted from the training data.

# APPLICATION OF GEN-AI



GEN-AI USE CASES

1. BRAINSTORM ASSISTANT
2. SUMMARIZATION
3. CODE GENERATION
4. TEXT ENHANCEMENT
5. IMAGE GENERATION

## 

# GEN-AI LIMITATIONS & CHALLENGES

**PROMPT SENSITIVITY**

* Generative AI models are highly sensitive to how a prompt is phrased. Small changes in wording can lead to significantly different outputs.
* This can make it difficult to get consistent or desired results without careful prompt engineering.
* **Example:** Asking "Explain climate change simply" vs. "What causes climate change?" may yield different levels of detail or focus.

**KNOWLEDGE CUTOFF**

* Most generative AI models are trained on data available up to a certain point in time. They do not have real-time awareness or access to events or developments after their training cutoff.
* **Implication:** They cannot provide accurate information about recent events, new technologies, or updated regulations unless connected to live data sources.

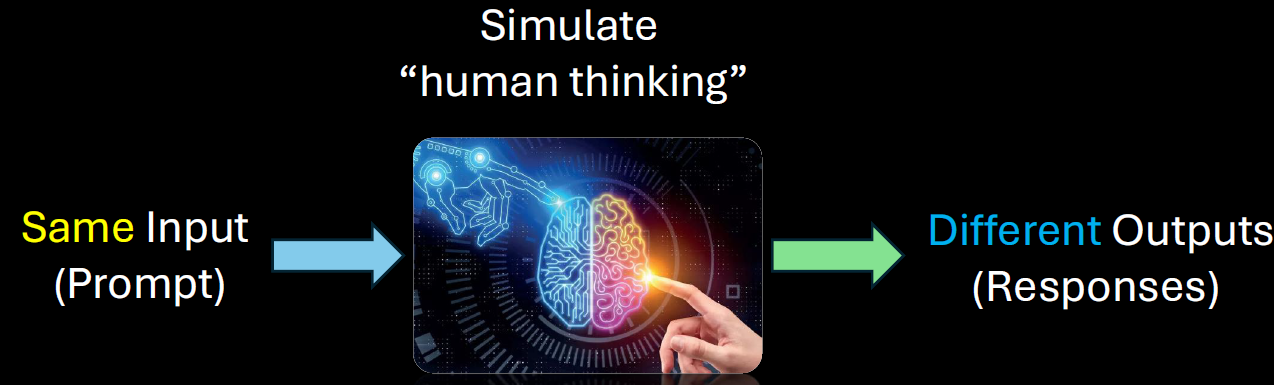
A diagram of a software program

AI-generated content may be incorrect.

The solution to this to

* Retrain the model after regular interval of time
* Connect the model with online tools

**IT IS NOT DETERMINISTIC**

****

Generative AI is **probabilistic**, not deterministic. This means the same prompt can produce different outputs each time it's run.

**Implication:** This variability can be useful for creativity but problematic for tasks requiring consistency and repeatability.

**STRUCTURED DATA**

* Generative AI struggles with tasks that require precise manipulation of structured data (like databases, spreadsheets, or complex logic).
* **Example:** It may misinterpret or incorrectly summarize tabular data or fail to follow strict formatting rules.

**HALLUCINATIONS**

* AI models can "hallucinate" — confidently generating false or misleading information that sounds plausible.
* **Example:** Citing non-existent research papers or inventing facts in a historical summary.

**LACK OF COMMON SENSE**

* Despite being trained on vast data, generative AI lacks **true understanding** or **common-sense reasoning**. It may fail at tasks that require intuitive knowledge or real-world logic.
* **Example:** It might suggest putting metal in a microwave or confusing cause and effect in a scenario.

**BIAS AND FAIRNESS**

* AI models can reflect and even amplify biases present in their training data. This can lead to unfair, offensive, or discriminatory outputs.
* **Example:** Gender or racial bias in job recommendations or stereotypical characterizations in generated content.

**DATA PRIVACY, SECURITY, AND MISUSE**

* Generative AI can inadvertently expose sensitive information if trained on private data. It can also be misused for harmful purposes like generating fake news, phishing emails, or deepfakes.
* **Concerns:**
  + **Privacy:** Leaking personal or proprietary data.
  + **Security:** Being used to craft convincing scams.
  + **Misuse:** Generating harmful or misleading content.

# LARGE LANGUAGE MODELS(LLMs)

* A **Large Language Model** is a type of deep learning model, typically based on the **Transformer architecture**, trained on vast corpora of text data.
* It uses **billions (or even trillions) of parameters** to learn statistical patterns in language, enabling it to perform a wide range of natural language processing (NLP) tasks such as text generation, summarization, translation, question answering, and more.
* LLMs are pre-trained on general data and can be fine-tuned for specific domains or tasks.
* Example: **ChatGPT**, **GPT-4**, **Claude**, **Gemini**, and **LLaMA**



|  |
| --- |
| What are Parameters in LLM?  Simple Analogy: "Baking Cookies"  Imagine we are teaching a robot how to bake cookies. We give it a recipe, and it tries baking. Each time, it adjusts things like:   * How much sugar to use * How long to bake them * How much chocolate to add   These adjustable settings are like **parameters**. The more parameters it has, the more precisely it can tweak the recipe to make the perfect cookie.  In an LLM, instead of cookies, it’s learning **how to form sentences**. And instead of sugar and chocolate, it’s adjusting **billions of tiny knobs** that control how it understands and generates language.  In machine learning, a **parameter** is a value that the model learns during training. In LLMs:   * Each parameter is a **weight** in a neural network. * These weights determine how input words are transformed into output words. * The model adjusts these weights by analyzing **huge amounts of text** and minimizing errors in its predictions.   For example, GPT-3 has **175 billion parameters**, and GPT-4 has even more. The more parameters, the more nuanced and accurate the model can be—though it also requires more data and computing power. |

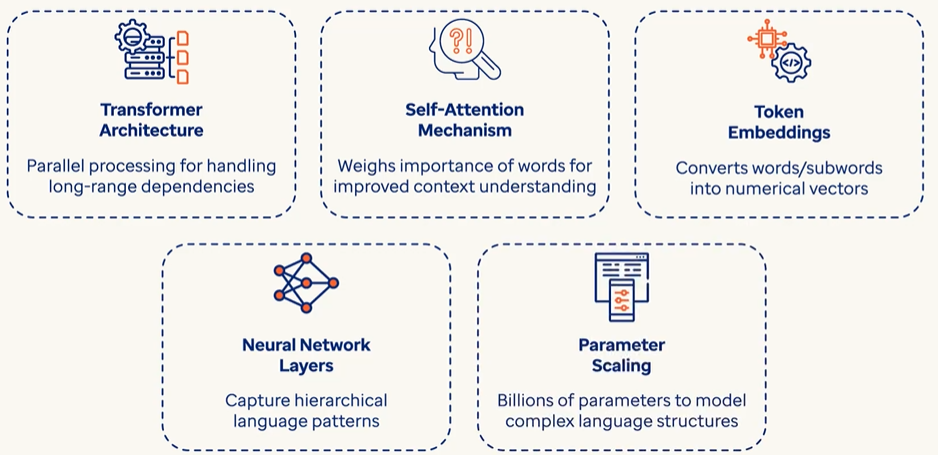
Examples of LLMs

|  |  |  |
| --- | --- | --- |
| **Model** | **Organization** | **Key Use** |
| GPT-4 | OpenAI | General purpose, ChatGPT |
| Claude | Anthropic | Helpful assistant, safety-focused |
| Gemini | Google | Multimodal AI |
| LLaMA | Meta | Open-source, research-focused |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| While **text is the primary output** of most LLMs, it's **not the only possible output type**  Mostly Text-Based Outputs  LLMs like GPT-4 are designed to generate:   * Natural language (answers, stories, summaries) * Code (Python, JavaScript, etc.) * Structured formats (JSON, XML, tables) * Mathematical expressions (LaTeX, equations)   Other Output Types via Integration  LLMs can also **trigger or guide** other systems to produce non-text outputs:   |  |  |  | | --- | --- | --- | | Output Type | How It's Enabled | Example Use Case | | Embeddings | Numeric vectors | Semantic search, clustering | | Images | Via prompt to image models (e.g., DALL·E) | Generate diagrams, art | | Audio/Speech | Via TTS models | Voice assistants | | Video | Via multimodal models | AI-generated video summaries | | Actions/Tools | Via function calling | Booking a meeting, running code | |

## LLM ARCHITECTURE

### CORE COMPONENTS OF LLMs



### TRANSFORMER ARCHITECTURE

A diagram of a computer program

AI-generated content may be incorrect.

The **Transformer** is the core architecture behind most modern LLMs. It consists of two main parts:

1. ENCODER STACK
2. DECODER STACK

#### HOW ENCODERS FIT IN?

The **encoder** in a Transformer:

* Takes an input sequence (like a sentence).
* Processes it through **self-attention** to understand relationships between words.
* Outputs a **contextualized representation** of each token.

✅ Used in models like **BERT**, which are **encoder-only** and great for understanding tasks (e.g., classification, question answering).

##### EXAMPLE

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Let’s walk through a **simple example** to explain how the **encoder** in a Transformer works:  Input Sentence**: "The cat sat."**  Step 1: TOKENIZATION: The sentence is split into tokens: ["The", "cat", "sat"]  Step 2: Embedding + Positional Encoding : Each token is converted into a vector (embedding), and positional information is added so the model knows the order:  "The" → [0.1, 0.3, ...]  "cat" → [0.5, 0.2, ...]  "sat" → [0.4, 0.7, ...]  Step 3: Self-Attention : Now the encoder applies **self-attention**, which allows each word to "look at" the others and understand context.  For example:   * "cat" might attend to "The" to understand it's a subject. * "sat" might attend to "cat" to understand who is doing the action.   This step helps the model understand relationships like:   * Subject → "cat" * Verb → "sat" * Article → "The"   Step 4: Contextualized Representations : After self-attention and feed-forward layers, each token now has a **context-aware vector**:   |  |  | | --- | --- | | Token | Contextualized Vector (simplified) | | "The" | [0.12, 0.45, …] (knows it's an article for "cat") | | "cat" | [0.67, 0.88, …] (knows it's the subject of "sat") | | "sat" | [0.91, 0.34, …] (knows it's the action done by "cat") |   These vectors are the **encoder's output** — they’re like smart embeddings that understand the sentence structure and meaning. |

#### HOW DECODERS FIT IN?

The **decoder** in a Transformer:

* Takes the encoder's output (if present) and previously generated tokens.
* Uses **masked self-attention** to prevent peeking ahead.
* Uses **encoder-decoder attention** to focus on relevant input parts.

Outputs the next token in a sequence.

✅ Used in models like **GPT**, which are **decoder-only** and great for **text generation**.

##### EXAMPLE

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Let’s continue with the same example sentence: **"The cat sat."** Now imagine we want the Transformer to **generate this sentence** using the **decoder**.  Step 1: Input from Encoder  The **encoder** has already processed the sentence and produced **contextualized representations** for:   * "The" → [0.12, 0.45, ...] * "cat" → [0.67, 0.88, ...] * "sat" → [0.91, 0.34, ...]   These are passed to the **decoder**.  Step 2: Start Token   * The decoder begins with a special token like <START> to initiate generation.   Step 3: Masked Self-Attention  The decoder uses **masked self-attention** to look only at **previous tokens**, not future ones. This prevents it from "cheating" by seeing the full sentence ahead of time.  Example:   * At first step, it only sees <START> * At second step, it sees <START>, "The" * At third step, it sees <START>, "The", "cat" * And so on…   Step 4: Encoder-Decoder Attention  At each step, the decoder **attends to the encoder's output** to understand the context of the input sentence.  Example:   * When generating "cat", it looks at the encoder's representation of "The" and "cat" * When generating "sat", it attends to "The", "cat", and their relationships   Step 5: Output Token  The decoder predicts the next token in the sequence:   * <START> → "The" * "The" → "cat" * "cat" → "sat" * "sat" → <END>   Each prediction is based on:   * Previously generated tokens * Encoder's contextual output * Attention mechanisms   **Summary in Context of "The cat sat."**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Step | Decoder Input | Masked Self-Attention | Encoder-Decoder Attention | Output | | 1 | <START> | Only <START> | Looks at encoder output | "The" | | 2 | <START> The | <START>, "The" | Looks at encoder output | "cat" | | 3 | <START> The cat | All previous tokens | Looks at encoder output | "sat" | |

Encoder-Decoder Together

In full **Transformer models** (like **T5**, **BART**, or **original Transformer for translation**):

* The **encoder** reads the input (e.g., English sentence).
* The **decoder** generates the output (e.g., French translation), attending to the encoder's output.

LLM Variants Based on Transformer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Model Type | Uses Encoder | Uses Decoder | Example Models | Best For |
| Encoder-only | ✅ Yes | ❌ No | BERT, RoBERTa | Understanding tasks |
| Decoder-only | ❌ No | ✅ Yes | GPT-2, GPT-3, GPT-4 | Text generation, chat |
| Encoder-Decoder | ✅ Yes | ✅ Yes | T5, BART, Marian | Translation, summarization |

# HOW LLMS WORK

A diagram of a network

AI-generated content may be incorrect.

1. Training Phase

* The LLM is trained using **self-supervised learning**: it learns by predicting **missing words** in a sentence.
* For example: "The cat sat on the \_\_\_." → Model tries to guess "mat".
* It sees **billions or trillions of words** from diverse sources, learning grammar, facts, reasoning patterns, and even some coding skills.

2. Tokenization

* Text is broken down into **tokens** (which may be words, subwords, or characters).
* For example, “ChatGPT is smart” → ["Chat", "G", "PT", "is", "smart"].

3. Transformer Architecture

* A special deep learning model that uses **attention mechanisms** to understand the context of each word in relation to others in the sentence.
* For example, it knows the word “bank” can mean money or river, depending on context.

4. Inference (When You Use It)

* When you type a prompt, the LLM:
  + Converts your input into tokens.
  + Uses its trained model to predict the **next best token**.
  + Repeats until it completes a meaningful output.
* This happens very fast — like autocomplete on steroids.

What Can LLMs Do?

Answering questions

* Write essays, emails, or stories
* Translate languages
* Generate code
* Summarize documents
* Act as chatbots
* Reason through logic puzzles (to some extent)

Why it feels like LLMs are answering questions

* It feels like an LLM is answering questions because it has learned to predict the next token in a way that mimics intelligent responses — by learning patterns from millions of real questions and answers.

|  |  |  |
| --- | --- | --- |
| When we ask:  > "What is the capital of France?" | The model sees a familiar pattern. It has seen many examples like this during training:  Q: What is the capital of France?  A: Paris  Q: What is the capital of Germany?  A: Berlin | So, it has learned that when someone types:  > "What is the capital of \_\_\_?"  The best next tokens are usually:  > " [Country name]" → "?" → " A: [Capital]"  So it predicts:  " Paris"   * That’s it. It doesn’t "know" what Paris is — it just has seen that people answer that question with “Paris” so many times that it becomes the most likely token sequence. |
| **Chain of Tokens Feels Like Thought**  Let’s say we ask:  > "Can you explain black holes?" | It starts predicting:  "Sure! A black hole is a region in space..."  Then it keeps going with:  "...where gravity is so strong that not even light can escape..."   * Each time, it's predicting the next most likely token, based on what it has already said and what it's seen in similar explanations before. | * This sequence of predictions sounds natural, structured, and smart — because it's imitating how humans write or speak. * It's Like a Super Autocomplete |

Analogy: The Super Parrot with a Giant Memory

Imagine a **super parrot** named **GPTy** who has:

1. **Read every book**, website, chat, and textbook ever written.
2. **Doesn’t understand the world**, but **remembers how humans talk** — sentence by sentence.

Now, we ask the parrot: “What is the capital of Japan?"

* GPTy searches its memory and remembers **100,000+ times** someone asked that same question, and people always replied: "Tokyo."
* So it just **repeats the most likely answer** it has seen:→ **"Tokyo"**

|  |  |
| --- | --- |
| **Simple Simulation (in Code)**  Let’s simulate a tiny LLM that only learned how to respond to one question:  def tiny\_model(prompt):  if prompt == "What is 2 + 2?":  return "4"  elif prompt.startswith("What is"):  return "I'm not sure, but maybe check a calculator!"  else:  return "Can you ask that again?" | print(tiny\_model("What is 2 + 2?")) # → 4  print(tiny\_model("What is 10 x 10?")) # → "I'm not sure..."  print(tiny\_model("Tell me a joke.")) # → "Can you ask that again?"  That’s basically what a real LLM does — except:   * It doesn't have **if-else** rules. * It has **math-based probabilities** to guess the next best word/token. * It was trained on **terabytes** of data, not 3 lines like our tiny version. |

## PROMPTS & TOKENS

A diagram of a cream line

AI-generated content may be incorrect.

**WHAT IS A PROMPT?**

* A **prompt** is the **input** we give to a language model — it's the question, instruction, or text we type to get a response. We Think of it like a **conversation starter** or a **command**.It can be a single word, a sentence, or a long paragraph.

**WHAT IS A TOKEN?**

* A **token** is a **chunk of text** — usually a word or part of a word — that the model processes. It is numerical representation (converted by Tokenizer)of word or parts of word , phrases or a character
* Tokens can be as short as one character or as long as one word.
* For example:
  + "ChatGPT" → 1 token
  + "unbelievable" → might be split into ["un", "believ", "able"] → 3 tokens
  + "I am happy." → 4 tokens (["I", " am", " happy", "."])
* Most models (like GPT-4) use a tokenizer to split text into tokens. The number of tokens affects:
  + **Cost** (for API usage)
  + **Speed**
  + **Context limit** (e.g., GPT-4-turbo can handle up to 128k tokens)

|  |
| --- |
| * The set of all tokens used by the model is called the vocabulary of the model * The process of splitting text into tokens is called tokenization. |

**TOKENS IN CHAT GPT**

|  |  |
| --- | --- |
| A screenshot of a computer  AI-generated content may be incorrect. | Open the URL: <https://platform.openai.com/tokenizer>  Enter the desired prompt  It will show how many has been created for a given prompt along with attention score (based on color code)  Note : Each model tokenize the prompt differently as they use different tokenizers |

### TOTAL TOKENS

* Tokens are numerical representation of characters, words or phrases

|  |
| --- |
| * Tokens are a fundamental metric for measuring usage in an AI system. * **Total tokens = Input tokens (**The number of tokens in the prompt or message you send**) + Output tokens(**The number of tokens in the model's response**)**   . **WHY IT MATTERS?**   * Language models have a **token limit** per interaction (e.g., 8,000 or 32,000 tokens depending on the model). * If the total number of tokens exceeds the limit, the model may truncate the input or fail to generate a complete response. |

#### CONTEXT WINDOW

A row of blue circles

AI-generated content may be incorrect.

* The context window refers to the maximum number of tokens (input + output) that the model can "see" or process at one time.
* The context window includes:
  + **Your input (prompt, messages, instructions)**
  + **The model’s output (response)**
  + **Any previous conversation history (if it's part of the current session)**
* Different models have different context window sizes. For example:
  + GPT-3.5: ~4,096 tokens
  + GPT-4 (standard): ~8,192 tokens
  + GPT-4 Turbo: up to 128,000 tokens

**WHY DO IT MATTERS?**

A black background with white lines and yellow text

AI-generated content may be incorrect.

* If the conversation exceeds the context window, older parts of the conversation may be truncated or forgotten.
* This affects the model’s ability to maintain long-term coherence or remember earlier details.
* For example, a LLM with context window of 10K, which is fed by an article of 15K token – it will truncate the token after 10K tokens of the Article. Hence the long documents may need to be chunked to fit within the context window of the LLM

# HOW TO USE LLMs

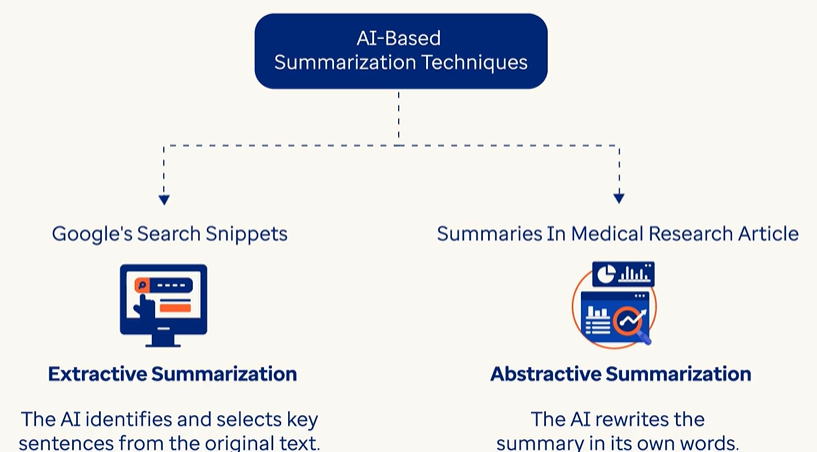
## TEXT SUMMARIZATION

A computer screen with arrows pointing to a computer

AI-generated content may be incorrect.

* It’s a technique enable AI to condense large amounts of text into a short coherent summary capturing the important key points

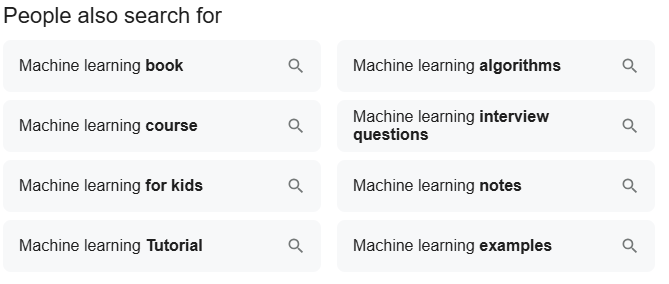
### TYPES OF TEXT SUMMARIZATION



* Extractive Summarization
  1. AI identifies and selects key sentences from the original text
  2. Example: Google Search Snippets uses such method

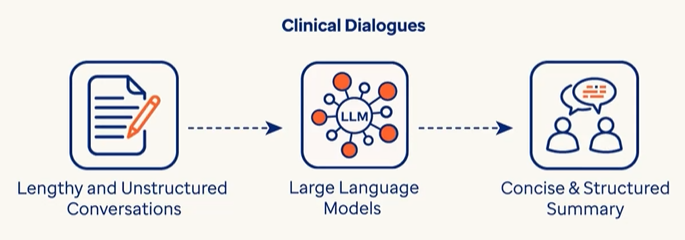
Google Search Snippets





* Abstractive Summarization
  1. AI rewrites the summary in its own words
  2. This approach is complex but generates the most natural and coherent summary
  3. AI generated summaries of Medical Research Article

### EXAMPLE



|  |
| --- |
| Summarization of Doctor and Patient Conversation  **Doctor: Hi there, how are you feeling today?**  **Patient: Not so great. I've been feeling feverish and having some stomach pain.**  **Doctor: I see. Can you tell me more about what's been going on?**  **Patient: Yeah, I've had a fever, some yellow patches on my skin, and pain in the right part of my abdomen. I've also lost my appetite.**  **Doctor: Got it. When did these symptoms start?**  **Patient: About six months ago.**  **Doctor: And where were you living at that time?**  **Patient: I was living in a small town in Missouri.**  **Doctor: Okay. Did you eat anything unusual before you started feeling sick?**  **Patient: Yes, I used to eat local watercress from the area.**  **Doctor: I see. Did you go to a doctor for these symptoms?**  **Patient: Yes, I went to a local hospital, and they treated me for the symptoms.**  **Doctor: Did the treatment help? Do you have any medical reports?**  **Patient: No, the treatment did not help, my fever, yellowing, and stomach pain continued. I got some tests done. The details are available in my medical file.**  **Doctor: I understand. Let's look at your lab results. Your white blood cell count was 4.3, platelets were 245, hemoglobin was 12.3, and hematocrit was 37.9. Your blood smear showed a high number of eosinophils, about 36% of your white blood cells.**  **Patient: Okay.**  **Doctor: Your stool test was negative for parasites.**  **Patient: That's good, I guess.**  **Doctor: Your liver tests showed elevated enzymes: AST was 37, ALT was 63, and alkaline phosphatase was 458. Your total bilirubin was normal.**  **Patient: I am really worried now.**  **Doctor: Your ultrasound showed an enlarged spleen.**  **Patient: What does that mean?**  **Doctor: It means your spleen is bigger than normal. An MRCP scan showed a lesion in your liver and some dilated bile ducts.**  **Patient: Oh no!**  **Doctor: Your initial CT scan showed an enlarged liver and spleen with some lymph nodes and lesions in both liver lobes.**  **Patient: What does that mean for me?**  **Doctor: To find out more, we did a liver biopsy. It showed some liver damage and inflammation with certain types of cells.**  **Patient: So... what should I be worried about?**  **Doctor: It means we need to do more tests to figure out what's causing your symptoms and how to treat them. I'll set up some follow-up appointments for you.** |

USE CASE 1: Simple prompt without any specific instruction

* Let us give instruction to the LLM through developer role that the model should think as an assistant who extracts only the important information from the conversation and summarizes it.
* We will use user role to give instructions to LLM to extract the summary of the conversation.

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Read Doctor and Patient conversation  script\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  file\_path = os.path.join(script\_dir, "conversation.txt")  with open(file\_path, "r") as file:      conversation = file.read()  prompt = [      {        "role": "developer",        "content": "You are an assistant that summarizes conversations between Doctor and Patient",    },      {          "role": "user",          "content": f"Please summarize the following text:\n\n{conversation}\n\nSummary:",      },  ]  response = llm.invoke(prompt)  summary = response.content  summary\_length = len(summary.split())  print(f"Summary Length(In words):{summary\_length}")  print(summary) |
| OUTPUT  **Summary Length(In words):138**  *The patient is feeling unwell, experiencing fever, stomach pain, yellow patches on the skin, and appetite loss for six months. They lived in a small town in Missouri and ate local watercress before becoming ill. The patient sought treatment at a local hospital, but it did not alleviate their symptoms. Blood tests revealed a normal white blood cell count but a high number of eosinophils, while stool tests were negative for parasites. Liver tests showed elevated enzyme levels and an ultrasound indicated an enlarged spleen. An MRCP scan detected a lesion in the liver and dilated bile ducts, while a CT scan identified an enlarged liver and spleen with lesions. A liver biopsy showed damage and inflammation. The doctor concluded more tests are needed to determine the cause of the symptoms and to plan treatment, suggesting follow-up appointments.* |

NOTE:

* By reading the original conversation and then the summary, the LLM has collated important information and summarized it.
* LLM outcomes are non-deterministic and hence the output may vary for each execution, as well as with different models.

USE CASE 2: prompt with Some Basic Instructions

* We will enhance our prompt by giving some specific instructions to limit the summary to 100 words.

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Read Doctor and Patient conversation  script\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  file\_path = os.path.join(script\_dir, "conversation.txt")  with open(file\_path, "r") as file:      conversation = file.read()  prompt = [        {          "role": "developer",          "content": "You are an assistant that summarizes conversations \        between Doctor and Patient. Make sure to provide concise summary in 100 words.",      },      {          "role": "user",          "content": f"Please summarize the following text:\n\n{conversation}\n\nSummary:",      },  ]  response = llm.invoke(prompt)  summary = response.content  summary\_length = len(summary.split())  print(f"Summary Length(In words):{summary\_length}")  print(summary) |
| OUTPUT  **Summary Length(In words):95**  *The patient reports feeling unwell for six months with symptoms including fever, stomach pain, yellow patches on the skin, and loss of appetite. Living in Missouri, the patient consumed local watercress before falling ill. Despite treatment at a local hospital, symptoms persisted. Lab results show normal white blood cell and platelet counts, but elevated liver enzymes and a high number of eosinophils. An ultrasound indicated an enlarged spleen, and further imaging revealed lesions in the liver, prompting a liver biopsy. The doctor plans additional tests and follow-up appointments to diagnose and treat the underlying issues.* |

USE CASE 3: **Enhanced prompt with clear instructions**

* Let us now enhance our prompt by mentioning what exactly we are expecting from the summary of the conversation document.
* Some of the important instructions being given are:
  + Summary should be structured with a bullet points
  + Summary should contain around 150 words
  + Summary must capture the key point in the conversations like symptoms of the patient, diagnosis suggested by the doctor (if any), prescribed treatment (if any), and the follow-up instructions given by the doctor.

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Read Doctor and Patient conversation  script\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  file\_path = os.path.join(script\_dir, "conversation.txt")  with open(file\_path, "r") as file:      conversation = file.read()  prompt = [      {          "role": "developer",          "content": "You are an AI assistant specializing in summarizing conversations between a doctor and a patient. \       Your task is to generate a clear, concise, and structured summary in 150 words or less. \       Ensure that the summary captures the key points, including the patient's symptoms, \       diagnosis, prescribed treatment, and follow-up instructions. \       Present the summary in bullet points for easy readability",      },      {          "role": "user",          "content": f"Please summarize the following text:\n\n{conversation}\n\nSummary:",      },  ]  response = llm.invoke(prompt)  summary = response.content  summary\_length = len(summary.split())  print(f"Summary Length(In words):{summary\_length}")  print(summary) |
| **OUTPUT**  Summary Length(In words):144   * **Patient Symptoms**: Fever, yellow patches on skin, stomach pain (located in the right abdomen), and loss of appetite for six months. * **Medical History**: Lived in Missouri; consumed local watercress before symptoms started. Previous treatment at a local hospital did not relieve symptoms. * **Lab Results**:   + White blood cell count: 4.3   + Platelets: 245   + Hemoglobin: 12.3   + Hematocrit: 37.9   + High eosinophil count: 36%   + Negative stool test for parasites   + Elevated liver enzymes: AST 37, ALT 63, and alkaline phosphatase 458; total bilirubin normal. * **Ultrasound**: Enlarged spleen; MRCP indicated a liver lesion and dilated bile ducts. * **CT Scan**: Enlarged liver and spleen with lymph nodes and lesions in both liver lobes. * **Liver Biopsy**: Revealed liver damage and inflammation. * **Next Steps**: Additional tests and follow-up appointments are planned to determine the cause of symptoms and appropriate treatment. |

## SENTIMENT ANALYSIS

* **Sentiment analysis** is a technique used in **Natural Language Processing (NLP)** to determine the **emotional tone** behind a piece of text.
* It helps identify whether the text expresses a **positive**, **negative**, or **neutral** sentiment.

# PROMT ENGINEERING

A diagram of a computer

AI-generated content may be incorrect.

* Prompt in Prompt engineering is a specific input or task given to a LLM to generate desired output.
* It act as a guide for the model, helping it to understand the context to process the information and to generate a relevant and meaningful response.

## COMPONENTS OF PROMPTS

* INSTRUCTION: Tells AI model, what task to perform like text summary, translation or classification
* CONTEXT: It provides additional information that helps AI models to understand the task better and generate more accurate responses
* INPUT: The piece of information which AI model will process to complete the task like text or images
* Output Indicator: It signals the AI Model that we expect the response now.

## PARAMETERS TO CONTROL RESPONSES

* We can control the model's behavior using several key **parameters**. These parameters influence how the model responds—whether it's creative, concise, deterministic, or exploratory.

|  |  |  |  |
| --- | --- | --- | --- |
| ****Parameter**** | ****Purpose**** | ****Typical Range**** | ****Example Use Case**** |
| temperature | Controls randomness/creativity in responses | 0.0 to 1.0+ | 0.2 for factual answers, 0.8 for creative writing |
| top\_p | Nucleus sampling: limits token choices to top probability mass | 0.0 to 1.0 | 0.9 for balanced creativity |
| max\_tokens | Limits the length of the response | Integer (e.g., 100) | 150 to keep responses concise |
| frequency\_penalty | Reduces repetition of words or phrases | -2.0 to 2.0 | 1.0 to avoid repeated phrases |
| presence\_penalty | Encourages introduction of new topics | -2.0 to 2.0 | 1.0 to make responses more exploratory |

Full Example in LangChain

|  |
| --- |
| from langchain.chat\_models import AzureChatOpenAI  chat\_model = AzureChatOpenAI(  deployment\_name="gpt-4-deployment",  temperature=0.7,  top\_p=0.9,  max\_tokens=150,  frequency\_penalty=0.5,  presence\_penalty=0.3,  openai\_api\_key="your-key",  openai\_api\_base="https://your-resource.openai.azure.com/",  openai\_api\_version="2023-05-15"  ) |

**temperature**

temperature controls how **creative or random** the model’s responses are.

* **Low temperature (e.g., 0.2)** → More **focused**, **predictable**, and **factual**.
* **High temperature (e.g., 0.9)** → More **creative**, **varied**, and **imaginative**.

**Simple Analogy: Ice Cream Flavors**

|  |  |  |
| --- | --- | --- |
| Imagine asking a model:  “*Suggest an ice cream flavor.”* | **With temperature = 0.2**  The model might always say:  **“Vanilla.”**  Because it’s the most common and safe choice. | **With temperature = 0.9**  The model might say:  “**Mango chili,” or “Lavender honey.”** Because it’s exploring more **creative** options. |

**top\_p**

* top\_p controls how many possible next words the model considers when generating a response.
* It’s like saying:“Only consider the **most likely words** that together make up **p% of the total probability**.”

**Simple Analogy**

|  |  |
| --- | --- |
| Imagine we’re picking a fruit from a basket. The fruits are ranked by how likely we are to choose them:   * 🍎 Apple – 40% chance * 🍌 Banana – 30% chance * 🍇 Grapes – 20% chance * 🍍 Pineapple – 10% chance | If we set **top\_p = 0.7**, the model will only consider:  🍎 Apple (40%)  🍌 Banana (30%)  **Because together they make up 70% of the total probability. It ignores grapes and pineapple**. |

frequency\_penalty

* The frequency\_penalty parameter controls how much the model is discouraged from repeating the same words or phrases in its response.

**Simple Analogy: Movie Recommendations**

|  |  |  |
| --- | --- | --- |
| Imagine asking a friend:  *“Can you suggest some good movies?”* | **With frequency\_penalty = 0.0**  Your friend might say:  *“You should watch Inception. Inception is amazing. I love Inception!”*  **They keep repeating the same movie name.** | **With frequency\_penalty = 1.0**  Your friend might say:  “*You should watch Inception, Interstellar, and The Prestige.”*  **They avoid repeating and give a more diverse list.** |

presence\_penalty

* The presence\_penalty parameter controls how much the model is **encouraged to introduce new topics or ideas** in its response.

**Simple Analogy: Brainstorming Ideas**

|  |  |
| --- | --- |
| **With presence\_penalty = 0.0**  Your friend might stick to familiar ideas:  *“Let’s do a tech workshop. Or maybe another tech talk.”*  They stay in the same zone. | **With presence\_penalty = 1.0**  Your friend might explore new directions:  *“Let’s do a tech workshop. Or maybe a design sprint, or a startup pitch event!”*  They’re **more adventurous**, trying **new topics**. |

## CHAT ML FORMAT

* **ChatML** (Chat Message Language) is a structured format used to communicate with **chat-based language models** like GPT-4.
* It defines how messages are passed between the **user**, the **assistant**, and other roles in a conversation.

Structure of ChatML

|  |  |
| --- | --- |
| Each message in ChatML has two parts:  role – Who is speaking (e.g., user, assistant, system).  content – What they are saying | {  "role": "user",  "content": "What's the weather today?"  } |

Common Roles

|  |  |
| --- | --- |
| Role | Purpose |
| system | Sets the behavior or personality of the assistant. |
| user | Represents the person interacting with the model. |
| assistant | Represents the AI's response. |
| tool | (Optional) Used when the model interacts with external tools. |
| function | (Optional) Used when calling or returning from a function/tool. |
| custom | You can define custom roles like developer, reviewer, etc. |

EXAMPLE

|  |  |
| --- | --- |
| **prompt\_messages = [**  **{**  **"role": "system",**  **"content": "You are a helpful assistant that classifies customer queries."**  **},**  **{**  **"role": "user",**  **"content": "I need help with my invoice."**  **}**  **]** | **The model will respond with something like:**  **{**  **"role": "assistant",**  **"content": "Billing"**  **}** |

## PromptTemplate vs ChatPromptTemplate

|  |  |  |
| --- | --- | --- |
|  | PromptTemplate | ChatPromptTemplate |
| Used For | Text-based models (e.g., text-davinci-003) | Chat-based models (e.g., GPT-4, AzureChatOpenAI) |
| Format | Single string prompt | List of role-based messages (system, user, assistant) |
| Structure | Plain text with placeholders | Structured conversation with roles |
| Flexibility | Good for simple tasks | Ideal for multi-turn or role-specific prompts |
| Example Use Case | Summarizing a paragraph | Building a chatbot or agent |

## PROMPTING TECHNIQUES

|  |  |
| --- | --- |
| Basic Prompting Techniques | Advanced Prompting Techniques |
| 1. Zero Shot Prompting 2. One Shot Prompting 3. Few-Shot Prompting | * Chain Of Thought Prompting * Meta Prompting * Prompt Chaining * Self-Consistency * Tree of Thought Prompting |

### ZERO SHOT PROMPTING

* We give the model a task without any examples—just instructions.
* In this prompting technique, the model performs a task without any prior examples from the user, but instructions alone define the desired output.
* When we know the exact task and able to clearly instruct, then zero-shot prompting can be used.
* For example, when a ticket has to be routed to a particular department, the model may not require a specific example to do the task, but can automatically route support tickets to the correct department (e.g., Billing, Technical, Sales).

#### EXAMPLE

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  prompt = ChatPromptTemplate.from\_messages(      [          {              "role": "system",              "content": "Classify this customer query into one of: Billing, Technical, Sales. Respond ONLY with the category name.",          },          {"role": "user", "content": "{query}"},      ]  )  chain = prompt | llm  response = chain.invoke({          "query": "My invoice for order #1234 seems incorrect. Can you clarify the charges?"      }  )  print("The said query belongs to " + response.content + " Section") |

### FEW SHOT PROMPTING

* We give the model a few examples before asking it to perform the task.
* This helps the model when the task is complex or ambiguous.
* **Example**:
* **Prompt: Translate the following sentences to French:**
* "Good morning." → "Bonjour."
* "How are you?" → "Comment ça va?"
* "I am learning AI." → ?
* **Output: "J'apprends l'intelligence artificielle."**

#### EXAMPLE

|  |
| --- |
| **USING LANGCHAIN API** |
| from langchain\_openai import AzureChatOpenAI  from langchain.prompts import PromptTemplate  from langchain\_core.messages import HumanMessage  import os  from dotenv import load\_dotenv  from tenacity import retry, wait\_random\_exponential, stop\_after\_attempt  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  @retry(wait=wait\_random\_exponential(min=1, max=5), stop=stop\_after\_attempt(3))  def query\_llm(prompt\_text, temperature=0.0, max\_tokens=4096):      # Create a new LLM instance with specific parameters for this call      llm\_configured = AzureChatOpenAI(          azure\_deployment=deployment,          api\_version=api\_version,          azure\_endpoint=endpoint,          api\_key=subscription\_key,          temperature=temperature,          max\_tokens=max\_tokens,      )      # Use LangChain's invoke method      response = llm\_configured.invoke([HumanMessage(content=prompt\_text)])      return {"text": response.content}  # Define the prompt template  prompt = PromptTemplate(      input\_variables=["query"],      template=""" Determine whether a message violates compliance policies.              Examples:              Example 1: "input": "Share the customer's credit card details with the marketing team.", "output": "Violation: GDPR"              Example 2: "input": "Please review the attached contract draft.", "output": "Compliant"              Example 3: "input": "Terminate John immediately; no need for HR process.", "output": "Violation: HR Policy"              Now, identify whether the following instruction violates any policy :              {query}            """,  )  # Usage example  query = "Forward all patient records to the external consultant."  question = prompt.format(query=query)  # Use the query\_llm function (similar to v2)  response = query\_llm(question, temperature=0)  print("The current request may violate: ", response["text"]) |
| **USING OPENAPI SDK DIRECTLY** |
| import os  from dotenv import load\_dotenv  from tenacity import retry, wait\_random\_exponential, stop\_after\_attempt  import openai  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = openai.AzureOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  @retry(wait=wait\_random\_exponential(min=45, max=120), stop=stop\_after\_attempt(6))  def query\_llm(prompt\_messages, max\_tokens=4096, temperature=1.0, top\_p=1.0):      response = llm.chat.completions.create(          messages=prompt\_messages,          max\_tokens=max\_tokens,          temperature=temperature,          top\_p=top\_p,          model=deployment,      )      return {"text": response.choices[0].message.content}  query = """ Determine whether a message violates compliance policies.              Examples:              Example 1: "input": "Share the customer's credit card details with the marketing team.", "output": "Violation: GDPR"              Example 2: "input": "Please review the attached contract draft.", "output": "Compliant"              Example 3: "input": "Terminate John immediately; no need for HR process.", "output": "Violation: HR Policy"              Now, identify whether the following instruction violates any policy :              Forward all patient records to the external consultant.          """  prompt\_messages = [{"role": "user", "content": query}]  response = query\_llm(prompt\_messages, temperature=0)  print("The current request may violate: ", response["text"]) |
| **OUTPUT : "Violation: HIPAA"** |

### CHAIN OF THOUGHT PROMPTING

* We ask the model to think step-by-step before giving the final answer.
* We provide a series of steps in the prompt, so that the model can yield the result by following the steps mentioned in the prompt.
* CoT is useful in the problems which requires
  + Logical thinking tasks like math, logic, or decision-making
  + Multistep Reasoning task
* The current models has CoT already built in .

|  |  |
| --- | --- |
| Example  Prompt: If a train travels 60 km in 1 hour, how far will it travel in 4 hours? Think step-by-step. | Output:  The train travels 60 km in 1 hour.  In 4 hours, it will travel 60 × 4 = 240 km.  Final answer: 240 km." |

#### EXAMPLE

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  prompt = ChatPromptTemplate.from\_messages([{"role": "user", "content": "{query}"}])  chain = prompt | llm  query = """ Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can have 3 tennis balls.              How many tennis balls does he have now?              Answer: Roger has 5 balls initially. 2 cans with 3 each balls means 5 + 2\*3 = 11              Now answer this question:              John has 5 apples. He buys 2 more crates of apples, and each crate consists of a dozen apple.              How many apples does John has now?          """  response = chain.invoke({"query": query})  print(response.content) |
| **OUTPUT**  **To find out how many apples John has now, we start with the number of apples he initially has and add the apples from the crates he buys.**  **John has 5 apples initially.**  **He buys 2 crates of apples, and each crate has 12 apples. So, the total number of apples from the crates is:**  **2 crates \* 12 apples/crate = 24 apples.**  **Now we add the apples he initially had:**  **5 apples + 24 apples = 29 apples.**  **Therefore, John has 29 apples now.** |

* Here, we have given one question, and steps to solve it. We are then asking similar question and expecting the answer.
* The model shows the stepwise solution for the new question posed at it, and finally calculates the answer.
* Note that, we have used one example here to train the model on the solution procedure, and hence this is known as **one-shot CoT prompting**.

#### EXAMPLE -2

|  |
| --- |
| **query = """ Let's think through this step by step.**  **Scenario:**  **- Patient has a PPO plan.**  **- Procedure: MRI (advanced imaging).**  **- Location: Out-of-network imaging center.**  **- The plan requires pre-authorization for advanced imaging.**  **Step-by-step:**  **1. Does the PPO plan require pre-authorization for advanced imaging?**  **2. Is the MRI considered advanced imaging?**  **3. Does it matter that it's out-of-network?**  **4. Who typically initiates it?**  **5. Give overall conclusion.**  **"""** |
| **OUTPUT** *Let's evaluate the scenario step by step:*  *1. \*\*Does the PPO plan require pre-authorization for advanced imaging?\*\**  *Yes, the scenario specifies that the PPO plan requires pre-authorization for advanced imaging procedures.*  *2. \*\*Is the MRI considered advanced imaging?\*\**  *Yes, an MRI is classified as an advanced imaging procedure, so it falls under the requirement for pre-authorization.*  *3. \*\*Does it matter that it's out-of-network?\*\**  *Yes, being out-of-network does matter. Many PPO plans have different rules and potential penalties for out-of-network services, including possibly higher out-of-pocket costs for the patient. However, the requirement for pre-authorization still applies regardless of whether the imaging center is in-network or out-of-network.*  *4. \*\*Who typically initiates it?\*\**  *The pre-authorization process is typically initiated by the healthcare provider (in this case, likely the physician ordering the MRI). The provider submits the necessary information to the insurance company for approval before the patient undergoes the procedure. Sometimes, the patient may need to be involved in providing information, but generally, it's the responsibility of the provider.*  *5. \*\*Give overall conclusion.\*\**  *In this scenario, the patient with a PPO plan needs to ensure that the MRI (considered advanced imaging) is pre-authorized because the plan mandates it for such procedures. The fact that the imaging center is out-of-network does not negate the need for pre-authorization; it may complicate coverage and costs, but the requirement remains. The healthcare provider requesting the MRI is typically responsible for initiating the pre-authorization process. To avoid unexpected expenses or denied claims, it’s essential to confirm pre-authorization and understand the implications of using an out-of-network facility*. |

### TREE OF THOUGHT(TOT) PROMPTING

* **Tree of Thought** prompting techniques is specially useful when solution to a problem has more than once solution
* Instead of jumping to a single answer we ask the model to evaluate all possible options and the give the final recommendations.

|  |  |
| --- | --- |
|  | * The model make use Breadth First Search algorithm which looks at each possible path and then come with a final recommendation. |

|  |
| --- |
| * Tree of Thought (ToT) prompting is an advanced prompting technique that encourages large language models (LLMs) to explore multiple reasoning paths instead of following a single linear chain of thought. * Inspired by human decision-making, ToT structures reasoning as a tree, where each node represents a partial solution and branches represent different possible next steps. * It combines the strengths of deliberation and evaluation, allowing the model to explore, backtrack, and select the most promising path, enhancing accuracy and coherence. * ToT prompting is particularly useful for complex problem-solving tasks such as math word problems, reasoning puzzles, coding, and multi-step decision-making. * By simulating internal deliberation, Tree of Thought prompting improves performance over standard chain-of-thought methods, especially when reasoning requires trial-and-error or comparison of alternatives. |

|  |  |
| --- | --- |
| Example  **Task**: “Find the best way to organize a 3-day AI workshop.” | **ToT Prompting Behavior**:   * Branch 1: Focus on lectures → Pros/Cons * Branch 2: Mix of lectures + hands-on labs → Pros/Cons * Branch 3: Panel discussions + networking → Pros/Cons   **Final Output**: “Option 2 is best because it balances learning and engagement.” |

#### EXAMPLE

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  import os  from dotenv import load\_dotenv  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  prompt = ChatPromptTemplate.from\_messages([{"role": "user", "content": "{query}"}])  chain = prompt | llm  query = """ You are a health insurance advisor evaluating the best plan for a patient.              Question: Should a patient with chronic diabetes and hypertension be offered a standard health plan or a specialized chronic care plan?              Think in multiple ways (Tree of Thought):              1. Think based on cost-effectiveness.              2. Think based on patient health outcomes and care coordination.              3. Think based on long-term insurance risk and sustainability.              Evaluate each path and provide a final recommendation with reasoning.              Display a tree-structure with proper blocks to show the paths.          """  response = chain.invoke({"query": query})  print(response.content) |
| **OUTPUT**  **Decision Tree: Evaluating Health Insurance Plans for a Patient with Chronic Diabetes and Hypertension**  **```**  **Evaluate Health Insurance Plans**  **/ \**  **Standard Health Plan Specialized Chronic Care Plan**  **/ \ / \**  **Cost-Effectiveness Health Outcomes Cost-Effectiveness Health Outcomes**  **/ \ / \ / \ / \**  **Lower Higher Suboptimal Optimal Lower Higher Suboptimal Optimal**  **/ \ / \ / \ / \**  **Long-Term Risk Treatment Long-term Best Care Long-Term Risk Treatment Long-term Best Care**  **and Sustainability Issues Adherence Coordination and Sustainability Issues Adherence Coordination**  **```**  **### Path Evaluations**  **1. \*\*Cost-Effectiveness:\*\***  **- \*\*Standard Health Plan:\*\***  **- \*\*Lower Cost:\*\* Lower premium but may not cover specialized treatments or frequent monitoring required for chronic conditions. This could lead to higher out-of-pocket costs.**  **- \*\*Higher Cost:\*\* If the standard plan leads to uncoordinated care and emergency visits due to uncontrolled conditions, overall expenses may rise.**  **- \*\*Specialized Chronic Care Plan:\*\***  **- \*\*Lower Cost:\*\* Upfront, the premium could be higher, but long-term savings could result from reduced emergency care and hospitalizations.**  **- \*\*Higher Cost:\*\* If the chronic care program does not reduce the need for more costly interventions, it may not be cost-effective.**  **2. \*\*Patient Health Outcomes and Care Coordination:\*\***  **- \*\*Standard Health Plan:\*\***  **- \*\*Suboptimal Outcomes:\*\* May not provide necessary ongoing care coordination and management; risks missing critical indicators of diabetes and hypertension.**  **- \*\*Optimal Outcomes:\*\* Only if the patient is highly proactive and self-managing, which is often unrealistic for patients with chronic issues.**  **- \*\*Specialized Chronic Care Plan:\*\***  **- \*\*Suboptimal Outcomes:\*\* If poorly designed, could lead to inadequate care.**  **- \*\*Optimal Outcomes:\*\* Tailored management, regular monitoring, and education improve adherence and reduce complications.**  **3. \*\*Long-Term Insurance Risk and Sustainability:\*\***  **- \*\*Standard Health Plan:\*\***  **- \*\*Long-Term Risk:\*\* Higher risk for insurers due to potential spike in claims when chronic patients neglect care.**  **- \*\*Sustainability:\*\* May experience financial pressure if costs escalate unexpectedly due to poorly managed care.**  **- \*\*Specialized Chronic Care Plan:\*\***  **- \*\*Long-Term Risk:\*\* Better risk management for the insurer, as these plans are designed to handle chronic conditions efficiently.**  **- \*\*Sustainability:\*\* More sustainable due to predictable costs associated with managed care programs.**  **### Final Recommendation:**  **\*\*Recommended Plan:\*\* \*\*Specialized Chronic Care Plan\*\***  **\*\*Reasoning:\*\***  **- \*\*Cost-Effectiveness:\*\* While the short-term premiums might be higher, the potential for reduced overall healthcare costs through preventive care and fewer emergency interventions makes it cost-effective in the long run.**  **- \*\*Patient Health Outcomes:\*\* The specialized care model offers tailored management, better health outcomes, and improved adherence, which is essential for managing chronic conditions effectively.**  **- \*\*Long-Term Risk:\*\* By investing in a care plan that targets chronic conditions, the sustainability of care for the patient and the insurer is improved, reducing the risk of unexpected healthcare costs in the future.**  **In conclusion, for a patient with chronic diabetes and hypertension, the benefits of a Specialized Chronic Care Plan far outweigh the potential downsides, making it the most suitable option for both patient care and insurance sustainability.** |

### SELF-CONSISTENCY PROMPTING

* Instead of asking the model for just one answer, you ask it multiple times (or in multiple reasoning “paths”), then take a vote or choose the most common/frequent answer.
* LLMs sometimes make small reasoning mistakes. By exploring multiple reasoning paths, you reduce the chance that you stick with a single flawed chain of thought.
* Analogy: Asking multiple friends the same question and trusting the answer most of them agree on.

How it works:

Prompt the model to produce step-by-step reasoning multiple times (different random seeds or temperature > 0).

Compare the final answers.

Pick the most frequent / consistent one.

Example:  
If solving a math problem:

Q: What is (24 ÷ 3) + (4 × 2)?

Run the reasoning 5 times — if 4 of them say “16” and 1 says “20,” you keep 16.

Use case:  
Math problems, logic puzzles, coding answers, or anything where correctness matters.

### META-PROMPTING

* Writing a prompt that tells the model how to create or improve prompts — in other words, prompting the AI to be better at prompting itself or others.
* It’s like telling the model: *“Before you answer, think about how to ask the best possible question.”*

How it works:

Your prompt can ask:  
*“You are a prompt engineer. Given this task, generate the best prompt that would lead to an optimal answer.”*

Then, you use the generated prompt for the actual task.

* Analogy: Asking a friend, “What’s the best way to ask this question so I get a great answer?”

Example:

Meta Prompt: "You are an expert prompt designer. Given the task 'Summarize a news article for a 10-year-old', write the best possible prompt for ChatGPT to accomplish this."

Output:  
*"Summarize the given news article in simple language so a 10-year-old can understand, using short sentences and clear examples."*

Use case:  
Optimizing instructions, refining queries, and automating prompt design.

### PROMPT CHAINING

* Breaking a complex task into multiple smaller prompts and feeding the output of one as the input to the next.
* Why:  
  LLMs can get lost in long, complex tasks. Breaking them into steps improves accuracy and allows mid-process corrections.
* Analogy: Asking one friend to gather info, another to analyze it, and another to present it nicely — step by step

How it works:

Prompt 1: Gather facts or extract data.

Prompt 2: Use those facts to do reasoning.

Prompt 3: Format the final result.

Example:  
Task: “Write a children’s story about renewable energy.”

Prompt 1: “List 5 fun facts about renewable energy for kids.”

Prompt 2: “Turn these facts into a story outline.”

Prompt 3: “Write the full story based on this outline.”

Use case:  
RAG pipelines, multi-step reasoning, document summarization, data-to-report flows.

### STATEFUL COMMUNICATION

* Stateful communication means, designing the prompts in such a way that the model maintains the awareness of past interaction and context.
* This allows coherent, personalized and context-sensitive responses.
* Stateful communication is also referred to as **multi-turn conversation.**
* Designing the prompts that maintain stateful communication is very essential is developing bots.
* Stateful communication can be achieved in two ways:
  + **Message History:** Most chat APIs require you to send prior messages along with the new one to maintain state.
  + **Longterm Memory:** Some systems use external memory like ChatGPT's memory feature, Custom app memory in LangChain or RAG systems.

|  |
| --- |
| Example:   * **User**: “Remind me what we decided about the workshop schedule.” * **Model**: “We agreed on a mix of lectures and labs over 3 days.”   Later...   * **User**: “Add a networking session on Day 2.” * **Model**: “Done. Your updated schedule now includes networking on Day 2.” |

#### EXAMPLE

|  |
| --- |
|  |
|  |

# MODEL TYPES (LLM TYPES)

## CLASSIFICATION BASED ON – HOW THEY ARE TRAINED

A diagram of a software development process

AI-generated content may be incorrect.

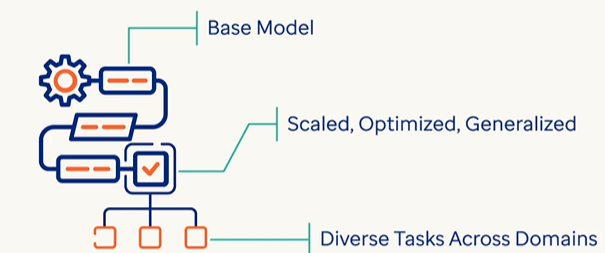
BASE MODEL

|  |  |
| --- | --- |
|  | * Base MODEL: The base model is outcome of pre-training phase on a large corpus of text using self-supervised learning (e.g., predicting the next word). * These models can process language but not optimized for further adaptation. |

|  |
| --- |
| * Self-supervised learning is a type of machine learning where a model learns from unlabeled data by creating its own labels from the data itself.   **HOW IT WORKS**  Imagine a sentence:  "The cat sat on the \_\_\_."  The model is trained to predict the missing word ("mat") using the rest of the sentence. No human labeled this data — the model learns by solving puzzles it creates from raw text.  **KEY FEATURES:**   * No manual labeling needed. * The model learns patterns, grammar, and meaning from large text corpora. * Used heavily in training base models like BERT, GPT, etc. |

* + Purpose: Learns general language patterns, grammar, facts, and reasoning.
  + Example: GPT-3 before any fine-tuning.
  + Limitation: Not optimized for specific tasks or instructions.

Foundational Model



* A broader term that includes base models and other large-scale models trained on diverse data.
* Built on base model – which are scaled , optimized for multitask adaptability
* It can perform diverse task across multiple domains without requiring task specific training.
  + Purpose: Acts as a foundation for building more specialized models.
  + Example: PaLM, GPT-4, LLaMA—used as starting points for downstream tasks.
  + Note: All base models are foundational, but not all foundational models are used as-is.

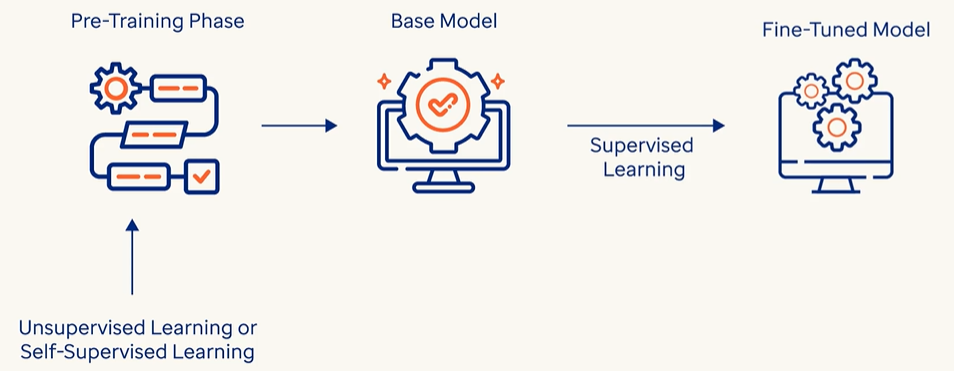
Instruction-Tuned MODEL



* A foundational or base model further trained to follow natural language instructions.
  + Purpose: Makes the model more helpful, safe, and aligned with human intent.
  + How: Trained on datasets like “prompt → response” pairs.
  + Example: InstructGPT, ChatGPT.

Fine-Tuned Model

* A model adapted to perform specific tasks or domains (e.g., legal, medical, customer support).
* The base model is further tuned using dataset using supervised learning.

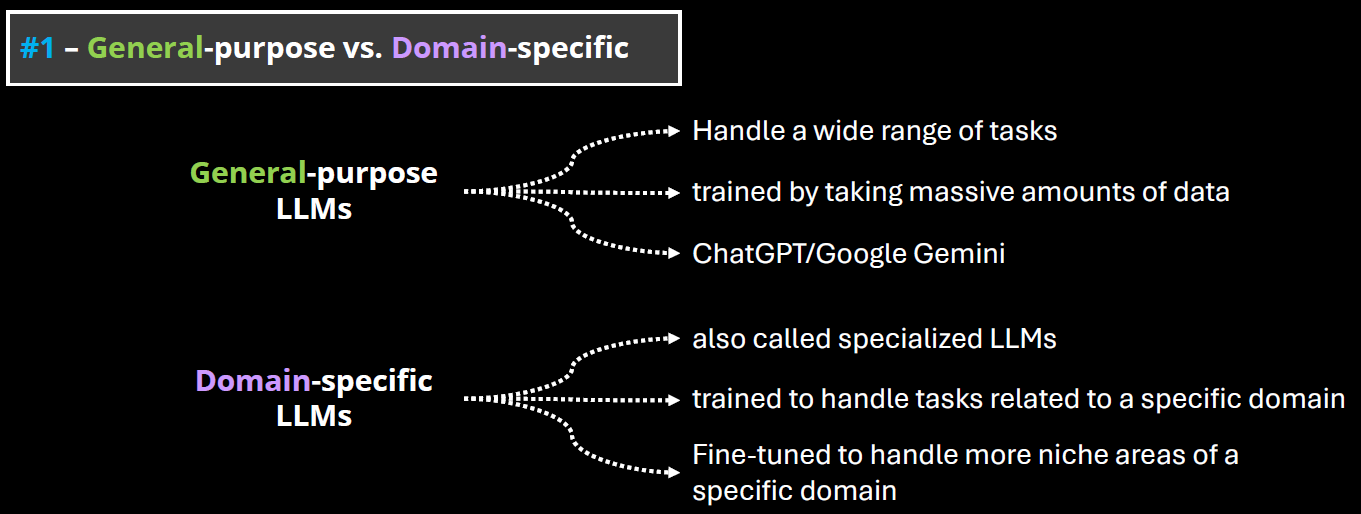


* Purpose: Improves performance on narrow use cases.
* How: Trained on labeled data or domain-specific examples.
* Example: A GPT model fine-tuned for legal document summarization.

## CLASSIFICATION BASED ON – HOW THEY ARE USED

### GENERAL PURPOSE AND DOMAIN-SPECIFIC LLMS

The distinction between General Purpose LLMs and Domain-Specific LLMs lies in their training data, capabilities, and intended use cases.:



#### GENERAL PURPOSE LLMS

* These are large language models trained on a broad and diverse dataset that spans many domains (e.g., science, literature, law, medicine, pop culture, etc.).
* Examples: GPT-4, Claude, Google Gemini, LLaMA.
* Strengths:
  + Versatility: Can handle a wide range of tasks (e.g., summarization, translation, coding, creative writing).
  + Adaptability: Can generalize well across different topics and user needs.
  + Scalability: Useful in applications where domain-specific knowledge is not required.
* Limitations: May lack deep expertise in specialized fields.

#### DOMAIN-SPECIFIC LLMS

* These are LLMs trained or fine-tuned on specialized datasets from a particular field (e.g., legal, medical, financial, scientific).
* Examples: Med-PaLM (medical), FinGPT (finance), BioGPT (biomedical), Legal-BERT (legal texts)
* Strengths:
  + High accuracy in domain-specific tasks.
  + Better understanding of terminology, context, and nuances in the field.
  + Often used in regulated industries where precision is critical.
* Limitations:
  + Limited generalization outside their domain.
  + May require frequent updates to stay current with domain knowledge.
  + Less flexible for multi-domain tasks.

#### OPEN AND CLOSED SOURCE LLMs

**The distinction between Open-Source and Closed-Source LLMs revolves around accessibility, transparency, control, and community involvement.**

**A diagram of a source

AI-generated content may be incorrect.**

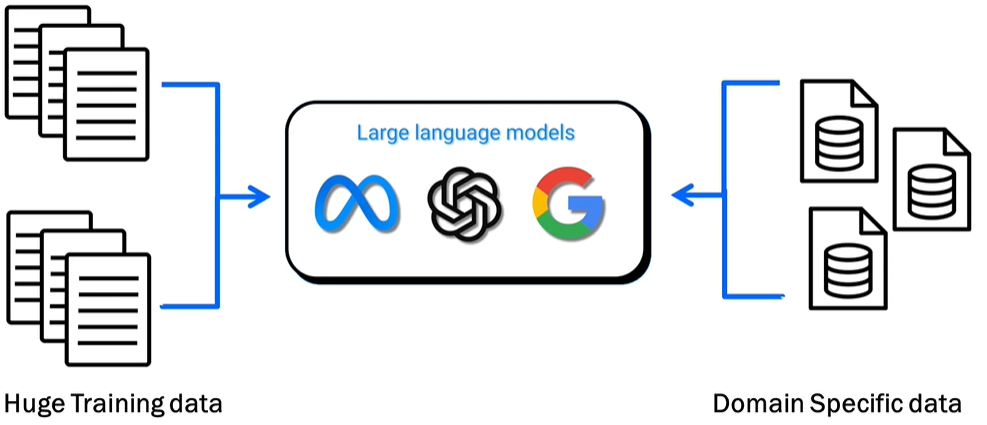
##### OPEN-SOURCE LLMS

* These models have their code, weights, and training data (or methodology) publicly available. Anyone can inspect, modify, or fine-tune them.
* Examples: Meta’s LLaMA (e.g., LLaMA 2, LLaMA 3), Mistral, Falcon,OpenChat, BLOOM (by BigScience)

CLOSED-SOURCE LLMS

* These models are proprietary. The model weights, training data, and architecture details are not publicly available.
* Examples: OpenAI’s GPT-4, Anthropic’s Claude, Google’s Gemini, , Cohere Command R+,Amazon Titan

## FINE TUNING



WHAT IS FINE TUNING

* Adjusting a pre-trained large language model (LLM) to provide better results on specific tasks or domain-specific datasets (e.g., healthcare, finance, etc.).

WHAT IS THE PURPOSE OF FINE TUNING

* Enhance the model's ability to generate focused and precise results on specialized datasets.
* Example: Fine-tuning an LLM on medical data improves its accuracy in answering medical queries compared to a general-purpose/pre-trained model.

### FINE TUNING TECHNIQUES

**1. Full Fine-Tuning**

**What it is**: Updating all model parameters using labeled data.

**Pros**: High flexibility and performance gains.

**Cons**: Computationally expensive, risk of overfitting, requires large datasets.

**Use case**: Domain-specific models (e.g., legal, medical).

**🧩 2. Adapter-Based Fine-Tuning**

**What it is**: Introduces small trainable modules (adapters) between layers of the frozen base model.

**Pros**: Efficient, modular, avoids catastrophic forgetting.

**Popular variants**: AdapterFusion, Compacter.

**Use case**: Multi-task learning, low-resource environments.

**🧠 3. LoRA (Low-Rank Adaptation)**

**What it is**: Injects low-rank matrices into the attention layers to reduce the number of trainable parameters.

**Pros**: Lightweight, fast training, minimal memory footprint.

**Use case**: Personalization, rapid prototyping.

**🧵 4. Prefix Tuning / Prompt Tuning**

**What it is**: Learns a fixed set of tokens (prefix or prompt) that steer the model’s behavior.

**Pros**: Extremely parameter-efficient, fast.

**Cons**: Limited flexibility compared to full fine-tuning.

**Use case**: Task-specific tuning, few-shot learning.

**🧪 5. Instruction Tuning**

**What it is**: Fine-tuning on datasets where tasks are framed as instructions.

**Pros**: Improves generalization across tasks, aligns model behavior with human intent.

**Use case**: Chatbots, general-purpose assistants.

**🧬 6. Reinforcement Learning from Human Feedback (RLHF)**

**What it is**: Uses human preferences to guide model outputs via reinforcement learning.

**Pros**: Aligns model with human values, improves helpfulness and safety.

**Cons**: Complex pipeline, expensive to scale.

**Use case**: Alignment-focused models like ChatGPT.

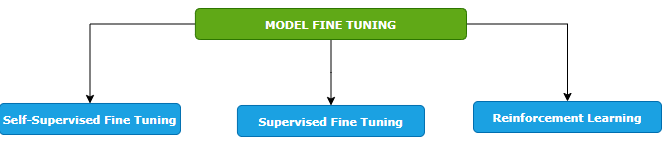
**🧮 7. Quantization-Aware Fine-Tuning**

**What it is**: Fine-tuning while converting model weights to lower precision (e.g., INT8).

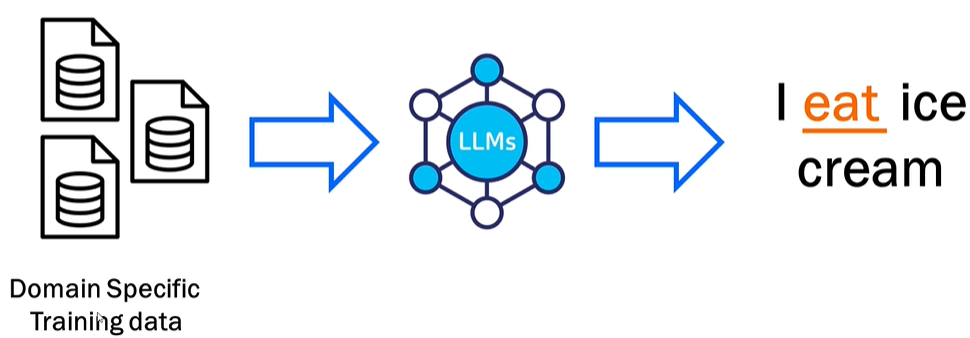
**Pros**: Reduces model size and inference cost.

**Use case**: Edge deployment, mobile devices.

### DIFFERENT WAYS TO FINE TUNNING A MODEL



#### SELF-SUPERVISED FINE TUNING



* Training the model on a domain-specific dataset. Means – we give the foundation model a big pile of training data that is specific to our domain, and the model will learn from it. In this way, the model learns to predict missing pieces of data.
* Example - when we say I ice cream, the model predicts that the missing word is eat.
* This is like how the foundation model is trained but the key difference here is that we are fine tuning the model by providing the domain specific data set. Example - Like if we want to train it on health care data set, we will pass it - the drug structure, scientific studies, all the documents that are related to drug and the model will learn from it and it would be able to generate content based on that.

#### SUPERVISED FINE TUNING

A screenshot of a computer

AI-generated content may be incorrect.

* The model is trained using a labeled dataset where both inputs and outputs are provided.
* Example: Input: "How do I find a broken bone?" → Output: "X-ray."
* Helps the model learn more precise responses based on labeled data.

#### REINFORCEMENT LEARNING

|  |  |
| --- | --- |
| A blue arrow with black text  AI-generated content may be incorrect. | A blue arrow pointing to a black arrow  AI-generated content may be incorrect. |
| **LOW SCORE FOR BAD RESULT** | **HIGH SCORE FOR GOOD RESULTS** |

* **Feedback-based learning**.
* The model generates outputs, and scores are assigned based on quality (high score for good results, low score for bad results).
* The model learns overtime from the feedback to improve predictions.

#### KEY FEATURES OF FINE TUNING

1. **STARTS FROM A PRE-TRAINED MODEL**: Fine tuning builds on top of a foundation model already trained on large datasets—it does not start from scratch.
2. **REQUIRES DOMAIN-SPECIFIC DATA**: We must provide good-quality, specific data for training tailored to your use case (e.g., drug data for healthcare).
3. **No Universal SOLUTION**: Each task/use case is unique, requiring case-specific implementation and variations.
4. **ITERATIVE PROCESS**: Fine tuning is repetitive and requires multiple cycles of iteration and adjustments for optimal results.

# EMBEDDINGS

* Embeddings are numerical representations of text—a way to convert words into numbers so machines can understand and process them.
* Machines do not inherently understand text; they operate using numbers. Embeddings allow machine learning models to interpret meaning, context, and relationships between words.

**KEY FUNCTIONALITY OF EMBEDDINGS**

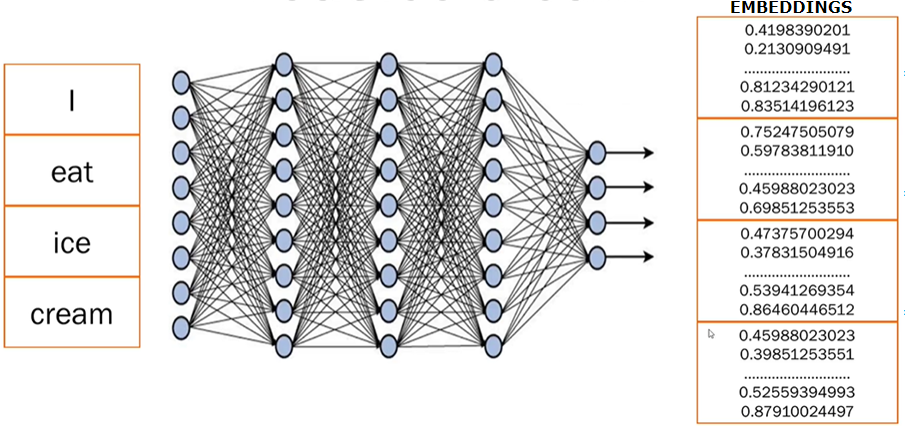
1. **CAPTURING MEANING**: Embeddings reflect the semantic meaning of words or sentences.
2. **CONTEXT UNDERSTANDING**: Embeddings account for the context of text (e.g., same word in different sentences/sentiments - “great” in sarcasm vs happiness).



1. **RELATIONSHIPS**:

* Words with strong associations (e.g., "ice" and "cream") have similar embeddings, reflecting their connection.
* Models understand the proximity(how close the words /sentences are) and sequence of words based on embeddings.

**HOW EMBEDDINGS ARE GENERATED**



1. **STEP 1:** **BREAKING TEXT INTO TOKENS**

* Sentences are broken into smaller pieces or tokens (e.g., splitting "I eat ice cream" into 4 tokens—"I", "eat", "ice", "cream").

2. **STEP 2: NEURAL NETWORK PROCESSING**:

* Trained **transformer models** analyze the text, generate embeddings, and capture meaning, context, and relations between tokens.

1. **STEP 3: NUMERICAL EMBEDDINGS:**

* Each token is converted into numerical data (random numbers).
* These numbers represent embeddings, storing all learned information about the word or sentence.
* Only the transformer model understands what these embeddings mean based on its training.

# VECTORS

A **vector** is a list of numbers that represents something—like a word, an image, or even a sentence—in a way that a computer can understand and work with.

|  |  |
| --- | --- |
| Simple Analogy | |
| Imagine we want to describe a fruit (say, an **apple**) using numbers:   * Sweetness: 8 * Crunchiness: 7 * Juiciness: 6   We could represent the apple as a vector: [8, 7, 6]  Now, a **banana** might be:[9, 3, 7]  These vectors help a computer compare fruits based on their features. | A diagram of a diagram  AI-generated content may be incorrect. |

In Language Models

When we talk about **words**, we turn them into vectors using **embeddings**. For example:

* “cat” → [0.12, -0.45, 0.88, ..., 0.03]
* “dog” → [0.10, -0.40, 0.85, ..., 0.05]

These vectors are **high-dimensional** (often 300 to 1,000+ numbers long) and capture the **meaning** of the word based on how it’s used in language.

## WHY ARE VECTORS USEFUL?

They allow computers to:

* **Compare** things (e.g., how similar two words or images are)
* **Search** by meaning (semantic search)
* **Cluster** similar items together
* **Feed data into machine learning models**

## VECTOR DATABASE

A diagram of a data processing process

AI-generated content may be incorrect.

* A **vector database** is a special kind of database designed to store and search **vectors (***which are just lists of numbers that represent things like text, images, or audio in a way that computers can understand*.)
* Primarily used for storing embeddings that represent complex data like images, text and audio in a form that machine can understand and process

Why Vectors?

* When you use **embeddings** (like we discussed earlier), we turn data (like the word *“cat”*) into a vector, such as:

**[0.12, -0.45, 0.88, ..., 0.03]**

* These vectors capture **meaning** and **context**. But once we have millions of them, we need a smart way to **store** and **search for** them efficiently. That’s where vector databases come in.

What Does a Vector Database Do?

It helps to:

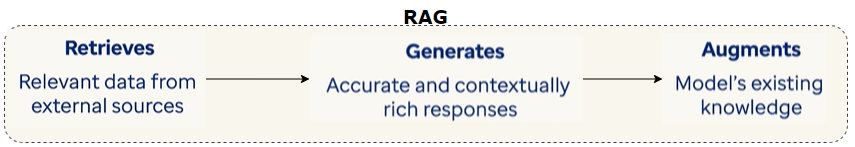
* **Store** millions or billions of embeddings
* **Search** for the most similar vectors (e.g., “find texts similar to this one”)
* **Rank** results by similarity (using distance metrics like cosine similarity)

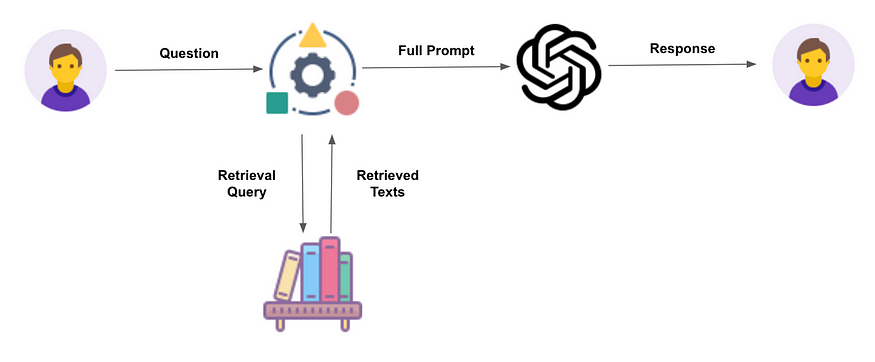
Real-World Example

Let’s say we run a **document search engine**:

1. We convert all the documents into vectors using an LLM.
2. We store those vectors in a vector database (like **Pinecone**, **Weaviate**, **FAISS**, or **Milvus**).
3. When a user asks a question, we:
   * Convert the question into a vector
   * Search the database for the **most similar document vectors**
   * Return the most relevant documents

# RAG (RETRIEVER AUGUMENTED GENERATION)





* **Retriever**: Fetches relevant information from a knowledge base.
* **Augmented**: Adds value by combining retrieved data with generative capabilities.
* **Generation**: Produces a response using a language model (e.g., ChatGPT, Gemini).

## RAG WORKFLOW

|  |
| --- |
| Step 1: Ask a Question –   * When we ask a question. The question goes into a knowledge base (like a smart library). * The knowledge source can be Databases, Articles or Websites * This knowledge base is often a vector database, which stores information in a format that helps find similar meanings.   Step 2: It Finds Relevant Text   * The system retrieves the most relevant documents or pieces of information that match the question. In the vector Database. This is called the retrieval step.   Step 3: It Builds a Full Input   * The retrieved information is combined with the original question. * This combination becomes a full prompt or input for the next step.   Step 4: AI Generates the Answer   * **Full prompt is sent to a Language Model (like ChatGPT or Gemini).** * **Note**: Retrieved data + original query = **prompt** for the **Language Model (LM)**. * The model reads both questions and the retrieved info and then generates a smart answer. * The response is based on internal data (from the knowledge base) and the intelligence of the AI model. |

## WHY RAG?

**RAG is important due to the following limitations of LLMs**

1. KNOWLEDGE CUT-OFF DATE
   1. The LLM model will have information up to their training cut-off date and lack of information beyond that point.
   2. Hence LLMs **can’t access real-time updates** or dynamic data, and they may miss recent changes in the organization or product.
2. LACK OF ACCESS OF ENTERPRISE DATA
   1. They lack access of enterprise specific data unless they are fine-tuned for customized for that enterprise

|  |
| --- |
| * Let’s say in banks or an enterprise company, the data will be sitting privately within companies private infrastructure or within companies’ network. * In this use-case - We don't want ChatGPT or any other LLMs to have information about private data. So that is where vector databases come into picture where the questions retrievals will happen from the company’s internal source. * Then – we are just using the capabilities of LLM to prepare a nice prompt to give a proper reply. |

Enterprise Use Cases

* **Customer support**: Pulls product details from internal DBs.
* **Educational tools**: Provides precise, sourced answers.
* **Banking/Enterprise**: Keeps sensitive data internal while leveraging LMs.

## BENEFITS

Access to Private/Internal Data

* RAG allows LLMs to use your organization’s internal documents (e.g., product manuals, policies, reports) that are not part of public training data.
* This makes responses more relevant and accurate for enterprise use.

Reduces Hallucinations

* LLMs sometimes make up answers when they don’t have enough information.
* RAG reduces this by grounding responses in real, retrieved documents.

Improves Contextual Accuracy

* RAG retrieves context-specific information before generating a response.
* This ensures the answer is tailored to the user’s query and environment.

Keeps Data Secure

* Sensitive data stays within private infrastructure.
* The LLM only sees the retrieved content, not the entire database—helping with data privacy and compliance.

Dynamic and Up-to-Date Responses

* Instead of relying on static training data, RAG can pull real-time or recently updated documents.
* This makes it ideal for fast-changing domains like tech support or policy updates.

## RAG ARCHITECTURE

A diagram of a computer system

AI-generated content may be incorrect.Step 1: Prepare Your Data

* Collect documents, images, videos, etc.
* Send them to an **Embedder**, which converts them into a format (vectors) that computers can search easily.

Step 2: Store the Data

* The Embedder sends these vectors to a **Vector Storage and Retrieval Engine** (like a smart library).

Step 3: User Asks a Question

* A person types a question into a **chat interface**.

Step 4: Process the Question

* The question goes to a **User Query module**, then to the **Embedder** to be converted into a vector (just like the data was).

Step 5: Search for Matching Info

* The query vector is sent to the **Vector Storage and Retrieval Engine**.
* It searches for the most relevant information based on meaning (semantic search).

Step 6: Generate the Answer

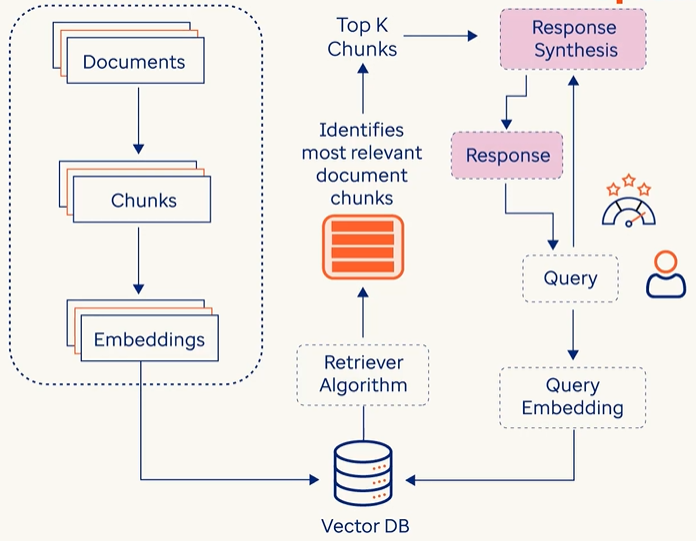
* The retrieved information is sent to a **Large Language Model** (like ChatGPT).
* The model uses both the question and the retrieved info to create a smart, accurate response.

Step 7: Show the Answer

* The response is sent back to the **chat interface**, where the user sees the final answer.

## RAG FRAMEWORK





|  |  |
| --- | --- |
| DATA INGESTION | * In this phase, the system gathers large volume of data from different data sources such as Databases, documents, API or online content * The main goal of data ingestion is to prepare external knowledge in a format that makes the system easier for retrieve and synthesis later * This knowledge comes for structure data like DB tables, unstructured data like text files or even programmatically generated by API calls |
| INDEXING & STORING | * At this stage, each document is split into multiple chunks from which corresponding embeddings (A numerical vector representation of textual chunks, which captures meanings and context) are created * Once the embeddings are created, each chunk along with its embeddings is stored and indexed in vector database |
| RETRIEVAL | * The primary objective of this step is to find relevant documents that has answers to user’s query submitted in natural language * The Query is the converted to embeddings to the vector DB * The Retriever Algorithm, then compare the user’s query and the embedding already stored in vector database * The Algo identified the most relevant document chunks which is like user query embeddings and finally it retrieves the Top K relevant chunks for Vector DB, which are the pieces of information which can answer user’s query |
| RESPONSE SYNTHESIS | * In this phase the Top K chunks and user’s query is passed Response Synthesis Module * This module has LLM which provides the response in natural language |
| QUERY OR CHAT ENGINE |  |
| EVALUATION |  |

## TOOLS TO BUILD RAG SYSTEMS

## LANGCHAIN

* **LangChain** is an open-source framework designed to help developers build powerful applications using **Large Language Models (LLMs)** like ChatGPT, Gemini, Claude, etc., by connecting them with **external data sources, tools, and workflows**.
* It is designed for integrating LMS or language models into applications.
* **LangChain Can be used to implement RAG**

|  |
| --- |
| ANALOGY   * Just like LEGO bricks can be assembled using instructions or imagination:   + LangChain provides **pre-built modules** for common tasks.   + Developers can **customize and combine** these modules to build complex AI workflows. * It’s **modular and flexible**, making it easy to mix and match components. |

**Why LangChain Is Useful**

* **Simplifies AI integration** into apps—no need to build from scratch.
* **Developer-friendly**: Reduces complexity of building AI-powered tools.
* Supports **multi-LLM setups** (e.g., combining ChatGPT and LLaMA).
* Enables **custom workflows** like chatbots, agents, and automation.

|  |
| --- |
| Example: Weather App with LangChain   * A user asks: “What’s the weather in Chandigarh today?” * LangChain:   1. Understands the query needs **real-time weather data**.   2. Calls an **external API or database** to fetch the latest info.   3. Passes the data to an LLM to **generate a natural response**.   4. Delivers: “It’s sunny and 40°C in Chandigarh today.” |