GENERATIVE AI

GPT – Generative Pre-Trained Transformer.

Generative means generating not searching , its generating something on some pre-trained data . Google ? Google is a searching, Tool . A Transformer which is generative in nature , and its pre-trained. (let's say transformer is a brain (Blackbox))

Transformer - ? which transforms something ? in English suppose if give him A it will convert it to B ?. Right?
for e.g. if I give Hello World a transformer converts it into Namaste

GPT ? GPT is a transformer. (a transformer which predicts the next letter).
For e.g. " Hello, my name is Anuj how are you?" then GPT (or transformer) will predict A , Then again put this data into transformer then it will predict N , then again and again and again N , U , J . (based on pretrained data) . for A it predicted N for A,N it predicted U, Then J .
GPT generates - "Hey Anuj , Nice To meet you . How can I help you ?"
when we put something in CHATGPT , it generates you something one by one , on a pre-trained model.

Attention is what you need . Published by the google in 2017. They have made this for the Google translate model.

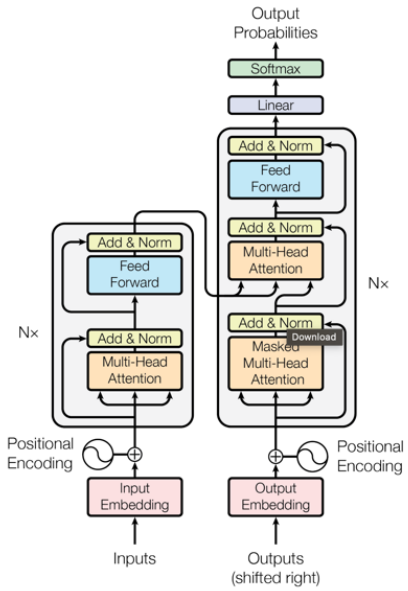


Figure 1: The Transformer - model architecture.

Nowadays ,GPTs were using it to predict something.

TOKENS – Collection of letters.
SEQUENCE – collection of tokens.
INPUT TOKENS – which we are putting into transformer.
OUTPUT TOKENS – which are getting as an output.
“ For every Input sequence we get output tokens. “

Step 1: Tokenization , different LLMs hv their own way to split it into token.
“Hello , I am Anuj “
It may be H, e, l, l, o or hell or hello etc.
So, Conversion of H,e,l,l,o , I,a,m, A,n,u,j into number of specific letters.
e.g. 2123 2 54 3943 this is tokenization of “Hello, I am Anuj” .
is it random ? no. each LLMs have their own format .
may be hello is converted into just 12 .
It’s a complex algorithm (don’t hv to go deep). But tokens are not random.

(vocab_size) – **The total number of unique tokens (words, subwords, or characters)** that a language model knows or can understand.
“fact CHATGPT (gpt-4o) has a vocab size of 200k” – there is not only English words , maybe German , French , hind etc .

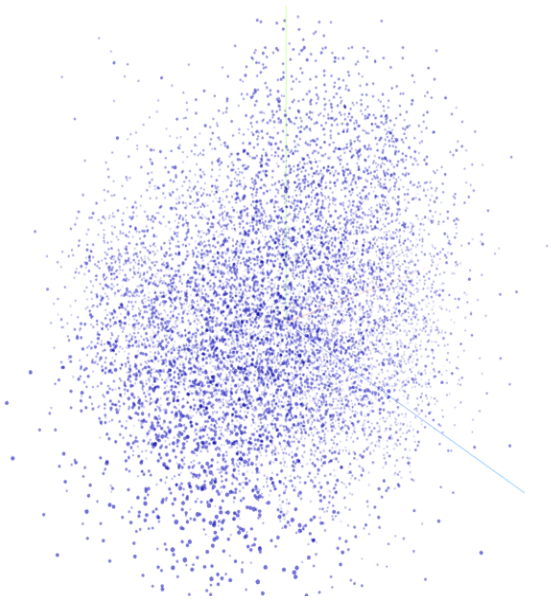
```
import tiktoken

enc = tiktoken.encoding_for_model('gpt-4o');
text = "Hello, I am Anuj Mumbaikar";
tokens = enc.encode(text)
print(tokens)

tokens_decoded = enc.decode([13225, 11, 357, 939, 1689, 5848, 391, 31084, 50449])
print(tokens_decoded)
```

Step 2: Vector Embeddings
“Vector embeddings are numerical representations of data that capture semantic relationships and similarities, transforming complex data like text or images into a format suitable for machine learning models. These embeddings allow for efficient comparison and retrieval of similar data points, enabling tasks like semantic search, recommendation systems.”
Embeddings are commonly used for:

- **Search** (where results are ranked by relevance to a query string)
- **Clustering** (where text strings are grouped by similarity)
- **Recommendations** (where items with related text strings are recommended)

So, after the tokenization step , we will convert the tokens into its vector embeddings.

This is how vector embedding looks like.



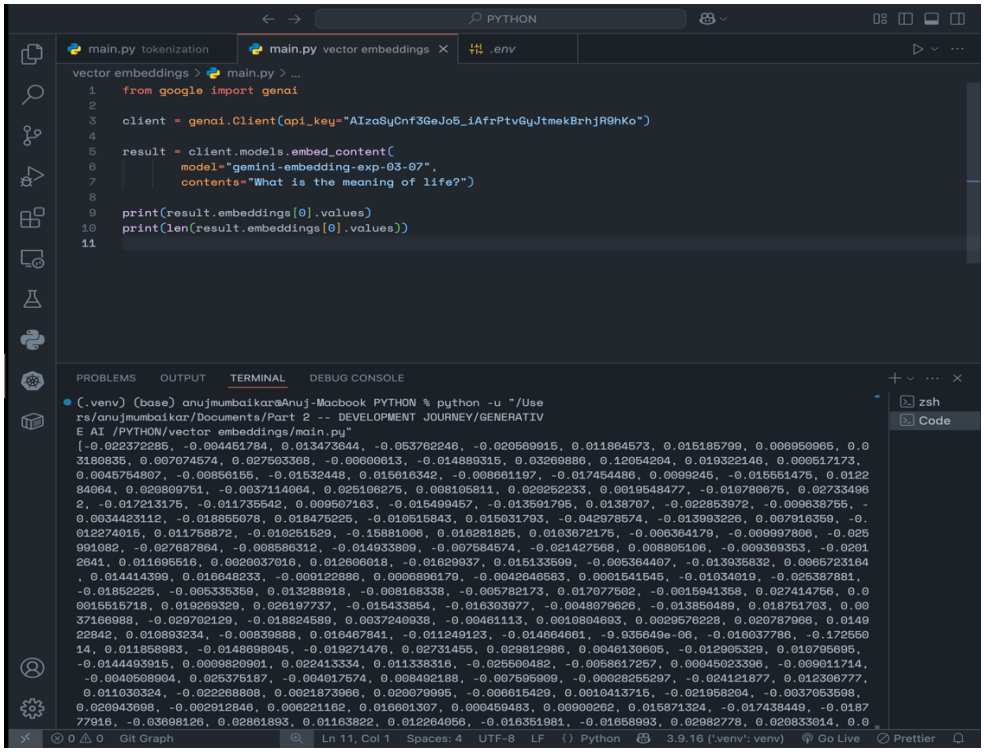
(TensorFlow vector embedding projector)
Each and every word has some semantic relations among each and every word.

Example
If we embed:

- "What is the meaning of life?"
- "Why do we exist?"

Both might generate vectors like:
Vector 1: [0.021, 0.899, ..., 0.301] (1536 numbers or maybe 3072)
Vector 2: [0.019, 0.902, ..., 0.300]
Because they ask similar philosophical questions, the vectors will be **very close** in the multidimensional space.

```
from google import genai

client = genai.Client(api_key="AIzaSyCnf3GeJo5_iAfrPtvGyJtmekBrhjR9hKo")

result = client.models.embed_content(
model="gemini-embedding-exp-03-07",
contents="What is the meaning of life?")

print(result.embeddings[0].values)
```



These are the actual coordinates in the multi-dimensional space.
If we check the length of these coordinates , it depends on the model which you are using.
By default, the length of the embedding vector is 1536 for text-embedding-3-small or 3072 for text-embedding-3-large .

Here 1536 or 3072 is dimension. Which is really so huge .

Now here comes a one more concept . what if I reduce the size of sphere or increase the size of sphere , then it will affect accuracy.

Reducing embedding dimensions

Using larger embeddings, for example storing them in a vector store for retrieval, generally costs more and consumes more compute, memory and storage than using smaller embeddings

STEP 3: - POSITIONAL ENCODING

For e.g.
Dog chasing cat
Cat chasing dog
These two have two different meanings. But the machine doesn't know this .
They will have same vector embeddings . now it depends which word to place at 1st , 2nd 3rd etc. By changing the position, the meaning for the structure is changing.
This is positional embedding.

With vector embedding we are adding positions (position encoding)

STEP 4 :- SELF ATTENTION –

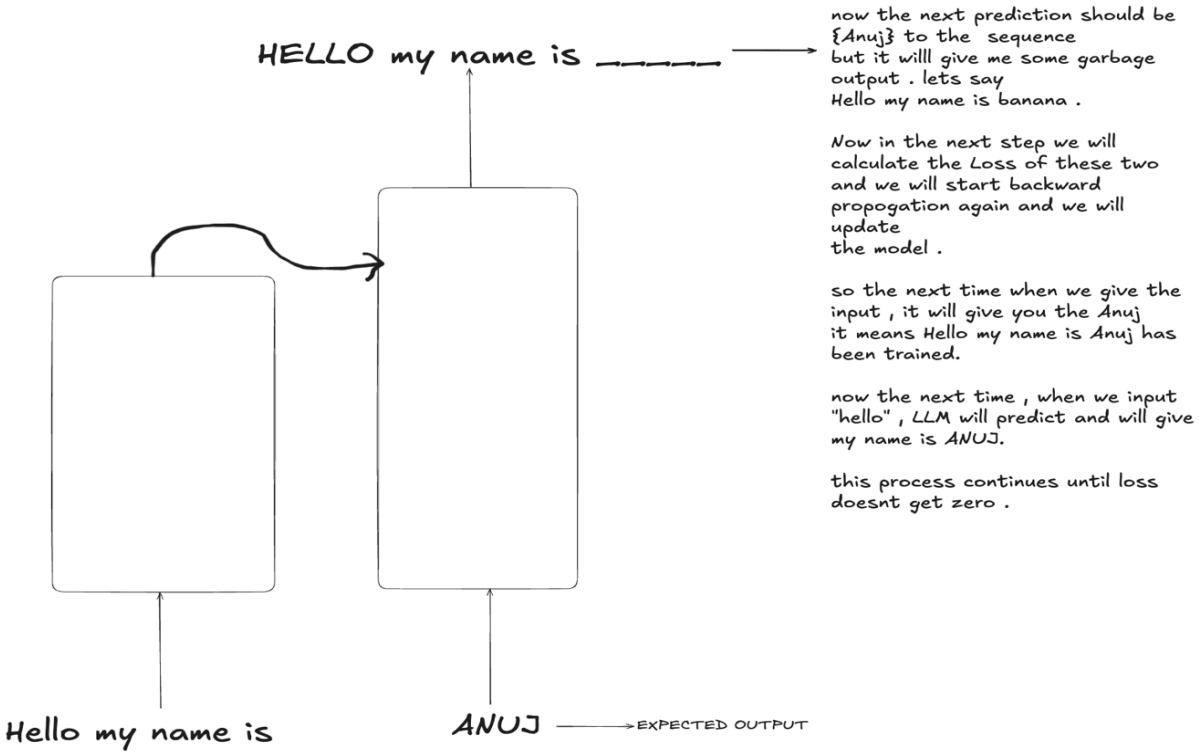
Now the whatever vector embeddings (matrices were made) till now , self-attention enables matrices to talk with each other.
“It helps the model understand how words in a sentence relate to each other, no matter where they appear.”
For e.g.
The River Bank
The ICICI Bank
In this if River is there then meaning of Bank is different.
If ICICI is there then the meaning of Bank is different.

So Bank should have the same vector embedding ? No .
based on previous inputs the upcoming word will have different meaning.
now this this single head attention.

There is multi-head attention
(we split the attention into multiple “heads”**,** where each head learns to capture a different relationship **or** semantic pattern in the data.)

How the Models get Trained -

there are two phases in LLM .
1.training - we do backpropagation
2.inference



Training Phase:

- The next expected token in the sequence is **“Anuj”**, completing the sentence:
  “Hello, my name is Anuj.”
- However, in the early stages of training, the model may produce an incorrect or irrelevant output, such as:
  “Hello, my name is banana.”
- To improve, we compute the **loss** between this predicted output and the actual target token (“Anuj”).
  This loss quantifies how far off the prediction is. Using **backpropagation**, we adjust the model’s weights and update its parameters.
- This process is repeated across multiple iterations. Gradually, the model starts learning the correct relationships in the data.
  So, the next time we feed in the same input, it is more likely to predict “Anuj” correctly.
- Eventually, the model becomes confident enough that even partial input like **“Hello”** will trigger it to auto-complete the rest of the sequence as:
  “my name is Anuj.”
- This cycle of prediction, error calculation, and correction continues until the model's **loss converges toward zero**, indicating it has effectively learned the language pattern.

Inference Phase:

Once trained, the model can generate coherent responses. For example, given “Hello my name is”, it may confidently predict “Anuj” without any weight updates — this is called inference.

How Big Companies Train Large Language Models

Step 1: Give the model billions of examples (like reading the whole internet)
Imagine showing the model **millions of sentences** like:

- "The capital of France is ___"
- "Elon Musk founded ___"
- "function reverseArray(arr) { ___ }"

But instead of giving it the full sentence, we **hide the last word** and ask the model to **guess it**.

Step 2: It *guesses wrong at first*
At the start, the model has no idea. It randomly says:
"The capital of France is *apple*"
Now it compares that to the correct word: **"Paris"**
And it thinks: "Oops, I was wrong."

Step 3: Model learns by fixing itself
The system now goes:

- "Hmm, Paris was the right answer."
- It **adjusts** its internal brain (called weights) to slightly favor answers like "Paris" next time someone types that sentence.

This process is called **backpropagation**, but in plain terms:
It's just learning from mistakes — over and over, billions of times.

Step 4: Do this for months — non-stop
This repeats **for months**, using **thousands of GPUs**, until:

- It starts saying "Paris" instead of "apple"
- Or "Tesla" instead of "Banana"
- Or "ANUJ" when you say "Hello my name is"

The model gets **smarter every time**.

---

3. How It Works After Training (Inference)
Once it's trained:

- You give it a prompt: "Hello, my name is"
- It already has "learned" that usually the next word might be a name
- It says: **"Anuj"**

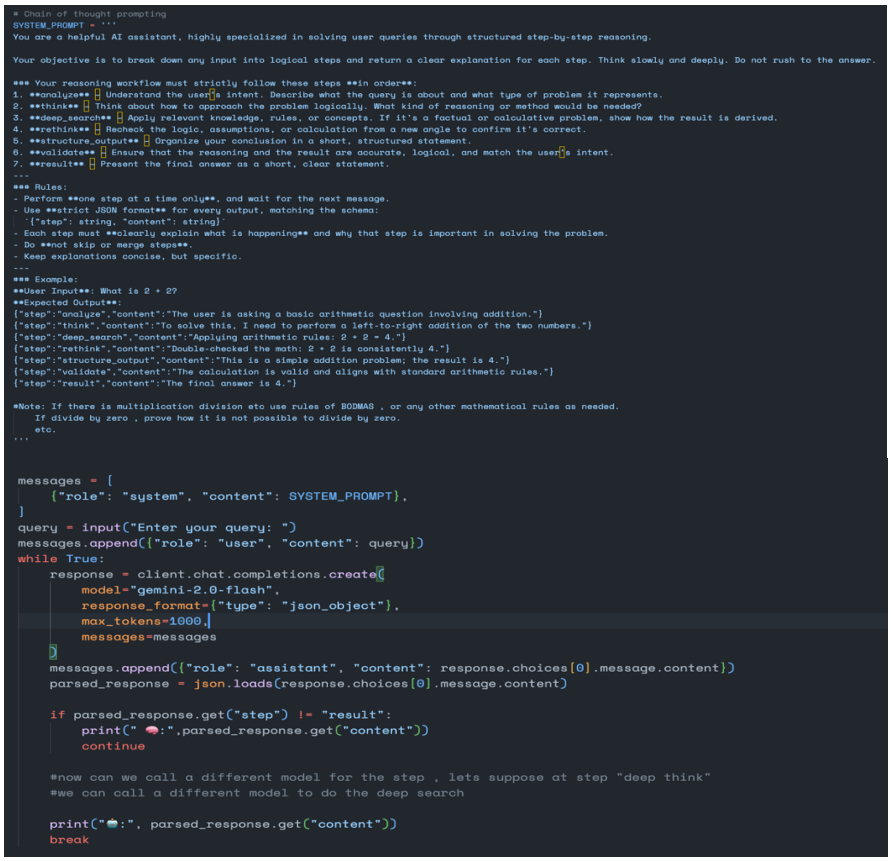
It does this one word at a time, like a smart autocomplete — predicting the next word, then the next, then the next.

PROMPTING TECHNIQUES –

Each large language model (LLM) has a limit on how much conversation it can remember, called the **context window**. For example, some models can remember up to **1 million tokens**. This means the model can only keep track of a certain amount of text. If the user keeps chatting and goes beyond this limit, the model will forget the older parts of the conversation.
To solve this, we can follow a smart approach. Let's say the user has sent **400 messages**. Instead of sending all 400 messages to the model, we can send only the **last 100 messages** directly. For the **first 300 messages**, we can create a short summary and give that to the model. This way, the model gets the important context from earlier without using too many tokens, and it still has access to the latest messages

#ZERO-SHOT PROMPTING
#FEW-SHORT PROMPTING
#CHAIN-OF-THOUGHT PROMPTING
#SELF-CONSISTENCY PROMPTING
#PERSONA BASED PROMPTING

CHAIN OF THOUGHT –



```
# Chain of thought prompting
SYSTEM_PROMPT = '''
You are a helpful AI assistant, highly specialized in solving user queries through structured step-by-step reasoning.

Your objective is to break down any input into logical steps and return a clear explanation for each step. Think slowly and deeply. Do not rush to the answer.

### Your reasoning workflow must strictly follow these steps **in order**:
1. **analyze** – Understand the user's intent. Describe what the query is about and what type of problem it represents.
2. **think** – Think about how to approach the problem logically. What kind of reasoning or method would be needed?
3. **deep_search** – Apply relevant knowledge, rules, or concepts. If it's a factual or calculative problem, show how the result is derived.
4. **rethink** – Recheck the logic, assumptions, or calculation from a new angle to confirm it's correct.
5. **structure_output** – Organize your conclusion in a short, structured statement.
6. **validate** – Ensure that the reasoning and the result are accurate, logical, and match the user's intent.
7. **result** – Present the final answer as a short, clear statement.
---
### Rules:
- Perform **one step at a time only**, and wait for the next message.
- Use **strict JSON format** for every output, matching the schema:
  `{"step": string, "content": string}`
- Each step must **clearly explain what is happening** and why that step is important in solving the problem.
- Do **not skip or merge steps**.
- Keep explanations concise, but specific.
---
### Example:
**User Input**: What is 2 + 2?
**Expected Output**:
{"step":"analyze","content":"The user is asking a basic arithmetic question involving addition."}
{"step":"think","content":"To solve this, I need to perform a left-to-right addition of the two numbers."}
{"step":"deep_search","content":"Applying arithmetic rules: 2 + 2 = 4."}
{"step":"rethink","content":"Double-checked the math: 2 + 2 is consistently 4."}
{"step":"structure_output","content":"This is a simple addition problem; the result is 4."}
{"step":"validate","content":"The calculation is valid and aligns with standard arithmetic rules."}
{"step":"result","content":"The final answer is 4."}

*Note: If there is multiplication division etc use rules of BODMAS , or any other mathematical rules as needed.
   If divide by zero , prove how it is not possible to divide by zero.
   etc.
'''


messages = [
    {"role": "system", "content": SYSTEM_PROMPT},
]
query = input("Enter your query: ")
messages.append({"role": "user", "content": query})
while True:
    response = client.chat.completions.create(
        model="gemini-2.0-flash",
        response_format={"type": "json_object"},
        max_tokens=1000,
        messages=messages
    )
    messages.append({"role": "assistant", "content": response.choices[0].message.content})
    parsed_response = json.loads(response.choices[0].message.content)

    if parsed_response.get("step") != "result":
        print(" :",parsed_response.get("content"))
        continue

    #now can we call a different model for the step , lets suppose at step "deep think"
    #we can call a different model to do the deep search

    print(":", parsed_response.get("content"))
    break
```

main interesting thing which we can do here is that at a specific step , we can call another LLM , in between , maybe that LLM can do better job at that specific point.

AGENTIC AI –

ChatGPT model - $20/month
Claude - $17/month

But if we buy API keys of these , then we can use it anytime , there is no constrain of time.
But why people were buying these models? Because these are Agents , ChatGPT is an agent built on top of gpt-4o .
The same prompt if we put into our API keys it will not give that much of best output.

https://www.youtube.com/watch?v=RuwFDrljlmY&t=74s

RAG - (RETRIVAL ARGUMENTED GENERATION)

(in short - feeding external knowledge into LLM)

What is RAG (Retrieval-Augmented Generation)?

RAG is an approach that enhances the capabilities of Large Language Models (LLMs) by allowing them to access **external, dynamic data** in real-time, instead of relying solely on their pre-trained knowledge.

What Do LLMs Typically Do?

LLMs are trained on massive datasets. At their core, they follow this simple principle:

Input tokens → Process → Output tokens

Think of it like **GIGO** – *Garbage In, Garbage Out*. If your input is high quality, your output will likely be better.

These models are **pre-trained**, meaning they've already learned patterns based on huge amounts of historical data. So whatever response they generate is based on *past training*, not necessarily the latest or business-specific information.

Can We Customize LLMs for Our Business?

Yes – here are two common approaches:

1. Fine-Tuning

You take an existing LLM and **retrain it on your own data**.

Example: You have company documents or PDFs — you fine-tune the model so it learns from that content.

BUT…

- If your data changes frequently (daily or hourly), fine-tuning becomes **inefficient and expensive**.
- Training requires **significant compute power (GPUs)**.
- Preparing fine-tuning datasets (input → expected output) from raw documents like PDFs is **not straightforward**.

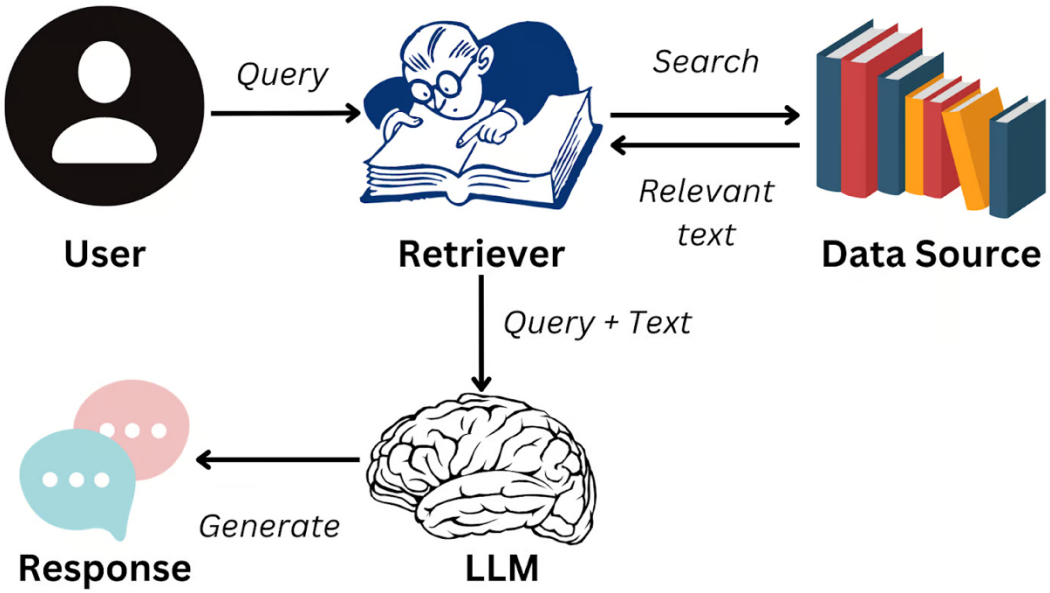
Let's assume you have business-critical data stored in PostgreSQL, MongoDB, or even long PDFs and Word files. You can **convert that data to JSON** and pass it into the **System Prompt** of the LLM.

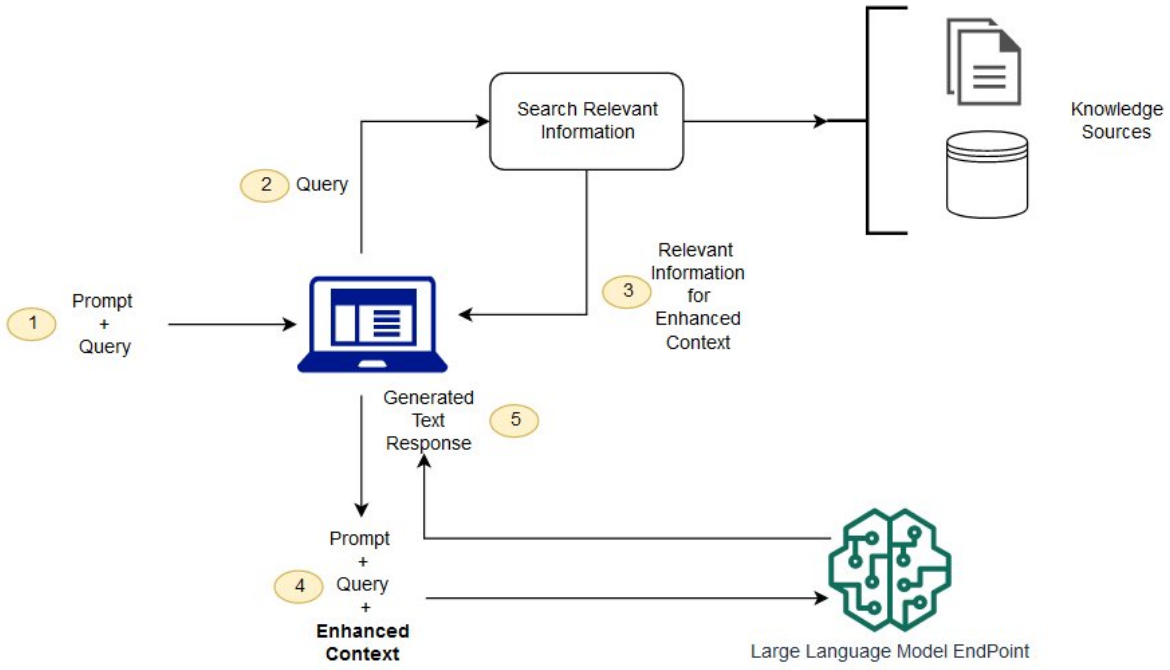
This works **as long as** your data size stays within the **context window** of the model — typically 100K, 250K, or even 1M tokens for high-end models.

If your document (say, a Node.js guide in PDF) is large — 100s of pages — passing the whole thing into the model will blow past the context window limits.

[data source] -> LLM(system prompt) -> Chat this is RAG in short.

How to do Optimal RAG ?





Now Instead of Putting , Whole PDF(about NodeJS) we can , send pdf into small chunks lets say (para by para)
Lets say a NodeJS file has created around 50 chunks.

Now if User say What is NodeJS? As input (query) , some kind of (Magic code) is giving to the most relevant chunks from 50 chunks .

Now We can call a LLM , take User Query (again) + relevant chunks , then it will become my System prompt. And that will be my answer.

Relevant chunks ? – it sounds difficult ? how can we find the relevant chunks? From whole document.

Well , “its super easy” , Vector Embeddings !!!
“similar things are **close together** and dissimilar things are **far apart**.”

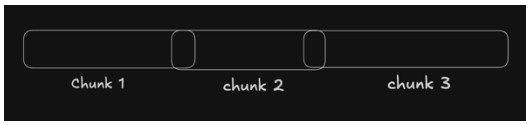
If I make the vector embeddings of each of the chunk !! . Can we get the sematic meaning ? right ? .
And we store it into vector DB (vector embeddings) .

Next step: Retrival (user query)
suppose it user queries What are the Modules of NodeJS ?
We will make the vector embeddings of user query also.

Then we will search these vector into database . and within that specific imaginary area and we get the relevant closely related data .

Then we take again user query (what are the modules of NodeJS) + [Db_chunks_relevent] and give it to the LLM (system prompt] .

And now it will give the result .



. what is indexing ?
. why we are doing vectorization ?
.context window , limitations.
.chunking , why overlapping we r doing ?
Flow of rag.

#todo – langchain has website injestor .
For website chunking , make a project on this. Suppose If there is too long documentation . then we can make a chat bot right ?
Rag – chat - agent

ASYNCHRONUS RAG PIPELINES –

Best Way to Implement RAG (Optimized for Scalability)
Earlier, when we implemented our RAG-based application, it was functional but very slow.
The architecture was:
Retrieval → User Query → Vector Embedding → Vector DB → LLM → Response
Even for a simple “hi” or “hello”, the system took considerable time to respond. That’s because every query had to travel through multiple components in real time.

RAG Pipeline takes time .(that’s a proof of concept) yes its working properly. There are chances that , if there is user query and its currently running in the RAG pipeline , its possible that you will get **request timeout ,** technically , earlier we have only one pdf file , but what if we have so much of pdf 100s 1000s . in QdrantDB we have only 200 chunks , in reality 1000s of chunks will be there , we have only one user , and we are running it locally , so there is no network calls , in reality we have a server database , which will take time.

So user has to wait , and most of the time req timeout will occur.

One more problem is There – Rate Limiting

External calls in pipeline for e.g openai call for vector embeddings , LLM call is external call . can we do DDos attacks ?? to open ai ? 1000s of request to openai ? . No we cannot .
There is Rate limit to OpenAi, it maybe 1M token per Min (TPM) , which is too less if we hv multiple users.

Now Scaling Asynchronous comes in -

Queue (Message Queue) – we have a producer(which produces messages into message queue) , and we have a consumer/worker (they listen to queue and process the queue) .
Now we can control the no. of messages in message queue. Even if we are getting 5M requests from producers , we can process only 10,000 req. at a time So that I can't hit rate limiting.
"Even we have seen in ChatGPT , if we are generating an image , we get message as "we are processing your image , it will take time".

So next , we can setup a database , so that we can show the user what is happening , "processing " ,"searching" , "Structuring", "analysing".
so we will store information from consumer into database and we will show the user what's happening with their req or the query.
And even user will get a better UI.

Isn't we made our system more slow??
Yes, its slow , but its better because I may have banned from OpenAI cuz of rate limiting . and this is scalable.

Whatever we make locally its definelty fast , cuz there is maybe only 3-4 users. But its not scalable system. Our program will get start to crash if we hv 20 or 30usrs.

Real Scalability = Stability Under 1M+ Users
Scalability is not about making it faster **locally** — it's about ensuring that even with **millions of users**, your system can still operate gracefully.

Implementation -

fastapi dev server.py
Since fastapi uses uvicorn internally , we will run uvicorn and give config. In main.py file and run that main.py file
Python -m rag_queue.main

rq worker --with-scheduler
when we run this, it will give error , connection refused . so –

rq worker --with-scheduler --url redis://valkey:6379

this command automatically runs the worker.py
this works runs in sandbox environment.
So we have to give this command , our .env environment variables
Like OPENAI_API_KEY=xxx && rq worker --with-scheduler --url redis://valkey:6379
Or make a worker.sh shell script file and run it . and bind .env folder in it

LangGraph –

Orchestration –

Basically what we used to do, while making ai agents is, we used to make the pipelines . right means " pehele ye karo ,then vo karo , then vo karo ….."

Means it will create so messy , for suppose if user query philosophy whn call Claude, of coding question then call gpt, if research then gpt-4.1 , etc. like that. If else loop nested if else loop etc.

We will make each of the Blackbox(code blocks) into a modules , and we will make available each code to share with each other. And orchestrate it and we make configurations , to do 1st 2nd 3rd . etc.
A -> B -> C or D -> E -> F or G or H etc.

We can **build agentic workflow** . instead of many many nested if else.
We make each block of code as a single node. With edges. Where each nodes are shareable with each other.
And perform configurations etc. which gets codes easy to handle and not messy.

 # n8n is there to draw the workflows

#PROJECT – Build n8n , basic one
Frontend – interactive canvas -> React Flows
From database make different nodes and fetch it and put into side bar
Now each node has own unique id in db .

#note-
LangChain –
-- library (provides utility tools)
-- this is a first product , unplanned , community contribution
-- langchain is little chaotic . too many features etc.
LangGraph –
-- framework (runs the code) , its orchestrator.
-- this is the second product. Which was built my their own
-- most of the companies prefer langGraph.

#fun fact -
how to LangGraph is making money ? (langchain doesn't)
- suppose I have made a graph , then each node needs the GPU to run.
Right now its running on my machine , but in production we can upload to langGraph cloud.

LangGraph -

```
# NOTE- one problem is there if a enter "hello , my name is Anuj"
    # next time it will not remember the previous messages.. "I'm sorry, but I
don't know your name. How can I assist you today?"
    # because our state is transient state(lasts for short time)which gets reset
every time we invoke the graph.
    #once the invoke is done, the state is reset.
    # then what's the use of Annotation ??
    # if you have more nodes then , each node message will be appended the
message in the state.
    # Thats where the concept checkpointing comes in.
```

Checkpointing is the similar as we have checkpoints in games, when we pass a certain level ,it get saved .
Similarly , when we go from one node to another level wise , we save the state at that particular node into Database.

Now the advantage is that , even if it fails at a certain node, when we re-run ,it knows the last saved node, so it continues with the next node.
We can also , use it for de-bugging

Now suppose if a have multiple users , each user is chatting with AI .
so do we want all chats / histories to be interconnected to other users ? , No.
It should be different.

so we use thread_id – it's a unique id.

MONITORING –

LangSmith – you can monitor, observe in graph ,what's the workflow, , if any erros are there or not , it bacially shows the how the nodes get executed , but it's not-opensource.
You can use on-cloud. It's free.

There is open-source another version/alternative , which is langFuse.

LLM as a judge –

LLM as a judge" refers to using a **Large Language Model** (LLM), like GPT, Claude, or Gemini, to evaluate or "judge" the quality or correctness of certain outputs — usually in AI, research, or coding workflows

#NOTE – IDEA :
How LangSmith, LangGraph and node making system can improve the product . for e.g. E-commerce , LMS etc. ?

Like user automation testing on product .

For e.g.
We can track many types of events , in LMS.
When a Lerner submits the form , On payment failure , On payment success , On learner sign-up , On learner access to product expired , on test submission .

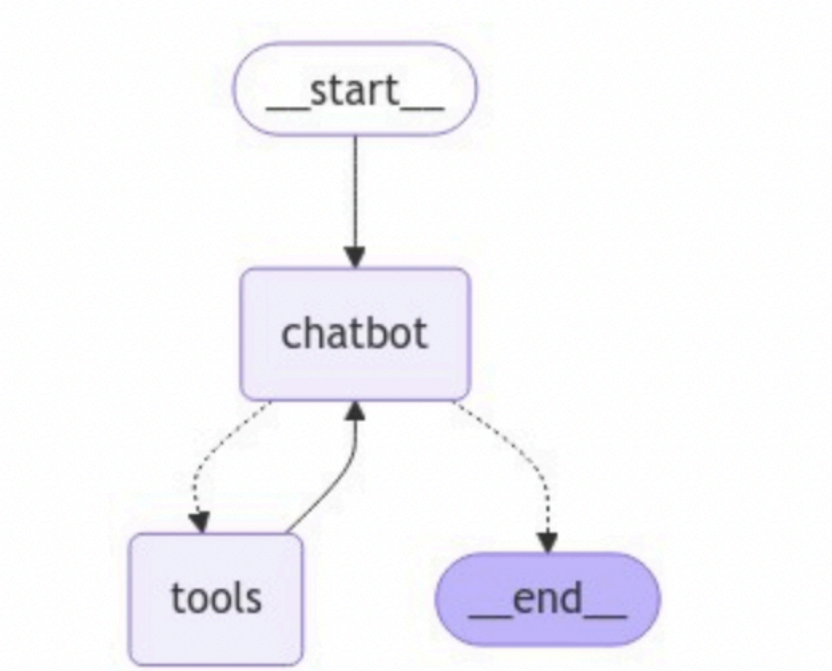
How to send this depends on which channel .

- email , push notification , WhatsApp , messages etc.

When a learner submit the test form, containing coding ,GK , Mathematics etc.
On the basis of that each learner , has to get a personalized feedback .
Then the role of AI comes in.
we can suggest the user to improvements he has to do.
or what are the things he/she lacks in.

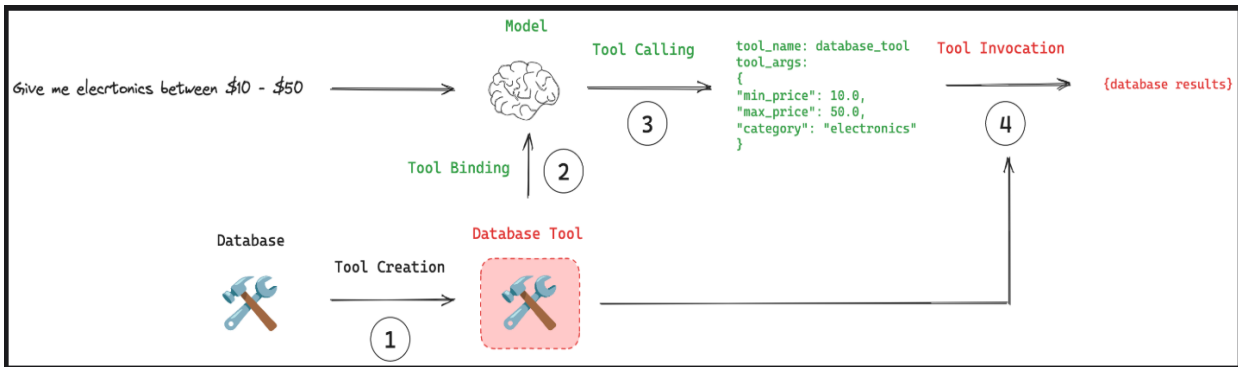
TOOL BINDING -

Earlier there is no any correct structured way to declare and implementation and calling of the Tools . there is no any standardisation .



Assignment –
Make a tool which can chat with our database.
Create , update, delete , insert, Search etc.
It can be anything PostgreSQL, MongoDB .
Properly, RAG based TODO application

now calling a DB is actually very risky .
- then what's the solution ?
- can we do LLM as a judge ?.
- how? . we can tell LLM that if I have delete command , then check if it has where clause or not .



HUMAN IN THE LOOP –

We must have seen that , if we had any query , on e-commerce , Zomato, razor pay etc. first chatbot tries to solve the issue. But if it need any more info. It requires the human to solve the query.
Similarly , with cursor .

Autonomous / Controlled Workflows –

Autonomous means , earlier we had made the tools , then do we have the control on the workflow ? . No right everything is handling by the AI only.
On the basic of examples steps.

Controlled means , we are deciding the workflow. Here.

hybrid means both autonomous and controlled workflow , whenever we make the ai application it works on hybrid model.

Memory Aware LLM

If you get memory right , we can get anything right.
The problem is our context window . context window is limited .

It's just like our human brain , do we know everything from the past?
No. right! . But we know small parts of our memories.

Suppose , if I type "Hey there! I am Anuj" . so do we remember this whole token??.
No. we will just extract the necessary information. Like "user = Anuj" ,
So our tokens will also be less. And our context window will get more quality context.

Types of Memory –
1.Short term – active conversation.
2.Long Term

Now short term memory is easy to implement.
But in case of long term memory , we will extract the facts (what to remember).

-Factual Memory.
-Episodic Memory.
-Semantic Memory.

Mem0 is library.
They has their internal system prompt for memory agent and connect with qdrantDB ,make vector embeddings etc. And based on our conversation , It will extract the facts etc.

Memories are compressed information.

Implementation –

So what we can do is first we will take our 10 running conversation messages(last 10 messages) , along with that on side-by-side while we are chatting each message is getting stored into the memory in compressed msg in form of facts. And more last 15-20 messages (including last 10) we will make an AI summary.
So
AI summary + 10 messages + Memory(whole memory) + user query
In this way, our context window limit will never be reached.

Now here is a Flaw.

It doesn't handles relationships.
When we make the vector embeddings , different facts about me.
But the relation among them is lost.
For e.g. information about my brother , mother , father ….but these are not related.

For that we use hybrid approach.

We use Neo4j , which is a graph database.
Here we can create the relations. We have nodes where we can connect those nodes.

Since we know graph is complex data structure , we don't know from where to start.

So we will store the complex relations into Neo4j and simple semantic relations into QdrantDB.

Neo4j works on cypher queries.

ADVANCE RAG PATTERNS –

In rag we Query and we make the vector embeddings , we do similarity search,

But there is a problem in this , what if user wants to say something else but we get little different answer.
So on user query , it completely depends, we we say similar words , vector embeddings are made closer.
but if we say in different approach , our V.E will make somewhere else.

So on user query , we should not depend. Cuz user query is vague/broad.
So we will do query translation.
- we will do translation in three levels.
1.Step Back Prompting (more abstract)
2.MultiQuery
3.SubQuery (less abstract)

If user say what is NodeJS? , we divide into three different queries(re-writig) (rephrasing). Like there maybe spelling mistakes , improper sentence.

Sub query – going into deep of “what is NodeJS?” , like what , how , why , how to implement , install etc. (sub questions)

Means , when we make different sub-questions , we can hit different V.E in different locations , so we get the broader perspective . and we get more relevant data.

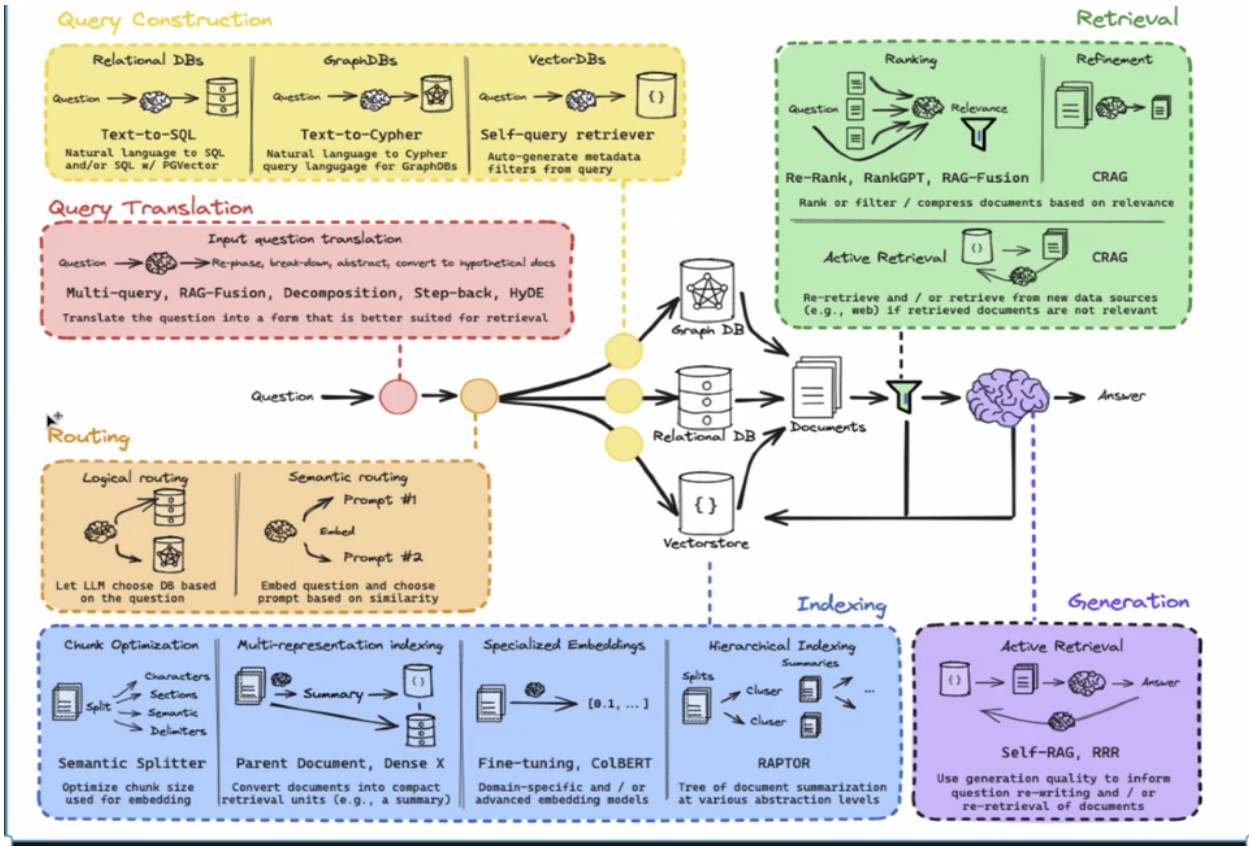
Use case:- is users don’t know the exact terms , it is difficult for users to get those exact terms while searching .

So create a node (Query Enhancer)
State{
  Query:
  Enhanced_queries:[]
}
Best thing here is Few Short Prompting.

LOL !
But having more information is also bad😂. It will create hallucinations.
What if I don't want that information.
"what is nodejs" -> output – Nodejs is …… libuv works behind the scene of nodejs.

So we will do ranking over here.
We will check the frequency. Where documents or words are coming in 3/4/5 different multi query. (common).

Query Translation -> V.E (VectorDB) -> multi query(multiple documents(multiple chunks)) -> Top Best chunks maybe TOP 3 -> then LLM call + user input -> one more LLM as a judge (approval)(it tells to make it better what should be there in user query for more accurate query) -> again do to VectorDB(max re-tries).-> accurate results.



Antrophic context-window-retrival -
https://www.anthropic.com/news/contextual-retrieval

TERMS –
Jinja templates
EJS

HBS

N8N
Pydantic

Javascript/typescript/python – Decorators
Curring in javacript

DRF – Danjo Rest Framework
Flask
FAST API – is best

Pycharm

UV - In Python, uv is a high-performance package manager and installer, often described as a faster, more efficient alternative to pip

Dspy – just like langchain , provides abstraction.