OpenAI Cohort

Page No.
Date

1) Intro to AI World → embedding.py, tokenization.py

Type: Input Query ⟹ Phase 1: Input & Encoding

- encoder
- decoder
- Loss
- Backpropagation
- Softmax
- Knowledge Cutoff
- Vocab Size
- Temperature

Tokenization
Vector Embeddings
Semantic Meaning
Positional Encoding
Self Attention

} part of "Transformer"

Refer article : medium.com/@anshmit86/decoding-ai-jargons-with-chai-95317669a528

2) Prompting → chat.py, chat-2.py, chat-3.py, chat-3.auto.py, chat-gemini.py

a) Zero-Shot prompting: The model is given a direct question/task without prior examples

b) Few-Shot prompting: The model is provided with a few examples before asking it to generate a response

c) Chain-of-Thought (CoT) prompting: The model is encouraged to break down reasoning step by step before arriving at an answer.

d) Self-Consistency Prompting: The model generates multiple responses & selects the most consistent or common answer.

e) Instruction Prompting: The model is explicitly instructed to follow a particular format or guideline.

f) Direct answer prompting: the model is asked to give a concise & direct response without explanation.

g) Persona-based prompting: the model is instructed to respond as if it were a particular character or professional.

h) Role-playing prompting: The model assumes a specific role & interacts accordingly.

) Contextual prompting: The prompt includes background information to improve response quality

) Multimodal prompting: The model is given a combination of text, images, or other modalities to generate a response.

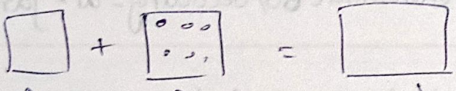
3) Agent → weather-agent.py, Pipeline.ipynb, ollama_api.py, sum.py, docker-compose.yml

App → Ollama → Models , tokenizer → transformer → detokenizer model

4) Fine Tuning: → finetune.ipynb

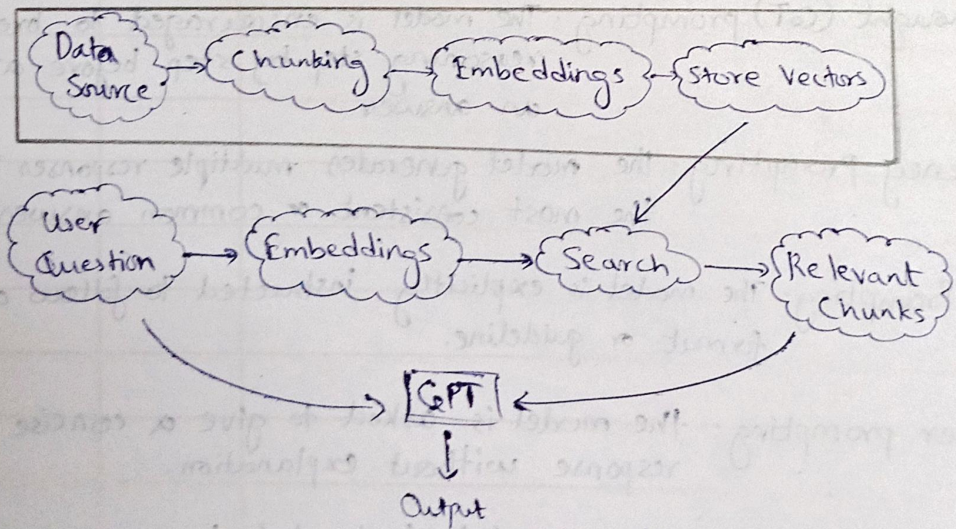
- Full Parameter Fine tuning → after loss func. → changes all params using back tracking
- LoRA

↳ loss func. → ex:



↑ original model ↑ where there are differences ↳ new fine tune model (no backtracking) but (more storage req.)

5) RAG : → rag_1.py , docker-compose.db.yml



Rag chain:

```
loader = pdf_loader(path)
splitter = text_splitter()
embedding = OpenAI()
qdrant = Qdrant_vector()
chain = loader | splitter | embedding | qdrant
chain.invoke(pdf_path)
```

6) Query Transformation & Translation

Basic RAG ⇒
- Indexing
- Retrieval
- Generation

Advance RAG ⇒
- Query Transformation
- Routing
- Query Construction
- Indexing
- Retrieval
- Generation

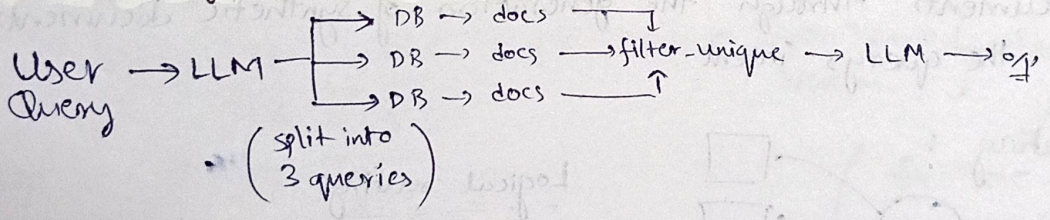
More abstraction

(Users)

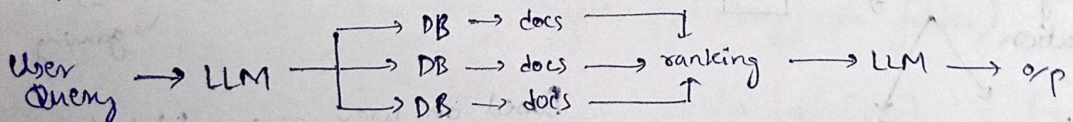
{Question} → Re-write (RAG Fusion, Multi Query)

Less abstraction

1) Parallel Query (Fan Out) Retrieval:



2) Reciprocate Rank Fusion:



3) Few Shot prompting:

{More Abstract} ← Question → {Less Abstract}

ex: Query → "When was the last time a team from Canada won the Stanley Cup as of 2002"

More Abstract → "Which years did a team from canada won the stanley cup as of 2002"

4) Query Decomposition:

Abstract → Less Abstract

(Step back prompting)

CoT

(problem ko break down karo)

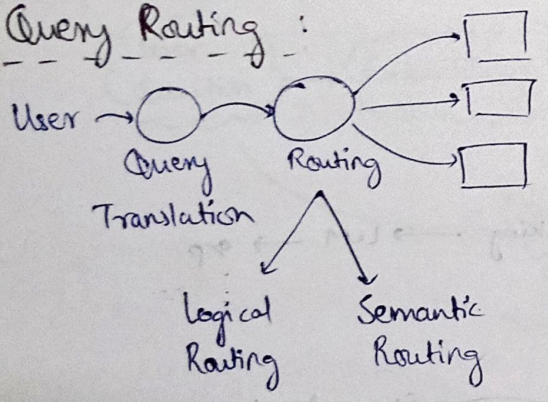
chain of thought

- For the given query, generate step by step plan how to answer this.
- Based on prev conv {question} ---

5) HyDE: Hypothetical Document Embeddings

- For large models
- HyDE enhances info. retrieval by closing the gap btw user queries & relevant documents through the generation of synthetic documents that encapsulate the query's intent.

7) Query Routing:



Logical routing:

ex: I have 3 data source → Python, JS, gaming.

Based on user query {} which is the best data source I should look on.

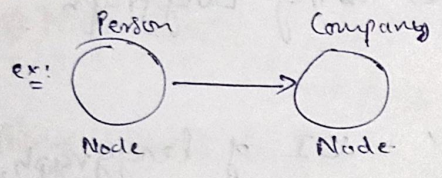
Note: Avoid mixing 3 data sources into 1 data source, so that we get good accuracy.

Semantic Routing: works on semantic meaning.

ex: We have 3 system prompts & user gave a prompt. Now based on user's prompt → choose best system prompt.

8) Knowledge Graph: → mem.py

- Vector Embeddings are good for semantic search, but they loose on relations.
- We use graph database to show relations → ex: Neo4j



(use ChatGPT to write) → uses Cypher Query Language

ex:

PDF → Chunking

for entity in ents:
MERGE(e: Char())
CREATE n-[rel]→(d)

- MemO → to store memories & retrieve.

9) Tracing AI Apps: → weather_agent.py, docker-compose.langfuse.yml

↳ Langsmith (closed source) → Open Source Version → Langfuse

(similar to OpenTelemetry) (for AI apps)

(for 'role-based' access, we need to pay)

10) LangGraph Orchestration: → graph.py (in langgraph folder)

Library vs Framework

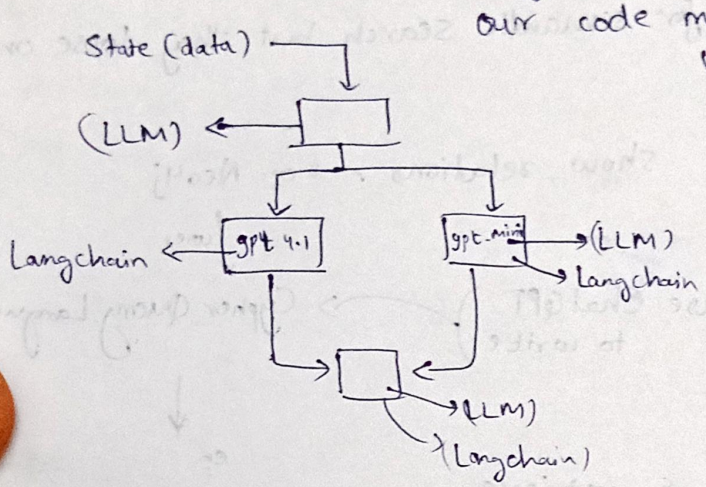
(we can use as per our wish) (we have to follow rules as per framework)

- LangGraph = Framework

- Tavily Search API → search engine built for AI agents
- Langgraph alternative → tensorlake

↳ What it does: → Previously we used to write all code in single file, but langgraph helps in making our code modular.

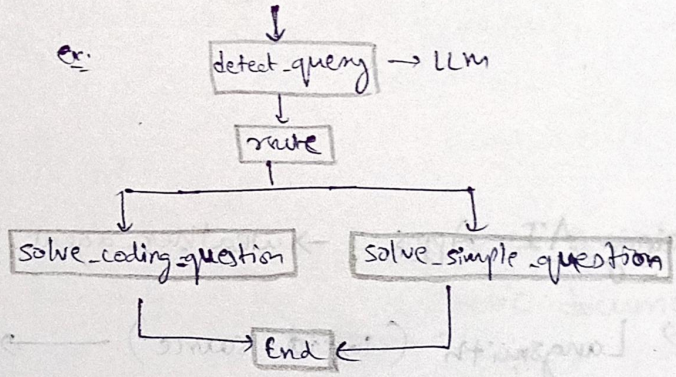
↳ Now we can write individual components & then we can just connect via nodes & edges using Langgraph.



- 'n8n' = UI of langgraph.

- Nodes = Code Block
- Edges = (flow)
  (define)

ex.



- "Flux" AI → Image Generation

11) LangGraph - Checkpointing → langgraph repository → graph.py, main.py

→ graph.stream = gives event at every node

→ graph.invoke = gives event after end of the graph (processing all the nodes)

Checkpointing: • each node, while running, saves the output it generates into the database.

- MongoDBSaver checkpointing - the previous 'state' from DB gets loaded each time when we are chatting. It is not like context or RAG, it is like continuing previous chat. It maintains the 'state'.

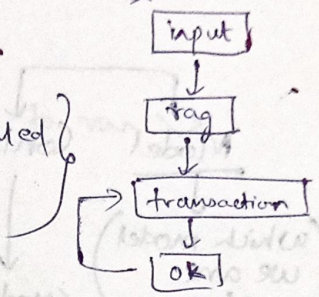
- Langgraph = State ka snapshot
  mem0 = makes a graphDB wrapper

Page No.
Date

Human in the loop: when we have already made a graph & now before transaction, we need to ask human

"Confirm the payment".

So, we are stopping in middle of graph, & this is called "human in the loop"



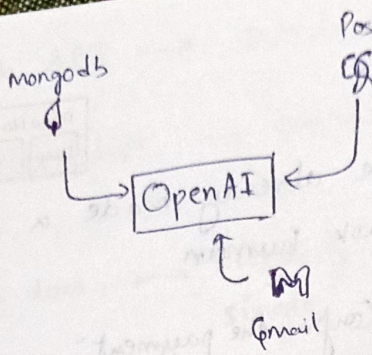
13) MCP Class {12th → Cursor proj.}

→ /mcp folder → & cross checking in another IDE

Things Learnt so far:

- LLM
- Transformer Architecture
- Tokenization
- Vectors & Vector embeddings
- Fine tuning
- OpenAI & Gemini
- Ollama
- Hugging face transformers (.ipynb files)
- Agents & Agentic Workflows
- Chain of Thought model from scratch
- Tool binding
- Mini Cursor Project
- External source connectivity
- Step-Plan-Execute-React Agent
- Langchain
- QdrantDB, Neo4j DB
- Knowledge Graph
- Mem0 memory DB
- Langgraph
- Human in the loop interruptions
- MongoDB checkpointing for LLM state management
- Query Transformations & Translations
- RAG & RAG pipeline
- Query translation, Query Routing, Reciprocal Ranking Doc for Rag optimisation.

- Voice to Voice Agents
- Voice based AI Agent
- MCP servers
- A2A protocol intro
- Security & Guardrails
- Langfuse self deployment & Langsmith



: If we want our model to communicate with mongodb, postgres, gmail → we have a common protocol aka MCP.

(tool)

So instead of defining from scratch, companies just gave an abstraction & this is basically MCP.

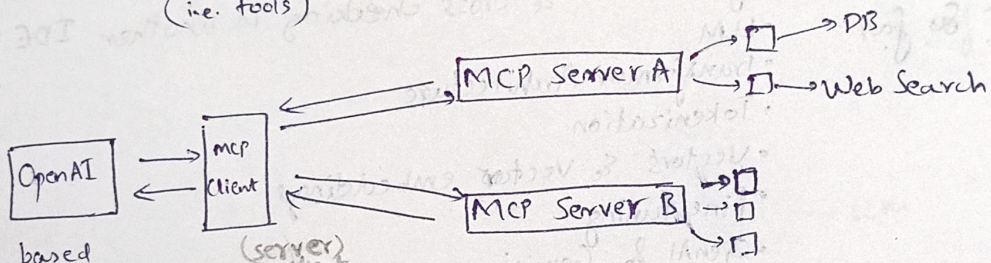
Model → (which model we are using)

Context → (Model gets context (i.e. tools))

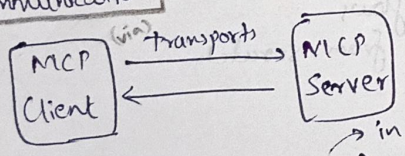
Protocol → (set of rules) → ex:

MCP server - Google
↳ send_email(from, to, subject)
↳ Prompt → Send email to xyz ...

Flow:



#Communication



this works only on local machine

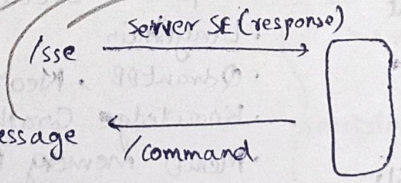
2 transports
↳ STDIO
↳ Server Sent Events (SSE) → Refer MCP docs

(Hosting MPC on our own server)

(standard input output - stdio) → in 'C' (stdio.h)

refers to the default communication channels a program uses for receiving data (input) & displaying results (output)

make 2 routes.



* send command to '/message' & you will get o/p from '/sse'.
to connect
↳ give server URL (refer docs)

ex: Taking i/p & o/p from terminal.

• Whenever MCP client starts this command "list_tools" & the server sends all the tools & their description. (So MCP automatically detected tools.)

Standardized Protocol

- Refer MCP docs of Cursor (if you are integrating with Cursor)
  - ex: (from class) → (JS proj.)

```
{
  "mcpServers": {
    "server-name": {
      "command": "node",
      "args": "full file path which we need to run \>"
    }
  }
}
```

- For TypeScript code (written in class)
  - refer: github.com/modelcontextprotocol/typescript-sdk → readme.md
    - (can get from docs as well)

14) Guardrails & Security:

Guardrail is like open source & we can see that it is not storing data anywhere.

- Input Guardrail → we can modify sensitive data before calling the API
- Output Guardrail → you can make sure that LLM doesn't hallucinate. (LLM as a Judge)