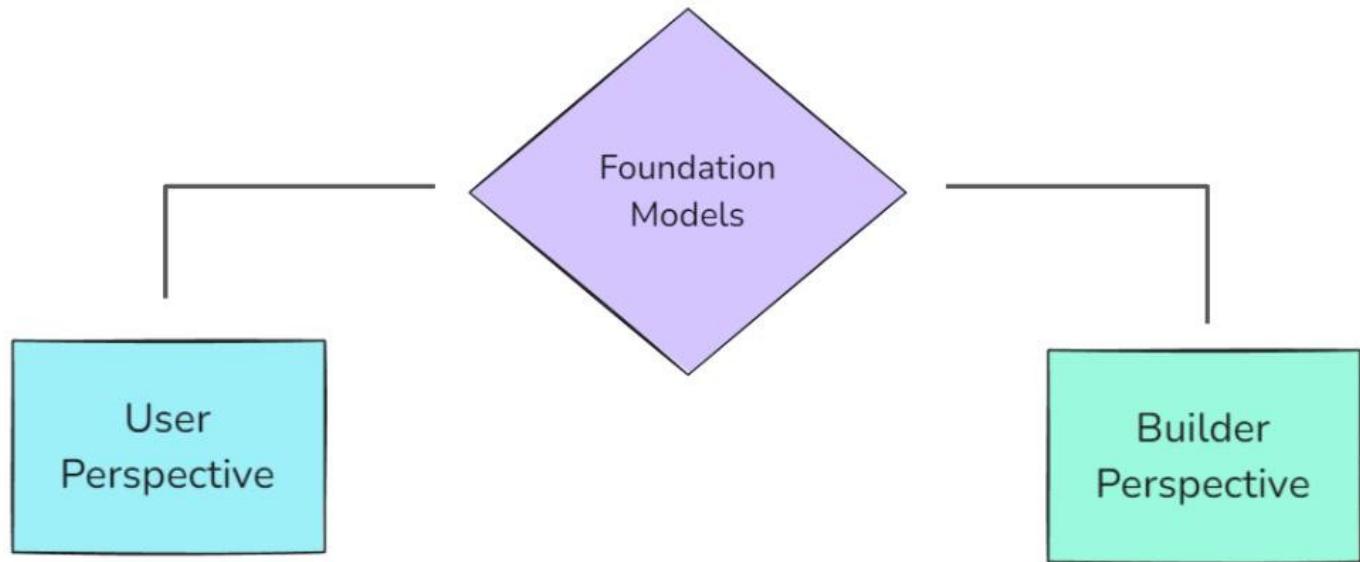
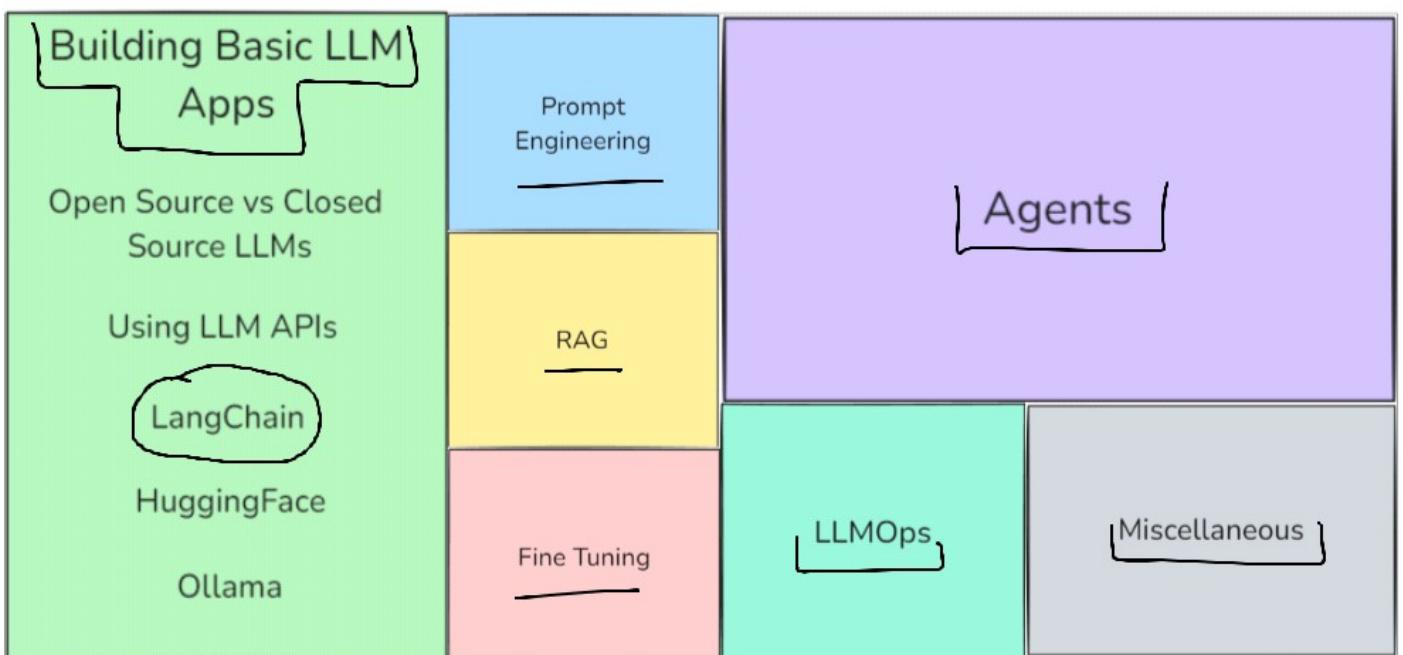


A little background!

29 January 08:28
2025



↓ use γ



0. Announcement Page 1

What is LangChain

29 08:
January 35
2025

LangChain is an open source framework that helps in building LLM based applications. It provides modular components and end-to-end tools that help developers build complex AI applications, such as chatbots, question-answering systems, retrieval-augmented generation (RAG), autonomous agents, and more.

- 1. Supports all the major LLMs
- 2. Simplifies developing LLM based applications
- 3. Integrations available for all major tools
- 4. Open source/Free/Actively developed
- 5. Supports all major GenAI use cases

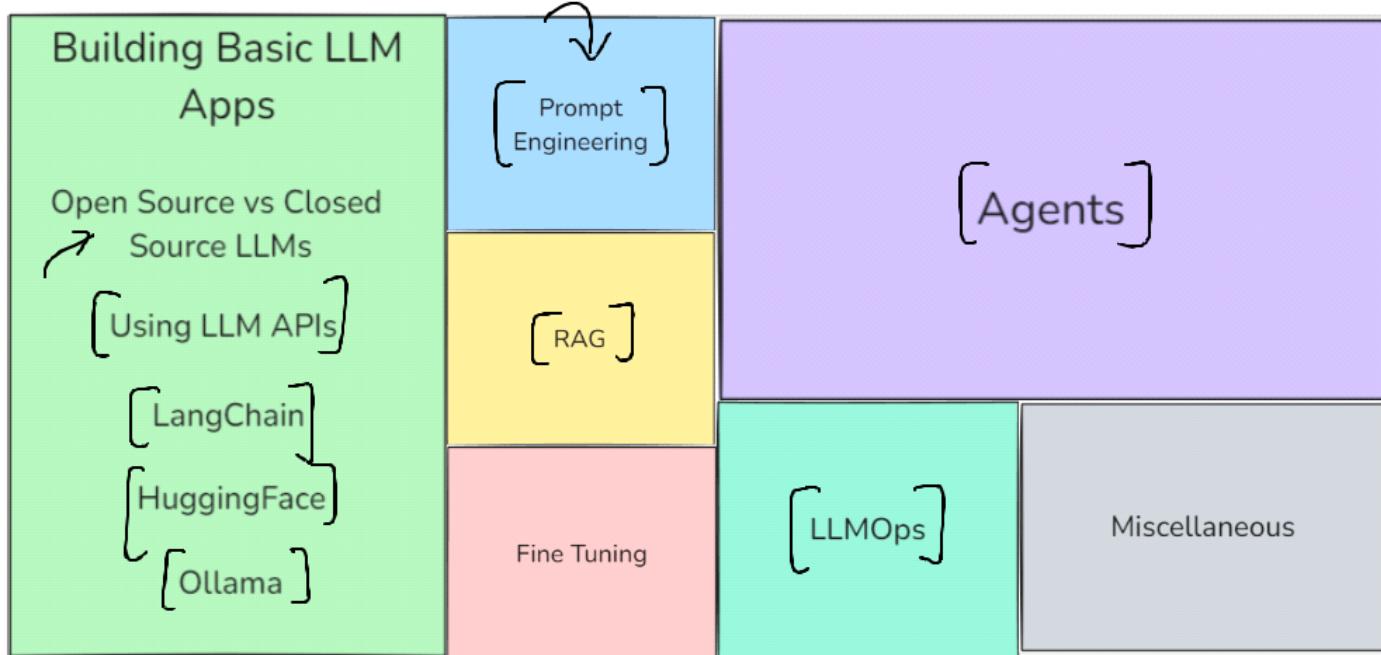
chains

```
arr[pivot], arr[j] = arr[j], arr[pivot]
return j
def quick_sort(arr, left, right):
    if left < right:
        split = partition
```

0. Announcement Page 2

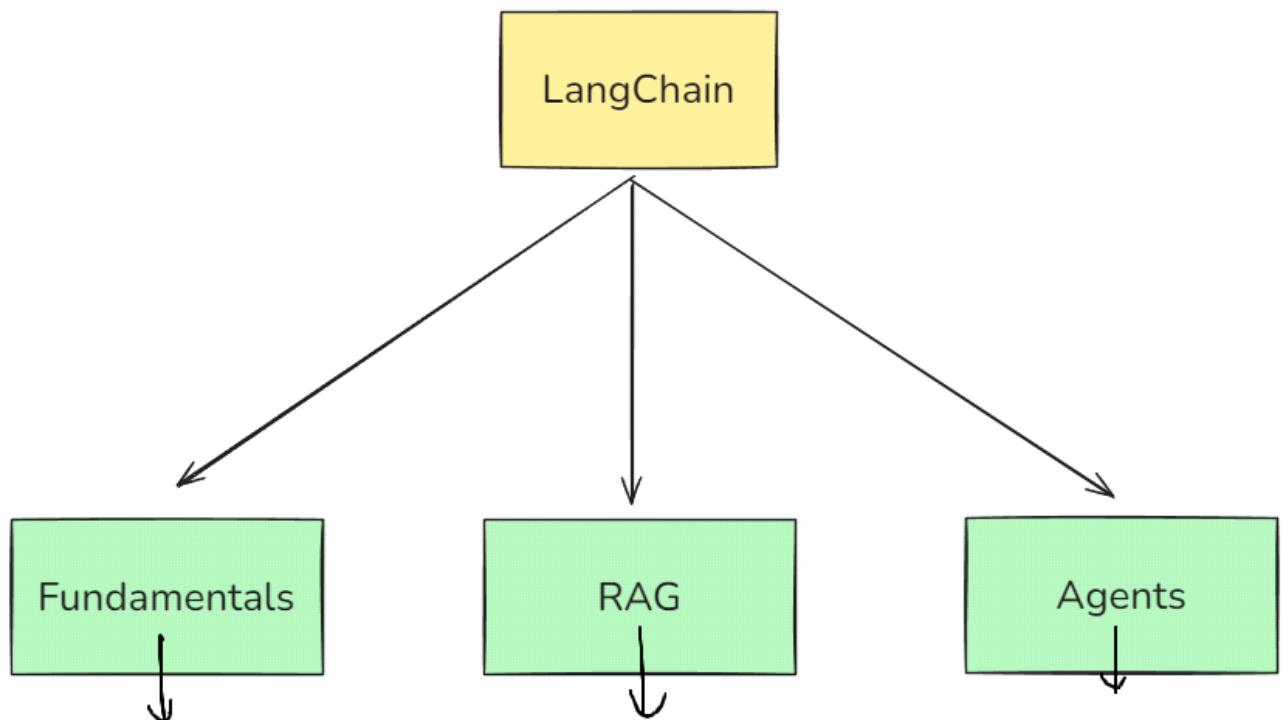
Why LangChain first

29 January 08:35
2025



Curriculum Structure

29 January 08:36
2025



0. Announcement Page 4

My Focus

29 08:
January 35
2025

| / 0.3 0

0.1 0.2

· 4



1. Updated information

2. Clarity ✓

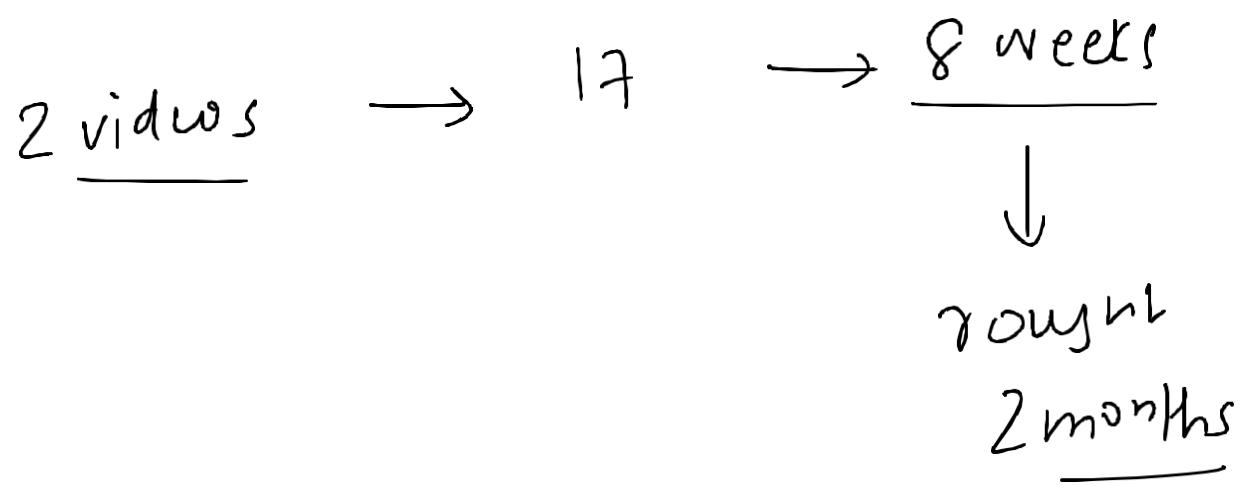
3. Conceptual understanding ✓

4. The 80 percent approach ✓

0. Announcement Page 5

Timeline

29 08:
January 37
2025



0. Announcement Page 6

What is LangChain

07 January
2025

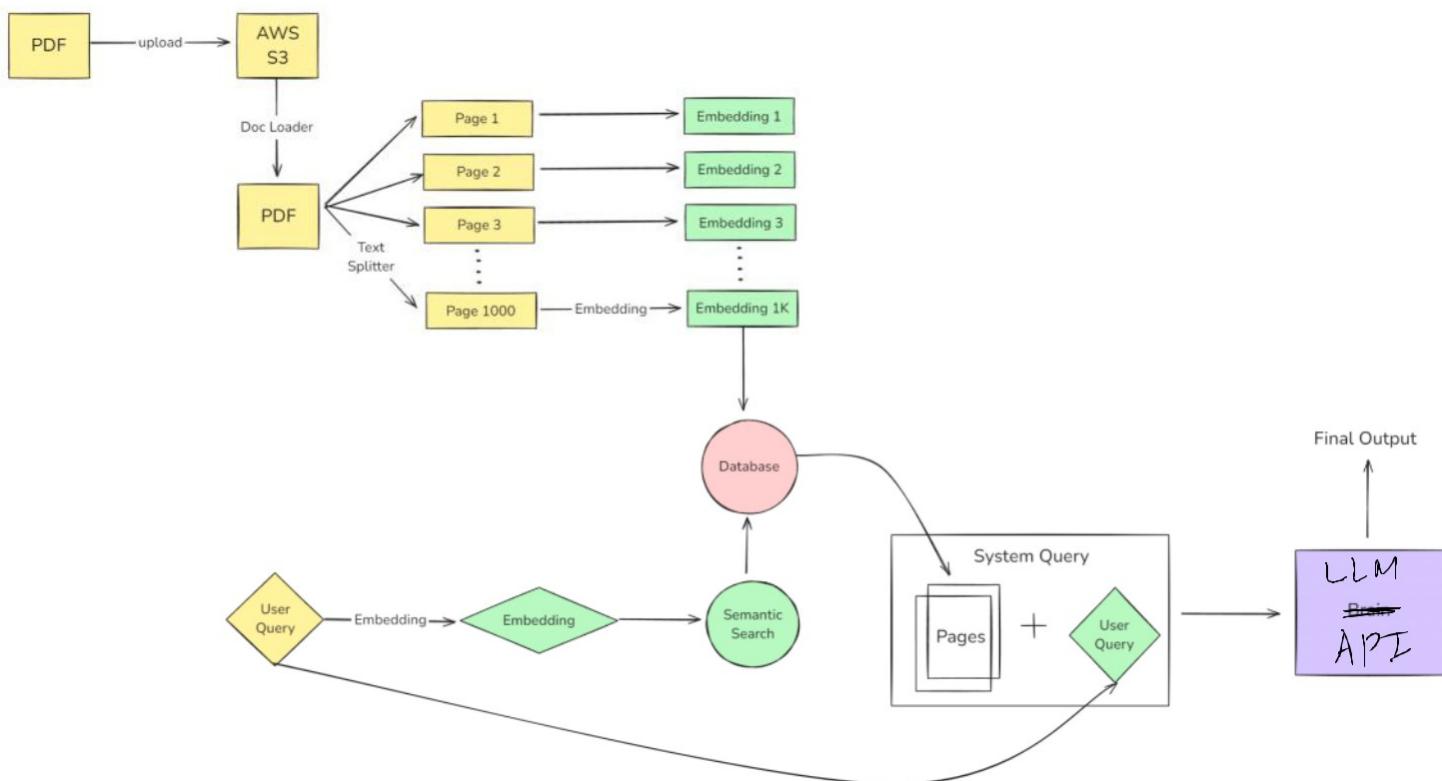
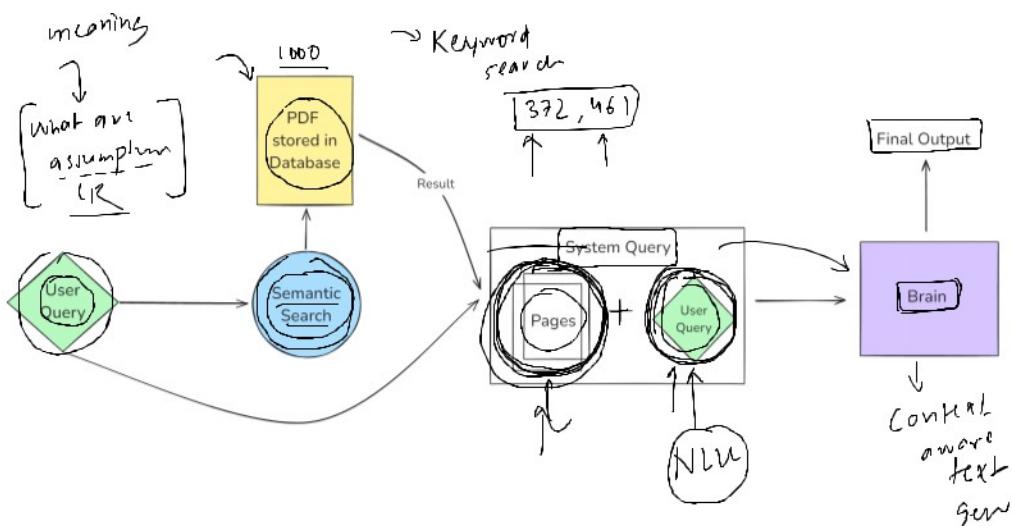
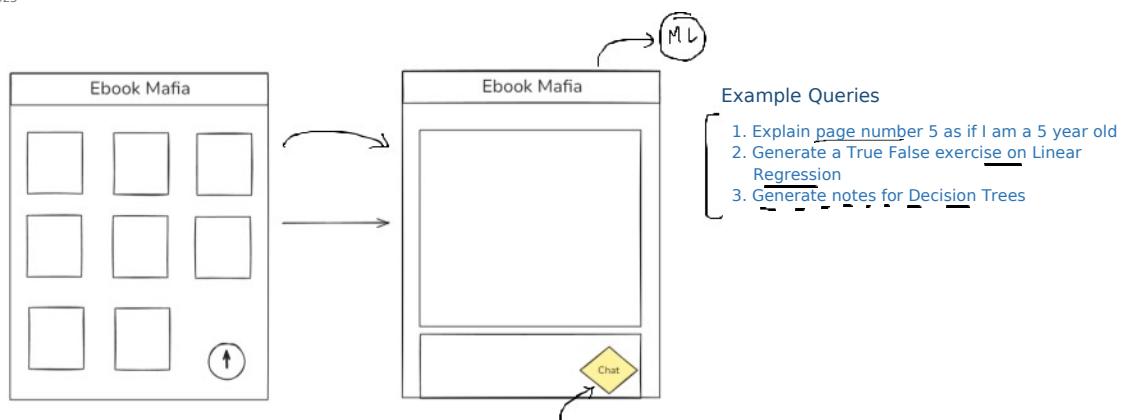
23:14

LangChain is an open-source framework for developing applications powered by large language models (LLMs).

Why do we need LangChain

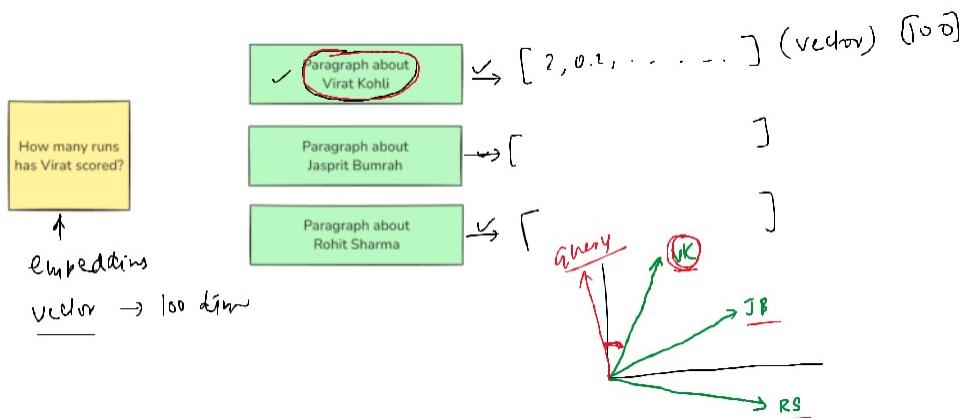
21 January 2025 23:34

2014



Semantic
Search

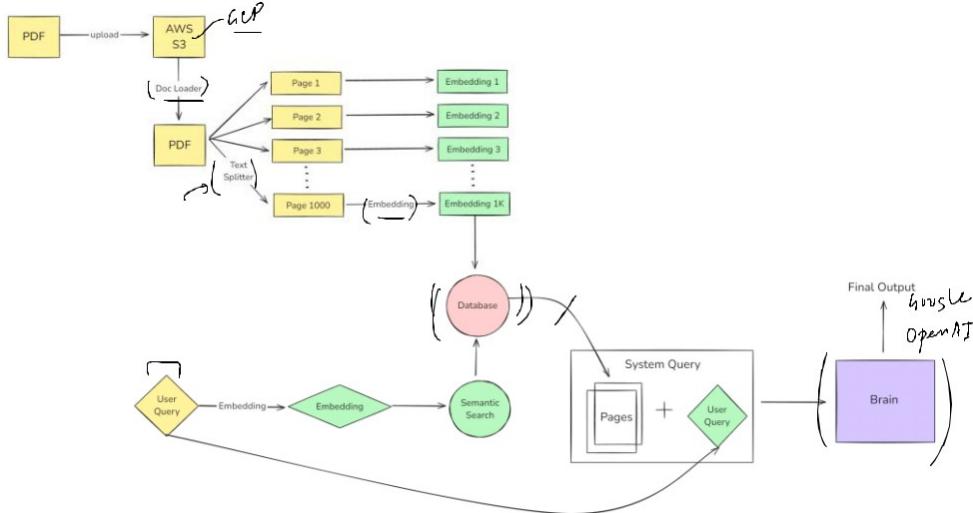
embedding \rightarrow vector (set-form)



Benefits

21 January 2025 23:34

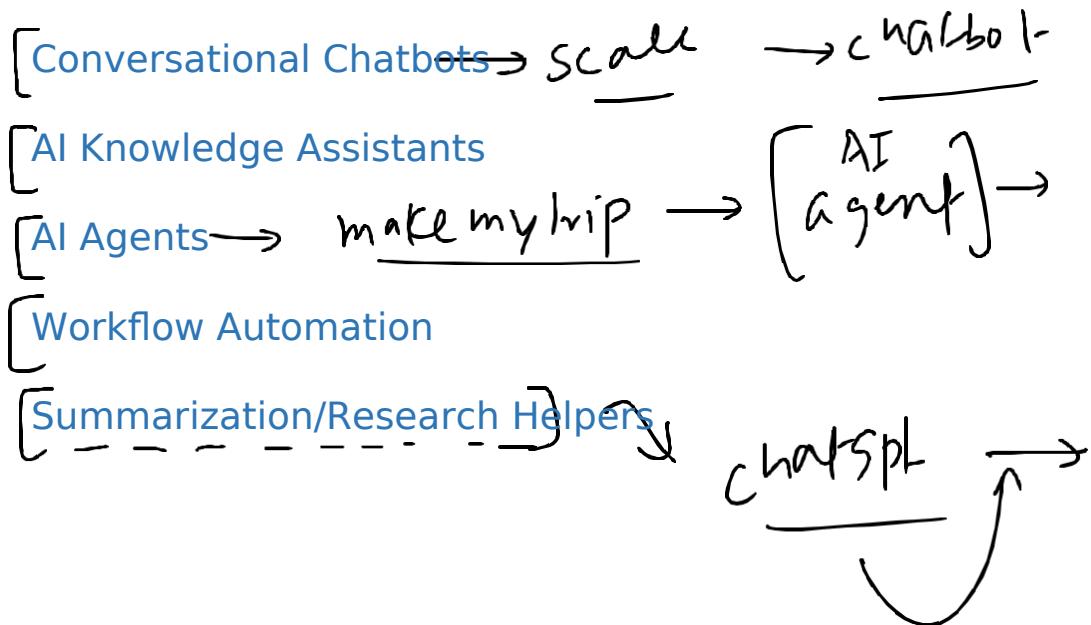
- Concept of chains
- Model Agnostic Development
- Complete ecosystem
- Memory and state handling



LLM
API

What can you build?

21 23:
January 34
2025



1. Introduction to LangChain Page 11

Alternatives

21 23:
January 34
2025

[LlamalIndex](#)
[Haystack](#)

1. Introduction to LangChain Page 12

Recap

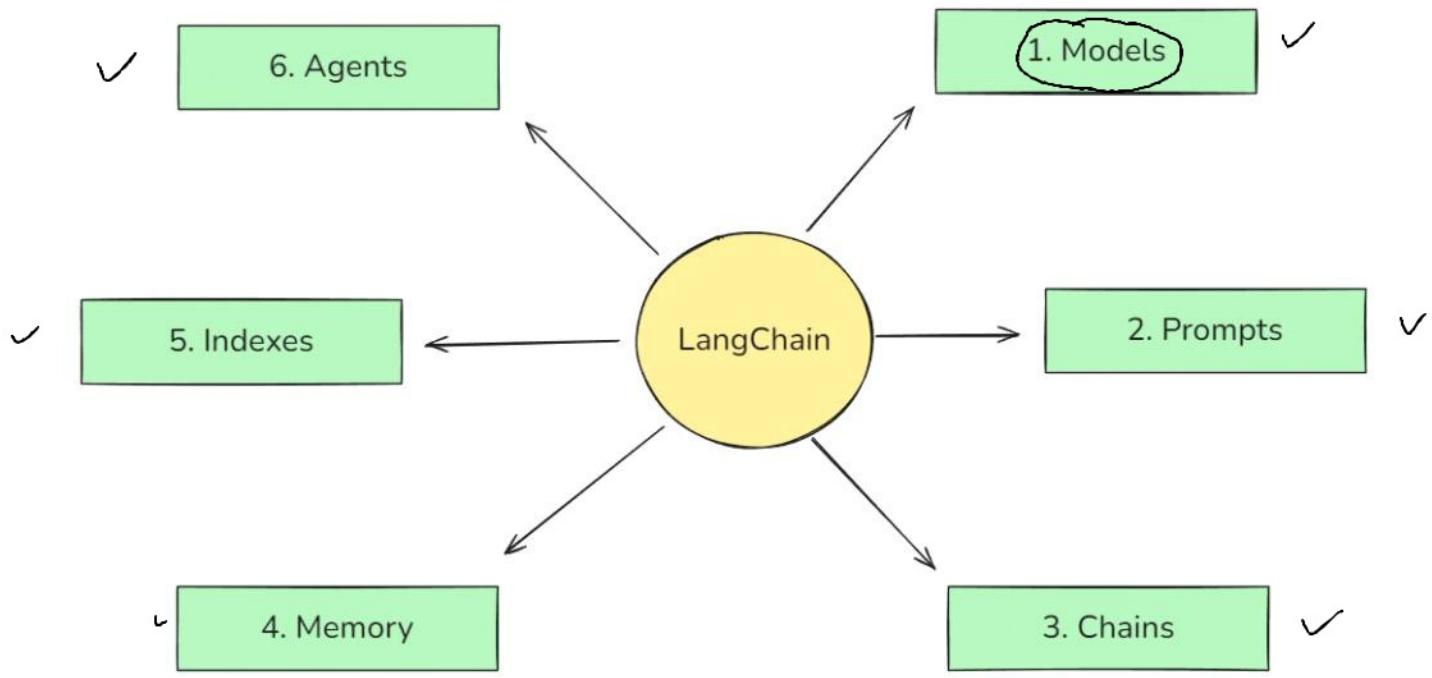
23 10:
January 29
2025

LangChain is an open-source framework for developing applications powered by large language models (LLMs).

2. LangChain Components Page 13

LangChain Components

23 January 2025 10:30



Models

23 January 2025 10:30

In LangChain, “models” are the core interfaces through which you interact with AI models.



```
openAI
```

from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

load_dotenv()

model = ChatOpenAI(model='gpt-4', temperat

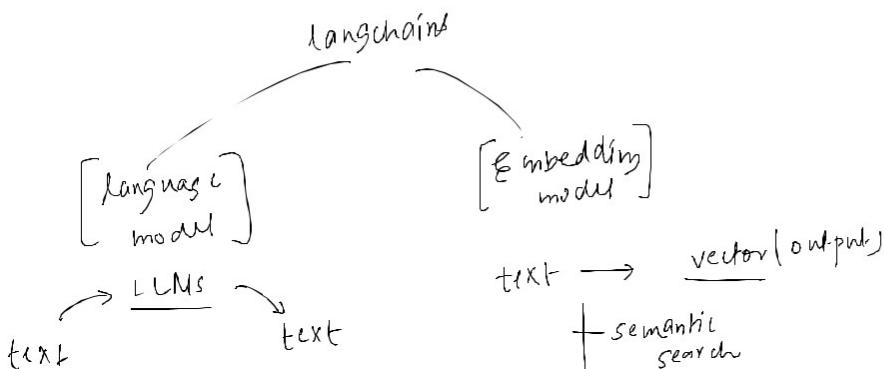
result = model.invoke("Now divide the resu

print(result.content)

```
from langchain_anthropic import ChatAnthropic
from dotenv import load_dotenv

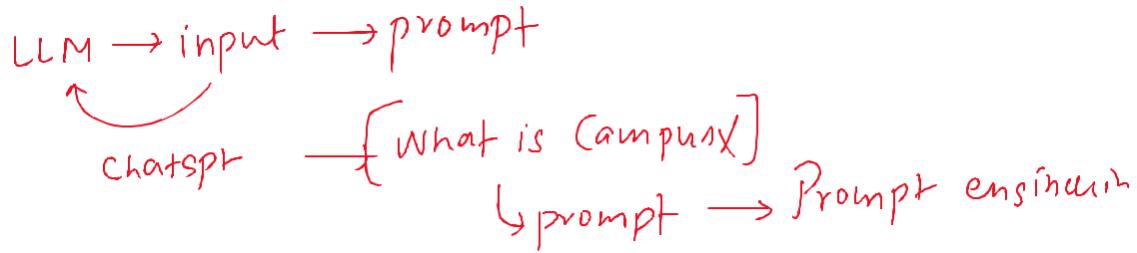
load_dotenv()

model = ChatAnthropic(model='claude-3-opus-20240229')
| result = model.invoke("Hi who are you")
|
print(result.content)
```



Prompts

23 January 2025 10:30



1. Dynamic & Reusable Prompts

```
from langchain_core.prompts import PromptTemplate

prompt = PromptTemplate.from_template('Summarize {topic} in {emotion} tone')

print(prompt.format(topic='Cricket', length='fun'))
```

2. Role-Based Prompts

```
# Define the ChatPromptTemplate using from_template
chat_prompt = ChatPromptTemplate.from_template([
    ("system", "Hi you are a experienced {profession}"),
    ("user", "Tell me about {topic}"),
])

# Format the prompt with the variable
formatted_messages = chat_prompt.format_messages(profession="Doctor", topic="Viral Fever")
```

3. Few Shot Prompting

```
examples = [
    {"input": "I was charged twice for my subscription this month.", "output": "Billing Issue"},
    {"input": "The app crashes every time I try to log in.", "output": "Technical Problem"},
    {"input": "Can you explain how to upgrade my plan?", "output": "General Inquiry"},
    {"input": "I need a refund for a payment I didn't authorize.", "output": "Billing Issue"},]

# Step 2: Create an example template
example_template = """
Ticket: {input}
Category: {output}
"""

# Step 3: Build the few-shot prompt template
```

```
few_shot_prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=PromptTemplate(input_variables=["input", "output"], template=example_template),
    prefix="Classify the following customer support tickets into one of the categories: 'Billing Issue', 'Technical Problem', or 'General Inquiry'.\n\n",
    suffix="\nTicket: {user_input}\nCategory:",
    input_variables=["user_input"],
)
```

Classify the following customer support tickets into one of the categories: 'Billing Issue', 'Technical Problem', or 'General Inquiry'.

Ticket: I was charged twice for my subscription this month.

Category: Billing Issue

Inquiry'.

Ticket: I was charged twice for my subscription this month.

Category: Billing Issue

] 1

Ticket: The app crashes every time I try to log in.

Category: Technical Problem

] 2

Ticket: Can you explain how to upgrade my plan?

Category: General Inquiry

] 3

←

Ticket: I need a refund for a payment I didn't authorize.

Category: Billing Issue

] 4

Ticket: I am unable to connect to the internet using your service.

Category:

Okay let's do this step by step, like I'm your teacher explaining in class.

We'll first set up **what the question is asking** and then go through each type of prompting in **LangChain**, with **examples** (like how you'd use them in real life).

Step 1: What is being asked?

The question is:

Explain the different types of prompting used in LangChain with examples, in a way that feels like a teacher teaching a student, using real-time situations.

So, our tasks are:

1. List the main types of prompting in LangChain.
 2. Explain each one in **simple terms**.
 3. Give **examples** (both technical + real-life analogy).
 4. Summarize at the end.
-

Step 2: Types of Prompting in LangChain

LangChain commonly uses these types of prompting:

1. **Zero-Shot Prompting**
2. **Few-Shot Prompting**

3. **Chain-of-Thought Prompting**
4. **Instruction Prompting**
5. **Contextual Prompting (with Memory & Retrieval)**
6. **Agent-based Prompting**

Let's go one by one.

1. Zero-Shot Prompting

- **Meaning:** You give the model a direct question/task without giving any examples.
- It must figure out the answer based only on its training knowledge.

LangChain Example:

```
from langchain.prompts import PromptTemplate
prompt = PromptTemplate.from_template("Translate the following English text
to French: {text}")
print(prompt.format(text="Good morning, how are you?"))
```

→ Output: "Bonjour, comment ça va ?"

Real-Life Teacher Example:

Imagine you're in class. A student who never solved a problem before is suddenly asked:

"Solve 12×8 ."

They have no examples, just the question. That's **zero-shot**.

Summary: Zero-shot = No examples, straight to the task.

2. Few-Shot Prompting

- **Meaning:** You give the model a few examples before asking it to solve the real problem.
- This helps guide its output style.

LangChain Example:

```
from langchain.prompts import FewShotPromptTemplate

examples = [
    {"word": "happy", "synonym": "joyful"}, 
    {"word": "sad", "synonym": "unhappy"}, 
]

example_prompt = PromptTemplate(
    input_variables=["word", "synonym"],
    template="Word: {word} -> Synonym: {synonym}"
)

few_shot = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix="Give synonyms of the following words:",
    suffix="Word: {input} -> Synonym:",
    input_variables=["input"],
)
print(few_shot.format(input="angry"))
```

→ Output: "Word: angry -> Synonym: mad"

Real-Life Teacher Example:

Like a teacher solving **2-3 problems on the board** before asking students to solve one on their own.

E.g., show:

- $12 \times 2 = 24$
- $12 \times 3 = 36$

Then ask the student: “Now solve 12×5 .”

Summary: Few-shot = Show some examples first, then ask the model.

3. Chain-of-Thought Prompting (CoT)

- **Meaning:** Asking the model to explain its reasoning step by step before giving the final answer.
- Helps in solving math, logic, coding, etc.

LangChain Example:

```
prompt = PromptTemplate.from_template(
    "Solve the math problem step by step: {question}"
)
```

```
print(prompt.format(question="If a book costs 120 rupees and I buy 3 books,  
how much do I pay?"))
```

→ Output:

- Step 1: One book costs 120.
- Step 2: Three books → $120 \times 3 = 360$.
- Final Answer: 360 rupees.

Real-Life Teacher Example:

When teaching math, teachers **show every step** instead of directly writing the answer.
That's **Chain-of-Thought prompting**.

Summary: CoT = Model explains reasoning step by step.

4. Instruction Prompting

- **Meaning:** You tell the model *how* to answer (tone, format, style).
- Helps control the response.

LangChain Example:

```
prompt = PromptTemplate.from_template(  
    "Explain Newton's first law in simple words, like teaching a 10-year-  
old."  
)  
print(prompt.format())
```

→ Output:

"Objects stay still unless pushed. A rolling ball keeps moving until something stops it."

Real-Life Teacher Example:

If I (teacher) say: *"Explain in simple words, no jargon, so even a kid can understand."*

That's **instruction prompting**.

Summary: Instruction = Control the style, tone, and format of answer.

5. Contextual Prompting (with Memory & Retrieval)

- **Meaning:** The model remembers past conversations or retrieves info from documents.
- Used for chatbots and Q&A systems.

LangChain Example:

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory()

memory.save_context({"user": "Hello"}, {"ai": "Hi, how can I help you?"})
print(memory.load_memory_variables({}))
```

→ Output: Conversation history stored.

Real-Life Teacher Example:

Imagine a student asks:

- Day 1: *“What is photosynthesis?”* Teacher explains.
- Day 2: Student asks: *“Can you explain it again in short?”*
Teacher remembers yesterday's topic. That's **contextual prompting**.

Summary: Contextual = Model remembers past or retrieves info.

6. Agent-based Prompting

- **Meaning:** The model acts as an “agent” that can use tools (calculator, search engine, Python code).
- It plans → decides → executes → returns result.

LangChain Example:

```
from langchain.agents import load_tools, initialize_agent
from langchain.llms import OpenAI

llm = OpenAI(temperature=0)
tools = load_tools(["serpapi", "llm-math"], llm=llm)
agent = initialize_agent(tools, llm, agent="zero-shot-react-description",
verbose=True)

agent.run("What is 15% of 240?")
```

→ Output: Agent uses math tool → 36.

Real-Life Teacher Example:

If a student asks: “*Teacher, what is the population of India in 2025?*”

Instead of remembering, teacher **checks Google first**, then answers.

That's **Agent Prompting** (model uses tools).

Summary: Agent = Model acts like a student who knows when to use calculator, books, or internet.

Step 3: Final Summary

Types of Prompting in LangChain

1. **Zero-Shot Prompting** → Direct task, no examples.
2. **Few-Shot Prompting** → Show some examples, then ask.
3. **Chain-of-Thought Prompting** → Step-by-step reasoning.
4. **Instruction Prompting** → Control style/tone/format.
5. **Contextual Prompting** → Uses memory or retrieves info.
6. **Agent-based Prompting** → Model acts like an agent, uses tools to answer.

Think of it like how a teacher teaches:

- Sometimes directly asks (zero-shot),
- Sometimes gives examples (few-shot),
- Sometimes shows steps (CoT),
- Sometimes explains in simple language (instruction),

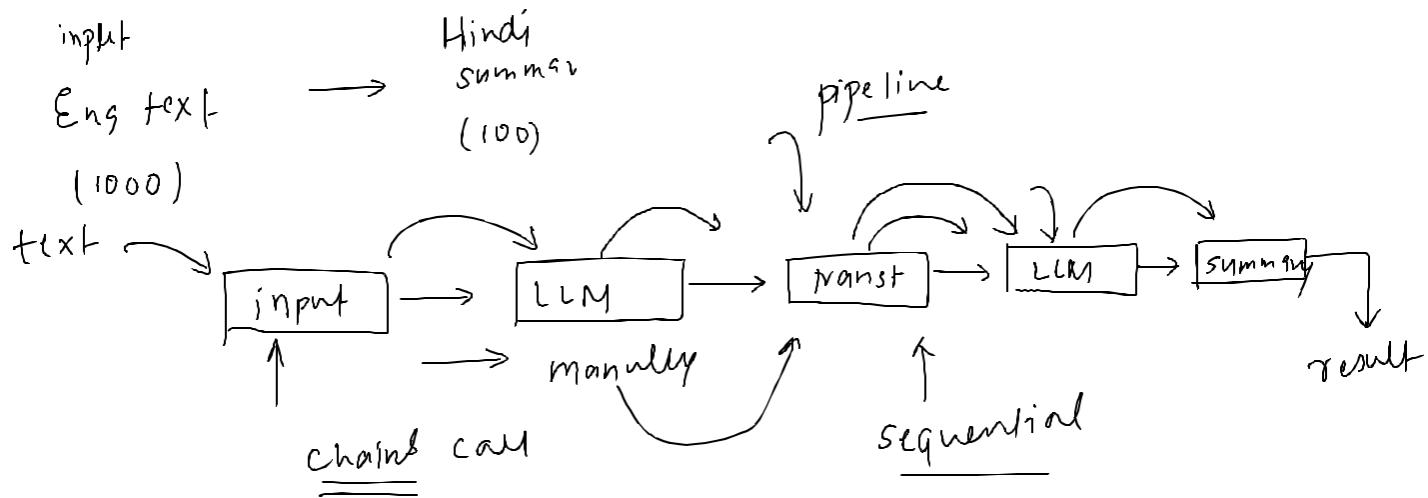
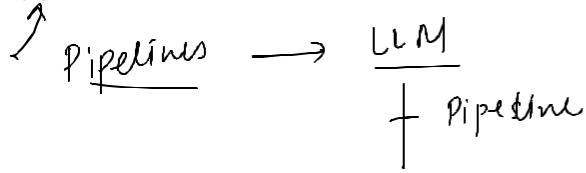
- Sometimes recalls past lessons (contextual),
 - Sometimes checks a reference book or calculator (agent).
-

Would you like me to also **draw a comparison table** (like Teacher Method vs Prompting Type in LangChain) so you can revise quickly?

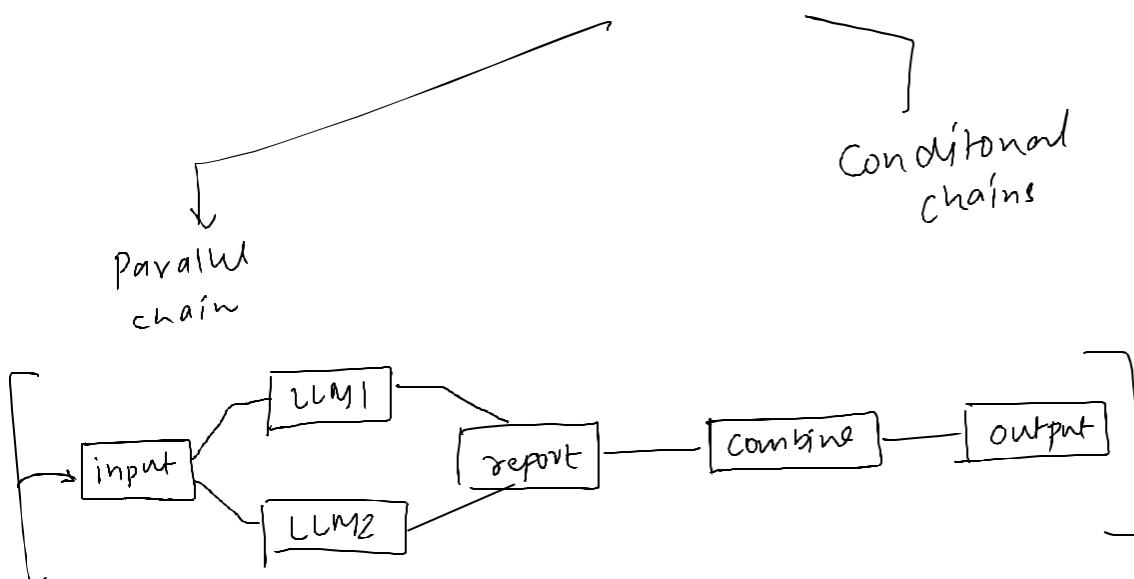


Chains

23 January 2025 10:31

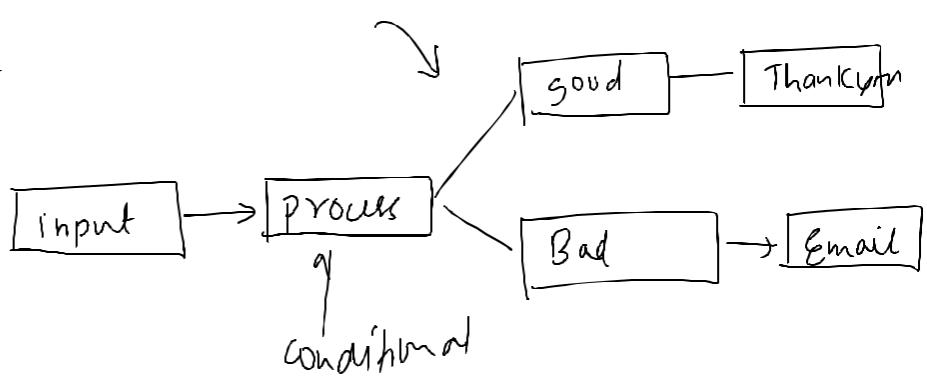


complex pipe



We can make the complex chains this is the ex

AI Agent
feedback



Alright, let's go step by step in a **teacher-style, notes-friendly format** so that you understand **Chains in LangChain** very clearly.

1. Introduction / Definition

Chains in LangChain are a way to **connect multiple components together** (like prompts, models, tools, and outputs) in a sequence.

Think of a chain as a pipeline where each step's output becomes the input to the next step.

2. Why it is important

- They allow you to build more complex workflows than a single model call.
 - Useful for tasks like summarization, question answering, reasoning, multi-step problem solving.
 - Make your code **modular and reusable** instead of writing everything from scratch.
-

3. How it works (intuitive explanation)

Imagine solving a math word problem:

- Step 1: Read the problem → extract numbers and conditions.
- Step 2: Perform calculations.
- Step 3: Summarize the final answer.

In LangChain, you can make each of these steps a **link in a chain**, and then connect them in order.

4. Types or Variants of Chains in LangChain

- a) **Simple Chain** - One input → One prompt → One output (like a single question-answer).
- b) **Sequential Chain** - Multiple steps, where each step feeds into the next.

- Example: Take user question → Search database → Generate answer.
 - c) **Parallel Chain** - Run multiple chains at once and combine results.
 - Example: Summarize a document in 3 styles (short, medium, long) at the same time.
 - d) **Router Chain** - Directs input to different chains depending on the question.
 - Example: If input is about math, send to math chain; if about history, send to history chain.
 - e) **LLMChain** - A very common chain where the step involves calling a language model with a prompt.
-

5. Code Examples (Python)

a) Simple LLM Chain:

```
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.llms import OpenAI

llm = OpenAI()
prompt = PromptTemplate(
    input_variables=["topic"],
    template="Explain {topic} in simple terms."
)

chain = LLMChain(llm=llm, prompt=prompt)
print(chain.run("quantum computing"))
```

b) Sequential Chain:

```
from langchain.chains import SimpleSequentialChain

chain1 = LLMChain(llm=llm, prompt=PromptTemplate(
    input_variables=["name"], template="Write a poem about {name}."
))
chain2 = LLMChain(llm=llm, prompt=PromptTemplate(
    input_variables=["poem"], template="Summarize this poem:\n{poem}"
))

overall_chain = SimpleSequentialChain(chains=[chain1, chain2])
print(overall_chain.run("LangChain"))
```

6. Real-life Analogies

- Think of chains like an **assembly line in a factory**. Raw material (input) enters, passes through machines (steps), and at the end you get the final product (output).
 - Another analogy: **Cooking recipe** - Step 1: Cut vegetables → Step 2: Cook → Step 3: Add spices → Step 4: Serve.
-

7. Applications / Use cases

- Question answering systems (where data retrieval + LLM reasoning is chained).
 - Multi-step summarization of long texts.
 - Document parsing → extracting facts → converting them into answers.
 - Chatbots that maintain context across steps.
 - Routing questions to specialized models.
-

8. Advantages and Limitations

Advantages:

- Modular and reusable.
- Easier debugging since each step is separate.
- Handles multi-step reasoning.

Limitations:

- Can become complex with too many chains.
 - Slower execution because multiple model calls are involved.
 - Requires careful design to avoid error propagation (a mistake in step 1 affects later steps).
-

9. Summary with Key Bullet Points

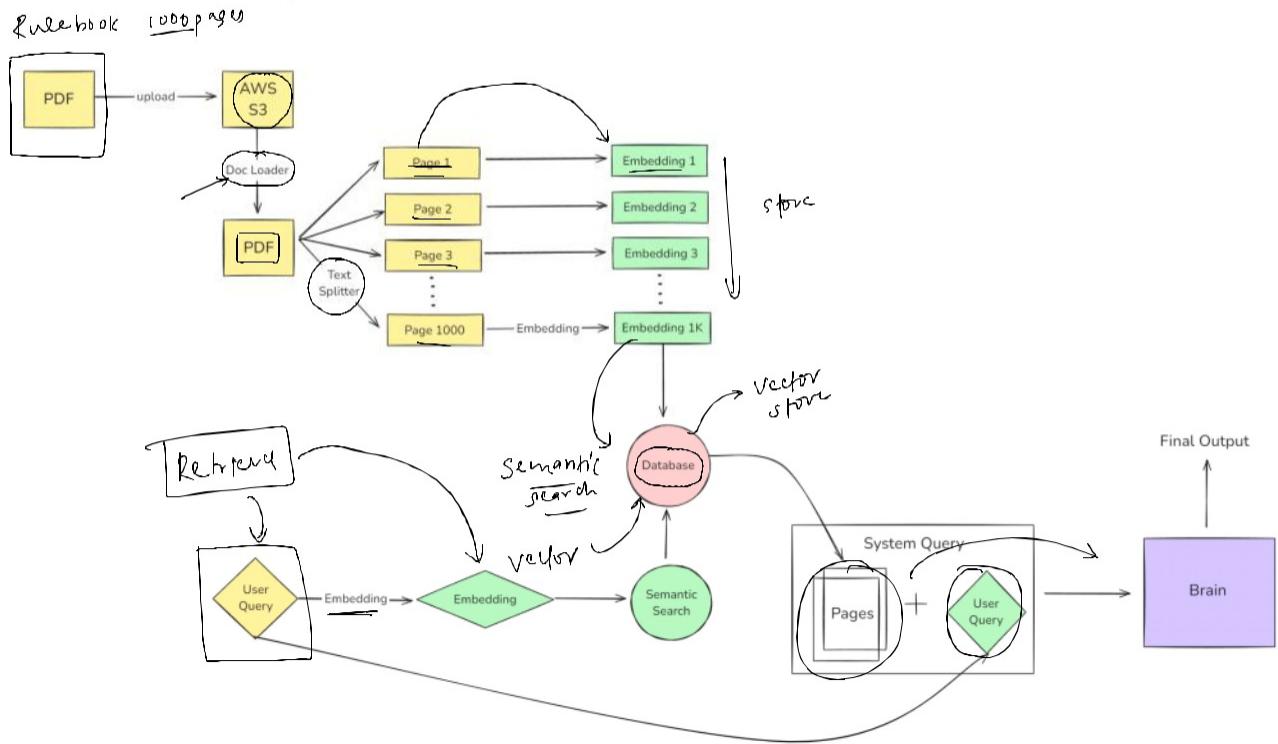
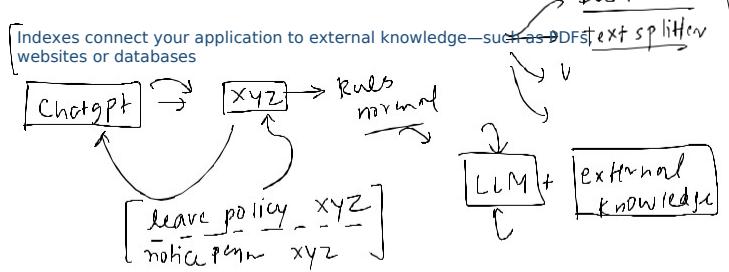
- Chains = sequence of connected steps in LangChain.
 - Purpose: Handle multi-step workflows beyond a single prompt.
 - Types: Simple, Sequential, Parallel, Router, LLMChain.
 - Example: LLMChain uses a prompt + model → output.
 - Analogy: Factory assembly line or cooking recipe.
 - Applications: Question answering, summarization, routing, chatbots.
 - Pros: Modular, reusable, multi-step reasoning.
 - Cons: Can be slow, complex, error-prone if not designed well.
-

Do you want me to also explain **Agents in LangChain** after this? Because usually, chains and agents are taught together (chains = fixed path, agents = flexible decision-making).



Indexes

23 January 2025 10:31



Got it. Let's carefully go over **Indexes in LangChain** in a clear classroom style with notes, examples, and even graphs for better understanding.

1. Introduction / Definition

Indexes in LangChain are data structures that help in **storing, organizing, and retrieving documents** efficiently so that they can be used by LLMs.

Since LLMs cannot handle very large text directly, indexes split the data into chunks and create a way to quickly search relevant parts when a user asks a question.

In short: Index = Smart way of connecting documents with an LLM.

2. Why it is important

- LLMs have **token limits** (cannot read full books or databases at once).
 - Indexes allow **efficient search** inside large text collections.
 - They make Question Answering (QA) systems possible on custom data.
 - Without indexes, retrieval would be very slow and inaccurate.
-

3. How it works (Intuitive Explanation)

Step 1: You load a set of documents (e.g., PDFs, text files, database records).

Step 2: The documents are split into smaller chunks (like paragraphs or sections).

Step 3: These chunks are converted into **embeddings** (vector representations).

Step 4: The embeddings are stored inside an index (like a library catalog).

Step 5: When a user asks a question, the system:

- Converts the query into an embedding.
- Searches the index for the closest matching chunks.
- Passes only those chunks to the LLM for generating an answer.

Graph (flow diagram):

User Query → Embedding → Search Index → Retrieve Top Relevant Chunks → Send to LLM
→ Final Answer

4. Types or Variants of Indexes in LangChain

a) **Vector Index (Vector Store Index)**

- Most common type.
- Uses embeddings + similarity search.

- Example: FAISS, Pinecone, Weaviate, Chroma.

b) **Keyword Index**

- Simple indexing using keyword matching (like inverted index in search engines).
- Faster but less semantic understanding.

c) **Hybrid Index**

- Combination of vector search + keyword search.
- More accurate because it balances semantic and keyword relevance.

d) **SQL/Knowledge Graph Index**

- Used when data is structured (databases or graph DBs).
- Query → translated into SQL/graph queries → relevant rows/nodes retrieved.

5. Code Example (Vector Index with Chroma)

```
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEMBEDDINGS
from langchain.document_loaders import TextLoader

# Step 1: Load documents
loader = TextLoader("notes.txt")
documents = loader.load()

# Step 2: Create embeddings
embeddings = OpenAIEMBEDDINGS()

# Step 3: Create index (vector store)
db = Chroma.from_documents(documents, embeddings)

# Step 4: Query the index
query = "What is LangChain?"
docs = db.similarity_search(query)

# Step 5: Print results
for d in docs:
    print(d.page_content)
```

6. Real-life Analogy

Think of a **library**:

- Books are split into **chapters** (document chunks).

- Each chapter is described in a **catalog card** (embedding).
 - When you search for a topic, the librarian looks into the **catalog index** to find the right chapters, instead of reading every book.
 - You then read only the relevant chapters, not the entire library.
-

7. Applications / Use cases

- Chatbots that answer from your company documents.
 - Question answering on research papers.
 - Legal or medical assistants that retrieve relevant case files.
 - Personal note search systems.
 - AI apps like “Chat with your PDF” or “AI-powered customer support.”
-

8. Advantages and Limitations

Advantages:

- Enables scalable use of LLMs on large datasets.
- Makes retrieval precise and fast.
- Flexible (can use vector, keyword, hybrid, or SQL).

Limitations:

- Requires embeddings (extra cost if using API-based embeddings).
 - Needs storage space for large datasets.
 - Accuracy depends on quality of embeddings and chunking strategy.
-

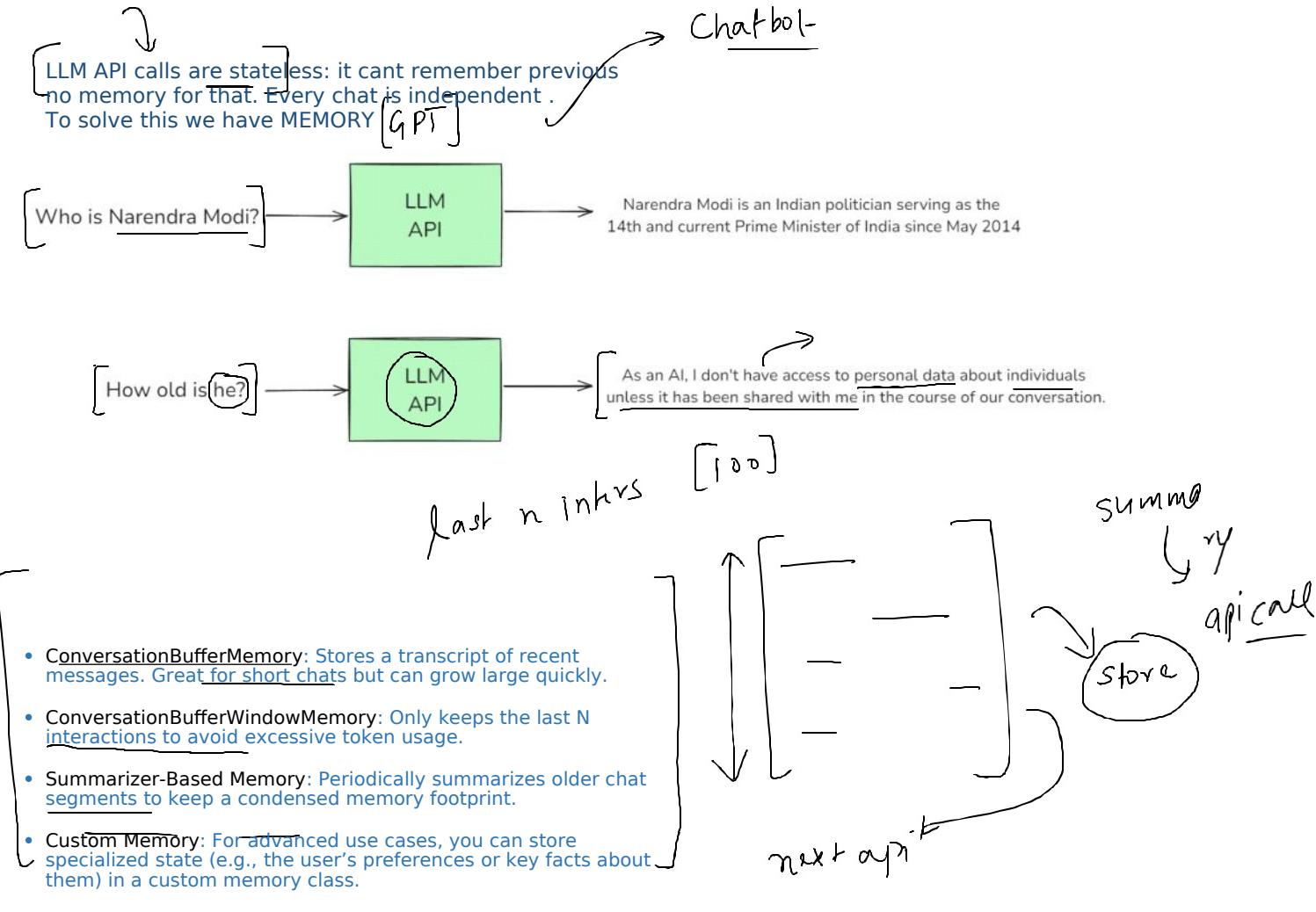
9. Summary (Key Points)

- Index = way to store and organize documents for efficient retrieval.
 - Main types: Vector Index, Keyword Index, Hybrid, SQL/Graph-based.
 - Works by chunking → embedding → storing → searching.
 - Real-life analogy: library catalog system.
 - Essential for QA systems, document chatbots, and large-scale information retrieval.
-

Do you want me to also draw a **graphical diagram (boxes and arrows)** of how an **Index works in LangChain** so it looks like a proper classroom flowchart?

Memory

23 January 2025 10:31



Perfect, let's now go through **Memory in LangChain** in the same teacher-style, step by step with full notes.

1. Introduction / Definition

Memory in LangChain refers to the component that allows the system to **remember past interactions in a conversation**.

By default, LLMs are stateless (they forget everything after each call). Memory solves this problem by keeping track of previous messages, context, or important variables, and feeding them back into the chain for continuity.

In simple words: **Memory = Chatbot's brain for remembering past conversations.**

2. Why it is important

- Without memory, every question would be treated as **new and unrelated**.
 - Memory makes interactions **contextual and human-like**.
 - Useful in chatbots, tutoring systems, customer support, and personal assistants.
 - Reduces redundancy (you don't have to repeat context every time).
-

3. How it works (Intuitive Explanation)

Step 1: User asks a question.

Step 2: LangChain stores this in memory (like writing in a notebook).

Step 3: Next time user asks something, LangChain fetches relevant past data from memory and appends it to the prompt.

Step 4: The LLM sees both old and new context → gives a coherent answer.

Graph (flow):

```
User Input → Add to Memory → Memory + Current Input → LLM → Output
          ↑                                ↓
          Memory gets updated after every turn
```

4. Types or Variants of Memory in LangChain

a) **ConversationBufferMemory**

- Stores all messages in memory as a buffer.
- Full chat history is sent each time.
- Simple but grows large with time.

b) **ConversationBufferWindowMemory**

- Stores only the last N messages.
- Prevents long prompts, saves cost.

c) **ConversationSummaryMemory**

- Instead of storing all text, it stores a **summary** of past conversations.
- Summarization keeps context compact.

d) **ConversationKGMemory**

- Stores information as a **Knowledge Graph**.
- Good for fact-based conversations (who, where, relations).

e) **VectorStoreRetrieverMemory**

- Stores conversations as embeddings in a vector database.
- Retrieves only relevant past pieces for current query.

5. Code Example (Simple Conversation Memory)

```
from langchain.llms import OpenAI
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

# Initialize LLM
llm = OpenAI(temperature=0)

# Add memory
memory = ConversationBufferMemory()

# Create conversation chain
conversation = ConversationChain(llm=llm, memory=memory, verbose=True)

# Example conversation
print(conversation.predict(input="Hi, my name is Hareesh."))
print(conversation.predict(input="What's my name?"))
```

Expected Output:

- First: "Hello Hareesh, nice to meet you!"
- Second: "Your name is Hareesh."

6. Real-life Analogy

- Think of memory like a **waiter in a restaurant**.
 - If the waiter remembers your past orders, they can recommend dishes you like.

- If they forget, you must repeat your preferences every time.
 - Similarly, memory lets a chatbot remember your past queries, so conversations feel natural.
-

7. Applications / Use cases

- Personal assistants (remembering user's preferences).
 - Customer support bots (remembering ticket details).
 - Educational tutors (keeping track of what student learned).
 - Therapy bots (long-term emotional support).
 - Games and storytelling bots (keeping story continuity).
-

8. Advantages and Limitations

Advantages:

- Makes chatbots more **human-like**.
- Saves user effort (no need to repeat info).
- Flexible options (buffer, summary, vector-based).

Limitations:

- Storing full history increases **token cost**.
 - Summarization may **lose details**.
 - Vector-based memory requires **extra setup (database)**.
 - Memory persistence across sessions needs special handling.
-

9. Summary (Key Points)

- Memory = mechanism for remembering past interactions in LangChain.
 - Types: Buffer, Window, Summary, KG, VectorStore.
 - Works by storing previous inputs/outputs and adding them to new prompts.
 - Real-life analogy: Waiter remembering your past orders.
 - Crucial for chatbots, personal assistants, tutors, and support systems.
-

Would you like me to also **draw a comparative chart (table)** of all memory types in LangChain, showing when to use which one? That way you'll have a ready reference like class notes.

Perfect. Let's make a **comparative chart of all memory types in LangChain**. This will serve like your classroom reference table.

Comparative Table of Memory Types in LangChain

Memory Type	What it Stores	When to Use	Advantages	Limitations
ConversationBufferMemory	Full chat history (all messages)	Small conversations, debugging, when full context is needed	Simple to use, complete context	Grows large, expensive in long chats
ConversationBufferWindowMemory	Only last N messages	Long chats where	Saves cost, avoids	Loses older context

Memory Type	What it Stores	When to Use	Advantages	Limitations
ConversationSummaryMemory	Only recent history matters	Long conversations where gist is enough	Huge prompts	May lose fine details
ConversationKGMemory	Facts in a knowledge graph (who, what, where)	Conversations involving relations (friends, fact-based Q&A)	Structured memory, better fact tracking	More complex setup
VectorStoreRetrieverMemory	Embeddings of conversations in vector DB	Very long histories, semantic recall (similar past topics)	Retrieves only relevant info, scalable	Needs vector DB, more setup effort

Graphical Comparison (Flow Concept):

ConversationBufferMemory → stores everything

ConversationBufferWindowMemory → stores last N messages

ConversationSummaryMemory → stores compressed summary

ConversationKGMemory → stores relationships in graph form

VectorStoreRetrieverMemory → stores embeddings in a searchable database

Quick Selection Guide:

- Use **BufferMemory** → if conversations are short.

- Use **WindowMemory** → if you only care about recent exchanges.
 - Use **SummaryMemory** → if conversations are long but gist is enough.
 - Use **KGMemory** → if tracking facts and relations is important.
 - Use **VectorMemory** → if you need semantic search in long conversations.
-

Do you want me to also give you a **Python example showing 2-3 types of memory side by side**, so you see how they behave differently in the same conversation?

Alright Let's make a **Python example comparing 3 common memory types** in LangChain:

- ConversationBufferMemory
- ConversationBufferWindowMemory
- ConversationSummaryMemory

This way you'll clearly see how each behaves differently in the same conversation.

Python Code Example:

```
from langchain.llms import OpenAI
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory,
ConversationBufferWindowMemory, ConversationSummaryMemory

# Initialize LLM
llm = OpenAI(temperature=0)

# 1. Buffer Memory (stores everything)
buffer_memory = ConversationBufferMemory()
buffer_conversation = ConversationChain(llm=llm, memory=buffer_memory,
verbose=True)

# 2. Window Memory (only last 2 messages)
window_memory = ConversationBufferWindowMemory(k=2)
window_conversation = ConversationChain(llm=llm, memory=window_memory,
verbose=True)

# 3. Summary Memory (keeps a running summary)
summary_memory = ConversationSummaryMemory(llm=llm)
```

```

summary_conversation = ConversationChain(llm=llm, memory=summary_memory,
verbose=True)

# --- Simulated Conversation ---

print("==== Buffer Memory Example ===")
print(buffer_conversation.predict(input="Hi, my name is Hareesh."))
print(buffer_conversation.predict(input="I am studying computer science."))
print(buffer_conversation.predict(input="What is my name?")) # Should remember
Hareesh

print("\n==== Window Memory Example ===")
print(window_conversation.predict(input="Hi, my name is Hareesh."))
print(window_conversation.predict(input="I am studying computer science."))
print(window_conversation.predict(input="What is my name?")) # Might forget
because only last 2 messages are stored

print("\n==== Summary Memory Example ===")
print(summary_conversation.predict(input="Hi, my name is Hareesh."))
print(summary_conversation.predict(input="I am studying computer science."))
print(summary_conversation.predict(input="What is my name?")) # Should recall
from summarized context

```

Expected Behavior (simplified):

1. **Buffer Memory**

- Remembers everything.
- Output → "Your name is Hareesh."

2. **Window Memory (last 2 messages only)**

- Might forget the name if it's not in the last 2 exchanges.
- Output → Possibly "I don't know your name."

3. **Summary Memory**

- Remembers by storing a summary like: "*User introduced as Hareesh, studying computer science.*"
- Output → "Your name is Hareesh."

Real-life Analogy:

- **Buffer** = Keeping every page of a diary.
- **Window** = Keeping only the last 2 pages.

- **Summary** = Keeping just a short summary on one page.
-

Do you also want me to extend this with a **VectorStoreRetrieverMemory example** (using embeddings + similarity search) so you can see how long-term recall works across conversations?

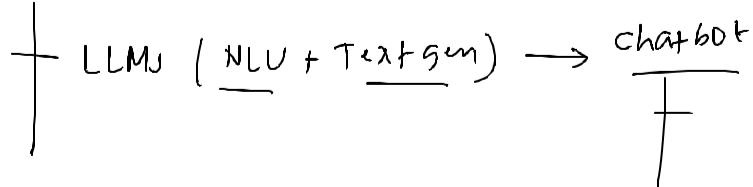
+ cont

Agents

23 January 2025 10:31

A as

AI Agents



chatbot

AI agent

↓
chatbot with super

travel website
+ Shimla Manali

24th Jan

API →
Ingrid
Book the flight

AI agent -

+ [Reasoning capabilities]
+ Tools

Tools → calculator → Weather API

User → AI agent.

→ can you tell
today's temp of Delhi
with 3

+ Chain of Thought

→ [Delhi temp]

→ 25°C

→ 25, 3, *

75 → 75

Recap

10 February 2025 09:21



+ Models

Models

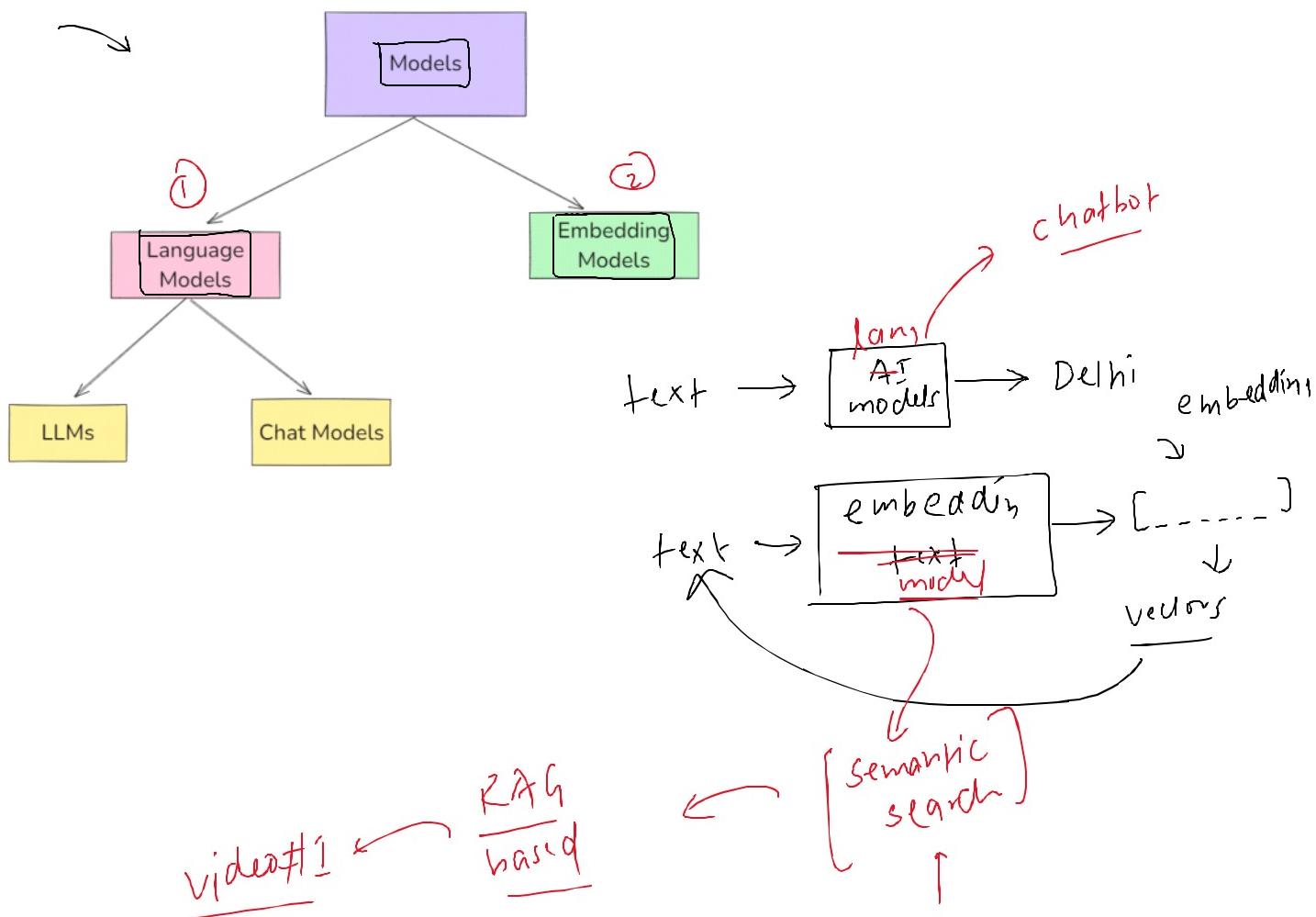
07 January 2025 23:15

AI
models

What are models :

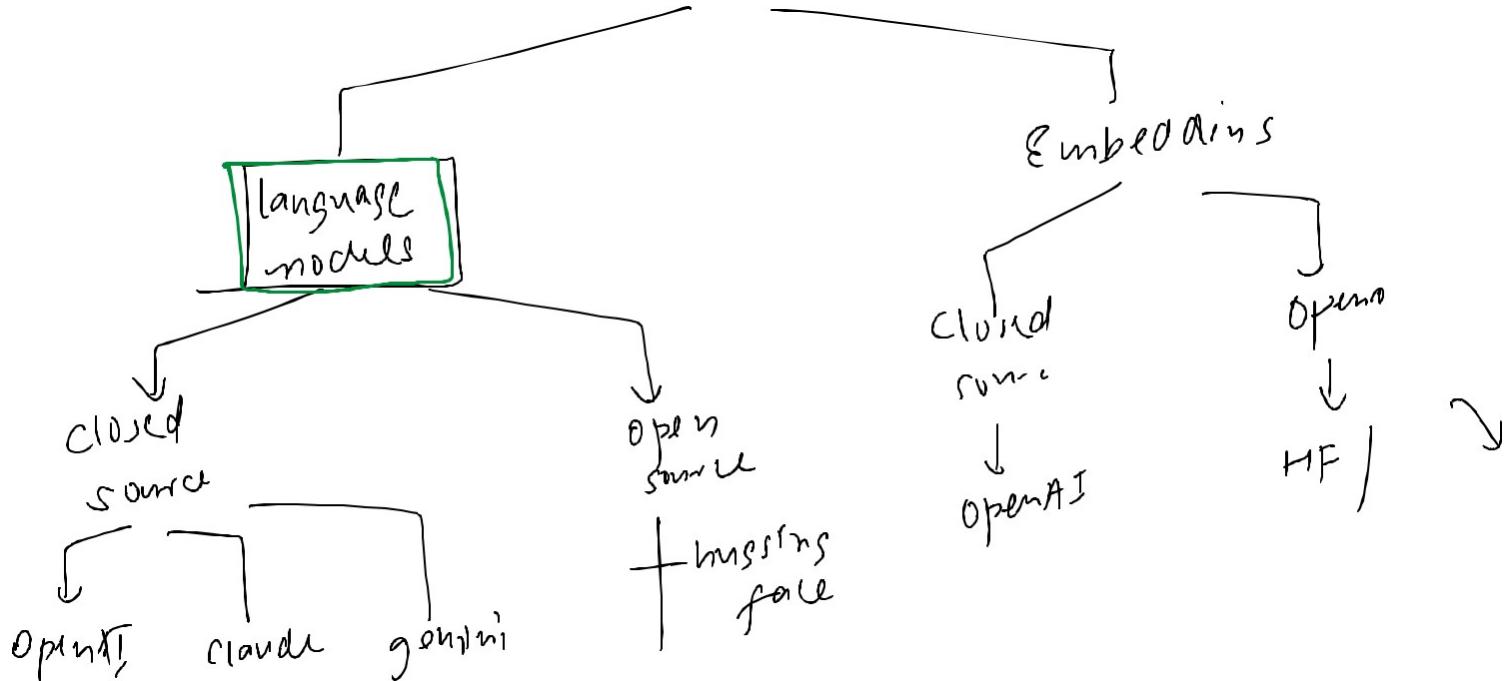
The Model Component in LangChain is a crucial part of the framework, designed to facilitate interactions with various language models and embedding models.

It abstracts the complexity of working directly with different LLMs, chat models, and embedding models, providing a uniform interface to communicate with them. This makes it easier to build applications that rely on AI-generated text, text embeddings for similarity search, and retrieval-augmented generation (RAG).



Plan of Action

10 February 09:21
2025



Got it — you're asking about **Models in LangChain**. Let me explain this in a teacher-style, notes-friendly format so you can directly add it to your notes.

Introduction / Definition

In LangChain, **Models** are the core components that generate or process text. They are essentially the bridge between LangChain and different language model providers (like OpenAI, Anthropic, Cohere, Hugging Face, etc.). Instead of you writing separate code for each provider, LangChain gives a unified interface to work with all of them.

Why it is Important

1. Models are the heart of any LLM application — they do the "thinking" or "generation."
 2. LangChain abstracts away provider-specific APIs so you can swap models easily (e.g., move from GPT-4 to LLaMA without rewriting logic).
 3. Models are standardized into categories so developers can know exactly how to use them.

How it Works (Intuitive Explanation)

Think of LangChain models as “adapters” or “wrappers”:

- You choose a model (say GPT-4).
- LangChain wraps it into a standard interface (predict, generate, stream, etc.).
- Now, whether you switch to Cohere or HuggingFace, the code calling `llm.predict("...")` stays the same.

It's like using a **universal charger**: one plug, but it works with different devices because of the adapter.

Types or Variants of Models in LangChain

1. **LLMs (Large Language Models)**

- Used for free-form text generation.
- Example: GPT-4, Cohere Command, Anthropic Claude.
- You call them when you need essays, reasoning, explanations, etc.

2. **Chat Models**

- Special kind of LLMs designed for conversational input/output.
- Example: OpenAI ChatGPT models, Anthropic Claude chat, etc.
- Input is a list of messages (system, user, assistant), not plain text.

3. **Text Embedding Models**

- Instead of generating text, they convert text into vectors (numeric arrays).

- Useful for semantic search, retrieval, similarity matching.
- Example: OpenAI text-embedding-ada-002, HuggingFace embedding models.

4. **Image / Multi-modal Models (emerging)**

- LangChain can also integrate with models that handle text-to-image or image-to-text tasks.

Code Examples (Python)

Example 1: Using an LLM

```
from langchain_openai import OpenAI

llm = OpenAI(model="gpt-3.5-turbo-instruct")
response = llm.invoke("Explain quantum physics in simple words.")
print(response)
```

Example 2: Using a Chat Model

```
from langchain_openai import ChatOpenAI
from langchain.schema import HumanMessage

chat = ChatOpenAI(model="gpt-4")
response = chat.invoke([HumanMessage(content="Hello, how are you?")])
print(response.content)
```

Example 3: Using an Embedding Model

```
from langchain_openai import OpenAIEMBEDDINGS

embeddings = OpenAIEMBEDDINGS(model="text-embedding-ada-002")
vector = embeddings.embed_query("Machine learning basics")
print(vector[:5]) # first 5 numbers
```

Real-life Analogy

- LLMs = Like a writer who can generate essays or articles.
- Chat Models = Like a conversational partner (you talk, it replies in context).
- Embedding Models = Like a librarian who gives each book a "number tag" so you can easily find similar books later.

Applications / Use Cases

- LLMs → Summarization, content generation, reasoning.
 - Chat Models → Chatbots, personal assistants, customer support.
 - Embedding Models → Search engines, RAG (retrieval-augmented generation), clustering similar texts.
-

Advantages and Limitations

Advantages:

- Unified interface across providers.
- Easy model switching.
- Supports text, chat, and embeddings.

Limitations:

- Performance depends on the underlying model/provider.
 - Some advanced provider-specific features may not be fully exposed.
 - Requires API keys (OpenAI, Cohere, etc.) for most models.
-

Final Summary with Key Points

- In LangChain, **Models** are the building blocks that interact with LLMs, chat models, and embeddings.
- They abstract away provider differences, giving you a standard interface.
- Three main types: **LLMs (text generation)**, **Chat Models (conversation)**, and **Embedding Models (vectorization)**.

- You can swap models easily without rewriting much code.
 - They are used in applications like chatbots, summarization, semantic search, and RAG systems.
-

Would you like me to also give you a **flow diagram** showing how these three model types fit into LangChain's ecosystem (like input → model → output)?

Perfect question — let's carefully break down the **difference between LLM models and Chat models in LangChain**, and when/where/how to use them. I'll explain in a teacher-style, notes-friendly way so you can paste this directly into your notes.

Introduction / Definition

1. **LLM Models**

- General-purpose large language models that take plain text as input and return plain text as output.
- Example: text-davinci-003, gpt-3.5-turbo-instruct.

2. **Chat Models**

- A specialized wrapper around LLMs that are trained and optimized for **conversation-like input/output**.
 - Example: gpt-3.5-turbo, gpt-4, claude-3-opus.
 - Input is not plain text, but a structured list of messages (system, user, assistant).
-

Why it is Important

- Choosing **LLM vs Chat Model** depends on your task.

- Wrong choice = awkward input handling or poor results.
 - LangChain makes both accessible, but you need to know their strengths.
-

How it Works (Intuitive Explanation)

Think of it like this:

- **LLM** = You give a prompt: "Summarize this text . . ." → it returns raw text.
- **Chat Model** = You create a dialogue setup:
 - System: "You are a helpful tutor."
 - User: "Explain transformers."
 - Assistant (model): "Transformers are..."

It's like the difference between:

- Asking a **speech writer** (LLM) to just write something.
 - Talking to a **friend in conversation** (Chat Model).
-

Key Differences

1. **Input Format**

- LLM → Plain string prompt.
- Chat Model → List of messages (system, user, assistant).

2. **Output Format**

- LLM → Raw string.
- Chat Model → Message object with content.

3. **Optimization**

- LLM → Good for tasks like summarization, classification, or plain generation.

- Chat Model → Better for multi-turn dialogue, context retention, and role-specific responses.

4. **Control**

- LLM → You control everything with prompt engineering in a single text string.
 - Chat Model → You control roles (system, user, assistant) to guide behavior.
-

Code Examples (Python)

Example 1: Using an LLM

```
from langchain_openai import OpenAI

llm = OpenAI(model="text-davinci-003")
response = llm.invoke("Write a short story about a robot learning emotions.")
print(response)
```

Example 2: Using a Chat Model

```
from langchain_openai import ChatOpenAI
from langchain.schema import HumanMessage, SystemMessage

chat = ChatOpenAI(model="gpt-4")
messages = [
    SystemMessage(content="You are a kind and patient tutor."),
    HumanMessage(content="Explain the difference between supervised and
unsupervised learning.")
]
response = chat.invoke(messages)
print(response.content)
```

Real-life Analogy

- **LLM** = Like giving an instruction to a **scribe**: “Write an essay on climate change.”
 - **Chat Model** = Like having a **conversation with a tutor**: “Can you explain climate change?” → Tutor explains → You ask another question → Tutor remembers.
-

When to Use

1. **Use LLM Models when...**

- You have one-shot tasks.
- Input is plain text (not dialogue).
- Examples: text summarization, code generation, classification, translation.

2. **Use Chat Models when...**

- You want natural conversation.
- You need to maintain context across turns.
- You want to assign roles (system = rules, user = query).
- Examples: chatbots, tutoring systems, assistants, role-based conversations.

Applications / Use Cases

LLM Models

- Summarization of articles.
- One-time Q&A without context.
- Code completion.
- Text classification.

Chat Models

- Customer support bots.
- AI tutors / companions.
- Multi-turn QA systems.
- Role-playing agents (doctor, lawyer, mentor, etc.).

Advantages and Limitations

LLM Models

- Simpler interface.
- Good for plain text → text.

- Less natural for dialogues.

Chat Models

- Better conversation handling.
 - Structured input (system/user roles).
 - Slightly more setup needed (messages list).
-

Final Summary with Key Points

- **LLMs** → Work with plain text input/output. Best for single-shot tasks like summarization, classification, or text generation.
 - **Chat Models** → Work with structured message lists, support roles, and maintain context across turns. Best for conversations, assistants, and dialogue-heavy apps.
 - **Rule of thumb:**
 - If you need “task → result” → use **LLM**.
 - If you need “dialogue → memory → context” → use **Chat Model**.
-

Do you want me to also make a **side-by-side comparison table** (LLM vs Chat Model) for quick revision?

Great — here's a **side-by-side comparison table of LLM Models vs Chat Models in LangChain**. This will be handy for quick revision in your notes.

LLM Models vs Chat Models in LangChain

Aspect	LLM Models	Chat Models
Definition	General-purpose large language models that take plain text input	Specialized LLMs optimized for conversation, with

Aspect	LLM Models	Chat Models
Input Format	and return plain text output Single string prompt (e.g., "Summarize this article.")	structured message input/output List of messages (System, User, Assistant)
Output Format	Raw string text	Message object (response.content)
Optimization	Suited for single-shot tasks, text completion, classification	Suited for multi-turn dialogue, role-based interactions
Control	Prompt engineering in one text block	Role control (system, user) + message history
Ease of Use	Simpler, fewer steps	Slightly more setup (need message objects)
When to Use	Summarization, code generation, classification, translation	Chatbots, tutoring systems, assistants, conversational flows
Applications	- Article summaries- Code completion- Sentiment analysis	- Customer support bots- Multi-turn Q&A- AI tutors or companions
Analogy	Like a scribe who writes exactly what you ask once	Like a tutor who remembers your past questions and continues the conversation
Limitations	Less natural for dialogue, no role separation	Slightly more complex setup, requires structured inputs

Final Key Rule:

- **Use LLM Models** → if you want plain text → text tasks (single-shot).

- **Use Chat Models** → if you want dialogue, memory, or role-based conversation.
-

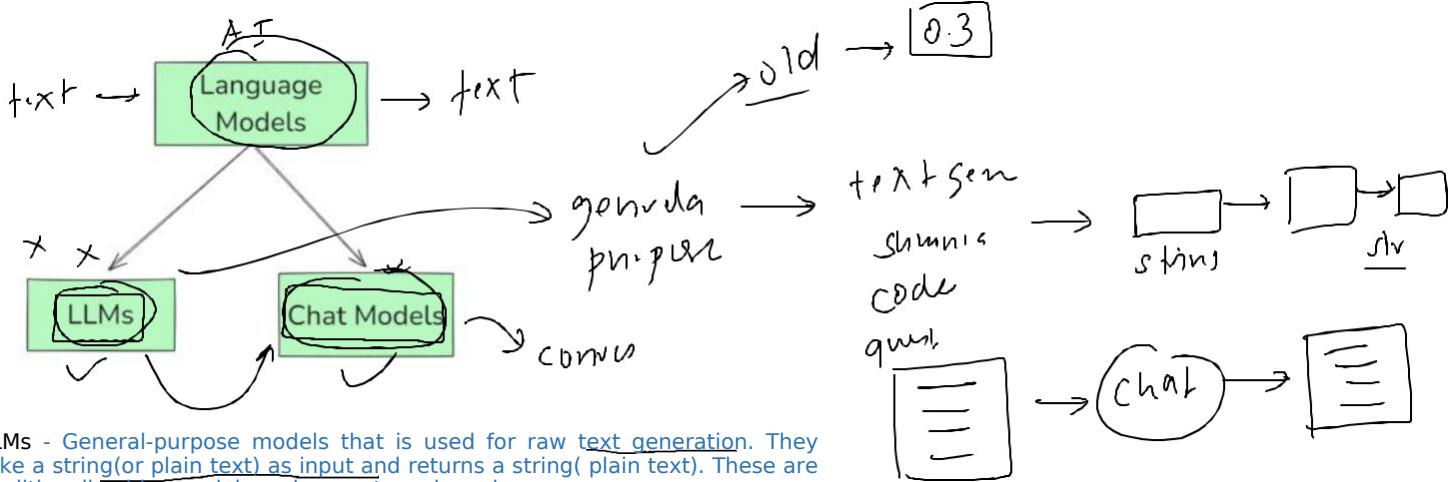
Do you also want me to add a **workflow diagram** showing:

User → LLM (task) vs User → Chat Model (multi-turn)?

That way you'll get a quick visual differentiation.

Language Models

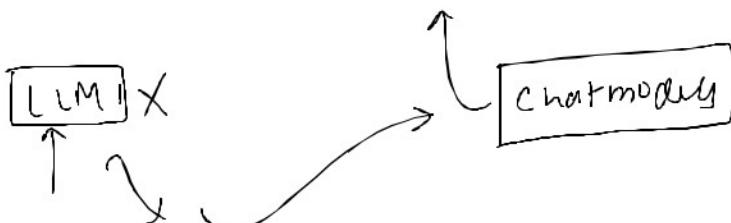
Language Models are AI systems designed to process, generate, and understand natural language text.



LLMs - General-purpose models that is used for raw text generation. They take a string (or plain text) as input and returns a string (plain text). These are traditionally older models and are not used much now.

Chat Models - Language models that are specialized for conversational tasks. They take a sequence of messages as inputs and return chat messages as outputs (as opposed to using plain text). These are traditionally newer models and used more in comparison to the LLMs.

Feature	LLMs (Base Models)	Chat Models (Instruction-Tuned)
Purpose	Free-form text generation	Optimized for multi-turn conversations
Training Data	General text corpora (books, articles)	Fine-tuned on chat datasets (dialogues, user-assistant conversations)
Memory & Context	No built-in memory	Supports structured conversation history
Role Awareness	No understanding of "user" and "assistant" roles	Understands "system", "user", and "assistant" roles
Example Models	GPT-3, Llama-2-7B, Mistral-7B, OPT-1.3B	GPT-4, GPT-3.5-turbo, Llama-2-Chat, Mistral-Instruct, Claude
Use Cases	Text generation, summarization, translation, creative writing, code generation	Conversational AI, chatbots, virtual assistants, customer support, AI tutors



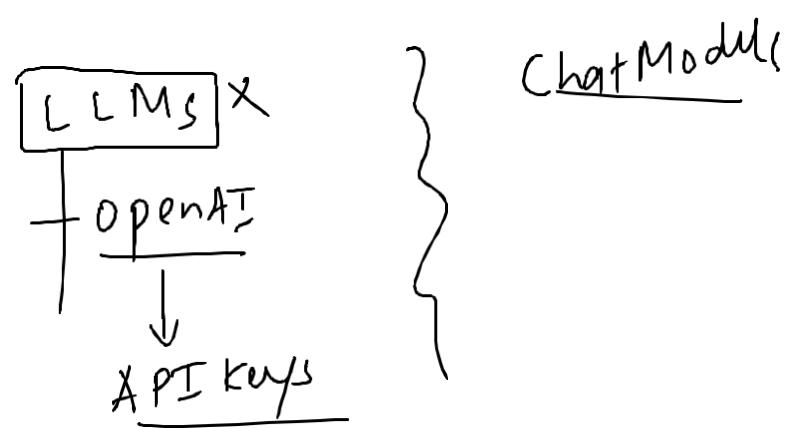
Setup

10 09:
February 21
2025

3. Models Page 26

Demo

1. OpenAI ✓
2. Anthropic ✓
3. Google ✓
4. HuggingFace ✓



`temperature` is a parameter that controls the randomness of a language model's output. It affects how **creative or deterministic** the responses are.

- Lower values (0.0 - 0.3) → More **deterministic** and predictable.
- Higher values (0.7 - 1.5) → More **random**, creative, and diverse.

Use Case	Recommended Temperature
Factual answers (math, code, facts)	0.0 - 0.3
Balanced response (general QA, explanations)	0.5 - 0.7
Creative writing, storytelling, jokes	0.9 - 1.2
Maximum randomness (wild ideas, brainstorming)	1.5+

Open Source Models

11 February 2025 08:58

Open-source language models are freely available AI models that can be downloaded, modified, fine-tuned, and deployed without restrictions from a central provider. Unlike closed-source models such as OpenAI's GPT-4, Anthropic's Claude, or Google's Gemini, open-source models allow full control and customization.

Feature	Open-Source Models	Closed-Source Models
Cost	Free to use (no API costs)	Paid API usage (e.g., OpenAI charges per token)
Control	Can modify, fine-tune, and deploy anywhere	Locked to provider's infrastructure
Data Privacy	Runs locally (no data sent to external servers)	Sends queries to provider's servers
Customization	Can fine-tune on specific datasets	No access to fine-tuning in most cases
Deployment	Can be deployed on on-premise servers or cloud	Must use vendor's API

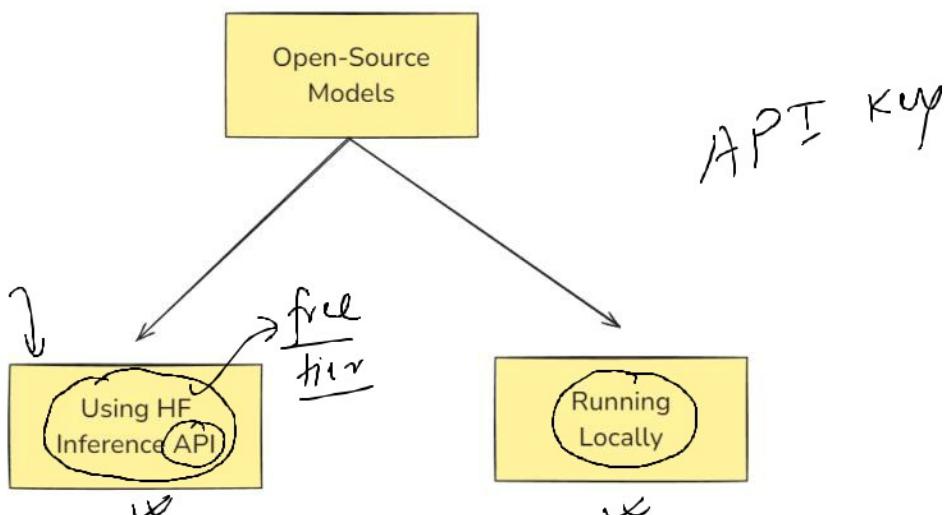
Some Famous Open Source Models

Model	Developer	Parameters	Best Use Case
LLaMA-2-7B/13B/70B	Meta AI	7B - 70B	General-purpose text generation
Mixtral-8x7B	Mistral AI	8x7B (MoE)	Efficient & fast responses
Mistral-7B	Mistral AI	7B	Best small-scale model (outperforms LLaMA-2-13B)
Falcon-7B/40B	TII UAE	7B - 40B	High-speed inference
BLOOM-176B	BigScience	176B	Multilingual text generation
GPT-J-6B	EleutherAI	6B	Lightweight and efficient
GPT-NeoX-20B	EleutherAI	20B	Large-scale applications
StableLM	Stability AI	3B - 7B	Compact models for chatbots

Where to find them?

HuggingFace - The largest repository of open-source LLMs

Ways to use Open-source Models

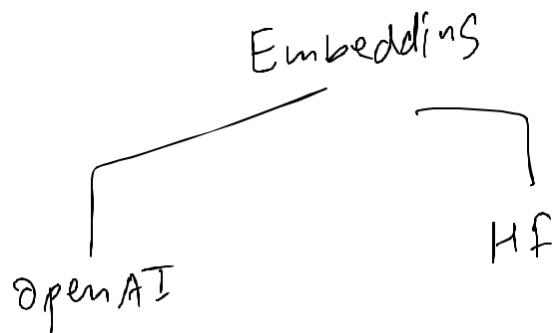
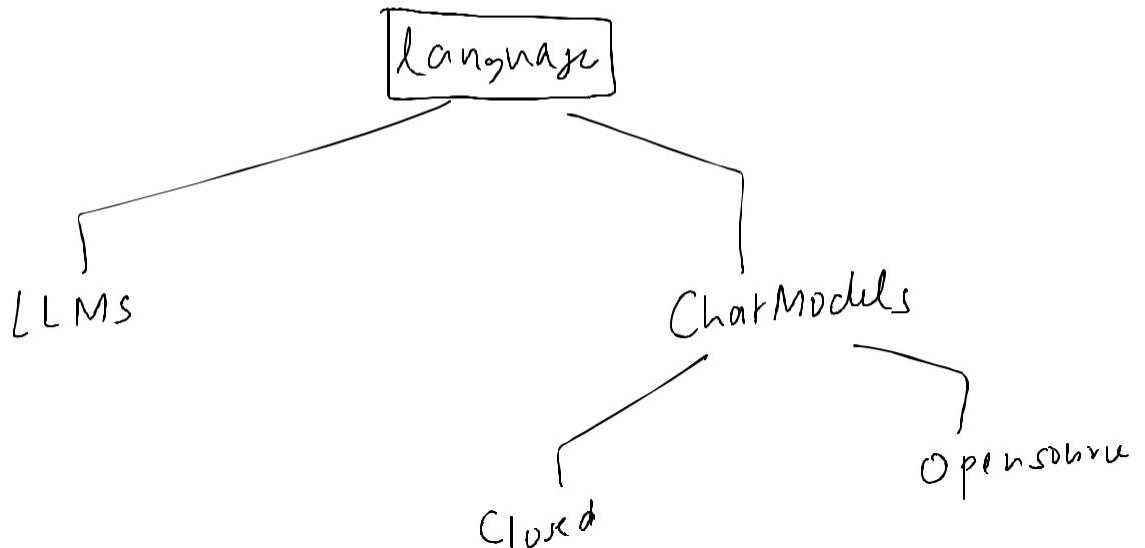


Disadvantages

Disadvantage	Details
High Hardware	Running <u>large</u> models (e.g., LLaMA-2-70B) requires <u>expensive</u> GPUs.

Diagram showing a bracket on the left side of the table, spanning the first four rows.

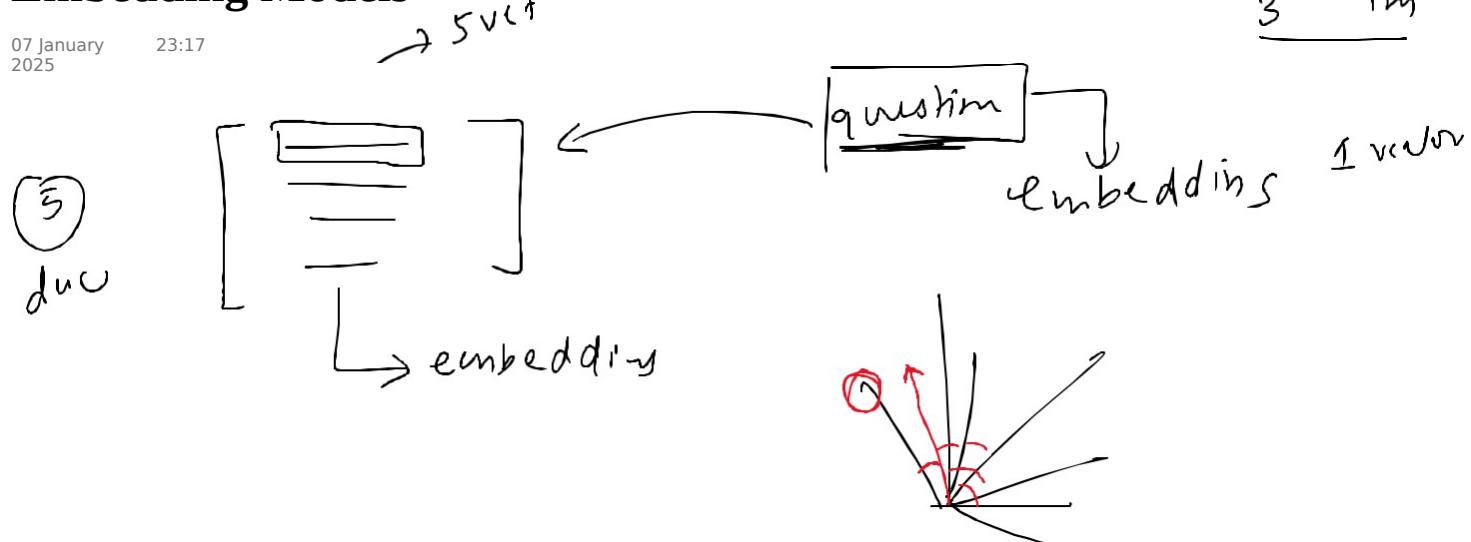
Disadvantage	Details
High Hardware Requirements	Running <u>large</u> models (e.g., LLaMA-2-70B) requires <u>expensive</u> GPUs.
Setup Complexity	Requires installation of dependencies like PyTorch, CUDA, transformers.
Lack of RLHF	Most open-source models don't have <u>fine-tuning</u> with <u>human feedback</u> , making them weaker in instruction-following.
Limited Multimodal Abilities	Open models don't support <u>images</u> , <u>audio</u> , or <u>video</u> like GPT-4V.



Embedding Models

07 January
2025

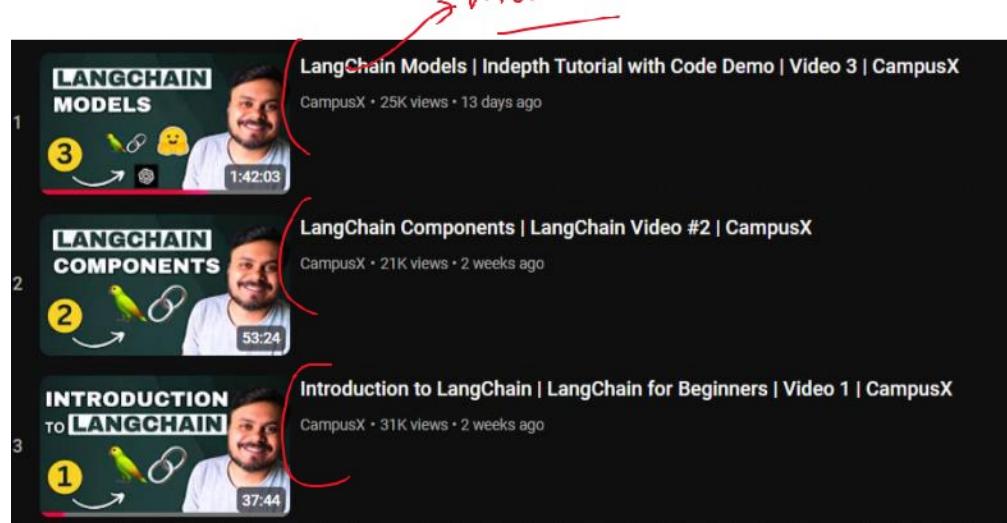
23:17



3. Models Page 30

Recap

24 February 2025 09:42



models

4 → Prompts

A mistake from my side!

24 09:
February 42
2025

temp → 0 → 2
↓ ↑ ↑

4. Prompts Page 32

Prompts

10 January 08:37
2025

Prompts are the input instructions or queries given to a model to guide its output.

```
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

load_dotenv()

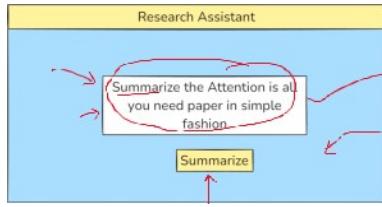
model = ChatOpenAI(model='gpt-4', temperature=1.5, max_completion_tokens=10)

result = model.invoke("Write a 5 line poem on cricket") ←
print(result.content)
```

text based
multimodal prompts
↓
image / sound / video

Static vs Dynamic Prompts

14 February 00:0
2025



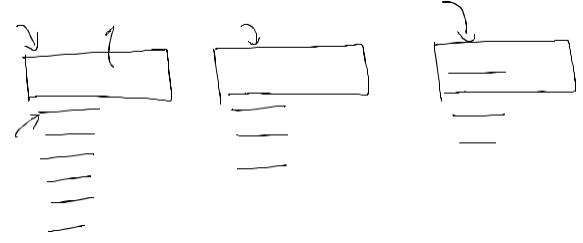
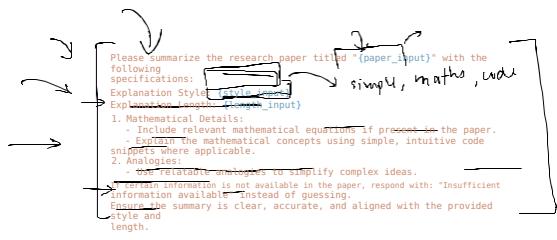
```
paper_input = st.selectbox("Select Research Paper Name", ["Select...", "Attention is All You Need", "BERT: Pre-training of Deep Bidirectional Transformers", "GPT-3: Language Models are Few-Shot Learners", "Diffusion Models Beat GANs on Image Synthesis"])

style_input = st.selectbox("Select Explanation Style", ["Beginner-Friendly", "Technical", "Code-Oriented", "Mathematical"])

length_input = st.selectbox("Select Explanation Length", ["Short (1-2 paragraphs)", "Medium (3-5 paragraphs)", "Long (detailed explanation)"])
```

```
style_input = st.selectbox("Select Explanation Style", ["Beginner-Friendly", "Technical", "Code-Oriented", "Mathematical"])

length_input = st.selectbox("Select Explanation Length", ["Short (1-2 paragraphs)", "Medium (3-5 paragraphs)", "Long (detailed explanation)"])
```



A **PromptTemplate** in LangChain is a structured way to create prompts dynamically by inserting variables into a predefined template. Instead of hardcoding prompts, PromptTemplate allows you to define placeholders that can be filled in at runtime with different inputs.

This makes it reusable, flexible, and easy to manage, especially when working with dynamic user inputs or automated workflows.

Prompt Template

10 08:
January 37
2025

A **PromptTemplate** in LangChain is a structured way to create prompts dynamically by inserting variables into a predefined template. Instead of hardcoding prompts, PromptTemplate allows you to define placeholders that can be filled in at runtime with different inputs.

This makes it reusable, flexible, and easy to manage, especially when working with dynamic user inputs or automated workflows.

Why use PromptTemplate over f strings?

1. Default validation
2. Reusable
3. LangChain Ecosystem

Got it Stark

Let's carefully go from the very beginning and explain **Prompts in LangChain** in your preferred structured, notes-friendly style.

=====

=====

PROMPTS IN LANGCHAIN

1. Introduction / Definition

A **Prompt** is the text instruction, context, or question given to a language model (like GPT, Mistral, or LLaMA) to make it perform a specific task.

In LangChain, prompts are not just plain strings. They are structured objects that can include templates, roles (system/human/AI), and formatting instructions. This makes prompts **reusable, dynamic, and easier to manage**.

2. Why it is Important

- The **same model** can give very different answers depending on how the prompt is written.
 - Good prompts reduce hallucination and guide the model to follow instructions.
 - LangChain provides **PromptTemplate** and **ChatPromptTemplate** so developers don't have to hardcode long strings every time.
 - Structured prompts = reproducible results.
-

3. How it Works (Intuitive Explanation)

Think of a prompt like a **question paper** for a student (the LLM):

- If the questions are clear, the student (LLM) will give correct answers.
- If the questions are vague, the answers will be random.

LangChain lets you create **templates with blanks** (placeholders) that can be filled dynamically with different values. This means you can ask the **same style of question** to the model many times with different inputs.

4. Types or Variants in LangChain

- a) **PromptTemplate** → For single-turn tasks with placeholders.

Example: "Tell me a {adjective} story about a {animal}".

- b) **ChatPromptTemplate** → For multi-turn or chat-based models, with role-based messages (system, human, AI).

c) **Few-shot Prompting** → Providing examples in the prompt so the model learns from patterns.

d) **Structured Output with OutputParser** → Used along with prompts to force the model's response into JSON, lists, or other formats.

5. Code Example (Python with PromptTemplate)

```
from langchain.prompts import PromptTemplate
from langchain_huggingface import HuggingFaceHub

# Define a prompt template with placeholders
template = "Translate the following sentence into French: {sentence}"

prompt = PromptTemplate.from_template(template)

# Fill the placeholder
final_prompt = prompt.format(sentence="I love programming with Python.")

print("Final Prompt:\n", final_prompt)

# Load LLM (Mistral on HuggingFace Hub)
llm = HuggingFaceHub(repo_id="mistralai/Mistral-7B-Instruct-v0.2")

# Send prompt to the model
response = llm.invoke(final_prompt)

print("\nModel Response:\n", response)
```

Expected Output:

Final Prompt:
Translate the following sentence into French: I love programming with Python.

Model Response:
J'aime programmer avec Python.

6. Real-life Analogy

Imagine you are a **teacher making worksheets** for students.

- The worksheet has blanks like: “The capital of {country} is ____.”
 - Each time you print the worksheet, you can fill {country} with “India”, “France”, “Japan”...
 - Same structure, different inputs.
That's exactly how **PromptTemplate** works.
-

7. Applications / Use Cases

- Translating text (EN → FR, etc.).
 - Summarizing articles.
 - Question answering (FAQ bots).
 - Generating code snippets.
 - Chatbots with personalities (system prompts like “You are a helpful teacher”).
-

8. Advantages and Limitations

Advantages:

- Prompts are reusable and modular.
- Reduces messy hardcoded strings in code.
- Easy to scale for multiple queries.

Limitations:

- Prompt quality strongly affects results.
 - Trial-and-error may be needed to get best phrasing.
 - Prompts are not enough alone; sometimes need fine-tuning or retrieval augmentation.
-

9. Final Summary with Key Points

- Prompts = instructions to LLMs.
- In LangChain, prompts are **structured objects** (not just plain strings).
- **PromptTemplate** → placeholders for dynamic inputs.
- **ChatPromptTemplate** → role-based prompts for chat models.
- Good prompts = better control over model outputs.

- Think of them as “question papers” or “worksheets” given to the AI student.
- =====
- =====

Would you like me to now give a **side-by-side comparison** of PromptTemplate vs ChatPromptTemplate with code so you can clearly see the difference?

Excellent question Stark.
Let's go step by step and answer clearly:

=====

=====

WHY USE PROMPTTEMPLATE INSTEAD OF JUST F-STRINGS

1. Introduction / Definition

- **f-strings**: A quick way in Python to insert variables into a string.
 - **PromptTemplate**: A LangChain class that does the same basic thing, but with extra **features** designed specifically for working with LLMs.
-

2. Why it is Important

While f-strings can format text, PromptTemplate adds:

- **Validation** → Checks if you passed all required variables.
- **Flexibility** → Works seamlessly inside LangChain pipelines (Chains, Agents).
- **Reusability** → Templates can be stored, reused, and shared easily.
- **Integration** → Plays nicely with output parsers, chat prompts, and few-shot examples.

So, PromptTemplate is like a **professional toolkit** for prompts, while f-strings are just a simple hammer.

3. How it Works (Intuitive Explanation)

- f-strings = “direct formatting” → “Translate {sentence}”.
- PromptTemplate = “managed formatting” → LangChain keeps track of placeholders, variables, and formatting, ensuring correctness and smooth pipeline usage.

4. Example with f-string vs PromptTemplate

Using f-string:

```
sentence = "I love programming with Python."
prompt = f"Translate the following sentence into French: {sentence}"
print(prompt)
```

Using PromptTemplate:

```
from langchain.prompts import PromptTemplate

template = "Translate the following sentence into French: {sentence}"
prompt = PromptTemplate.from_template(template)

final_prompt = prompt.format(sentence="I love programming with Python.")
print(final_prompt)
```

Both output:

Translate the following sentence into French: I love programming with Python.

So far, **same result**.

5. Where PromptTemplate Wins (Advanced Use)

a) Validation

```
prompt.format()  
# Raises error: Missing value for 'sentence'
```

With f-strings, Python would just throw a `NameError`, but PromptTemplate tells you **exactly which variable is missing**.

b) Multiple Inputs

```
template = "Tell me a {adjective} story about a {animal}"  
prompt = PromptTemplate.from_template(template)  
print(prompt.format(adjective="funny", animal="dog"))
```

→ Works cleanly when you have **many variables**.

c) Integration with LangChain Chains

PromptTemplate can plug directly into `LLMChain`, `ChatPromptTemplate`, `Agent`, etc.

With f-strings, you'd have to manually manage string formatting every time.

d) Few-Shot Prompting (dynamic examples)

PromptTemplate supports inserting **lists of examples** programmatically — f-strings cannot handle this cleanly.

6. Real-life Analogy

- f-strings = Writing a one-time note on paper.
Quick and dirty.
- PromptTemplate = A **reusable form** with blanks (like a worksheet). You can print and reuse it with different data many times. Plus, it checks you filled in all blanks correctly.

7. Applications / Use Cases

- When you only need one quick prompt → f-strings are fine.
 - When building a chatbot, RAG pipeline, or agent → PromptTemplate is better, since it's structured and integrates with the rest of LangChain.
-

8. Advantages and Limitations

Advantages:

- Structured, validated, reusable.
- Works seamlessly in LangChain pipelines.
- Scales well for large projects.

Limitations:

- Slightly more boilerplate than f-strings.
 - Overhead may feel unnecessary for very small scripts.
-

9. Final Summary

- For **tiny scripts or debugging**, f-strings are fine.
 - For **serious LangChain apps**, PromptTemplate is better because:
 - Validates inputs
 - Handles multiple variables
 - Reusable in chains/agents
 - Supports advanced features like few-shot prompts
-
-

So: **f-string = quick & dirty; PromptTemplate = scalable & safe.**

Do you want me to also show how PromptTemplate directly connects into an LLMChain, so you see the real advantage over f-strings?

4. Prompts Page 35



Messages

14 February 00:02
2025

use v:

At:

uuu. . . -

You: - - - - -

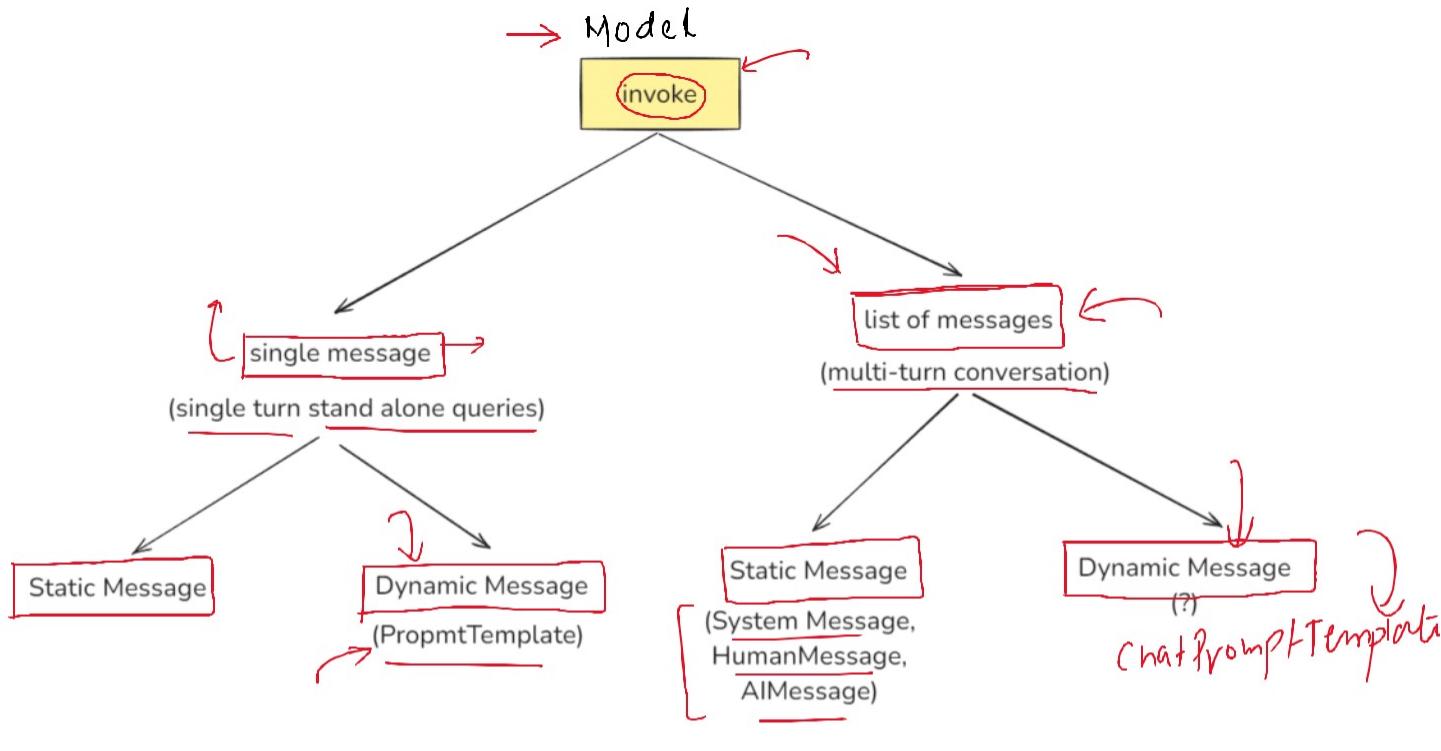
AI: - - - - -

} |
+-----+ System m/s
+-----+ Human m/s
+-----+ AI messages

4. Prompts Page 36

Chat Prompt Templates

14 February 2025 00:02



System Message → dynamic

You are a helpful [domain] expert

Human message

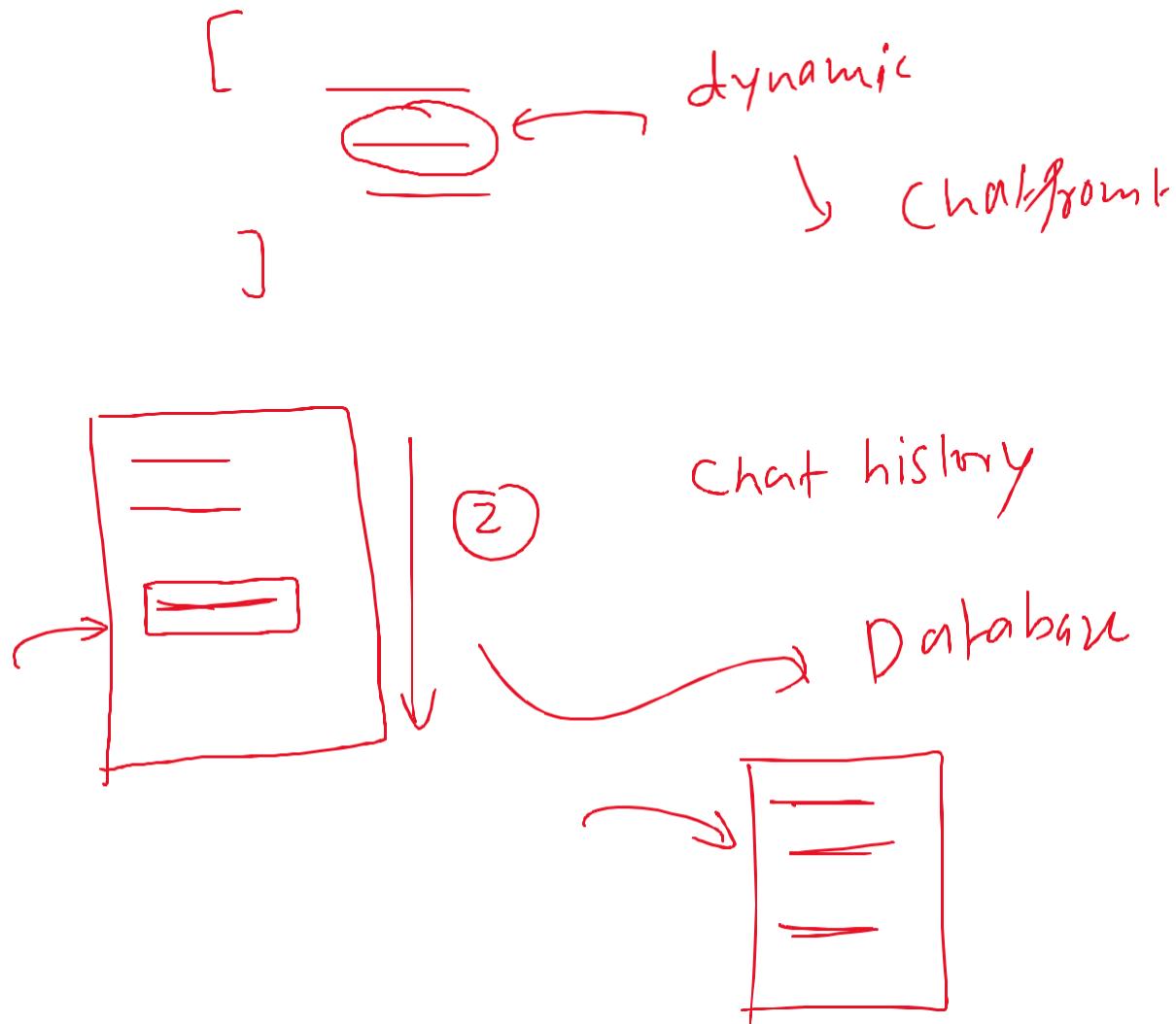
explain about [topic]

[]

Message Placeholder

14 February 2025 00:09

A **MessagesPlaceholder** in LangChain is a special placeholder used inside a **ChatPromptTemplate** to dynamically insert chat history or a list of messages at runtime.

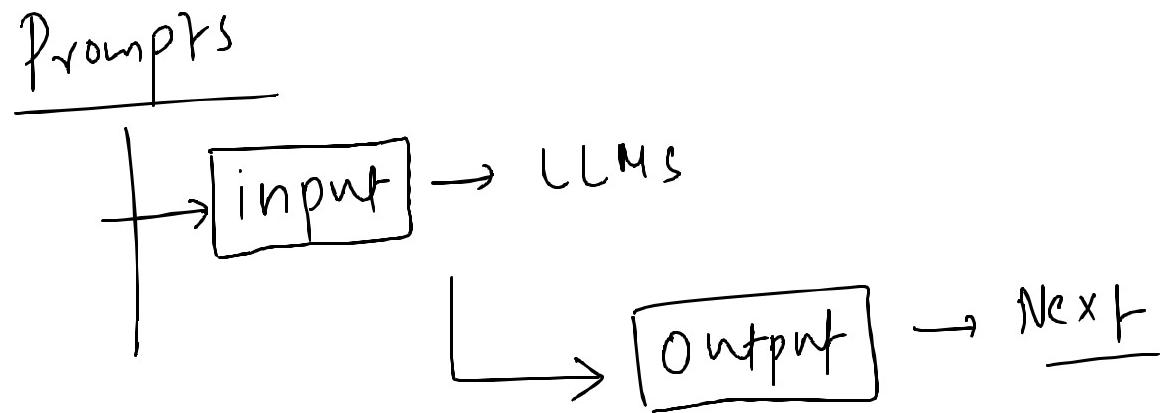


placeholder

4. Prompts Page 38

Recap

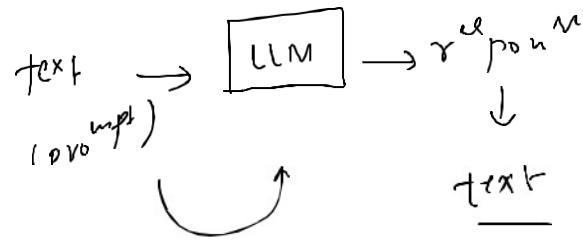
28 February 00:12
2025



Structured Output

28 February 00:13
2025

In LangChain, structured output refers to the practice of having language models return responses in a well-defined data format (for example, JSON), rather than free-form text. This makes the model output easier to parse and work with programmatically.



[Prompt] - [Can you create a one-day travel itinerary for Paris?]

[LLM's Unstructured Response]

Here's a suggested itinerary: Morning: Visit the Eiffel Tower.

Afternoon: Walk through the Louvre Museum.

Evening: Enjoy dinner at a Seine riverside café.

[JSON enforced output]

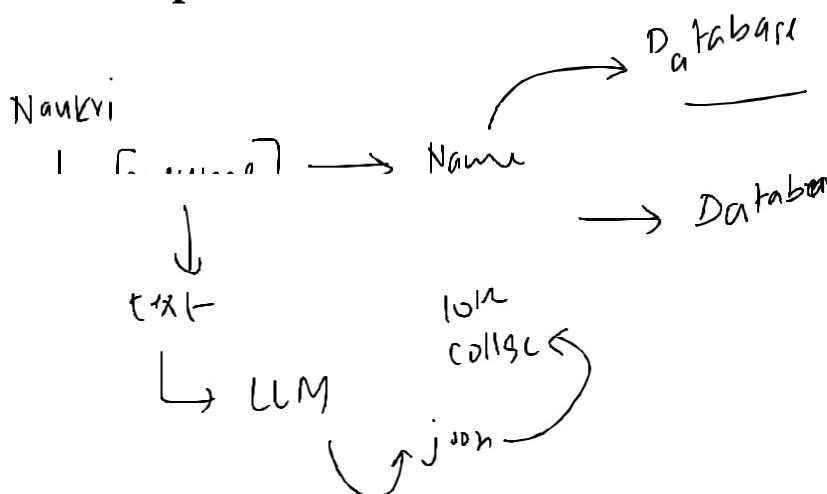
```
[  
  {"time": "Morning", "activity": "Visit the Eiffel Tower"},  
  {"time": "Afternoon", "activity": "Walk through the Louvre Museum"},  
  {"time": "Evening", "activity": "Enjoy dinner at a Seine riverside café"}]
```

Source -
<https://www.linkedin.com/pulse/structured-outputs-from-langs-langchain-output-parsers-vijay-chaudhary-wgjqc>

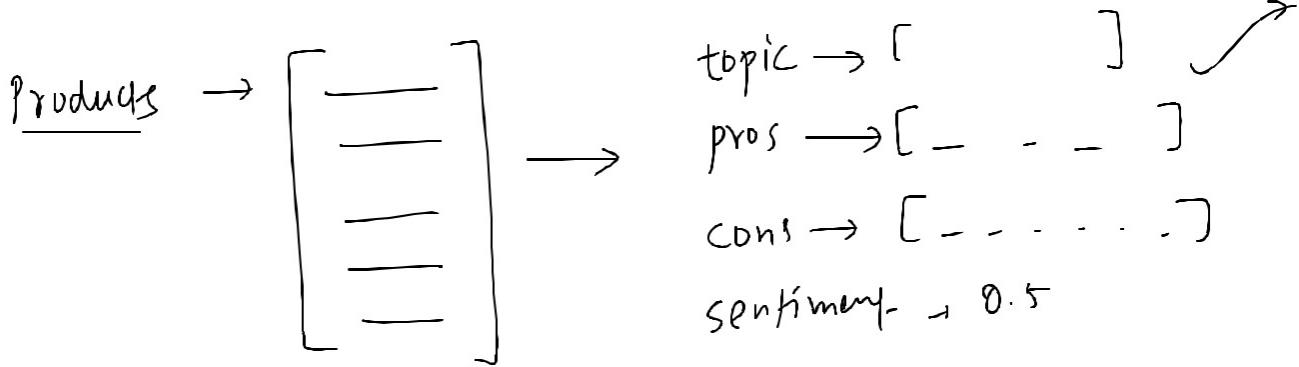
Why do we need Structured Output

28 February 00:13
2025

Data Extraction
API building
Agents



Amazon



(2)

→ Agents → Chatbots

Tools

calculator

Agent → print → struct-

numbers

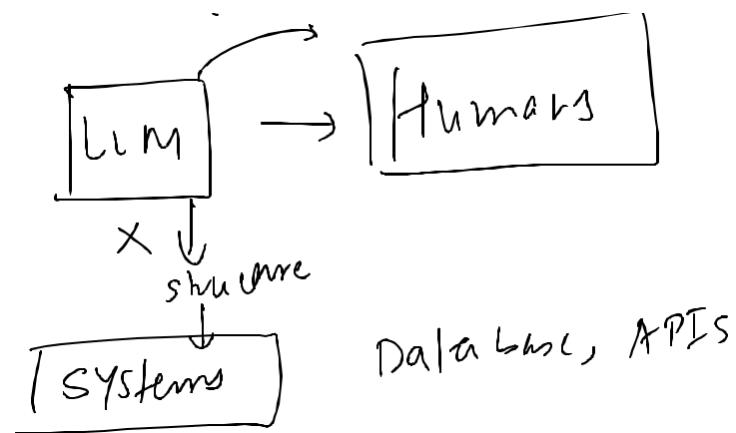
calculator

↓

Structured Output

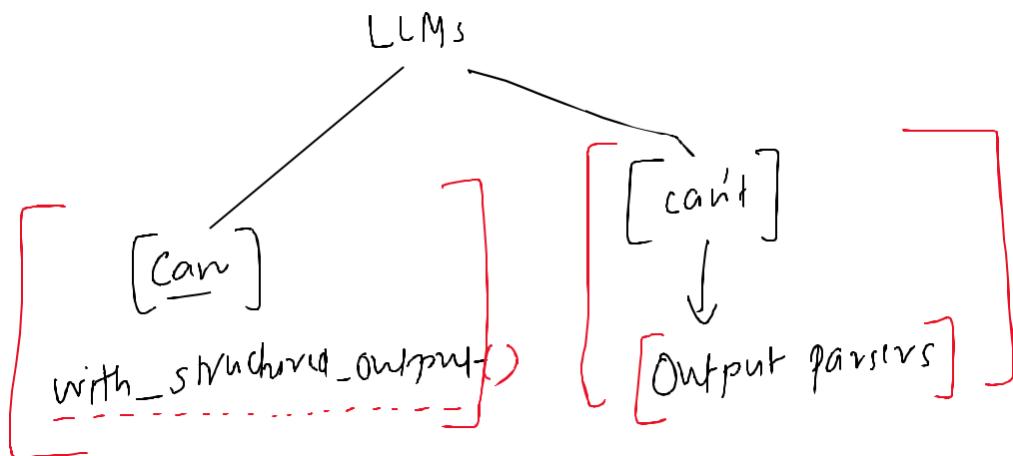
text

Diagram



Ways to get Structured Output

28 February 00:14
2025



dt for

with_structured_output

28 February 00:15
2025

(model_inference)



TypedDict

Pydantic

json-schema

You are an AI assistant that extracts structured insights from text. Given a product review, extract: - Summary: A brief overview of the main points. - Sentiment: Overall tone of the review (positive, neutral, negative). Return the response in JSON format.

TypedDict

01 March 12:59
2025

TypedDict is a way to define a dictionary in Python where you specify what keys and values should exist. It helps ensure that your dictionary follows a specific structure.

Why use TypedDict?

- It tells Python what keys are required and what types of values they should have.
- It does not validate data at runtime (it just helps with type hints for better coding).

person = { name: str
 age: int }
 sir

-> simple TypedDict
-> Annotated TypedDict
-> Literal
-> More complex -> with pros and cons

age

age = '25'

validation define

dict ~

person

class person
 name: str
 age: int

reviews

LLM

{ summary: —
 sentiment: —, —, — }

)

Pydantic

01 March 12:59
2025

Pydantic is a data validation and data parsing library for Python. It ensures that the data you work with is correct, structured, and type-safe.

Basic example ✓

Default values

Optional fields

Coerce →

Builtin validation

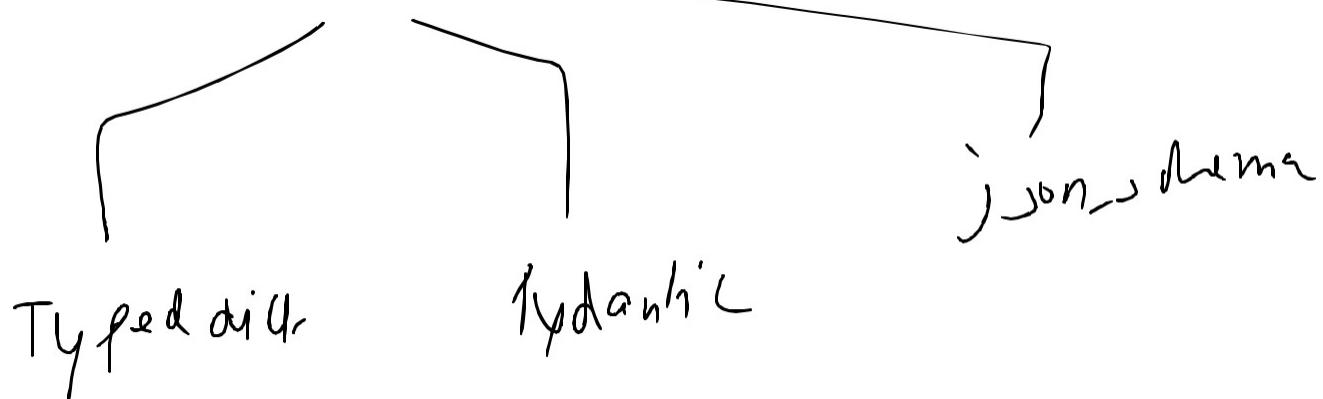
Field Function -> default values, constraints, description, regex expressions →

Returns pydantic object -> convert to json/dict

Json

01 March
2025

18:45



When to use what?

01 12:
March 59
2025

✓ Use `TypedDict` if: ✓

- You only need type hints (basic structure enforcement).
- You don't need validation (e.g., checking numbers are positive).
- You trust the LLM to return correct data.

✓ Use `Pydantic` if:

- You need data validation (e.g., sentiment must be `"positive"`, `"neutral"`, or `"negative"`).
- You need default values if the LLM misses fields.
- You want automatic type conversion (e.g., `"100"` → `100`).

✓ Use `JSON Schema` if:

- You don't want to import extra Python libraries (Pydantic).
- You need validation but don't need Python objects.
- You want to define structure in a standard JSON format.

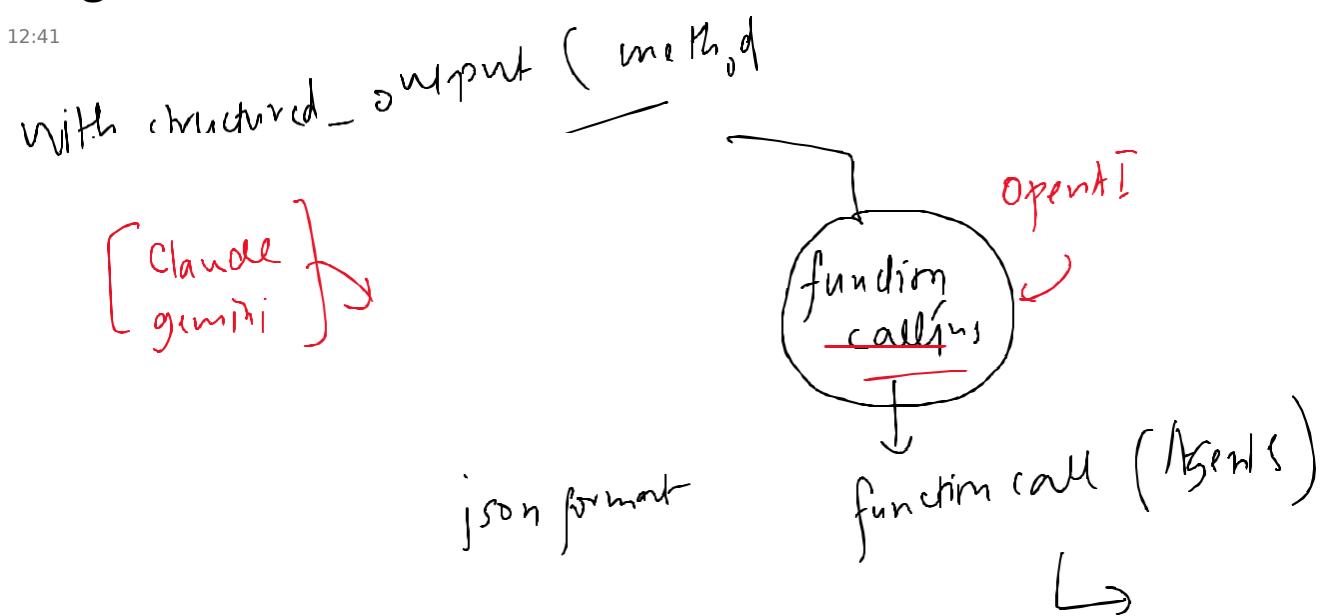
🚀 When to Use What?

Feature	TypedDict ✓	Pydantic 🚀	JSON Schema
Basic structure ✓	✓ —	✓ —	✓ —
Type enforcement —	✓ —	✓ —	✓ —
Data validation	✗	✓	✓
Default values	✗	✓	✗
Automatic conversion	✗	✓	✗
Cross-language compatibility	✗	✗	✓

5. Structured Output Page 48

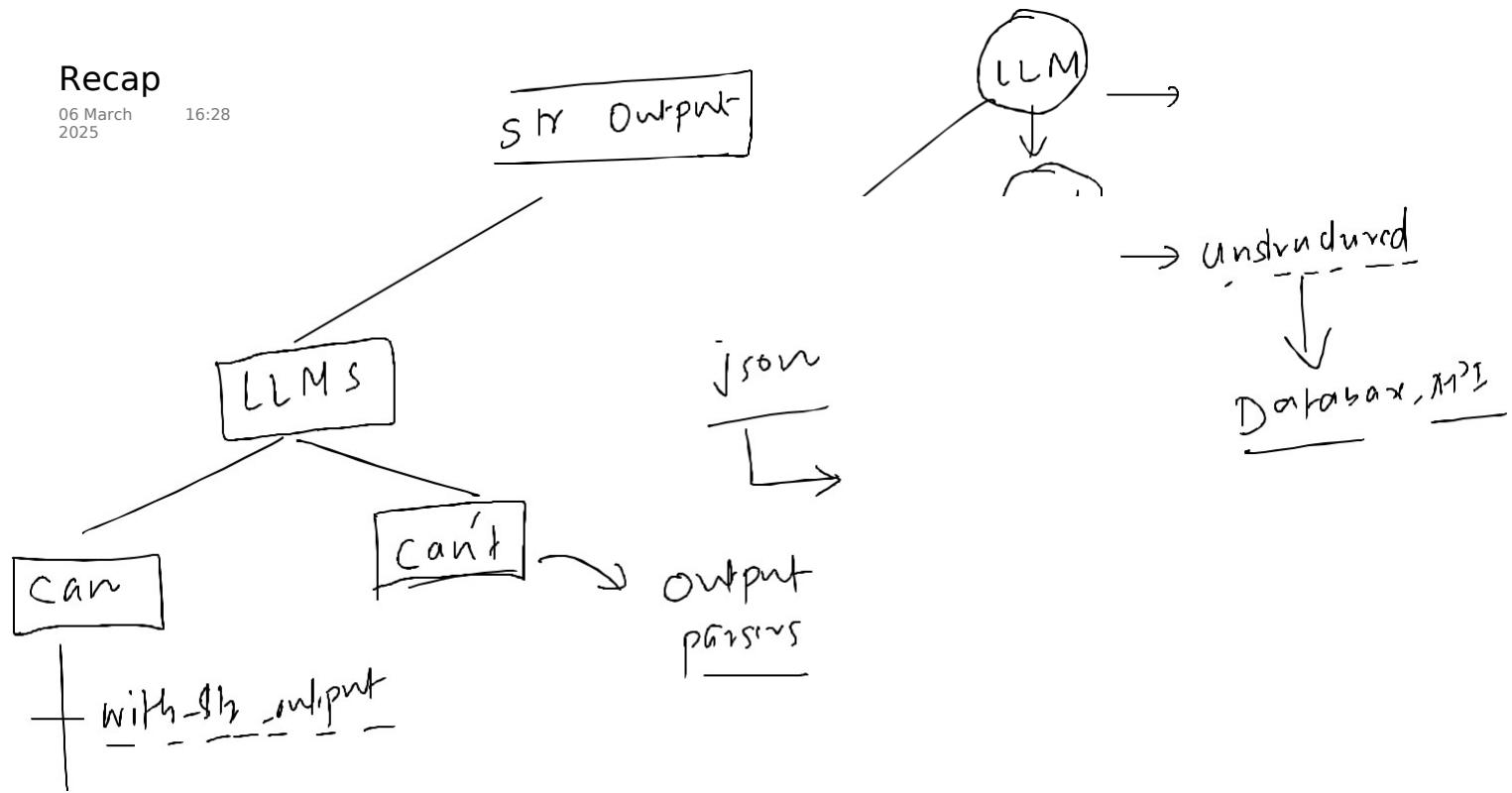
A few things to remember!

03 March
2025



Recap

06 March 2025 16:28



Output Parsers

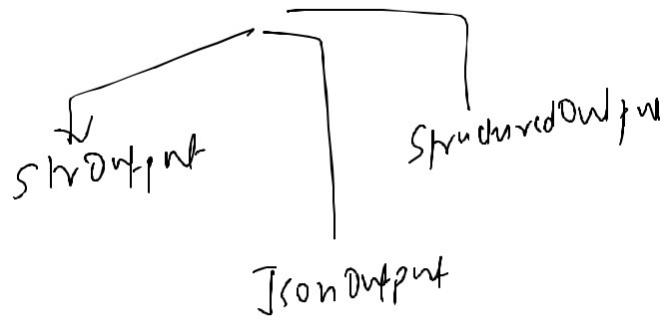
06 March 2025 16:29

Output Parsers in LangChain help convert raw LLM responses into structured formats like JSON, CSV, Pydantic models, and more. They ensure consistency, validation, and ease of use in applications.

Pydantic
o w, v, t
o 1

can

can't



StrOutputParser →

06 March 2025

16:29

→ result.content →

The StrOutputParser is the simplest output parser in LangChain. It is used

to parse the output of a Language Model (LLM) and return it as a plain

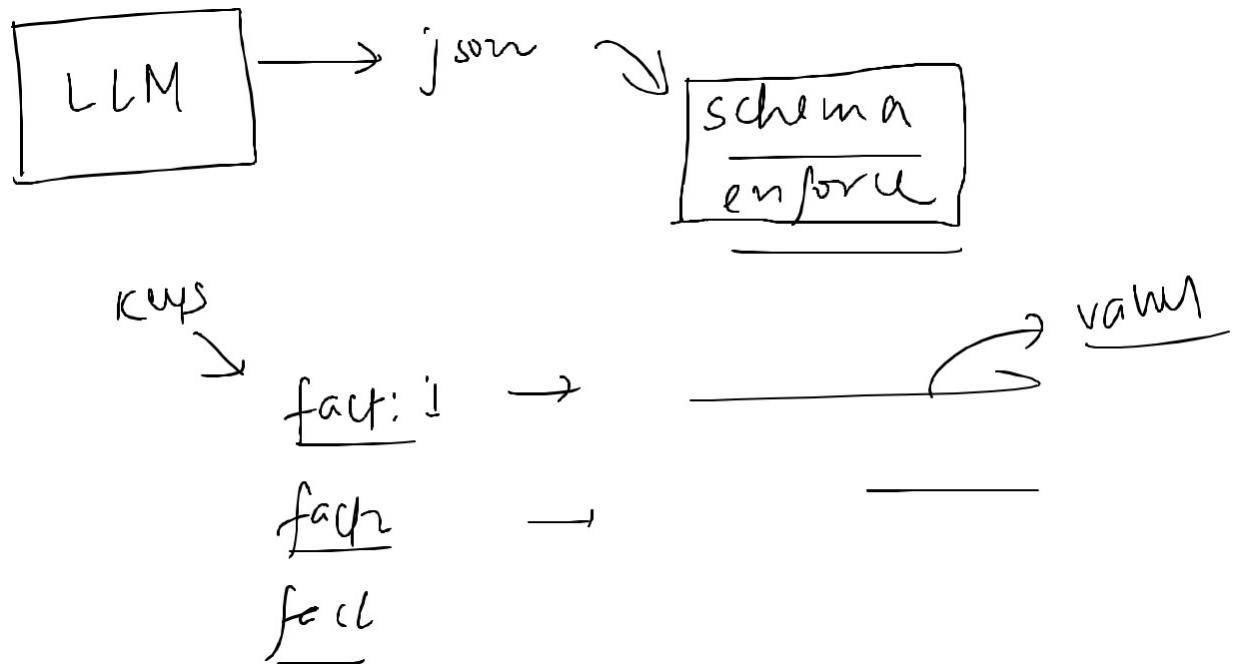
string.

Content: A Black hole is a region in space where gravity is so strong that nothing, not even light, can escape its pull. It is formed when a massive star collapses upon itself.
additional_kwargs={'refusal': None} response_metadata={'token_usage': {'completion_tokens': 37, 'prompt_tokens': 15, 'total_tokens': 52, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}}, 'model_name': 'gpt-3.5-turbo-0125', 'system_fingerprint': None, 'finish_reason': 'stop', 'logprobs': None} id='run-a7b90203-58f8-47c5-a01b-01184b6aec14-0' usage_metadata={'input_tokens': 15, 'output_tokens': 37, 'total_tokens': 52, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}



JSONOutputParser

06 March 16:29
2025



StructuredOutputParser

06 March
2025

StructuredOutputParser is an output parser in LangChain that helps extract structured JSON data from LLM responses based on predefined field schemas.

It works by defining a list of fields (ResponseSchema) that the model should return, ensuring the output follows a structured format.

Pydanchic
and
Jant

Data validation (str) name _____
(int) age 35 years → str
(str) city

6. Output Parsers Page 54

PydanticOutputParser

06 March 16:30
2025

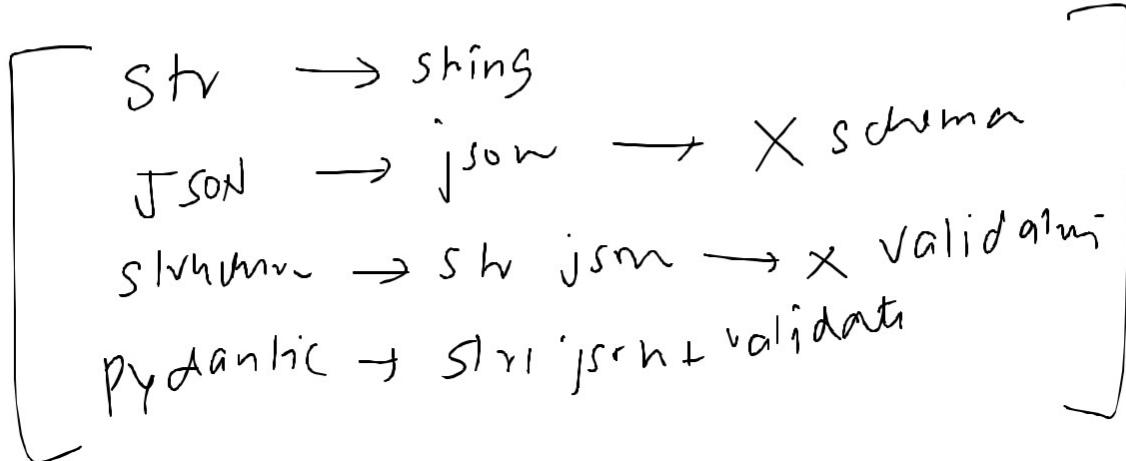
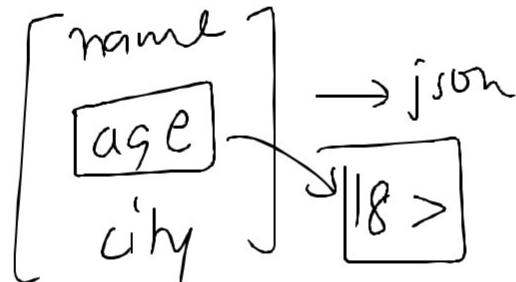
• What is PydanticOutputParser in LangChain?

PydanticOutputParser is a structured output parser in LangChain that uses Pydantic models to enforce schema validation when processing LLM responses.

→ Schema ↴
[Pydantic
object]

🚀 Why Use PydanticOutputParser?

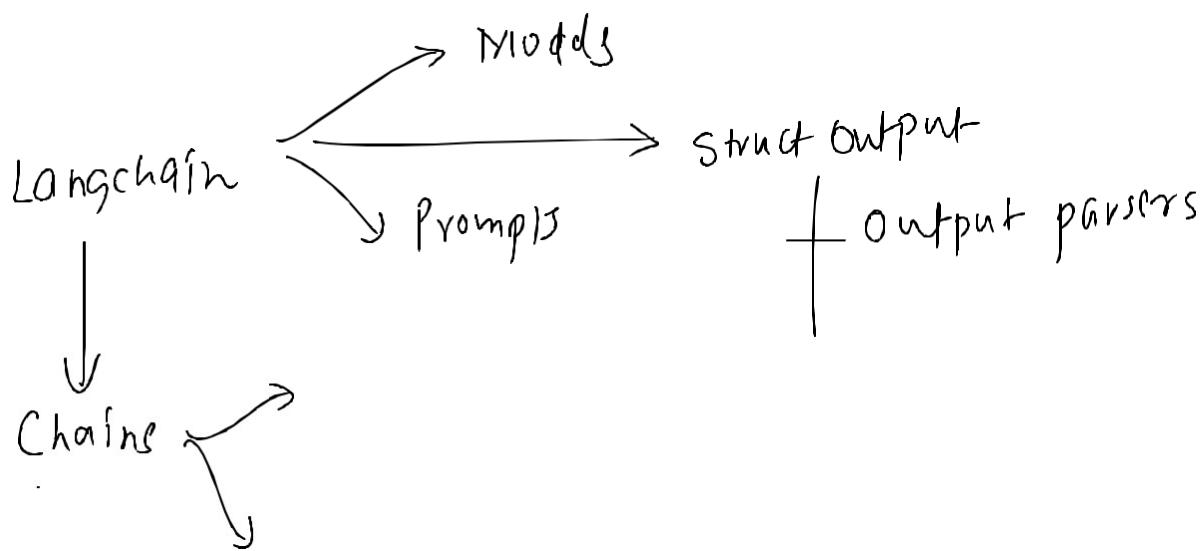
- ✓ Strict Schema Enforcement → Ensures that LLM responses follow a well-defined structure.
- ✓ Type Safety → Automatically converts LLM outputs into Python objects.
- ✓ Easy Validation → Uses Pydantic's built-in validation to catch incorrect or missing data.
- ✓ Seamless Integration → Works well with other LangChain components.



Chains

Recap

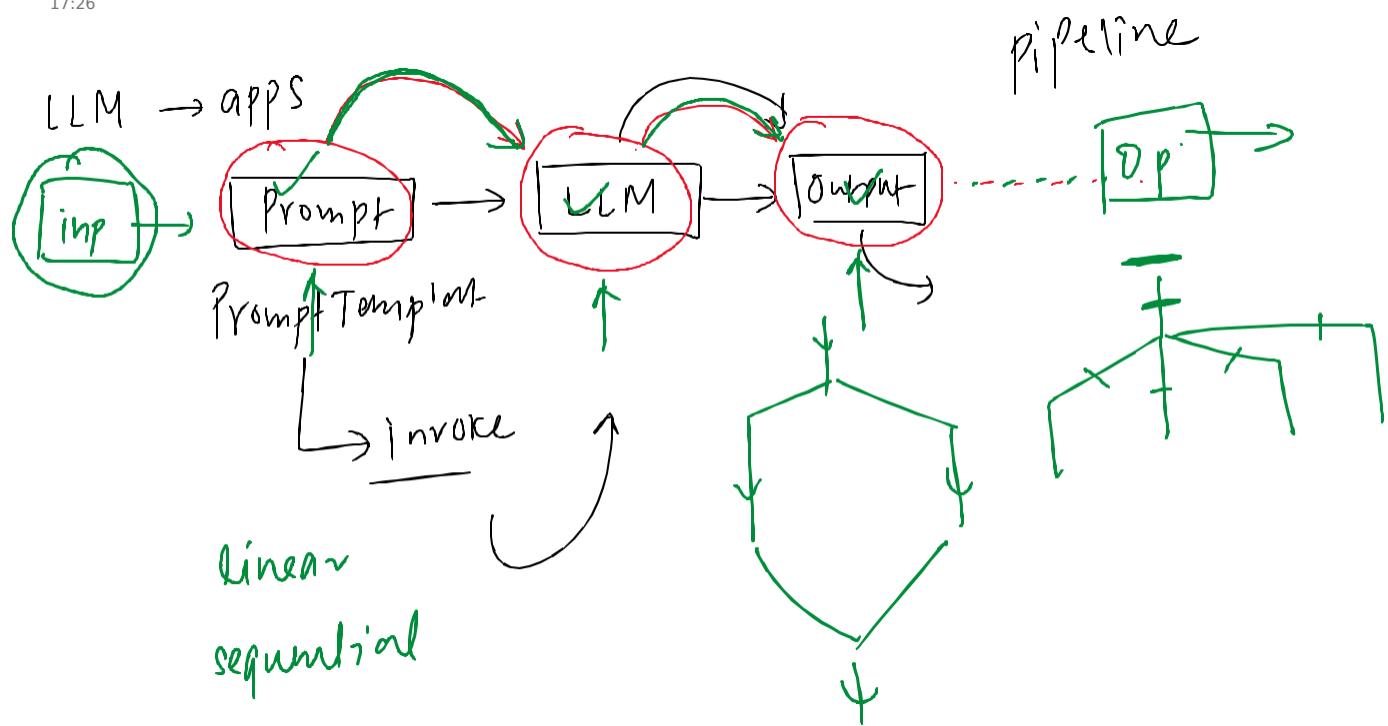
11 March 2025 17:26



7. Chains Page 56

What & Why

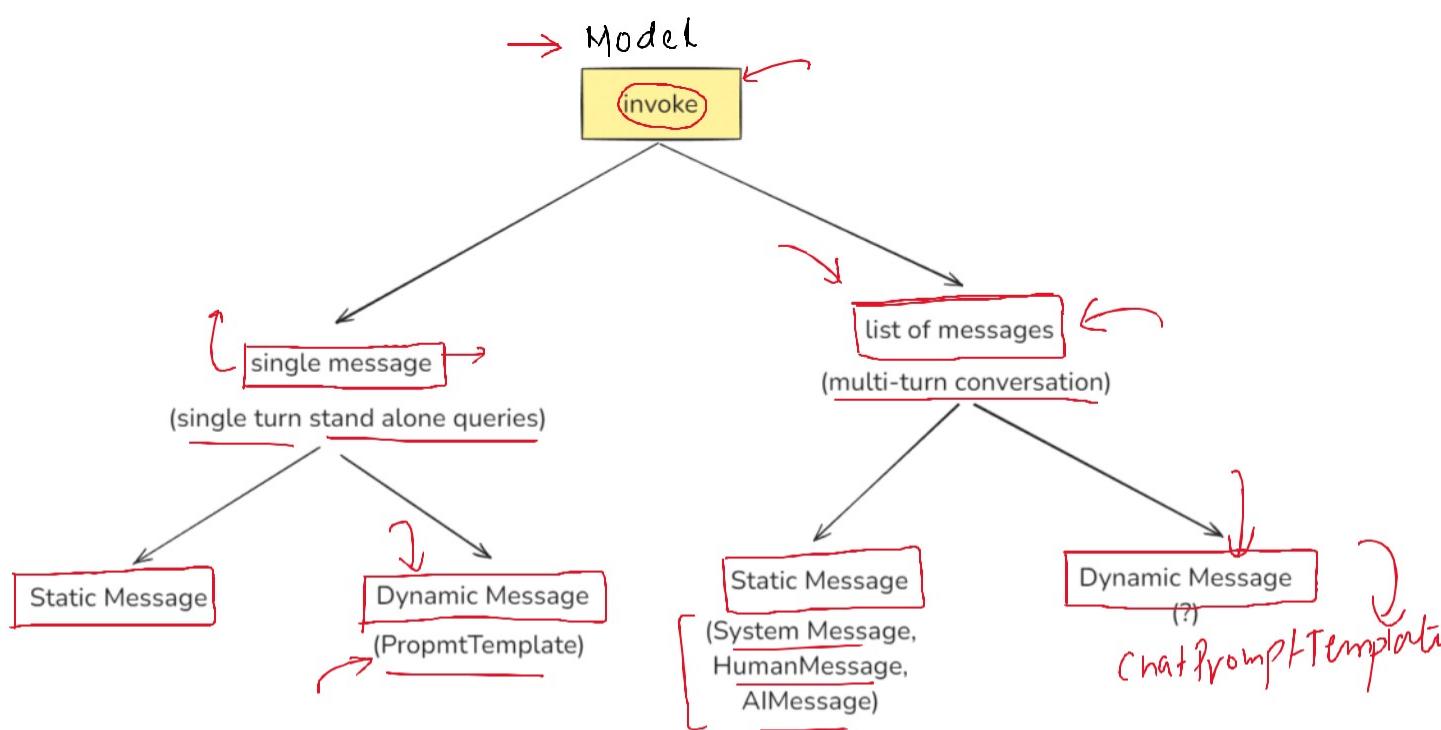
11 March 2025 17:26



simple chains

Simple Chain

11 17:
March 26
2025



System Message → dynamic

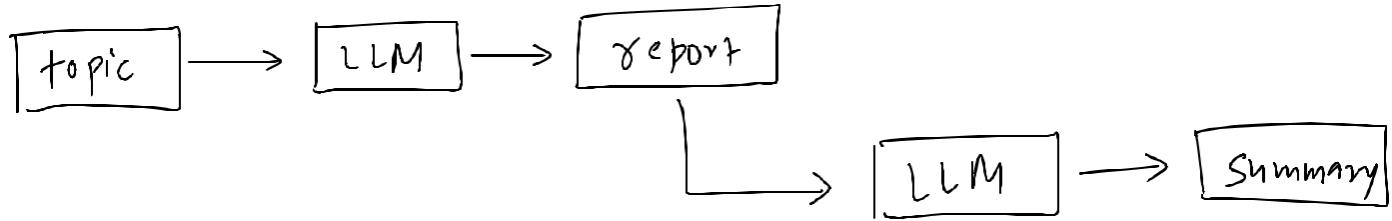
You are a helpful domain expert

Humanme

explain about topic

Sequential Chain

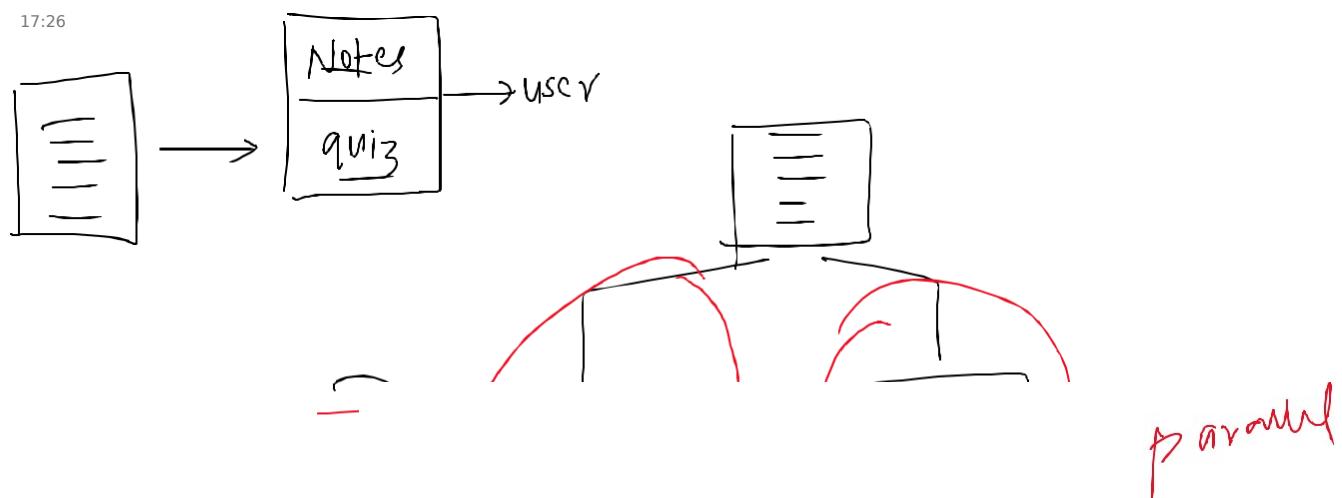
11 March 17:26
2025



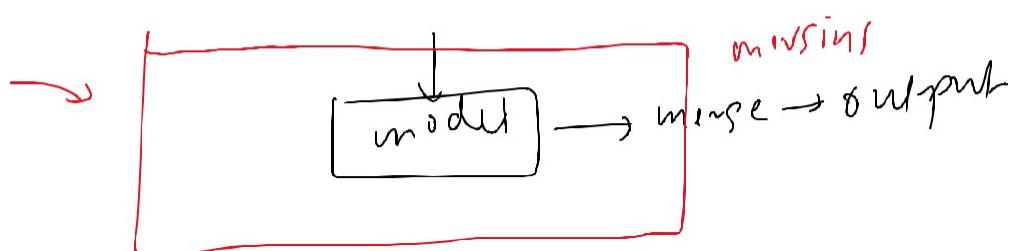
Parallel Chain

11 March 2025

17:26



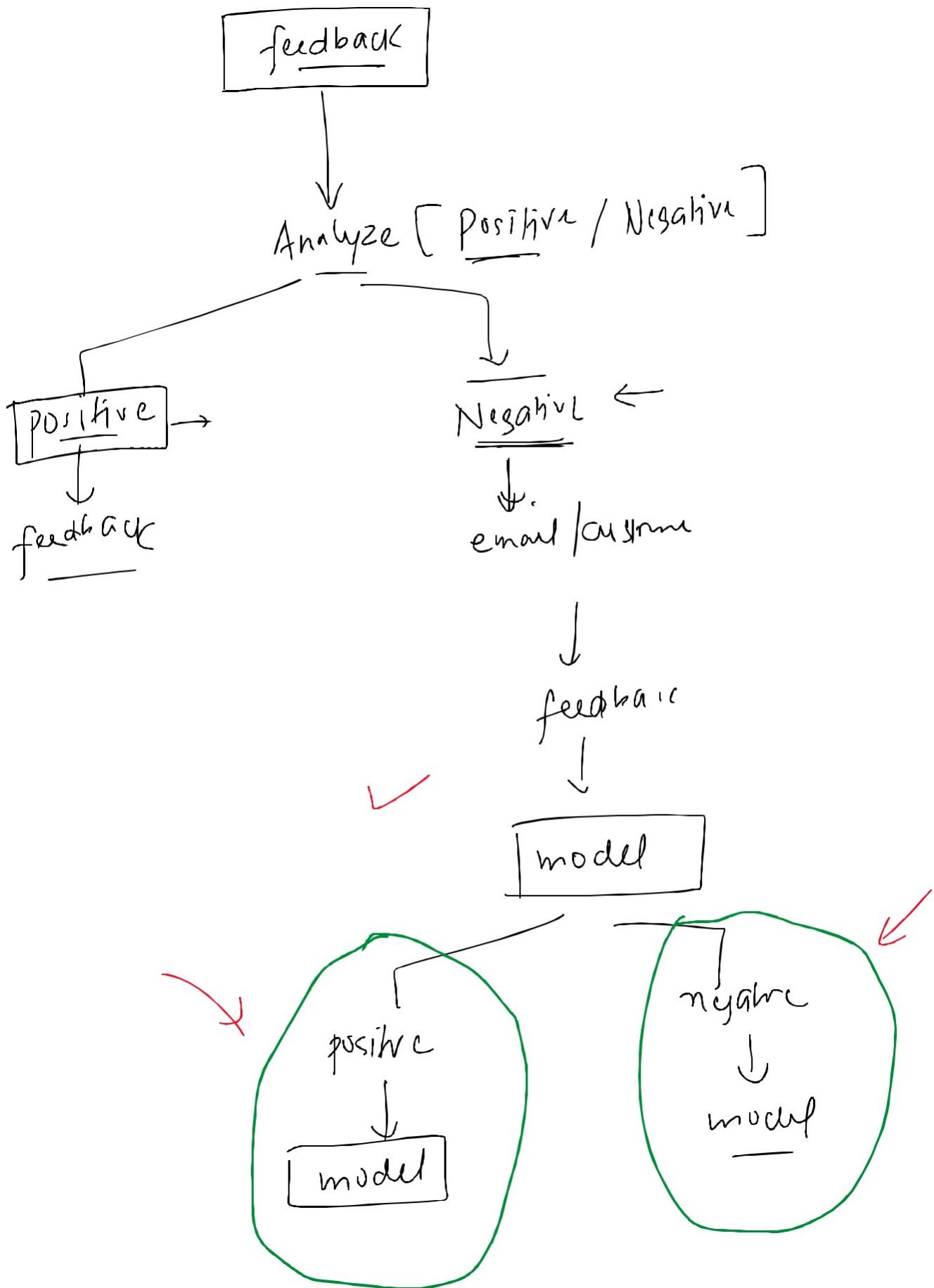
+



Conditional Chain

11 March 2025 17:27

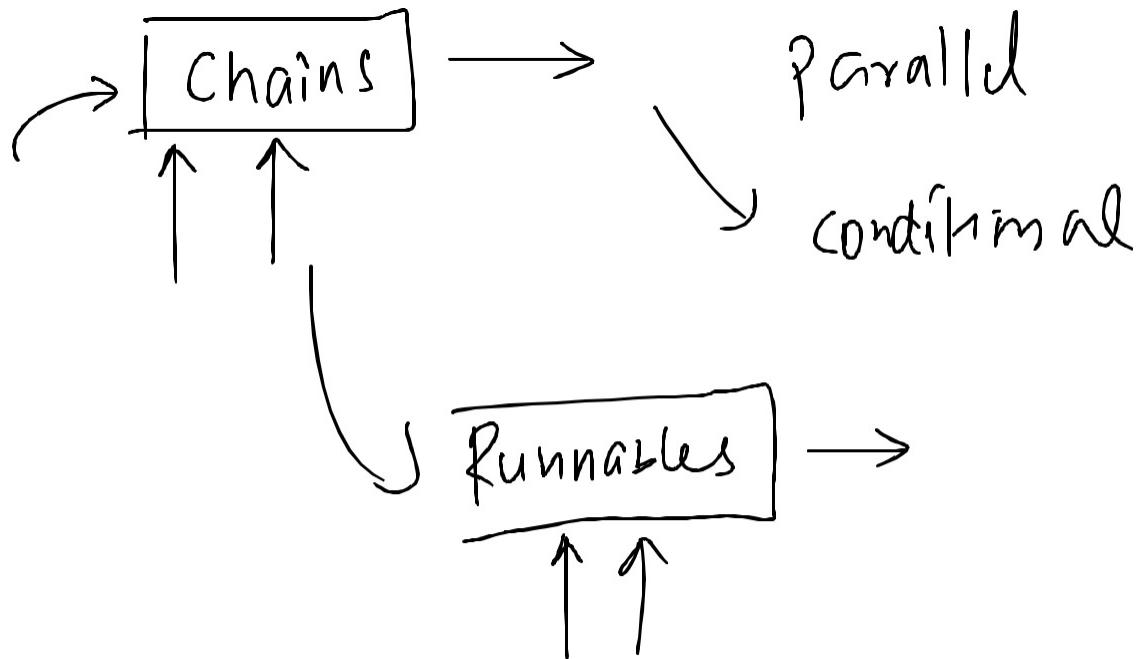
Agent



Recap

18
March
2025

16:
55

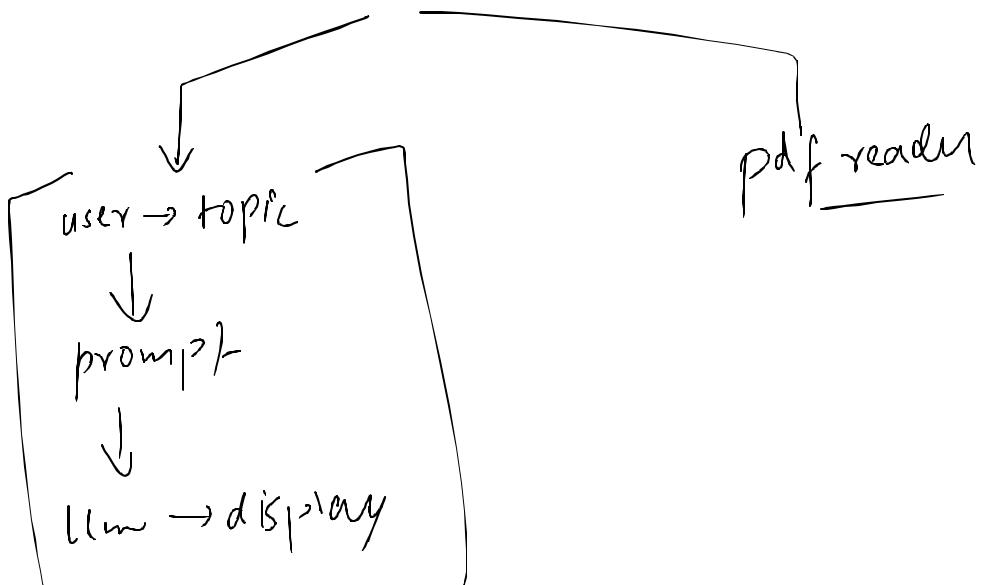
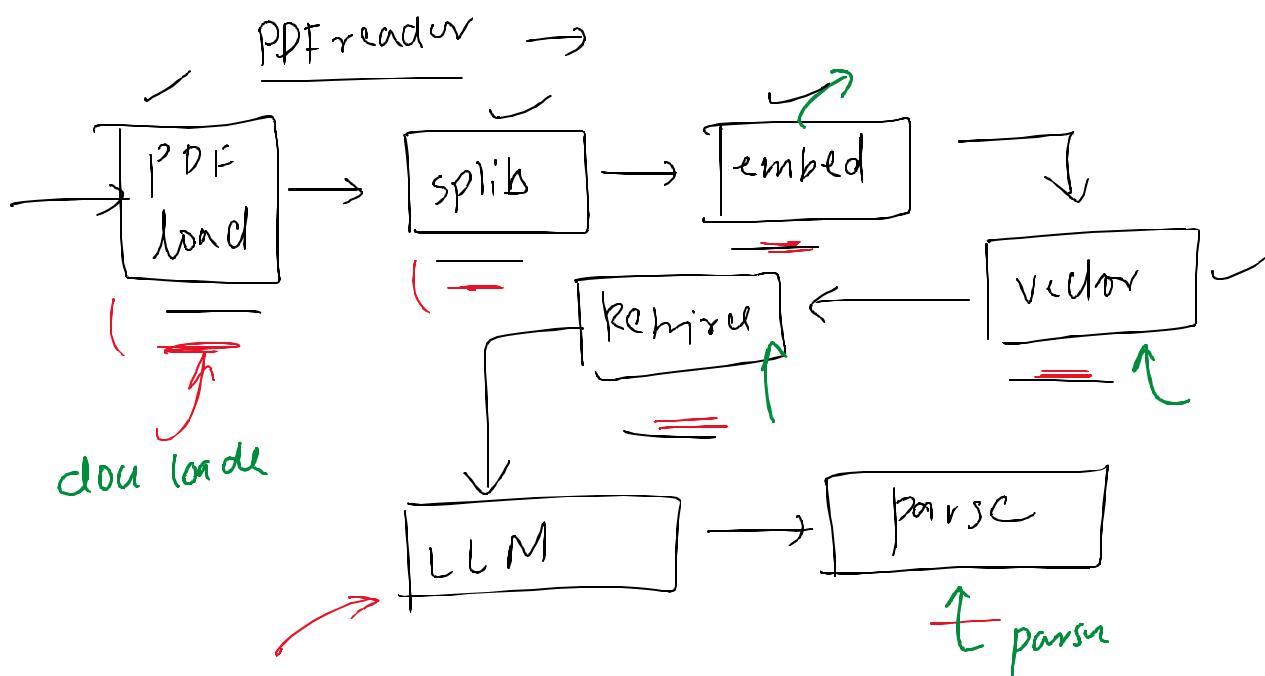
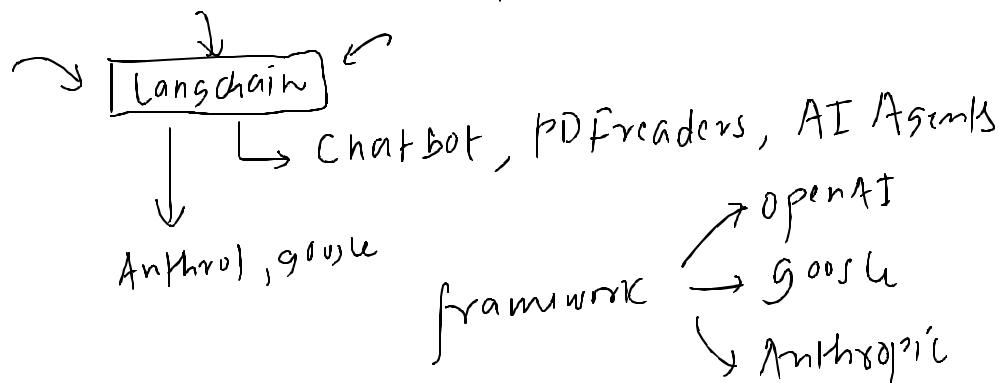


Runnables

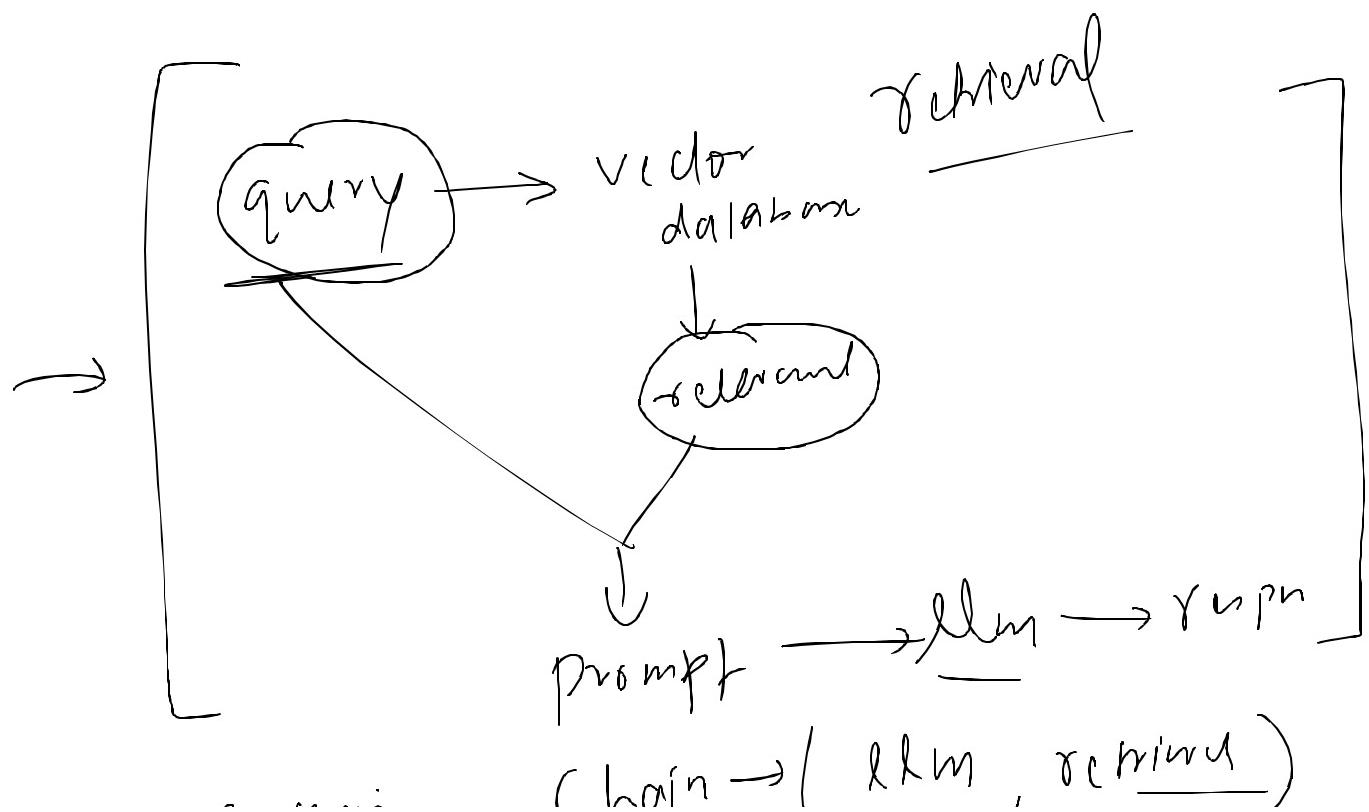
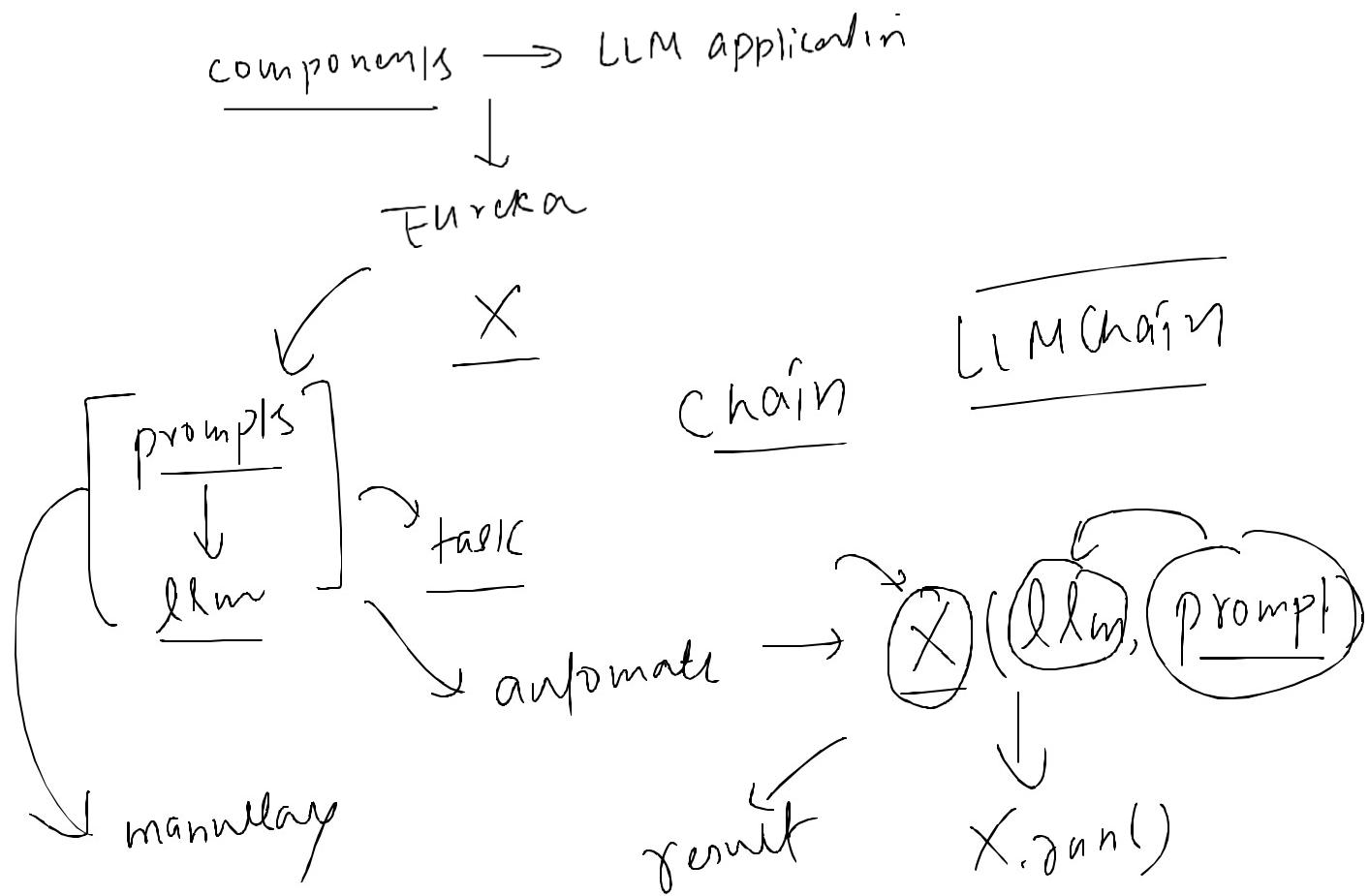
The Why 2022 → chatGPT → Nov

18 March 2025 16:59

API → LLM based App



$llm \rightarrow display$



Retrieve A Chain

Chain \rightarrow (LLM , retrieval)

LLM Chain

LLMChain \leftrightarrow LLMChain

Chain Name	Description
LLMChain	Basic chain that calls an LLM with a prompt template.
SequentialChain	Chains multiple LLM calls in a specific sequence.
SimpleSequentialChain	A simplified version of SequentialChain for easier use.
ConversationalRetrievalChain	Handles conversational Q&A with memory and retrieval.
RetrievalQA	Fetches relevant documents and uses an LLM for question-answering.
RouterChain	Directs user queries to different chains based on intent.
MultiPromptChain	Uses different prompts for different user intents dynamically.
HydeChain (Hypothetical Document Embeddings)	Generates hypothetical answers to improve document retrieval.
AgentExecutorChain	Orchestrates different tools and actions dynamically using an agent.
SQLDatabaseChain	Connects to SQL databases and answers natural language queries.



Problem

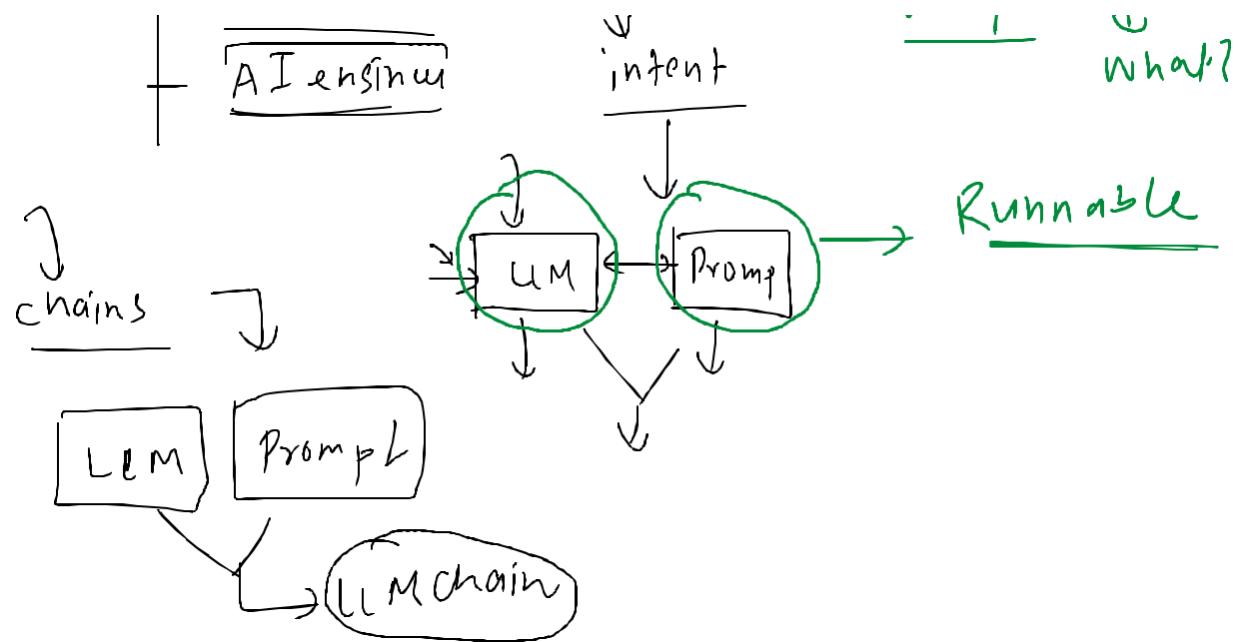
too many chains

+ codebase
+ AI engine

intent

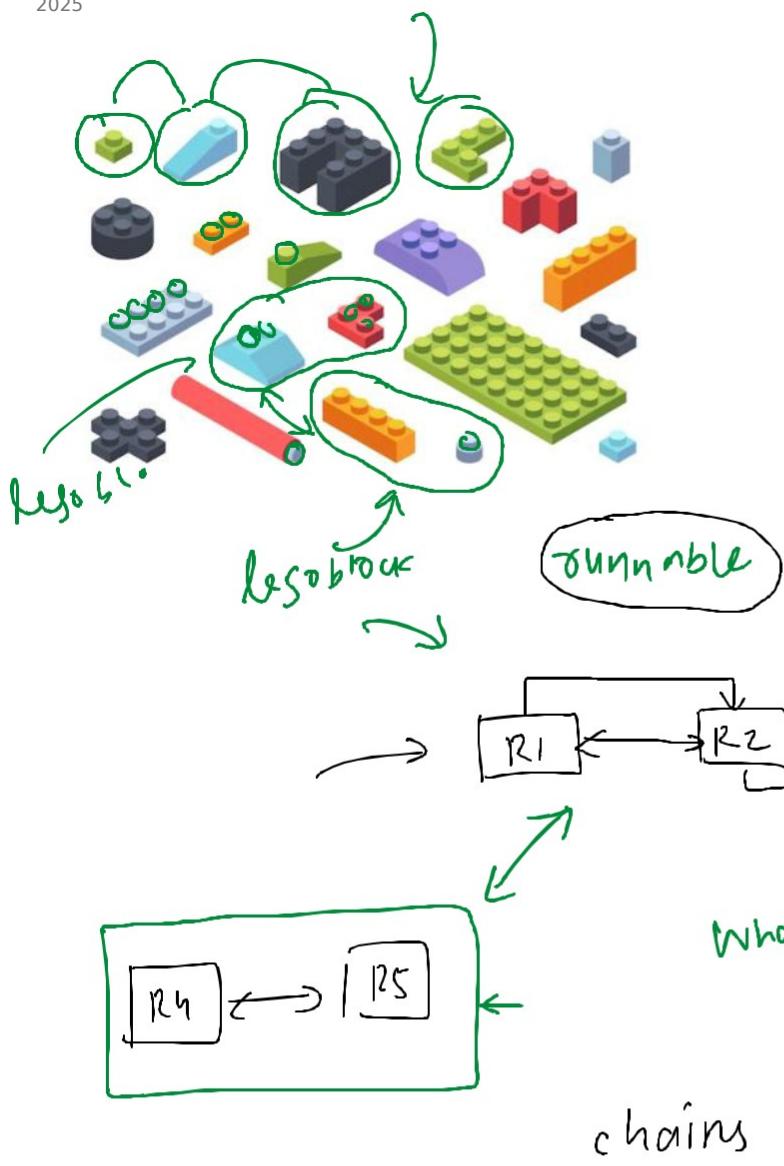
why?

what?



The What

19 March 2025 00:12



① unit of work

+ input
+ process
+ output

② common interface

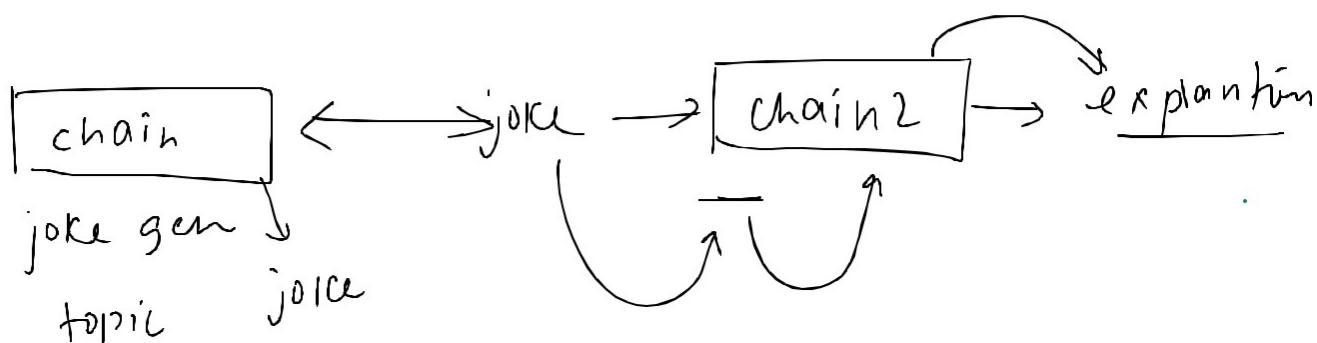
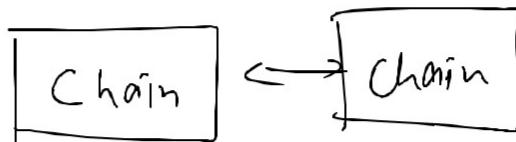
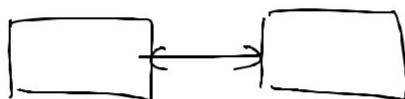
+ invoke() →
+ batch()
+ stream()

R3

③ connect

What → How

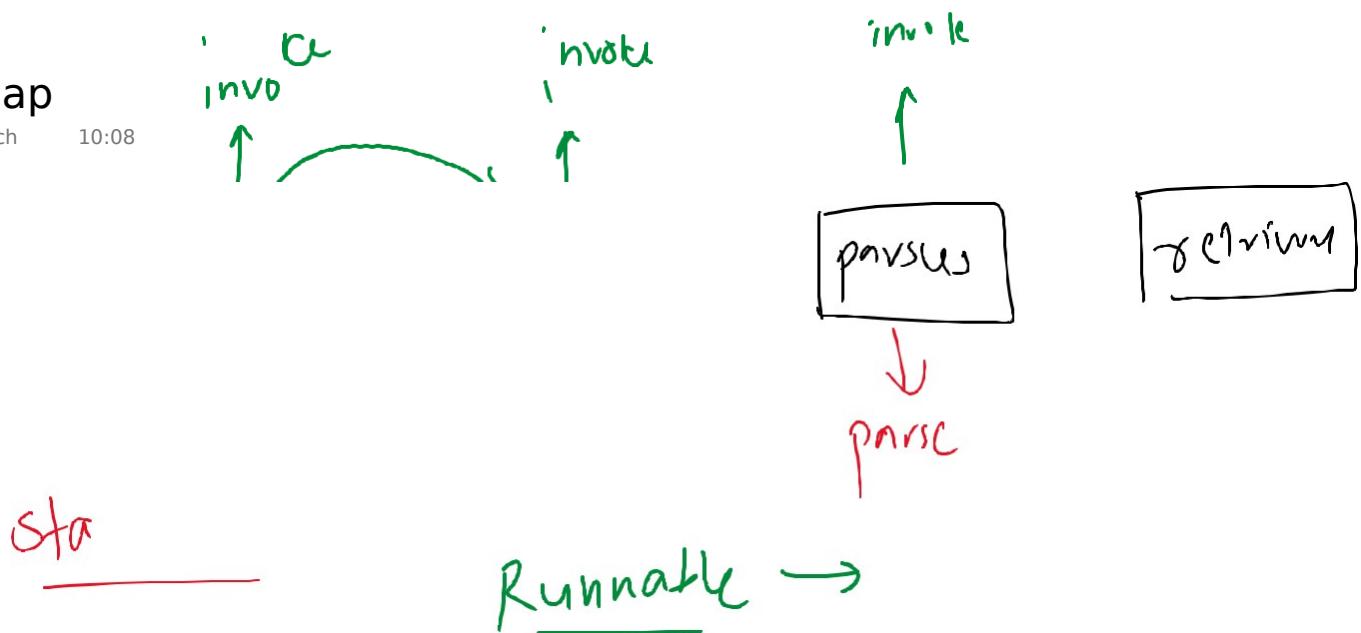
chains



Recap

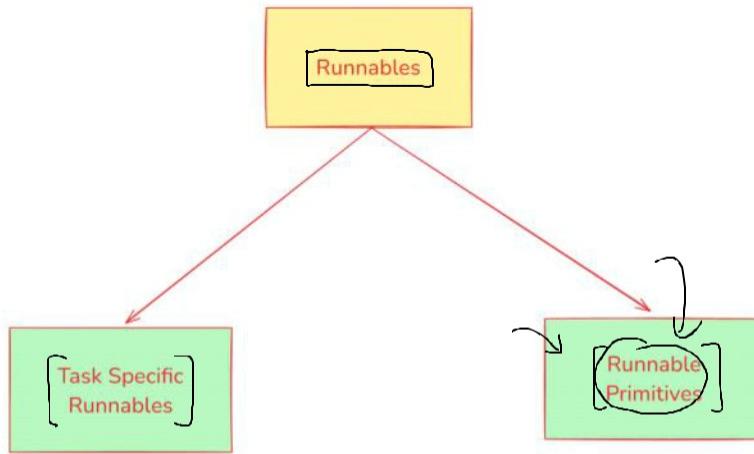
20 March
2025

10:08



Plan of Action

20 March 11:02
2025



- Definition:** These are core LangChain components that have been converted into Runnables so they can be used in pipelines.
- Purpose:** Perform task-specific operations like LLM calls, prompting, retrieval, etc.
- Examples:**
 - ChatOpenAI → Runs an LLM model.
 - PromptTemplate → Formats prompts dynamically.
 - Retriever → Retrieves relevant documents.

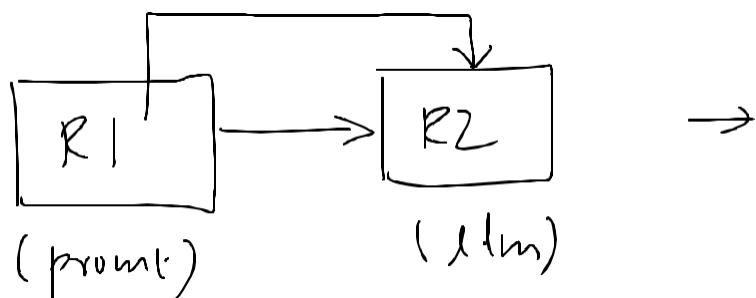
- Definition:** These are fundamental building blocks for structuring execution logic in AI workflows.
- Purpose:** They help orchestrate execution by defining how different Runnables interact (sequentially, in parallel, conditionally, etc.).
- Examples:**
 - RunnableSequence → Runs steps in order (| operator).
 - RunnableParallel → Runs multiple steps simultaneously.
 - RunnableMap → Maps the same input across multiple functions.
 - RunnableBranch → Implements conditional execution (if-else logic).
 - RunnableLambda → Wraps custom Python functions into Runnables.
 - RunnablePassthrough → Just forwards input as output (acts as a placeholder).

1. RunnableSequence

20 March 2025 12:10

RunnableSequence is a sequential chain of runnables in LangChain that executes each step one after another, passing the output of one step as the input to the next.

It is useful when you need to compose multiple runnables together in a structured workflow.



prompt ↗

juice

↓

use

9. LCEL Page 70

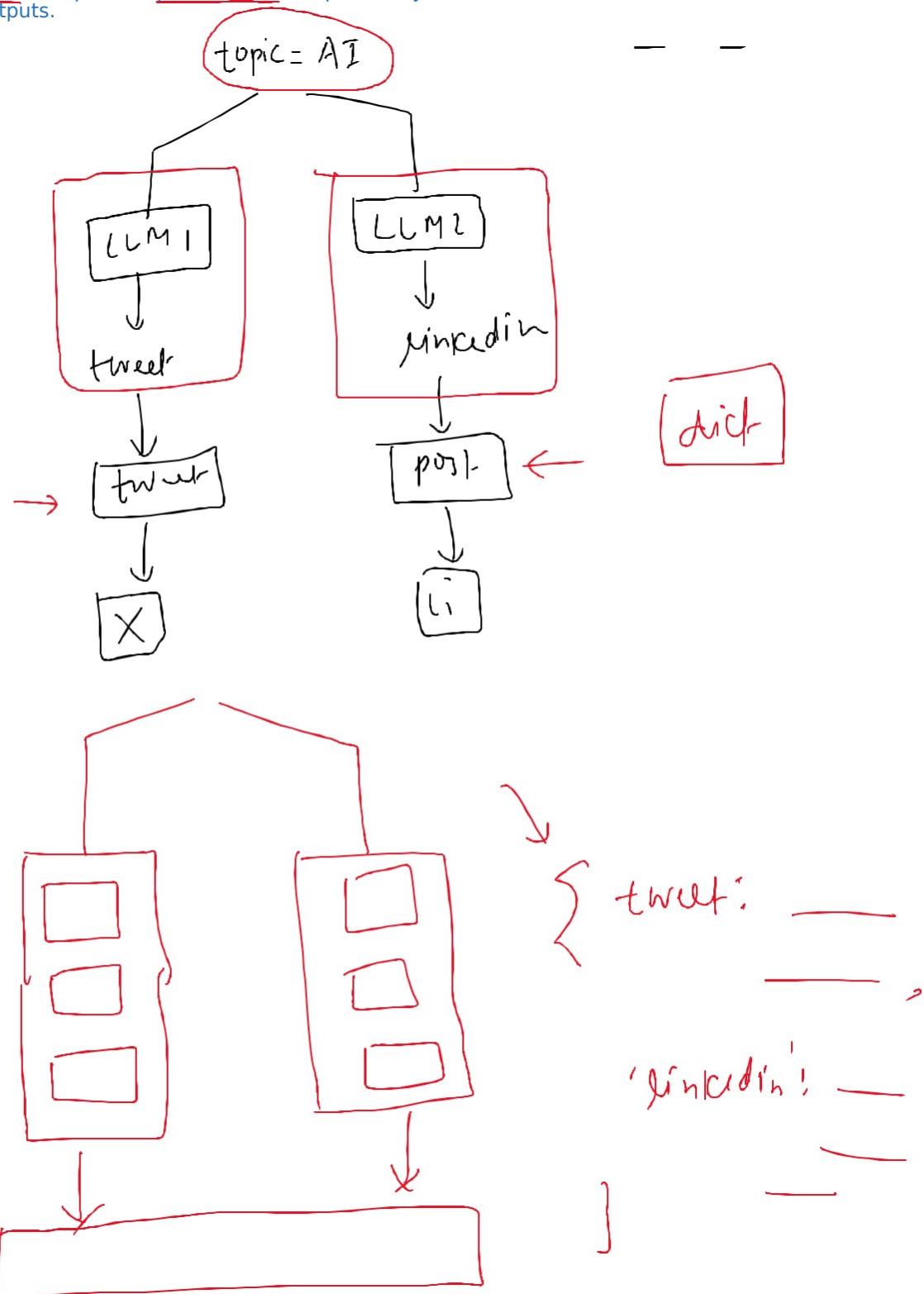
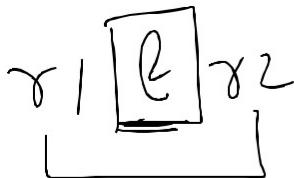
2. RunnableParallel

20 March 18:33
2025

RunnableParallel

RunnableParallel is a runnable primitive that allows multiple runnables to execute in parallel.

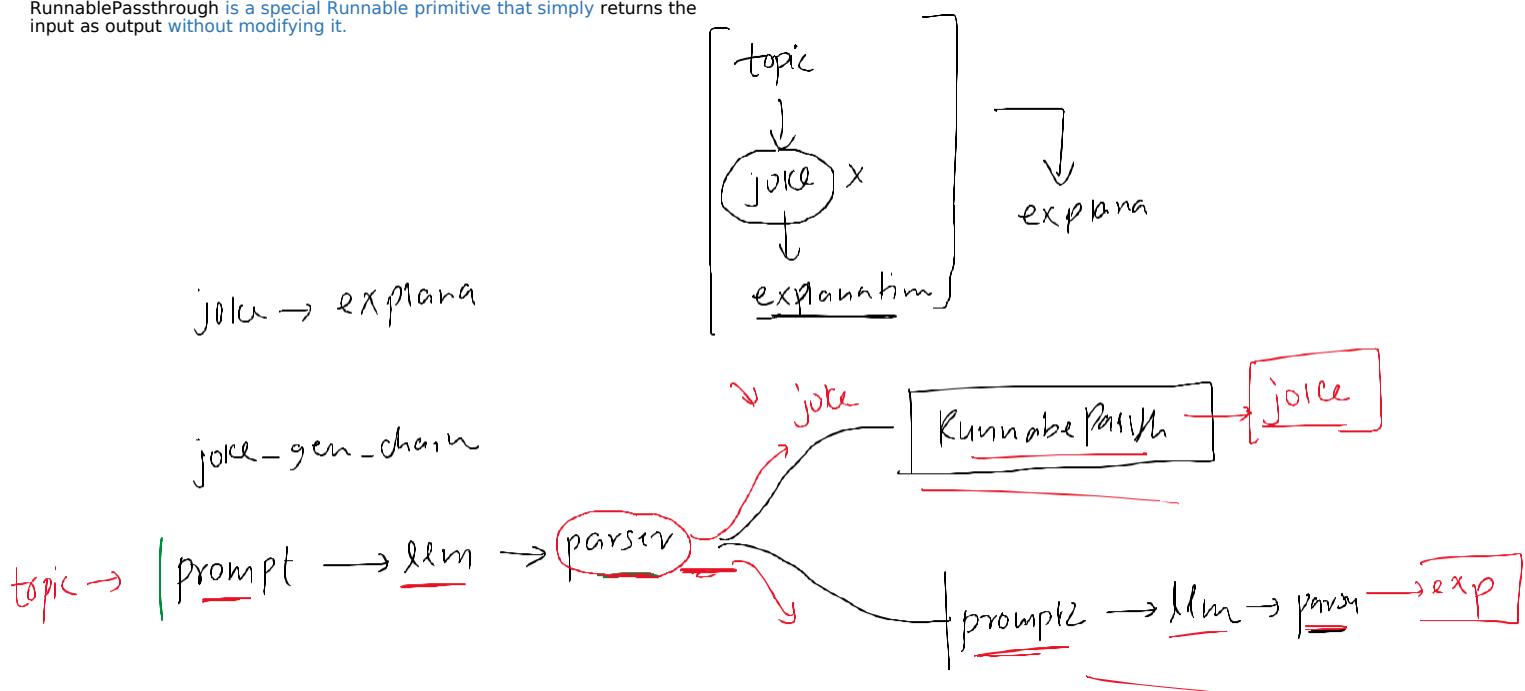
Each runnable receives the same input and processes it independently, producing a dictionary of outputs.



3. RunnablePassthrough

20 March 22:34
2025

RunnablePassthrough is a special Runnable primitive that simply returns the input as output without modifying it.



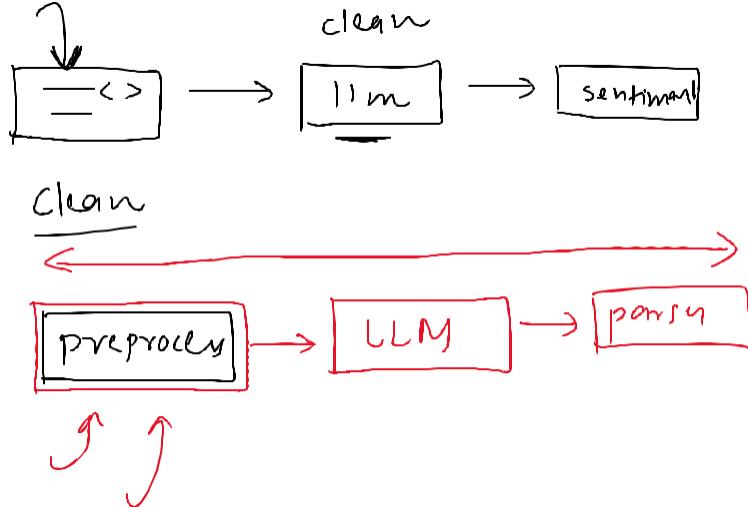


4. RunnableLambda

20 March 2025 23:18

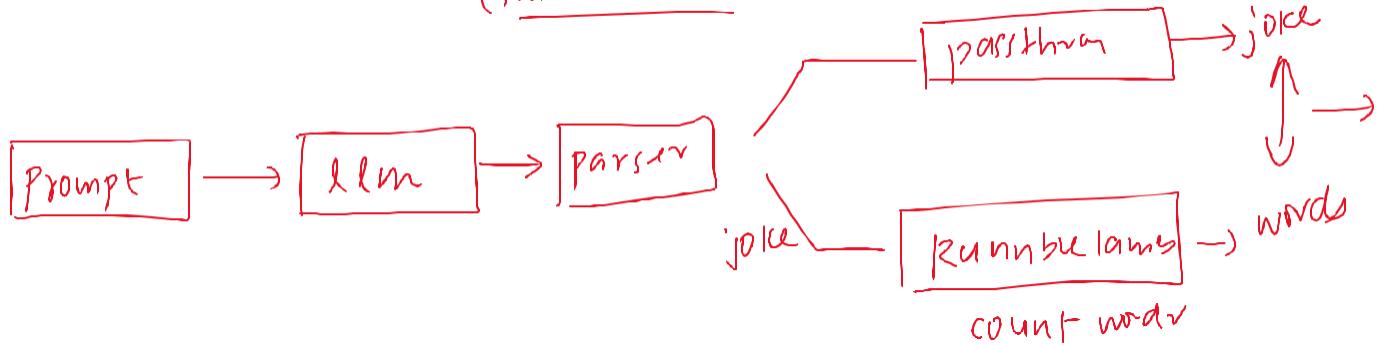
RunnableLambda is a runnable primitive that allows you to apply custom Python functions within an AI pipeline.

It acts as a middleware between different AI components, enabling preprocessing, transformation, API calls, filtering, and post-processing in a LangChain workflow.



joke → topic →

(Number → Words)

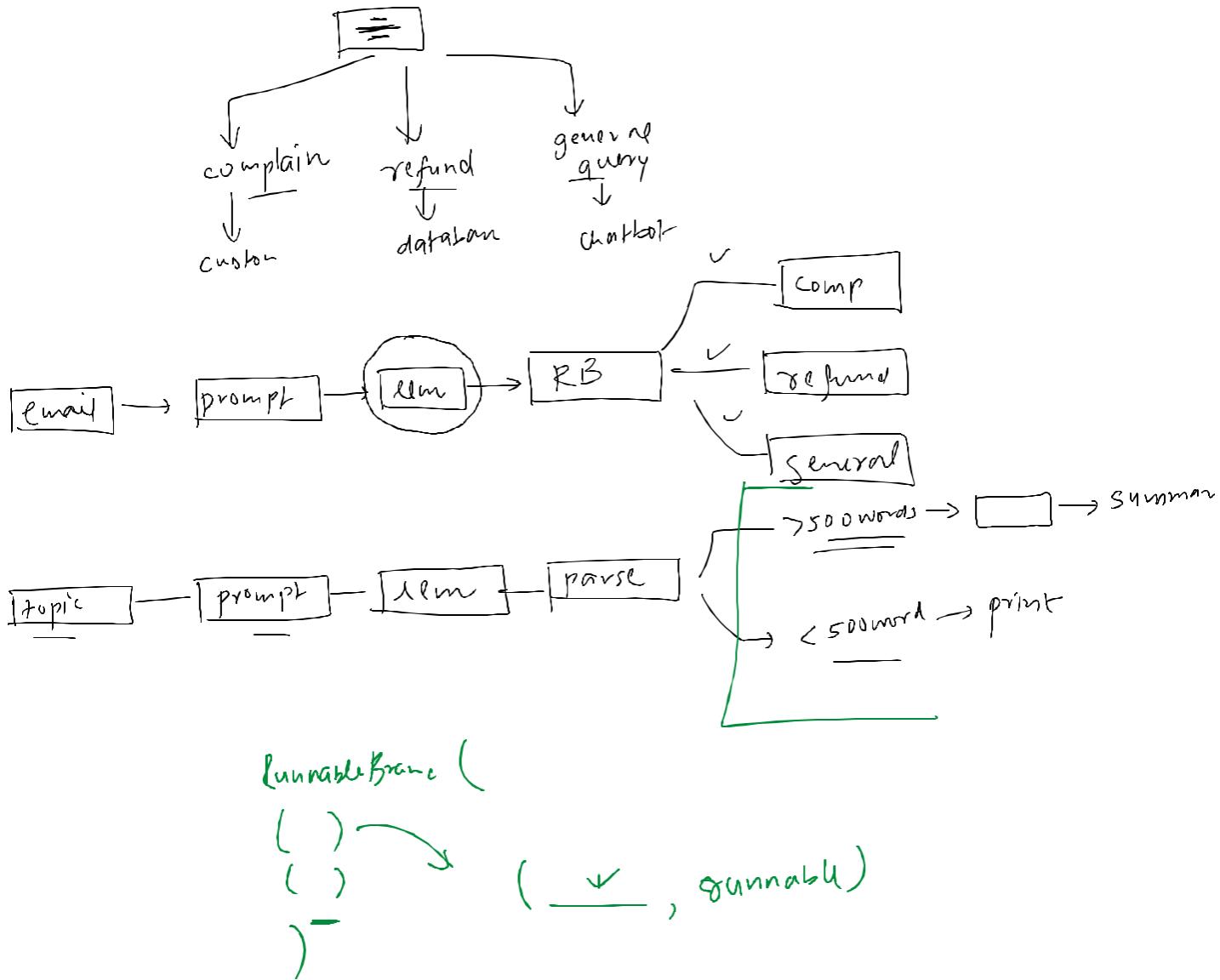


5. RunnableBranch

21 March 08:02
2025

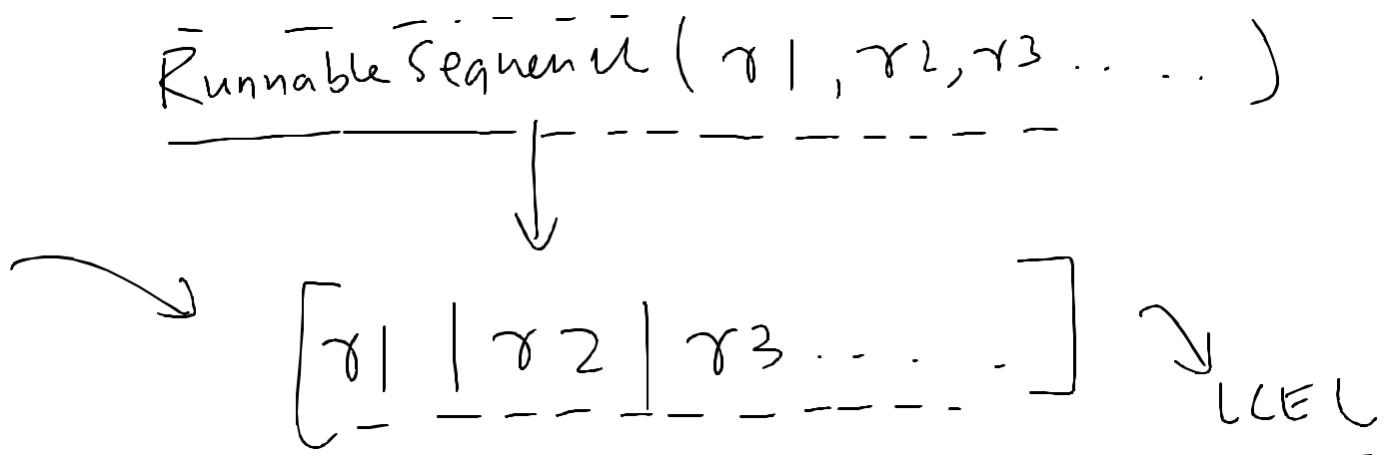
RunnableBranch is a control flow component in LangChain that allows you to conditionally route input data to different chains or runnables based on custom logic.

It functions like an if/elif/else block for chains — where you define a set of condition functions, each associated with a runnable (e.g., LLM call, prompt chain, or tool). The first matching condition is executed. If no condition matches, a default runnable is used (if provided).



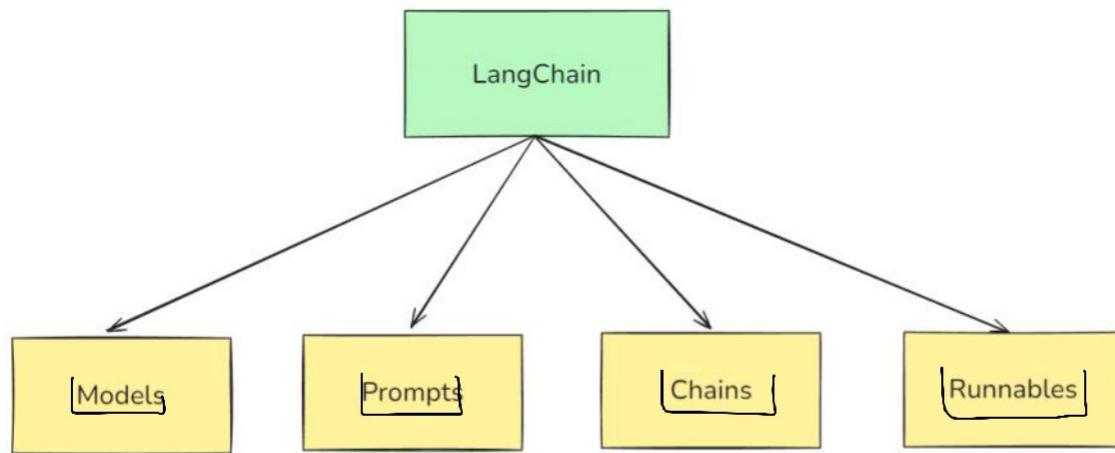
LCEL

21 March 08:39
2025



RAG

27 March 2025 10:59



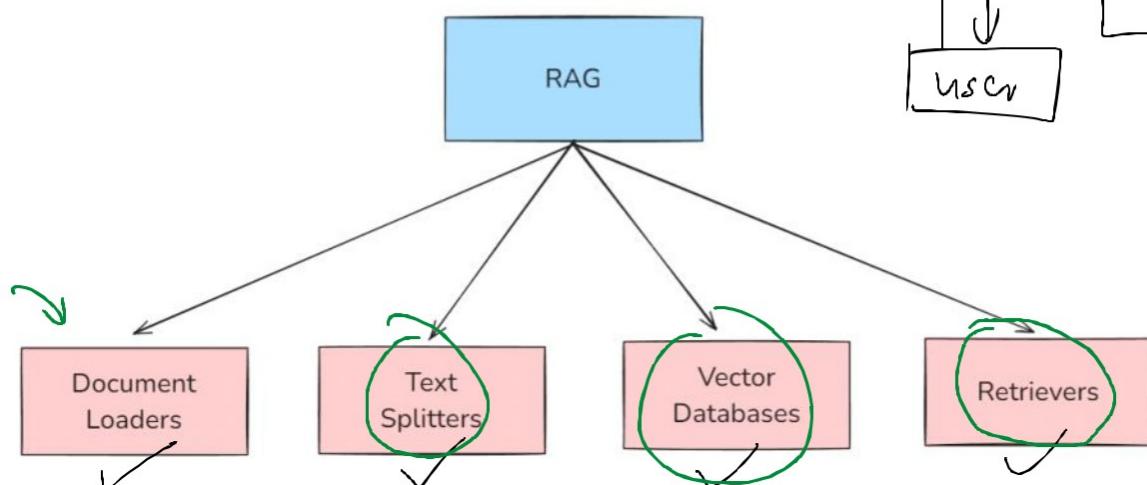
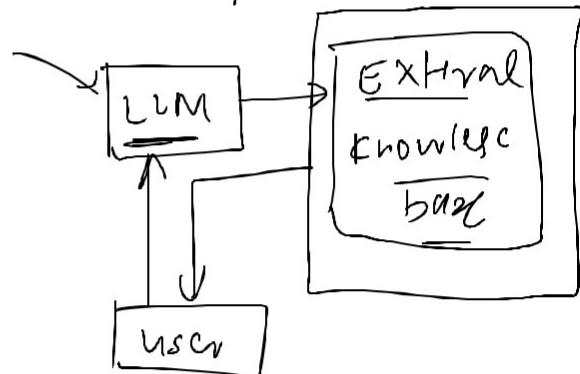
RAG is a technique that combines information retrieval with language generation, where a model retrieves relevant documents from a knowledge base and then uses them as context to generate accurate and grounded responses.

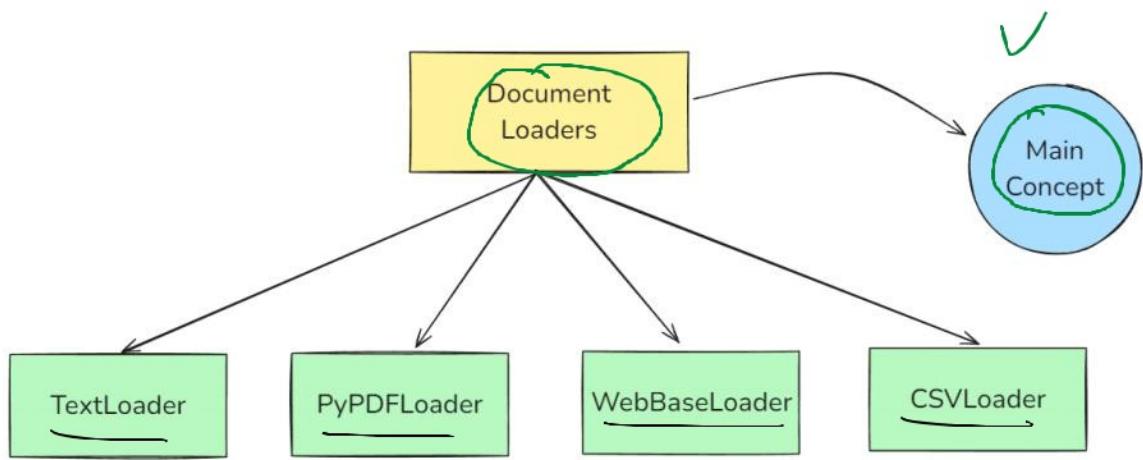
Benefits of using RAG

1. Use of up to date information
2. Better privacy
3. No limit of document size

Chatbots → ChatGPT

→ current affairs
→ personal data



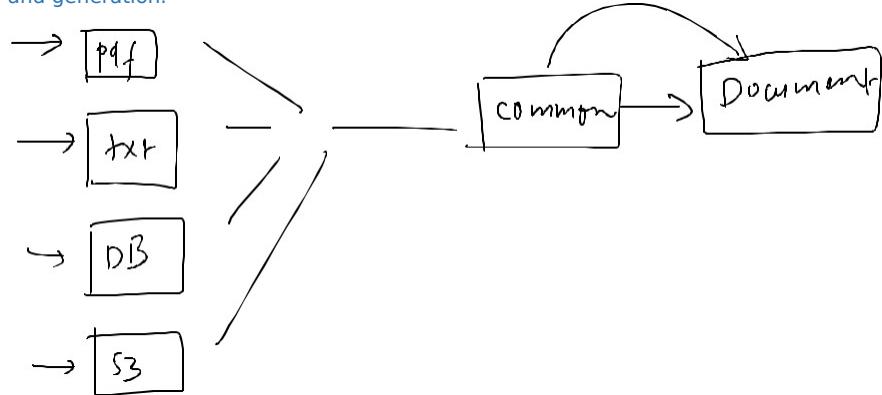


Document Loaders in LangChain

27 March 16:20
2025

Document loaders are components in LangChain used to load data from various sources into a standardized format (usually as Document objects), which can then be used for chunking, embedding, retrieval, and generation.

```
Document(  
  page_content="The actual text content",  
  metadata={"source": "filename.pdf", ...}  
)
```



TextLoader

27 March 16:50
2025

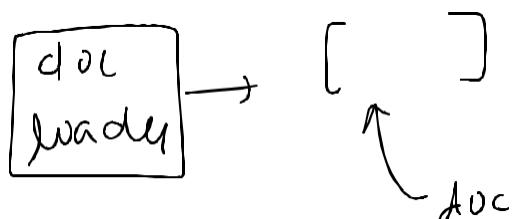
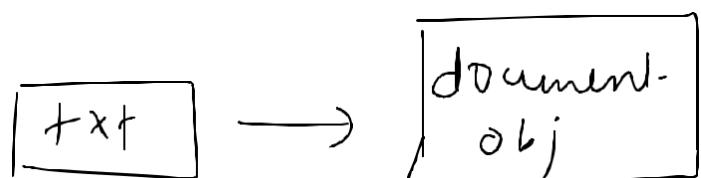
TextLoader is a simple and commonly used document loader in LangChain that reads plain text (.txt) files and converts them into LangChain Document objects.

Use Case

- Ideal for loading chat logs, scraped text, transcripts, code snippets, or any plain text data into a LangChain pipeline.

Limitation

- Works only with .txt files



10.Document Loaders Page 79

PyPDFLoader → text
 27 March 17:50 2025 → Scanned pdf

PyPDFLoader is a document loader in LangChain used to load content from PDF files and convert each page into a Document object.

```
↓ [ Document(page_content="Text from page 1", metadata={"page": 0, "source": "file.pdf"}),  

  Document(page_content="Text from page 2", metadata={"page": 1, "source": "file.pdf"}),  

  ...  

]
```

Limitations:

- It uses the PyPDF library under the hood — not great with scanned PDFs or complex layouts.

25 → pypdf
 ↓
 25 document

[—, —, —, —]

Use Case	Recommended Loader
Simple, clean PDFs	PyPDFLoader
PDFs with tables/columns	PDFPlumberLoader
Scanned/image PDFs	UnstructuredPDFLoader OR AmazonTextractPDFLoader
Need layout and image data	PyMuPDFLoader
Want best structure extraction	UnstructuredPDFLoader

DirectoryLoader

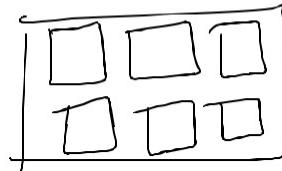
27 March 2025 18:44

DirectoryLoader is a document loader that lets you load multiple documents from a directory (folder) of files.

→ lanschain

Glob Pattern	What It Loads
"**/*.txt"	All <u>.txt</u> files in all subfolders
"*.pdf"	All <u>.pdf</u> files in the root directory
"data/*.csv"	All <u>.csv</u> files in the <u>data/</u> folder
"**/*"	All files (any type, all folders)

** = recursive search through subfolders





Load vs Lazy load

27 March 2025 Load() 23:51

- Eager Loading (loads everything at once).
- Returns: A list of Document objects.
- Loads all documents immediately into memory.
- Best when:
 - The number of documents is small.
 - You want everything loaded upfront.

→ gen
of docs

.LAZY LOAD()

- Lazy Loading (loads on demand).
- Returns: A generator of Document objects.
- Documents are not all loaded at once; they're fetched one at a time as needed.
- Best when:
 - You're dealing with large documents or lots of files.
 - You want to stream processing (e.g., chunking, embedding) without using lots of memory.

WebBaseLoader →

28 March 00:34
2025

WebBaseLoader is a document loader in LangChain used to load and extract text content from web pages (URLs).

It uses BeautifulSoup under the hood to parse HTML and extract visible text.

When to Use:

- For blogs, news articles, or public websites where the content is primarily text-based and static.

Limitations:

- Doesn't handle JavaScript-heavy pages well (use SeleniumURLLoader for that).
- Loads only static content (what's in the HTML, not what loads after the page renders).

10.Document Loaders Page 83

CSVLoader

28 01:
March 48
2025

CSVLoader [is a document loader used to load CSV files into LangChain Document objects — one per row, by default.](#)

10.Document Loaders Page 84

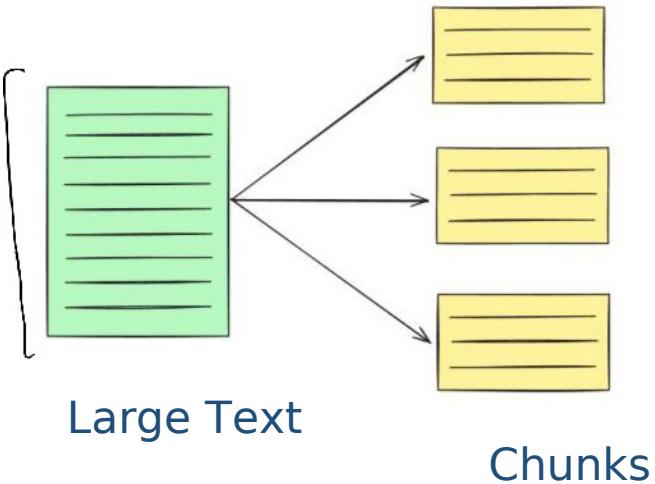
Other Document Loaders

28 01:
March 58
2025

Text Splitting

01 April 2025 18:10

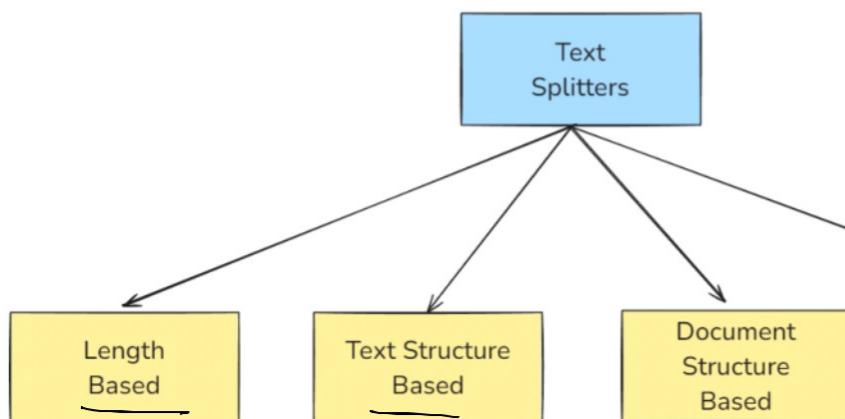
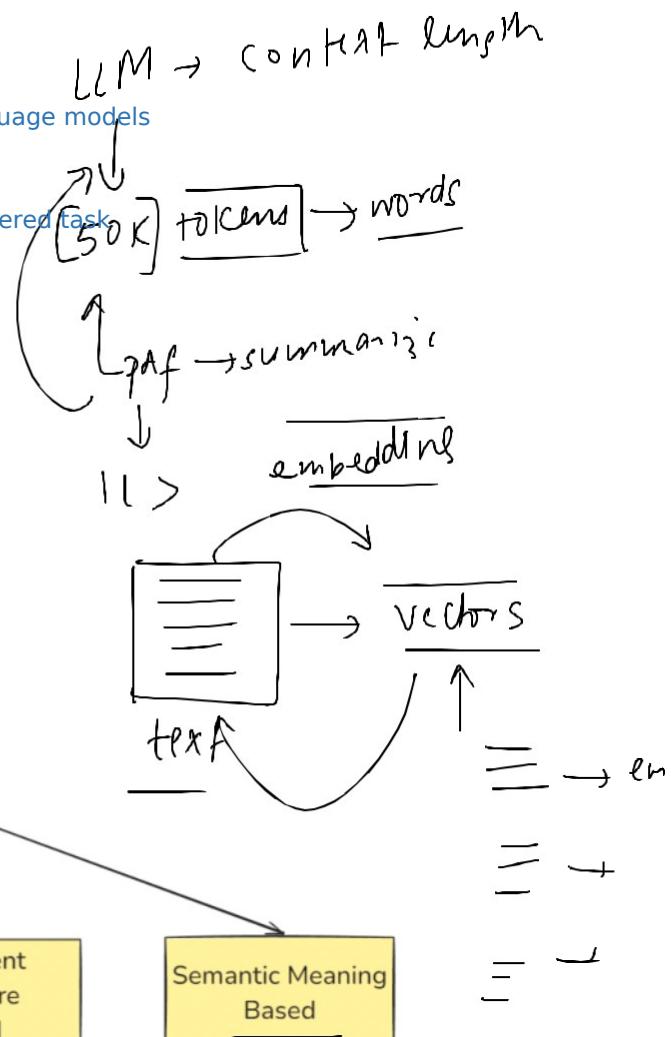
Text Splitting is the process of breaking large chunks of text (like articles, PDFs, HTML pages, or books) into smaller, manageable pieces (chunks) that an LLM can handle effectively.



- Overcoming model limitations: Many embedding models and language models have maximum input size constraints. Splitting allows us to process documents that would otherwise exceed these limits.
- Downstream tasks - Text Splitting improves nearly every LLM powered task

Task	Why Splitting Helps
Embedding	Short chunks yield more accurate vectors
Semantic Search	Search results point to focused info, not noise
Summarization	Prevents hallucination and topic drift

- Optimizing computational resources: Working with smaller chunks of text can be more memory-efficient and allow for better parallelization of processing tasks.

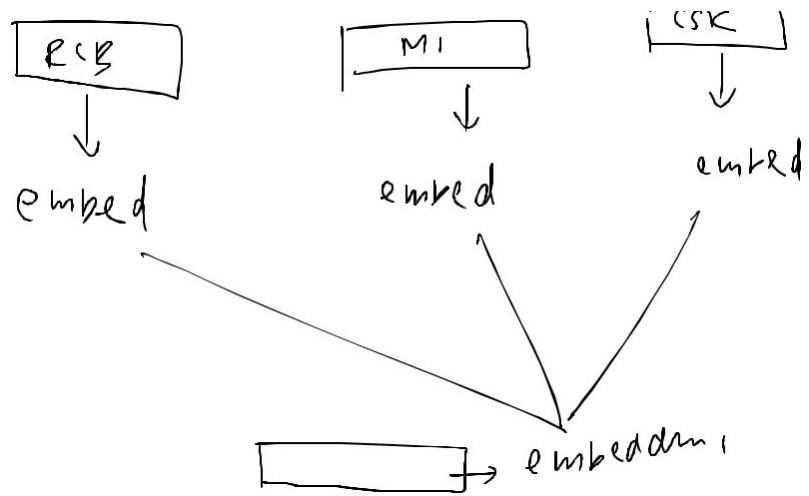


RCB

MI

CSK

11. Text Splitters Page 86



1. Length Based Text Splitting

01 April 2025 18:10

Space exploration has led to incredible scientific discoveries. From landing on the Moon to exploring Mars, humanity continues to push the boundaries of what's possible beyond our planet. These missions have not only expanded our knowledge of the universe but have also contributed to advancements in technology here on Earth. Satellite communications, GPS, and even certain medical imaging techniques trace their roots back to innovations driven by space programs.

Space exploration has led to incredible scientific discoveries. From landing on the Moon to exploring

g Mars, humanity continues to push the boundaries of what's possible beyond our planet. These missi

ons have not only expanded our knowledge of the universe but have also contributed to advancements in

n technology here on Earth. Satellite communications, GPS, and even certain medical imaging techniqu

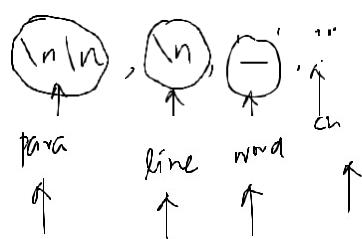
es trace their roots back to innovations driven by space programs.

2. Text-Structured Based

01 April 2025 18:10

My name is Nitish
I am 35 years old

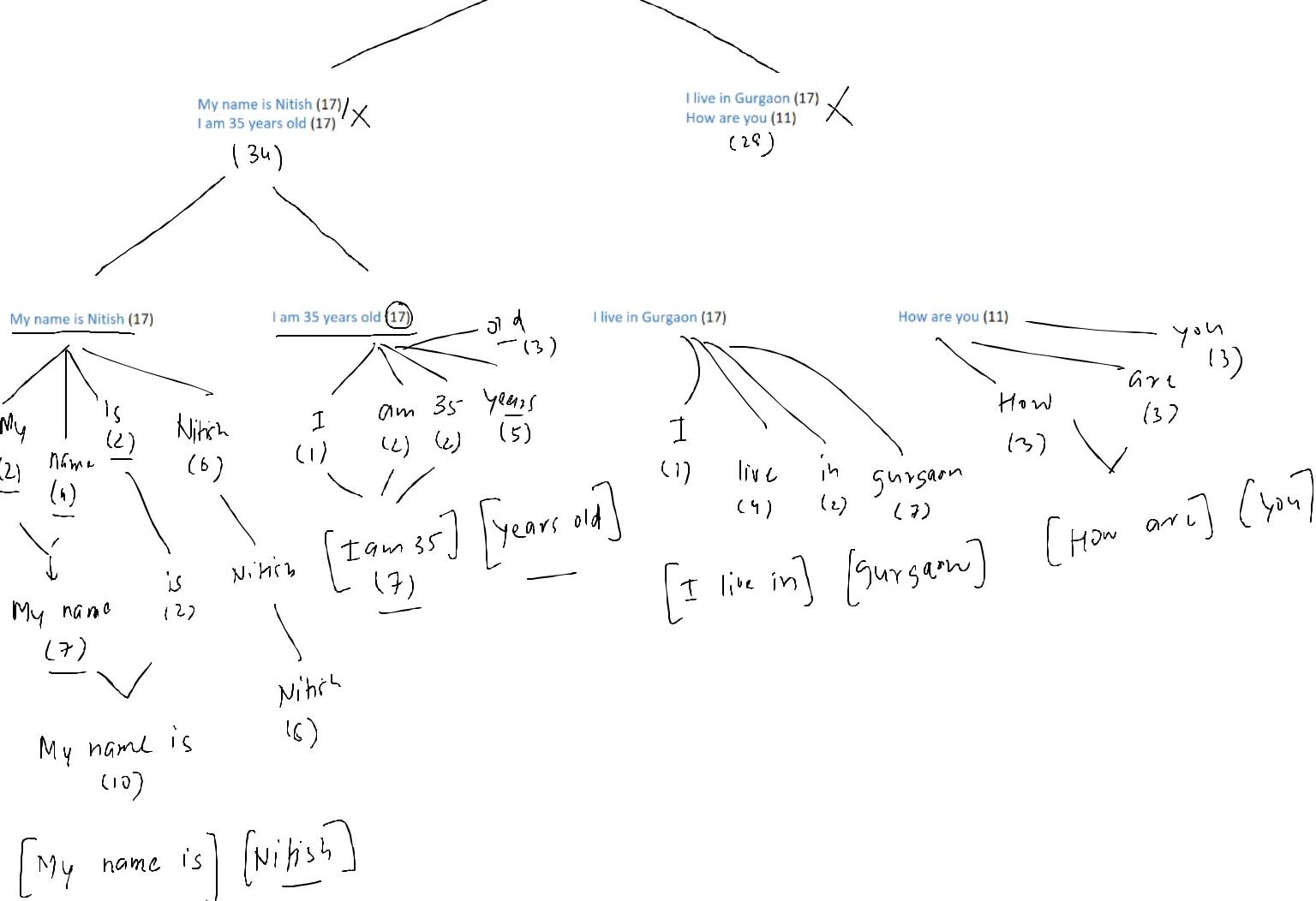
I live in Gurgaon
How are you



Allowed
[chunk_size = 10]

chunks
chunk_size = 10

[My name is Nitish (17)
I am 35 years old (17)
I live in Gurgaon (17)
How are you (11)]



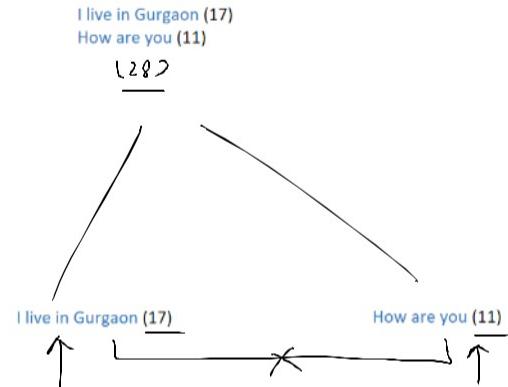
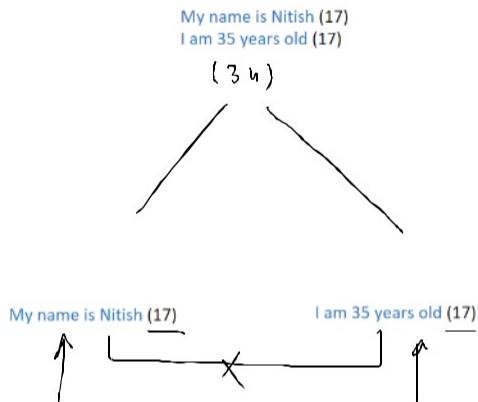
chunk size = 25

My name is Nitish (17)
I am 35 years old (17)

chunk size = 25

My name is Nitish (17)
I am 35 years old (17)

I live in Gurgaon (17)
How are you (11)



chunk size = 50

My name is Nitish (17)
I am 35 years old (17)

I live in Gurgaon (17)
How are you (11)

My name is Nitish (17)
I am 35 years old (17)

I live in Gurgaon (17)
How are you (11)

(34)

(28)

3. Document-Structured Based

01 April
2025 18:11

markdown \rightarrow HTML

```
# Project Name: Smart Student Tracker

A simple Python-based project to manage and track student data,
---  
## 🔎 Features
- Add new students with relevant info
- View student details
- Check if a student is passing
- Easily extendable class-based design
---  
## 💾 Tech Stack
- Python 3.10+
- No external dependencies
```

```
# First, try to split along Markdown headings (starting with level 2)
"\n#{1,6} ",
# Note the alternative syntax for headings (below) is not handled here
# Heading level 2
# -----
# End of code block
"```\n",
# Horizontal lines
"\n\\*\\*\\*\\*+\n",
"\n---+\n",
"\n___+\n",
# Note that this splitter doesn't handle horizontal lines defined
# by *three or more* of ***, ---, or ___, but this is not handled
"\n\n",
"\n",
" ",
"";
```

~~coll~~ →

```
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade # Grade is a float (like 8.5 or 9.2)

    def get_details(self):
        return f"Name: {self.name}, Age: {self.age}, Grade: {self.grade}"

    def is_passing(self):
        return self.grade >= 6.0
```

```
# Example usage
student1 = Student("Aarav", 20, 8.2)
print(student1.get_details())

if student1.is_passing():
    print("The student is passing.")
else:
    print("The student is not passing")
```

```
# First, try to split along class definitions
"\nclass ", —
"\ndef ", —
"\n\tdef ", —
# Now split by the normal type of lines
"\n\n", —
"\n", —
" ", —
" ", —
```


4. Semantic Meaning Based

01 April 2025

18:11

2 chunks

w)

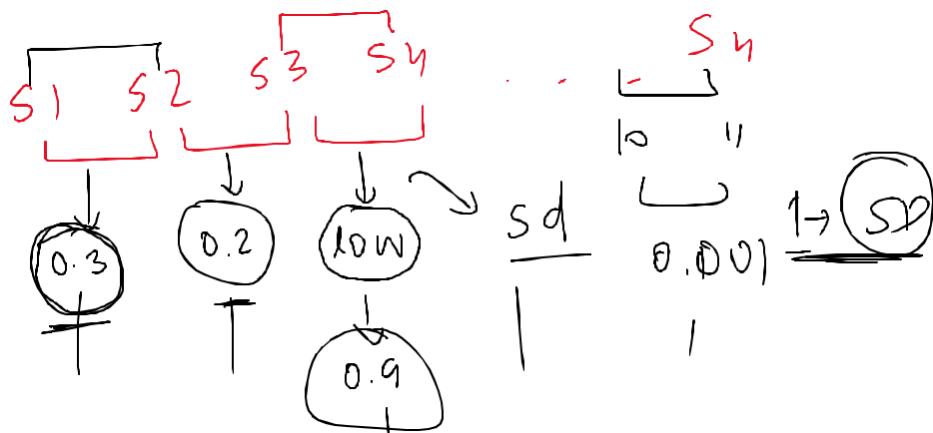
(le (structure)



Farmers were working hard in the fields, preparing the soil and planting seeds for the next season. The sun was bright, and the air smelled of earth and fresh grass. The Indian Premier League (IPL) is the biggest cricket league in the world. People all over the world watch the matches and cheer for their favourite teams.)

2 para x

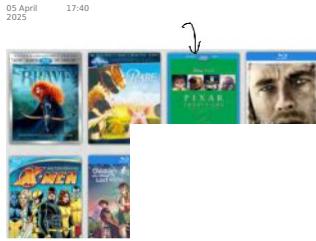
Terrorism is a big danger to peace and safety. It causes harm to people and creates fear in cities and villages. When such attacks happen, they leave behind pain and sadness. To fight terrorism, we need strong laws, alert security forces, and support from people who care about peace and safety.



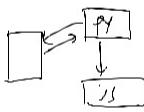
11. Text Splitters Page 92

Why Vector Stores?

05 April 2025 17:40

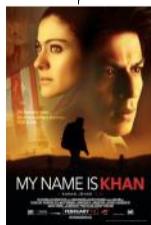


Movie id	Movie name	Director	Actor	Genre	Release Date	Outcome
M001	3 Idiots	Raju Hirani	Aamir Khan	Drama, Romance	2009	Super Hit
M002	Chennai Express	Rohit Shetty	Shah Rukh Khan	Romance, Comedy	2014	Super Hit
M003	Inception	C Nolan	L Di Caprio	Thriller, Sci-Fi	2009	Blockbuster
M004	Stree	Amar Kaushik	Rajkumar Rao	Horror, Comedy	2019	Hit

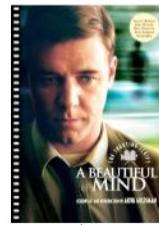


Keyword match

m_1



m_2



Movie id	Plot
M001	In the present day, Farhan receives a call from Chatur, saying that Rancho is coming. Farhan is so excited that he fakes a heart attack to get off a flight and picks up Raju from his home (who forgets to wear his pants). Farhan and Raju meet Chatur at the water tower of their college (ICE (Imperial College of Engineering), where Chatur informs them that he has found Rancho. Chatur taunts Farhan and Raju that now he has a mansion worth \$3.5 million in the US, with a heated swimming pool, a maple wood floor living room and a Lamborghini for a car. Chatur reveals that Rancho is in Shimla...

Movie id	Plot
M002	Rahul Mithaiwala (Shahrukh Khan) is a forty-year-old bachelor who lives in Mumbai. His parents died in a car accident when he was eight years old and was brought up by grandparents. His grandfather has a sweet-selling chain store - Y.Y. Mithaiwala. Before his birth centenary celebration, two of Rahul's friends suggest a vacation in Goa which he accepts. On the eve of the celebration, his grandfather dies whilst watching a cricket match. His grandmother tells him that his grandfather desired to have his ashes divided into two parts - to be immersed in the Ganges River and Rameswaram respectively. She requests Rahul to go to Rameswaram and immerse them. Rahul reluctantly accepts her request but was also eager to attend the Goa trip...

Movie id	Plot
M003	Dominick "Dom" Cobb (Leonardo DiCaprio) and business partner Arthur (Joseph Gordon-Levitt) are "extractors", people who perform corporate espionage using an experimental military technology to infiltrate the subconscious of their targets and extract information while experiencing shared dreaming. Their latest target is Japanese businessman Saito (Ken Watanabe). The extraction from Saito fails when sabotaged by a memory of Cobb's deceased wife Mal (Marion Cotillard). After Cobb's and Arthur's associate sells them out, Saito reveals that he was actually auditioning the team to perform the difficult act of "inception": planting an idea in a person's subconscious.

Movie id	Plot
M004	In the peculiar town of Chanderi, India, the residents believe in the myth of an angry woman ghost, referred to as "Stree" (Hindi for woman) (Flora Saini), who walks during Durga Puja festival. This is explained by the sudden disappearance of these men, leaving their clothes behind. She is said to stalk the men of the town, whispering their names and causing their disappearances if they look back at her. The whole town protects itself from Stree during the 4 nights of Durga Puja by writing "OO Stree, Kal Aana" on their walls. Additionally, men are advised to avoid going out alone after 10 PM, during the festival and to move in groups for safety. This practice, however, causes disappearances if they look back at her. This reflects a societal parallel to the precautions typically advised to women for their own protection.

Additional notes: In the town of Chanderi, India, women are advised to avoid going out alone after 10 PM, during the festival and to move in groups for safety. This practice, however, causes disappearances if they look back at her. This reflects a societal parallel to the precautions typically advised to women for their own protection.

Additionally, men are advised to avoid going out alone after 10 PM, during the festival and to move in groups for safety. This practice, however, causes disappearances if they look back at her. This reflects a societal parallel to the precautions typically advised to women for their own protection.

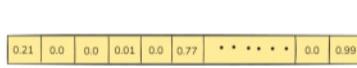
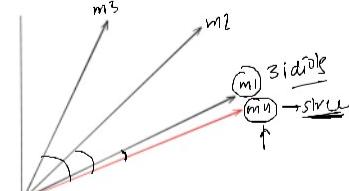


512 dim

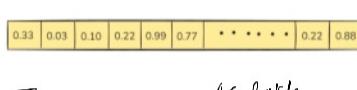


Compare similarity score →

512 dim



512 dim



512 dim

vectors

m_1

embeddings



512 dim



512 dim



512 dim

langchain-retrievers.ipynb

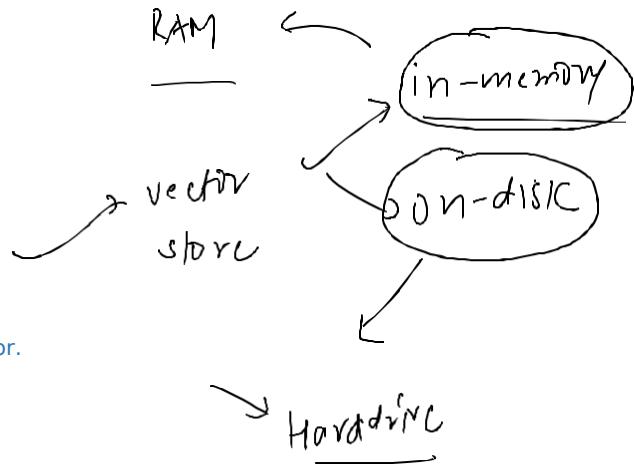
What are Vector Stores

05 April
2025 17:38

A **vector store** is a system designed to store and retrieve data represented as numerical vectors.

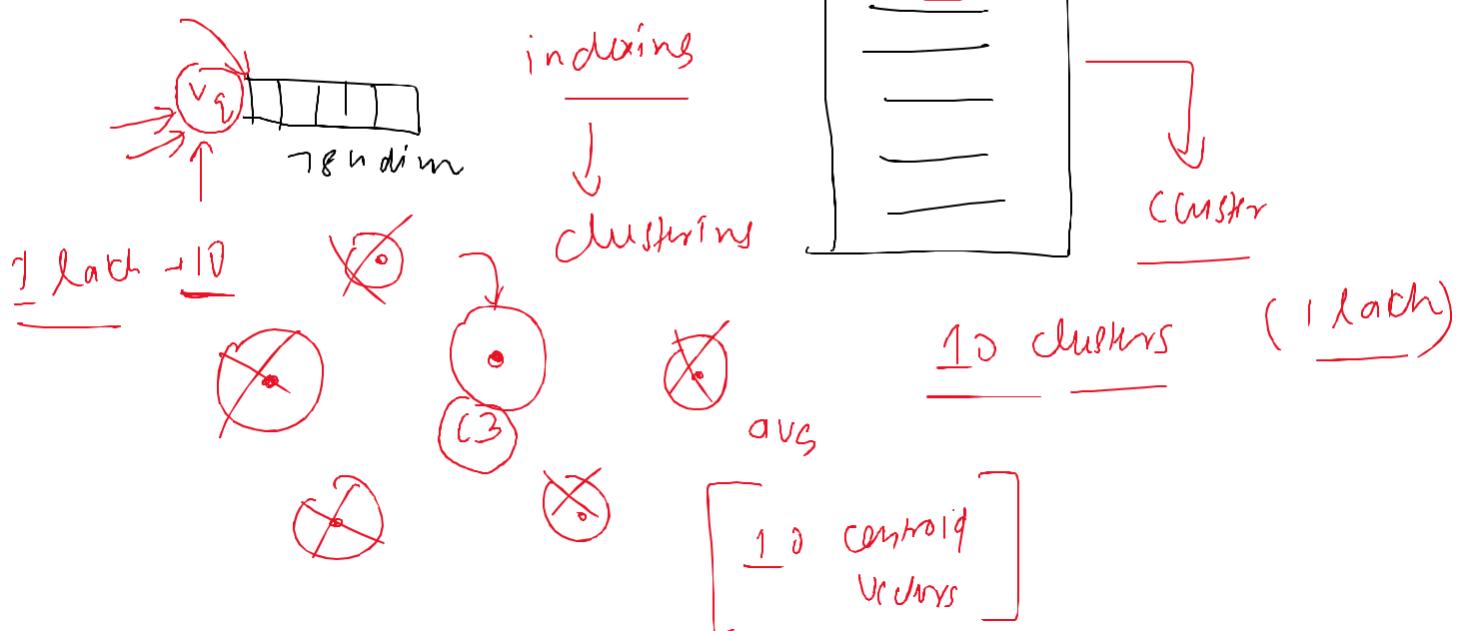
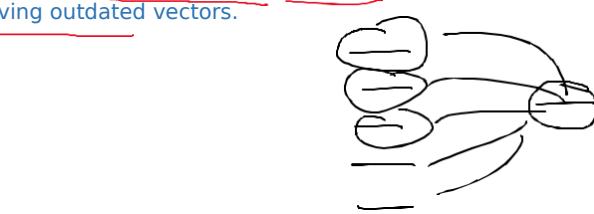
Key Features

1. Storage - Ensures that vectors and their associated metadata are retained, whether in-memory for quick lookups or on-disk for durability and large-scale use.
 2. Similarity Search - Helps retrieve the vectors most similar to a query vector.
 3. Indexing - Provide a data structure or method that enables fast similarity searches on high-dimensional vectors (e.g., approximate nearest neighbor lookups).
 4. CRUD Operations - Manage the lifecycle of data—adding new vectors, reading them, updating existing entries, removing outdated vectors.



Use-cases

- 1. Semantic Search ✓
 - 2. RAG →
 - 3. Recommender Systems ✓
 - 4. Image/Multimedia search



12. Vector Store Page 94langchain-retrievers.ipynb

Vector Store Vs Vector Database

05 April 2025 17:40

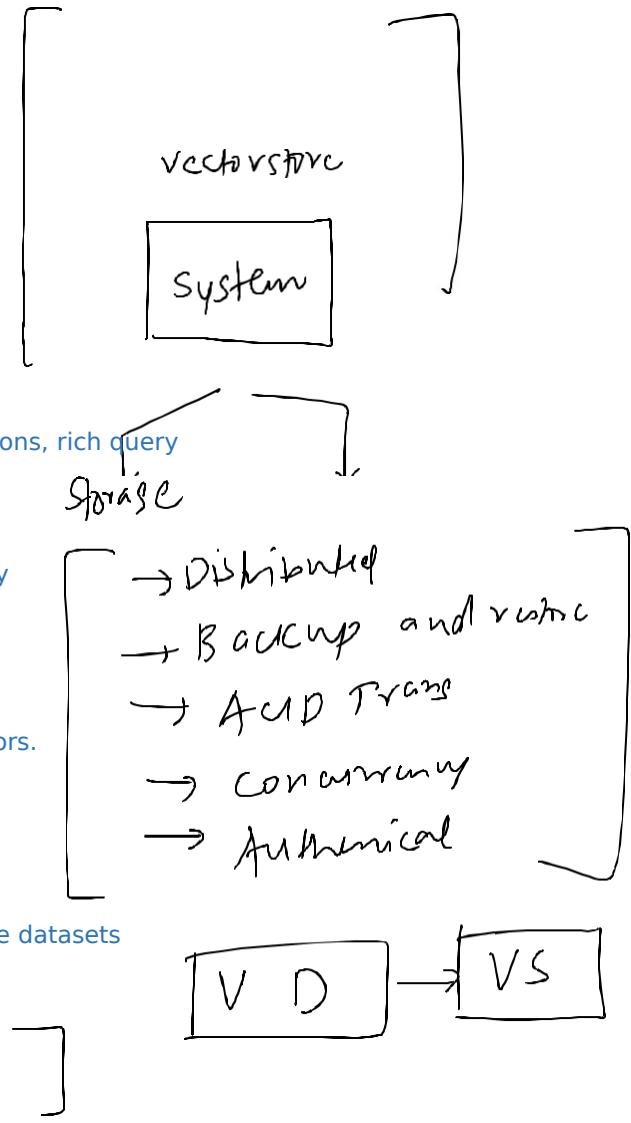
• Vector Store

- Typically refers to a lightweight library or service that focuses on storing vectors (embeddings) and performing similarity search.
- May not include many traditional database features like transactions, rich query languages, or role-based access control.
- Ideal for prototyping, smaller-scale applications
- Examples: FAISS (where you store vectors and can query them by similarity, but you handle persistence and scaling separately).

• Vector Database

- A full-fledged database system designed to store and query vectors.
- Offers additional "database-like" features:
 - Distributed architecture for horizontal scaling
 - Durability and persistence (replication, backup/restore)
 - Metadata handling (schemas, filters)
 - Potential for ACID or near-ACID guarantees
 - Authentication/authorization and more advanced security
- Geared for production environments with significant scaling, large datasets
- Examples: Milvus, Qdrant, Weaviate

A vector database is effectively a vector store with extra database features (e.g., clustering, scaling, security, metadata filtering, and durability)



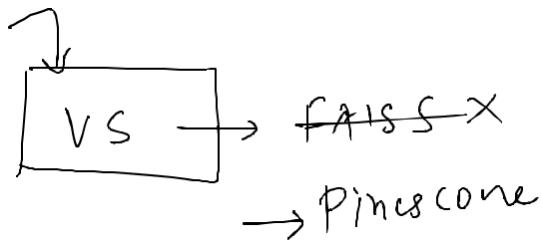
12. Vector Store Page 95

Vector Stores in LangChain

05 April 17:41
2025

- **Supported Stores:** LangChain integrates with multiple vector stores (FAISS, Pinecone, Chroma, Qdrant, Weaviate, etc.), giving you flexibility in scale, features, and deployment.
- **Common Interface:** A uniform Vector Store API lets you swap out one backend (e.g., FAISS) for another (e.g., Pinecone) with minimal code changes.
- **Metadata Handling:** Most vector stores in LangChain allow you to attach metadata (e.g., timestamps, authors) to each document, enabling filter-based retrieval.

from_documents(...) or from_texts(...)
add_documents(...) or add_texts(...)
similarity_search(query, k=...)
Metadata-Based Filtering

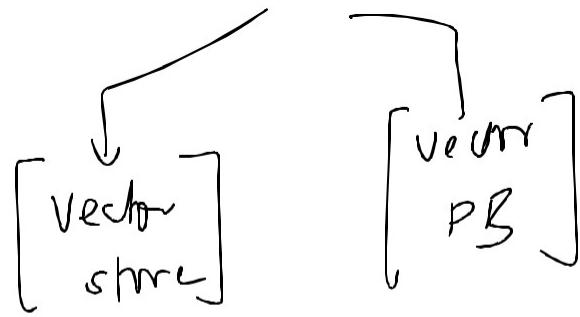
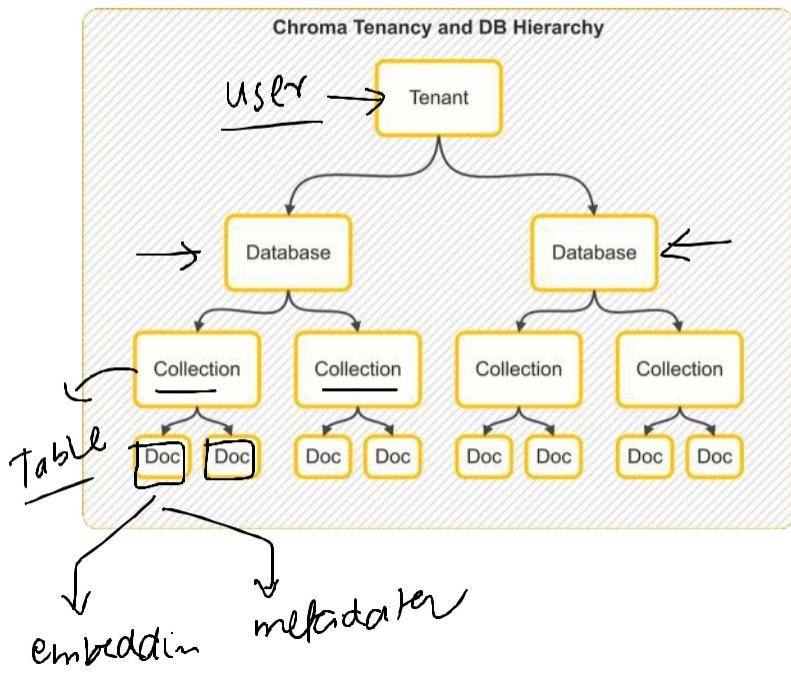


12. Vector Store Page 96

Chroma Vector Store

05 April 17:41
2025

Chroma is a lightweight, open-source vector database that is especially friendly for local development and small- to medium-scale production needs.



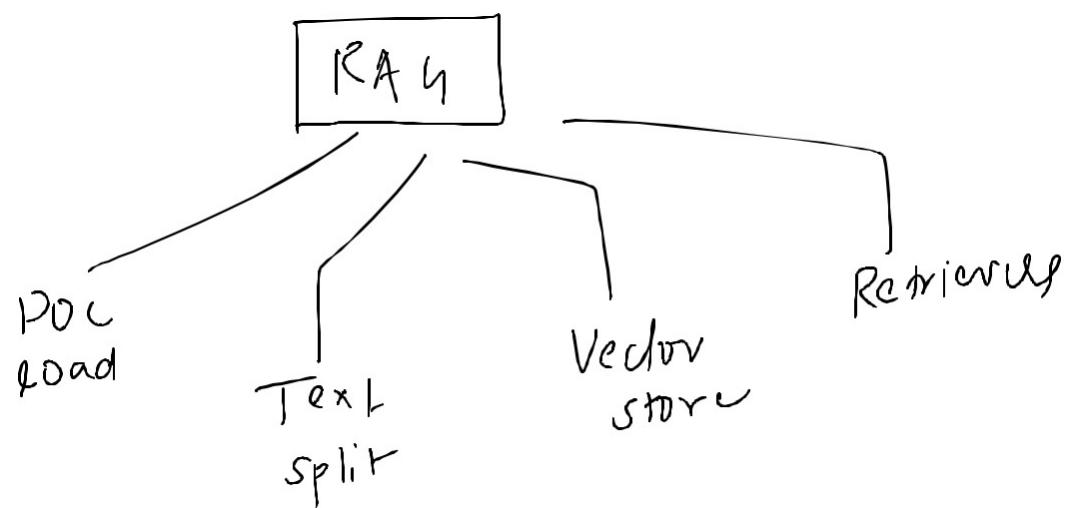
Code

05 April 17:
2025 41

12. Vector Store Page 98

Recap

12 April 13:51
2025



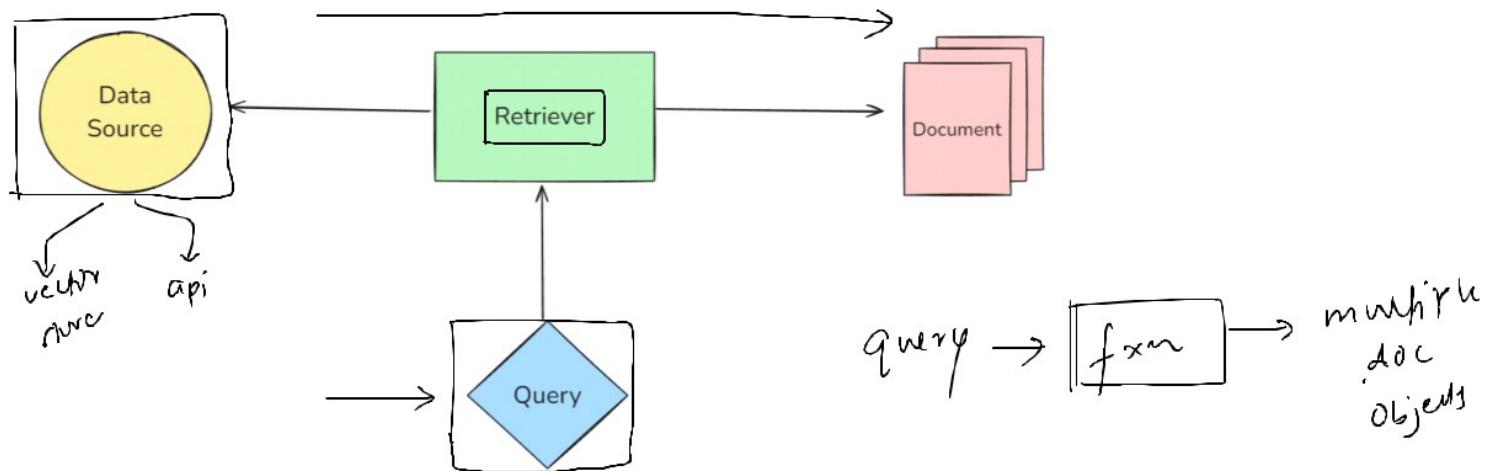
What are Retrievers

10 April 2025 07:56

A retriever is a component in LangChain that ~~fetches~~ relevant documents from a data source in response to a user's query.

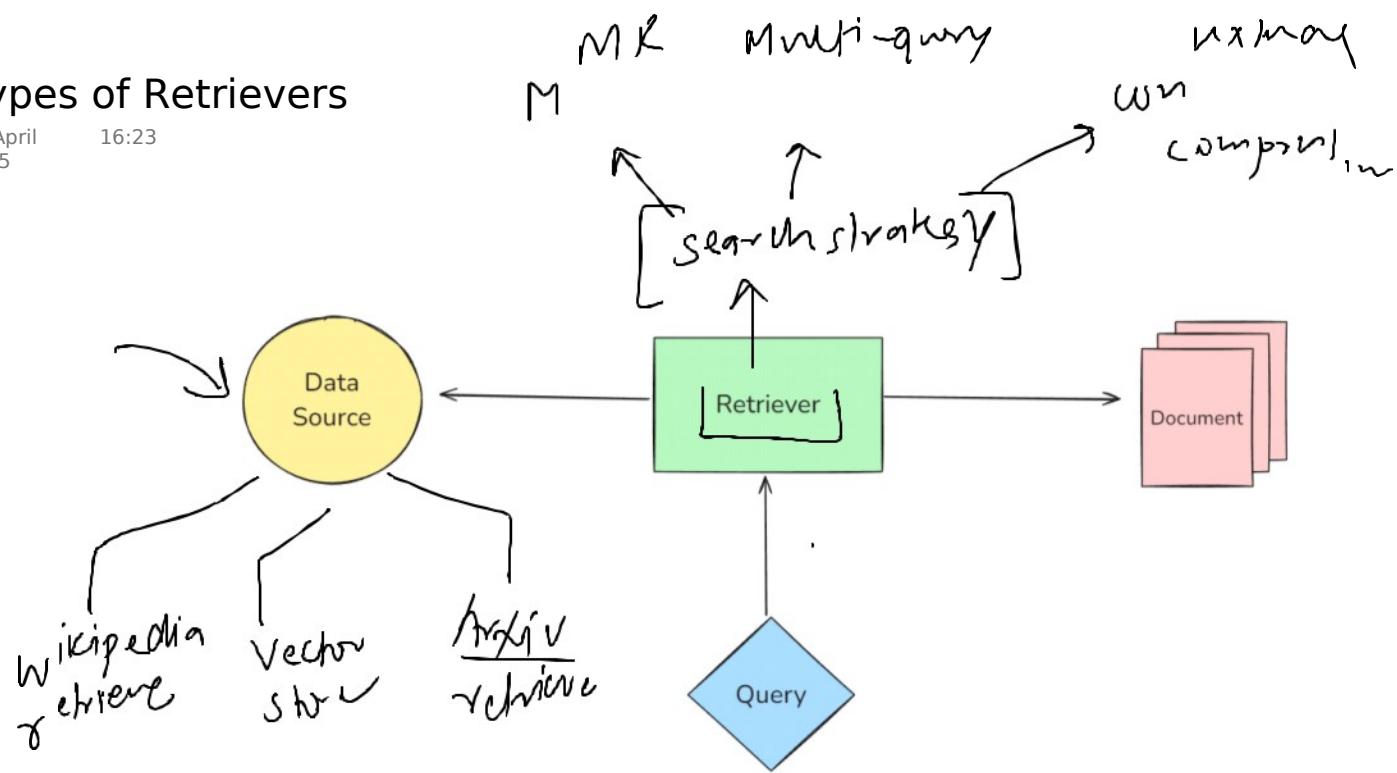
→ There are multiple types of retrievers

→ All retrievers in LangChain are runnables



Types of Retrievers

10 April 2025 16:23





Based on the Data Source they are diff types of retrievers:

Wikipedia Retriever → doc wa

10 April 2025 16:23

A Wikipedia Retriever is a retriever that queries the Wikipedia API to fetch relevant content for a given query. →

How It Works

1. You give it a query (e.g., "Albert Einstein")
2. It sends the query to Wikipedia's API
3. It retrieves the most relevant articles
4. It returns them as LangChain Document objects



Vector Store Retriever

10 April 2025 16:24

A Vector Store Retriever in LangChain is the most common type of retriever that lets you search and fetch documents from a vector store based on semantic similarity using vector embeddings.

How It Works

1. You store your documents in a vector store (like FAISS, Chroma, Weaviate)
2. Each document is converted into a dense vector using an embedding model
3. When the user enters a query:
 - It's also turned into a vector
 - The retriever compares the query vector with the stored vectors
 - It retrieves the top-k most similar ones

Based on the strategy of the retrieval:

Maximal Marginal Relevance (MMR)

10 April 16:24
2025

"How can we pick results that are not only relevant to the query but also different from each other?"

MMR is an information retrieval algorithm designed to reduce redundancy in the retrieved results while maintaining high relevance to the query.

💡 Why MMR Retriever?

In regular similarity search, you may get documents that are:

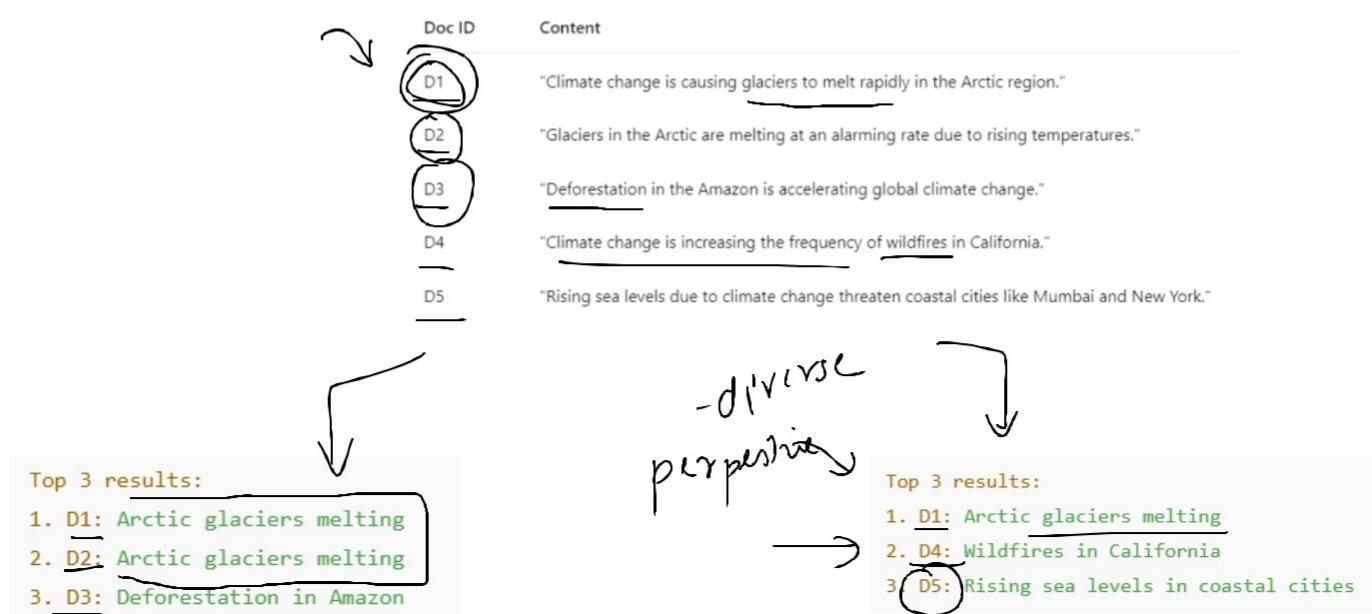
- All very similar to each other
- Repeating the same info
- Lacking diverse perspectives

MMR Retriever avoids that by:

- Picking the most relevant document first
- Then picking the next most relevant and least similar to already selected docs
- And so on...

This helps especially in RAG pipelines where:

- You want your context window to contain diverse but still relevant information
- Especially useful when documents are semantically overlapping



13. Retrievers Page 104

Multi-Query Retriever

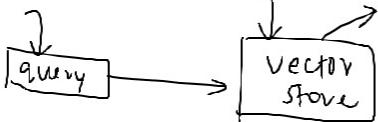
10 April 16:26
2025

Sometimes a single query might not capture all the ways information is phrased in your documents.

For example:

Query:

"How can I stay healthy?"



Could mean:

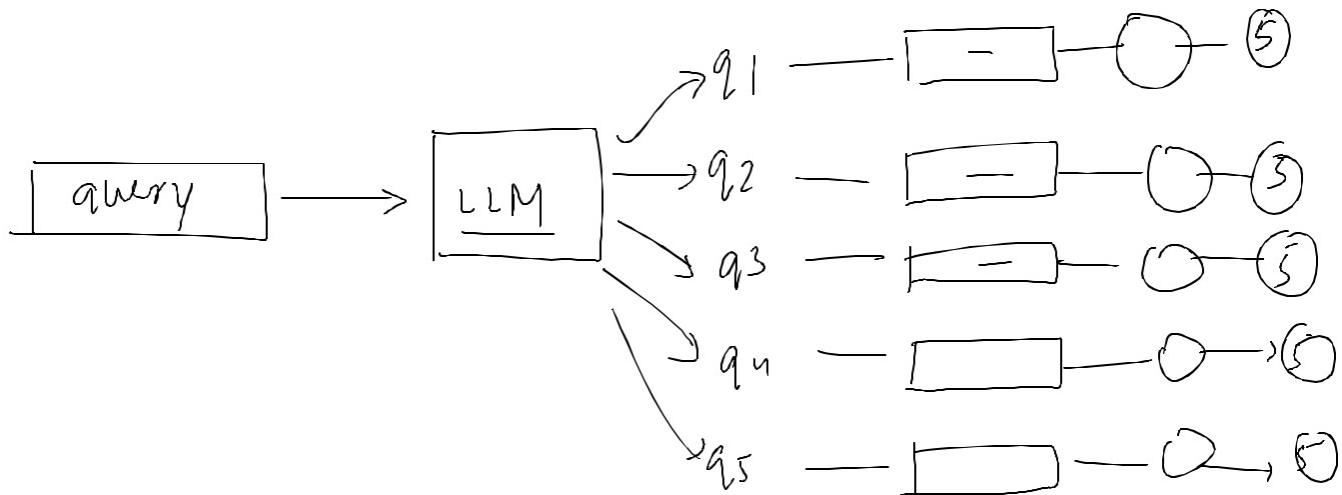
- What should I eat? ✓
- How often should I exercise? ✓
- How can I manage stress? ✓

A simple similarity search might miss documents that talk about those things but don't use the word "healthy."

"How can I stay healthy?"

- 1. "What are the best foods to maintain good health?"
- 2. "How often should I exercise to stay fit?"
- 3. "What lifestyle habits improve mental and physical wellness?"
- 4. "How can I boost my immune system naturally?"
- 5. "What daily routines support long-term health?"

1. Takes your original query
2. Uses an LLM (e.g., GPT-3.5) to generate multiple semantically different versions of that query
3. Performs retrieval for each sub-query
4. Combines and deduplicates the results



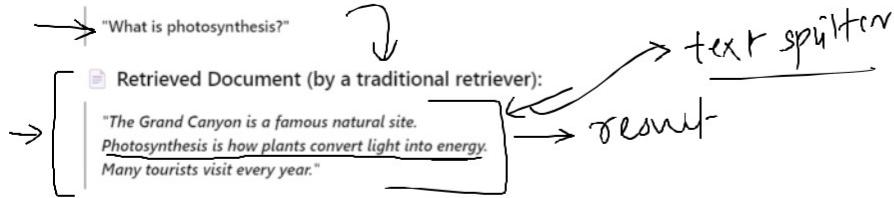
Here it combines the results from the each query and retrieves after deduplication the result.

Contextual Compression Retriever

10 April 2025 16:29

The Contextual Compression Retriever in LangChain is an advanced retriever that improves retrieval quality by compressing documents after retrieval — keeping only the relevant content based on the user's query.

Query:



Problem:

- The retriever returns the entire paragraph
- Only one sentence is actually relevant to the query
- The rest is irrelevant noise that wastes context window and may confuse the LLM

What Contextual Compression Retriever does:

Returns only the relevant part, e.g.

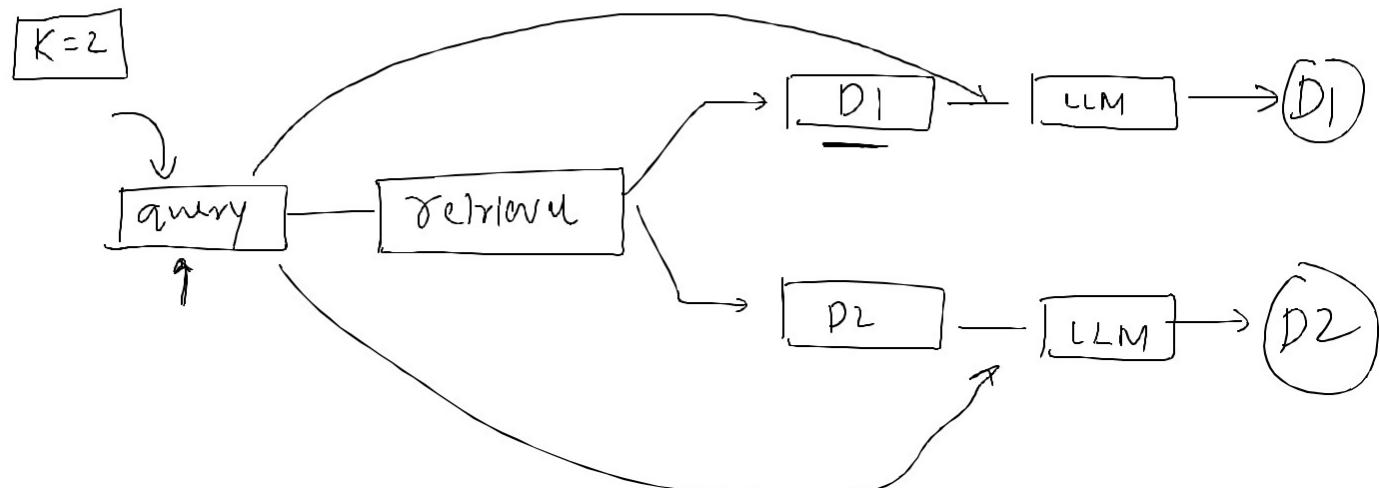
"Photosynthesis is how plants convert light into energy."

How It Works

- Base Retriever (e.g., FAISS, Chroma) retrieves N documents.
- A compressor (usually an LLM) is applied to each document.
- The compressor keeps only the parts relevant to the query.
- Irrelevant content is discarded.

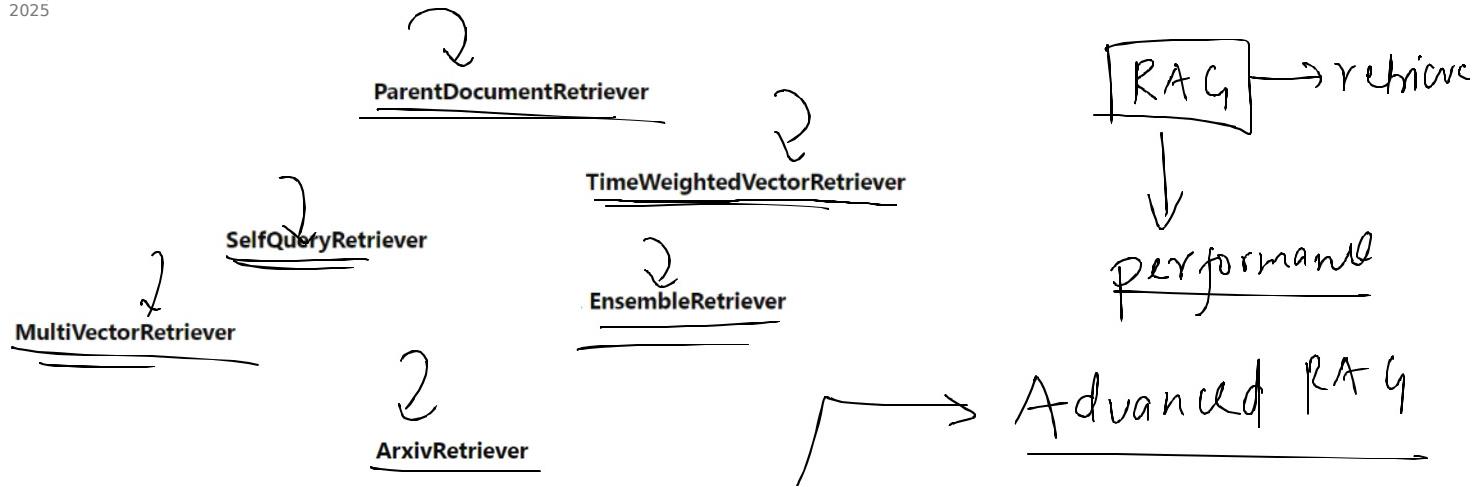
When to Use

- Your documents are long and contain mixed information
- You want to reduce context length for LLMs
- You need to improve answer accuracy in RAG pipelines



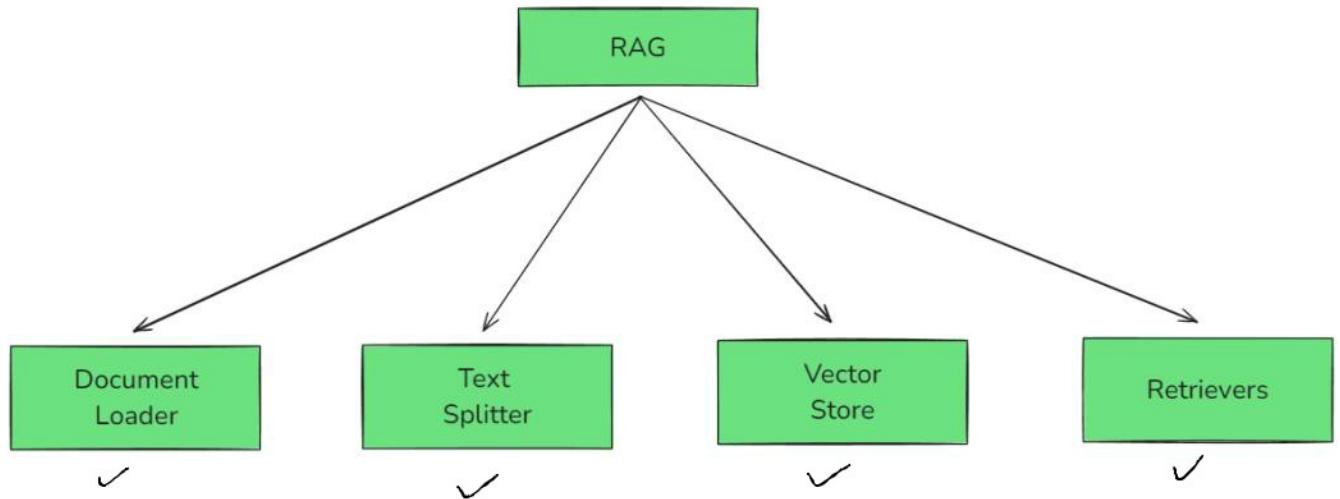
More Retrievers

10 April 2025 16:29



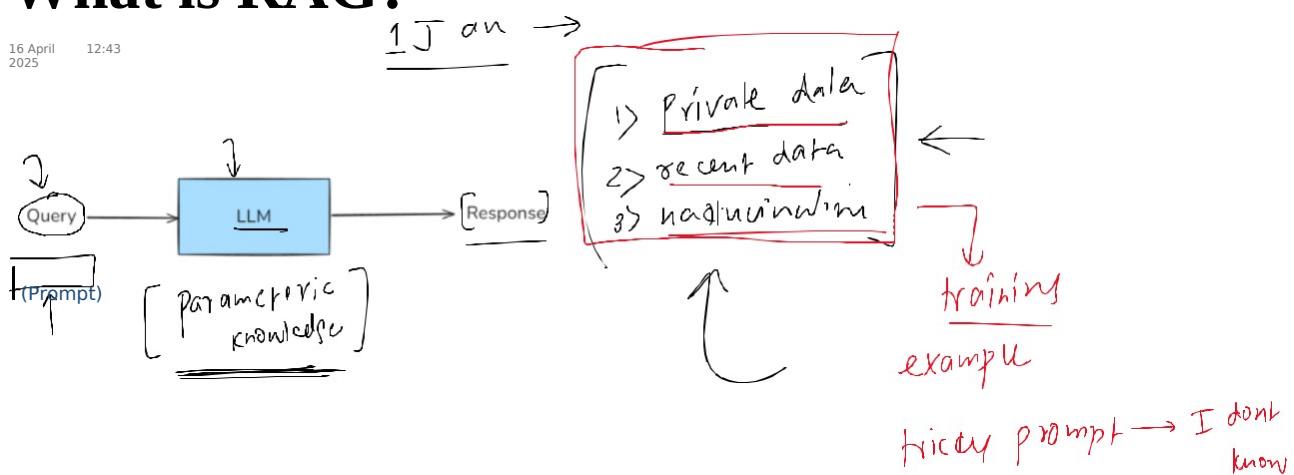
Recap

16 April 12:45
2025



What is RAG?

16 April 2025 12:43



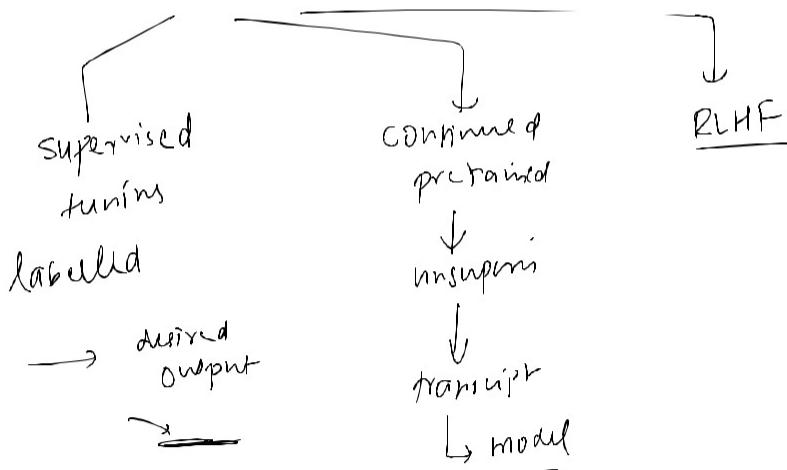
fine-tuning

pretrained **LLM**

↓
smaller → domain specific

student - LLM
finetunes → pretrain
train → finetuning

1. Collect data
A few hundred – few hundred-thousand carefully curated examples (prompts – desired outputs).
2. Choose a method
Full-parameter FT, LoRA/QLoRA, or parameter-efficient adapters.
3. Train for a few epochs
You keep the base weights frozen or partially frozen and update only a small subset (LoRA) or all weights (full FT).
4. Evaluate & safety-test
Measure exact-match, factuality, and hallucination rate against held-out data; red-team for safety.



- 1) LLM → train
- 2) technical expertise
- 3)

In context learning

In-Context Learning

is a core capability of Large Language Models (LLMs) like GPT-3/4, Claude, and Llama, where the model learns to solve a task purely by seeing examples in the prompt—without updating its weights.

LLM → em(^sant) np

Below are examples of texts labeled with their sentiment.
Use these examples to determine the sentiment of the final text.

Text: I love this phone. It's so smooth. → Positive
Text: This app crashes a lot. → Negative
Text: The camera is amazing! → Positive

LLM → emergent property

An **emergent property** is a behaviour or ability that suddenly appears in a system when it reaches a certain scale or complexity—even though it was not explicitly programmed or expected from the individual components.

$\begin{bmatrix} \text{GPT-1} \\ \text{GPT-2} \end{bmatrix} \rightarrow \text{prompts}$
GPT-3 (175B)

Context

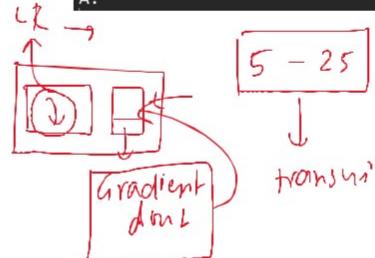
- Instead of just example tasks, retrieve background information, facts, documents, product manuals, etc.
- Inject that into the prompt to augment the model's knowledge

Use these examples to determine the sentiment of the final text.
Text: I love this phone. It's so smooth. → Positive
Text: This app crashes a lot. → Negative
Text: The camera is amazing! → Positive
Text: I hate the battery life. →

Label the named entities in the sentences (Person, Location, Organization):
Sentence: Elon Musk founded SpaceX.
Entities: [Elon Musk → Person], [SpaceX → Organization]

Solve the math problems step by step:

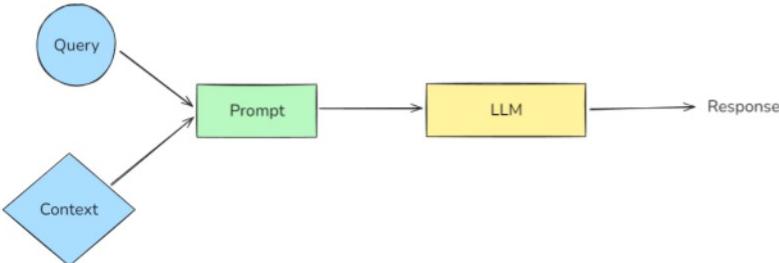
- Q: John has 3 apples. He buys 4 more. How many apples does he have now?
A: John had 3 apples. He bought 4 more. $3 + 4 = 7$. → 7
- Q: Sarah has 10 pens. She gives away 3. How many does she have left?
A: Sarah had 10 pens. She gave away 3. $10 - 3 = 7$. → 7
- Q: A pizza is cut into 8 slices. Alex eats 3 slices. How many are left?
A:



"""You are a helpful assistant.
Answer the question ONLY from the provided context.
If the context is insufficient, just say you don't know.

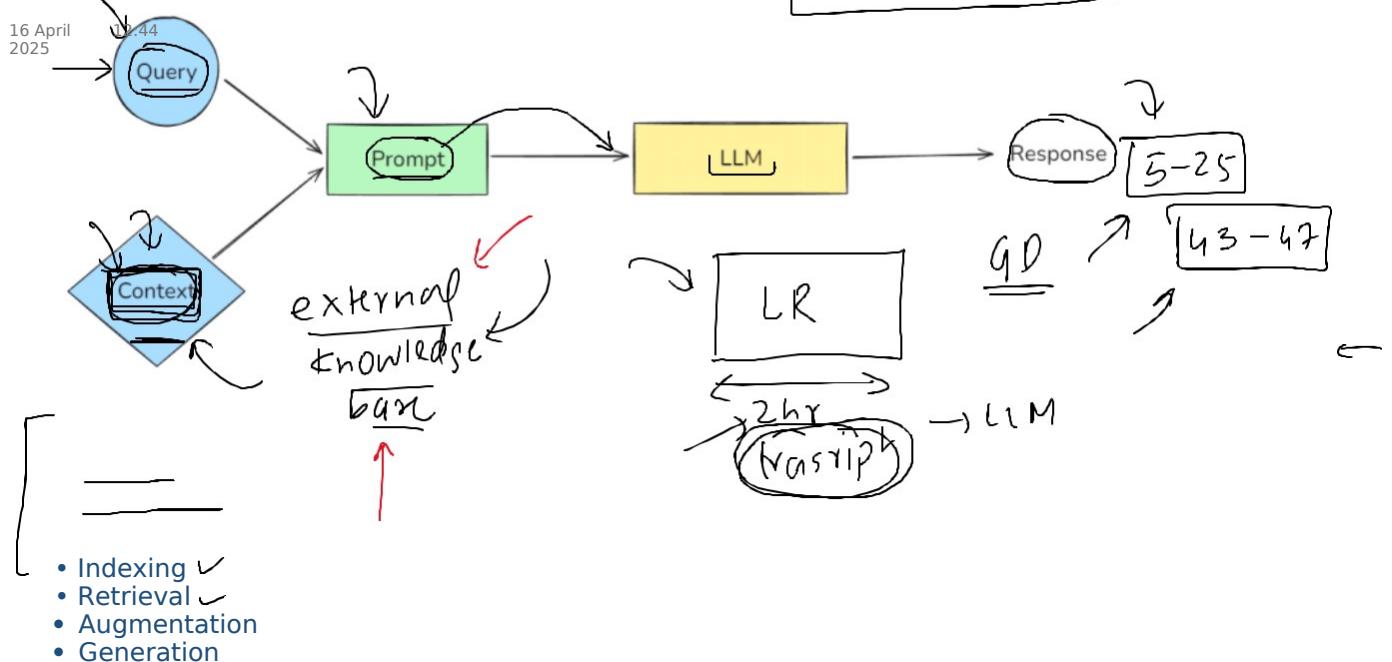
{context}
Question: {question}"""

RAG is a way to make a language model (like ChatGPT) smarter by giving it extra information at the time you ask your question.



Understanding RAG

RAG → Information retrieval + text generation



Indexing - Indexing is the process of preparing your knowledge base so that it can be efficiently searched at query time. This step consists of 4 sub-steps.

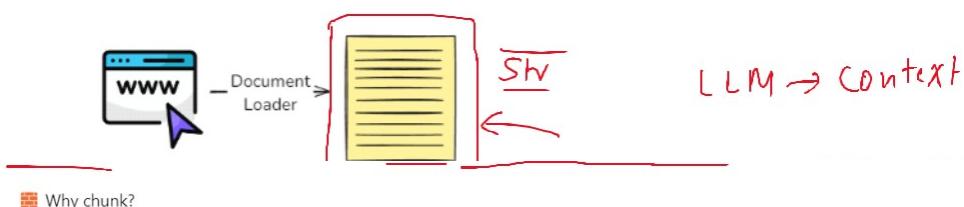
1. Document Ingestion - You load your source knowledge into memory.

Examples:

- PDF reports, Word documents
- YouTube transcripts, blog pages
- GitHub repos, internal wikis
- SQL records, scraped webpages

Tools:

- LangChain loaders ([PyPDFLoader](#), [YoutubeLoader](#), [WebBaseLoader](#), [GitLoader](#), etc.)

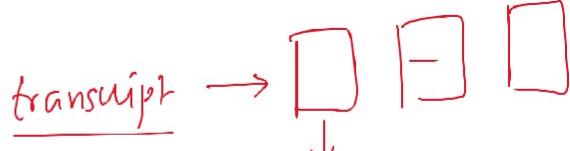
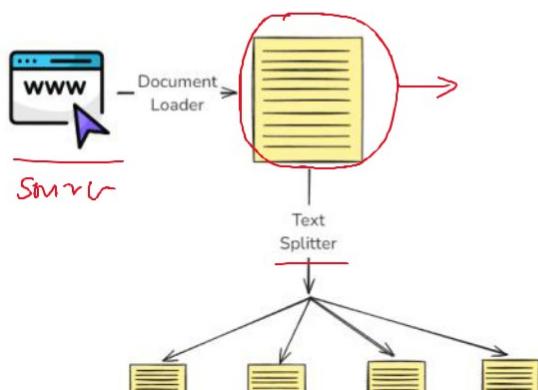


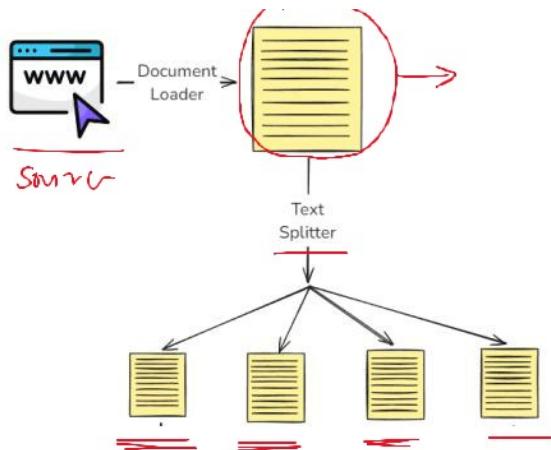
2. Text Chunking - Break large documents into small, semantically meaningful chunks

- Smaller chunks are more focused → better semantic search

Tools:

- [RecursiveCharacterTextSplitter](#), [MarkdownHeaderTextSplitter](#), [SemanticChunker](#)





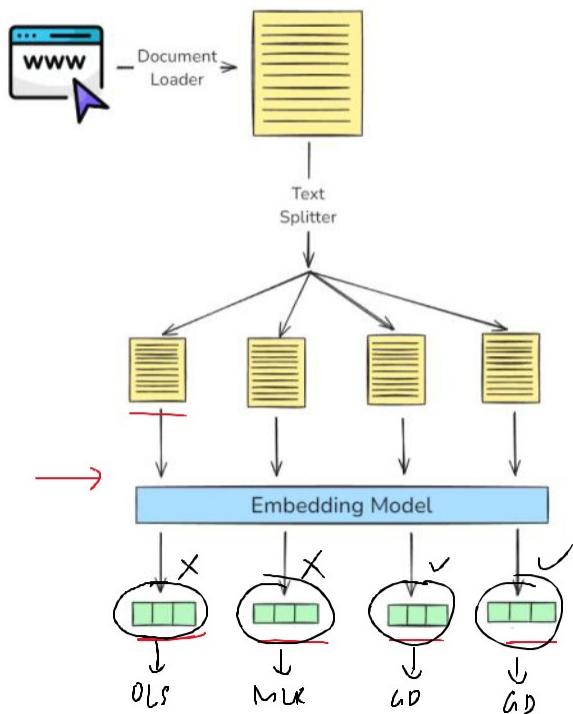
3. Embedding Generation - Convert each chunk into a dense vector (embedding) that captures its meaning.

💡 Why embeddings?

- Similar ideas land close together in vector space
- Allows fast, fuzzy semantic search

🛠 Tools:

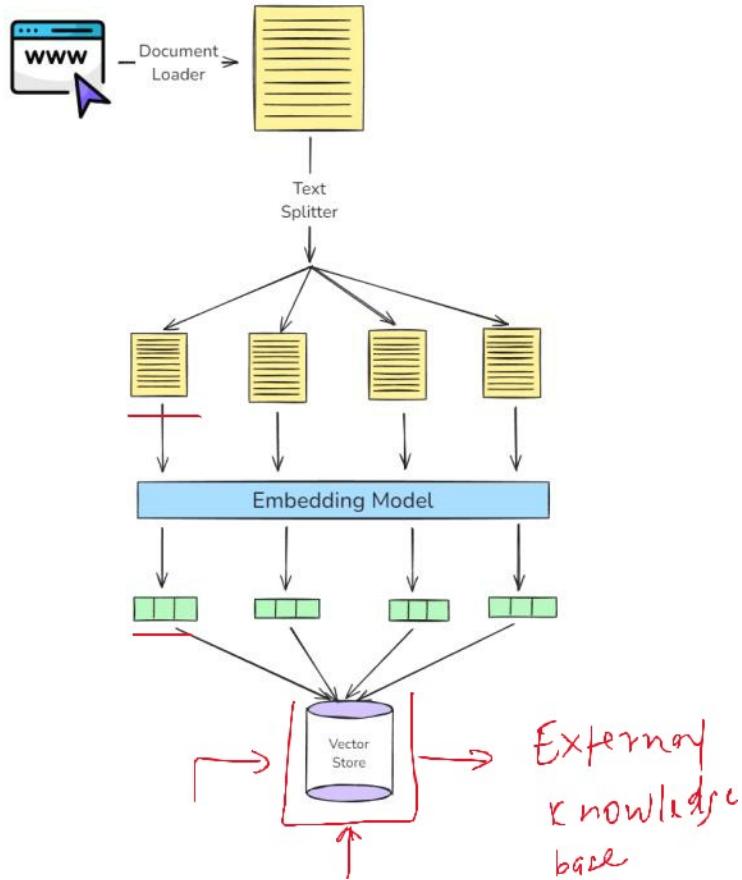
- [OpenAIEmbeddings](#), [SentenceTransformerEmbeddings](#), [InstructorEmbeddings](#), etc.



4. Storage in a Vector Store - Store the vectors along with the original chunk text + metadata in a vector database.

🛠 Vector DB options:

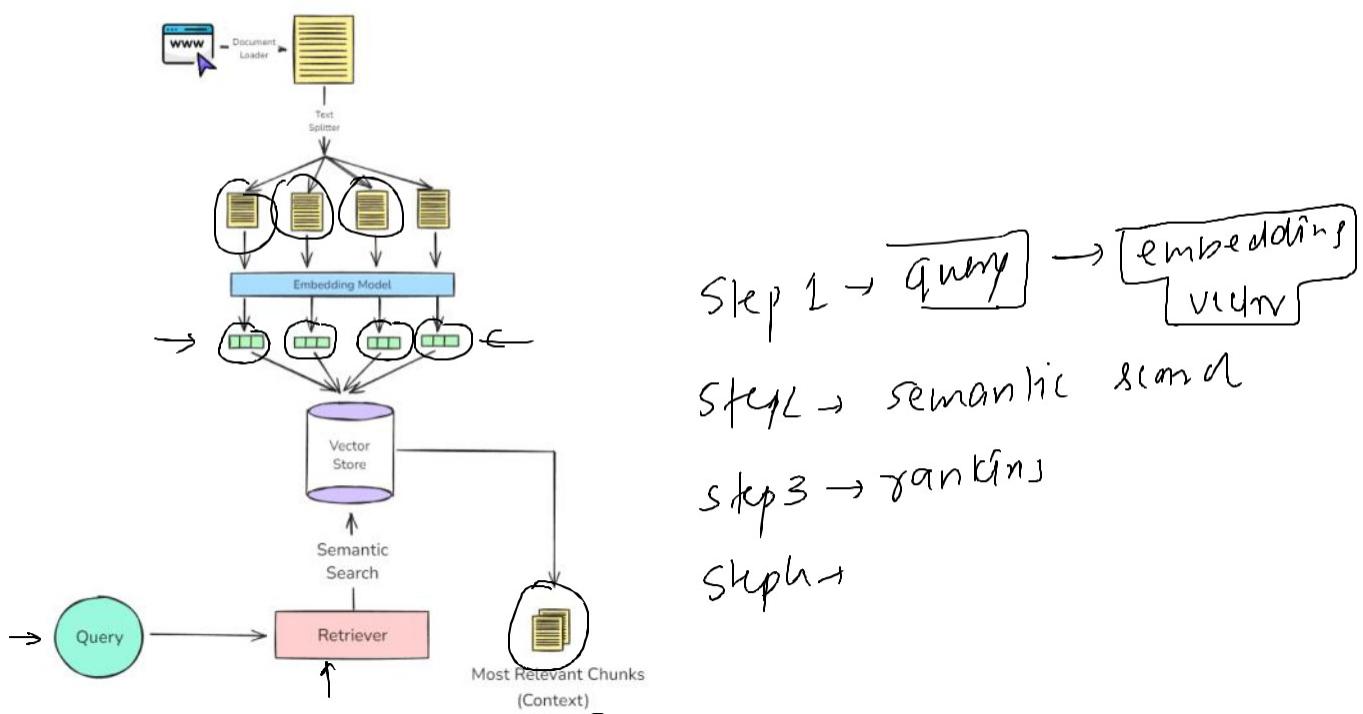
- Local: [FAISS](#), [Chroma](#)
- Cloud: [Pinecone](#), [Weaviate](#), [Milvus](#), [Qdrant](#)



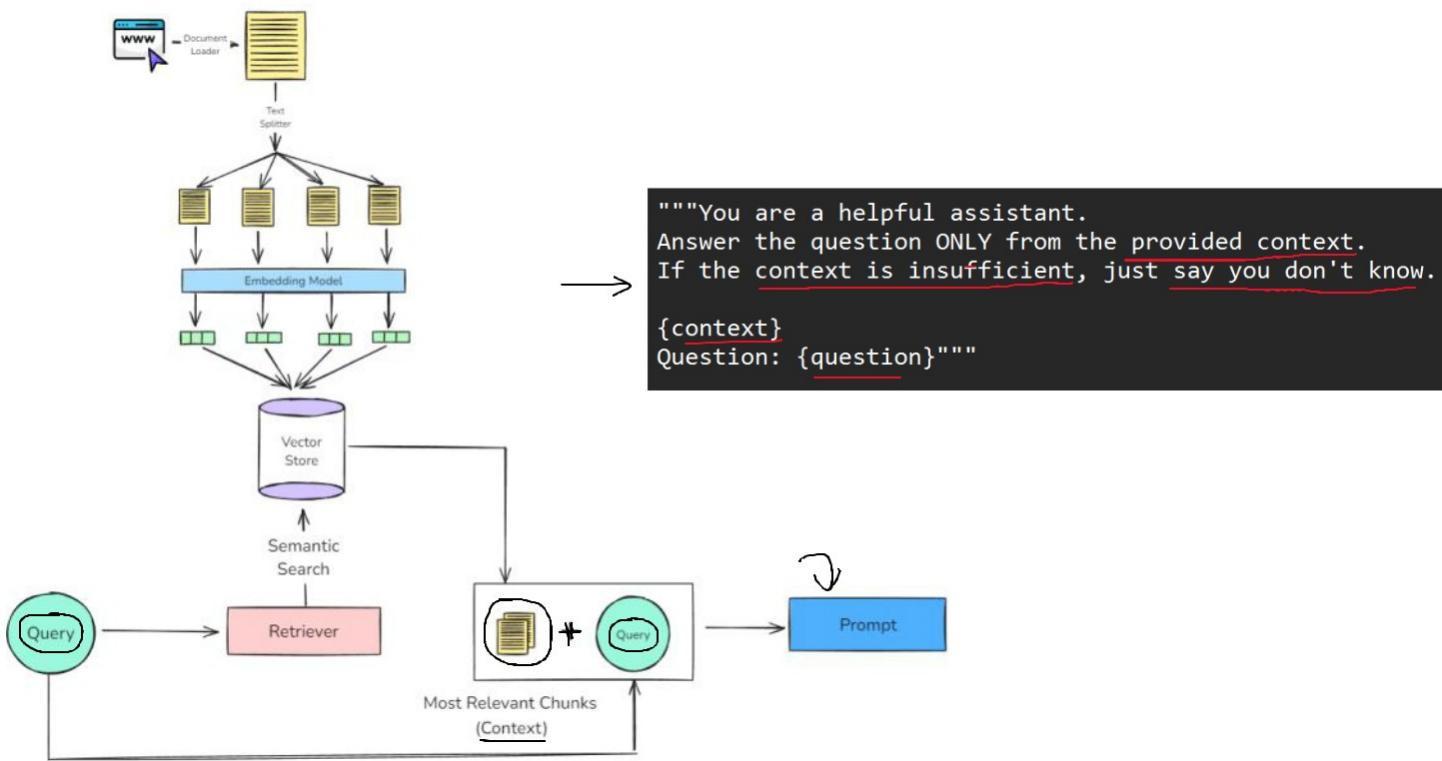
Retrieval - Retrieval is the real-time process of finding the most relevant pieces of information from a pre-built index (created during indexing) based on the user's question.

It's like asking:

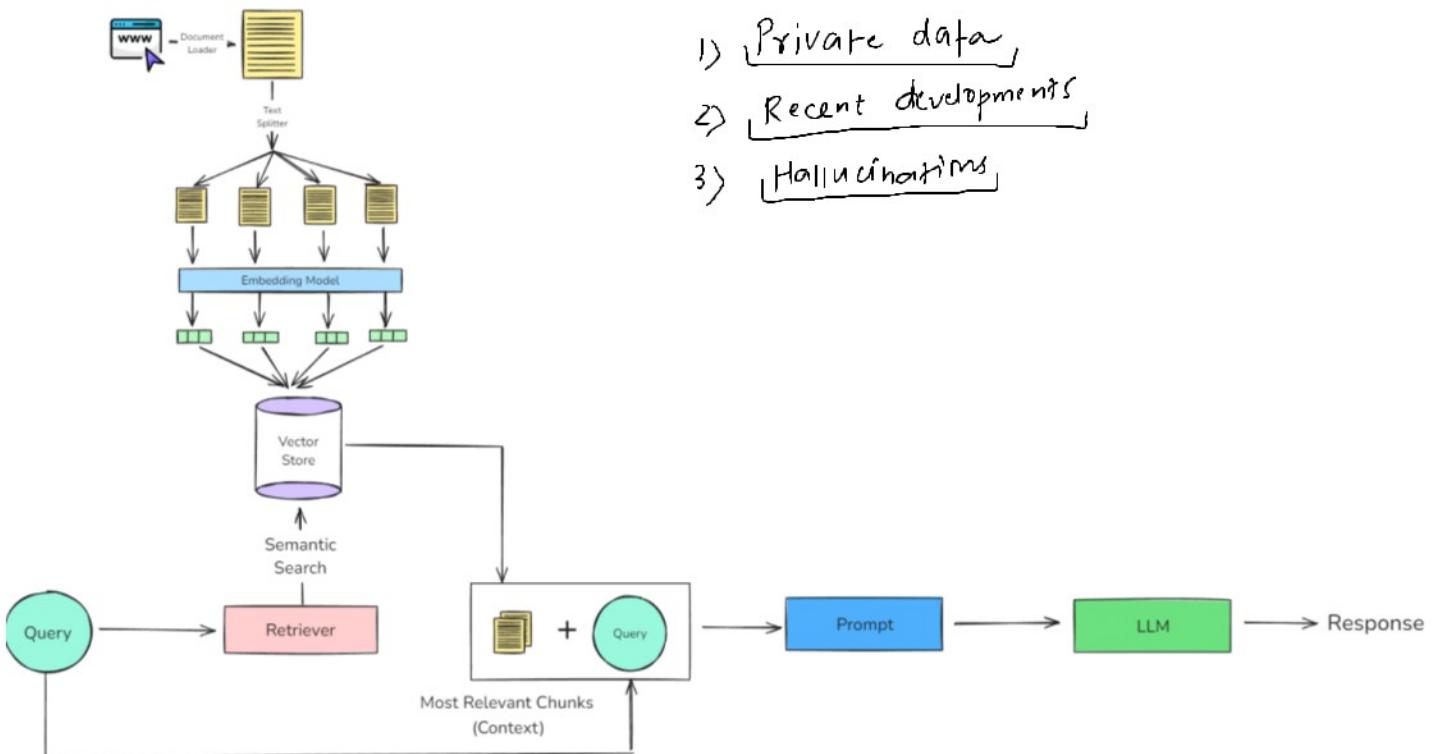
"From all the knowledge I have, which 3–5 chunks are most helpful to answer this query?"



Augmentation - Augmentation refers to the step where the retrieved documents (chunks of relevant context) are combined with the user's query to form a new, enriched prompt for the LLM.

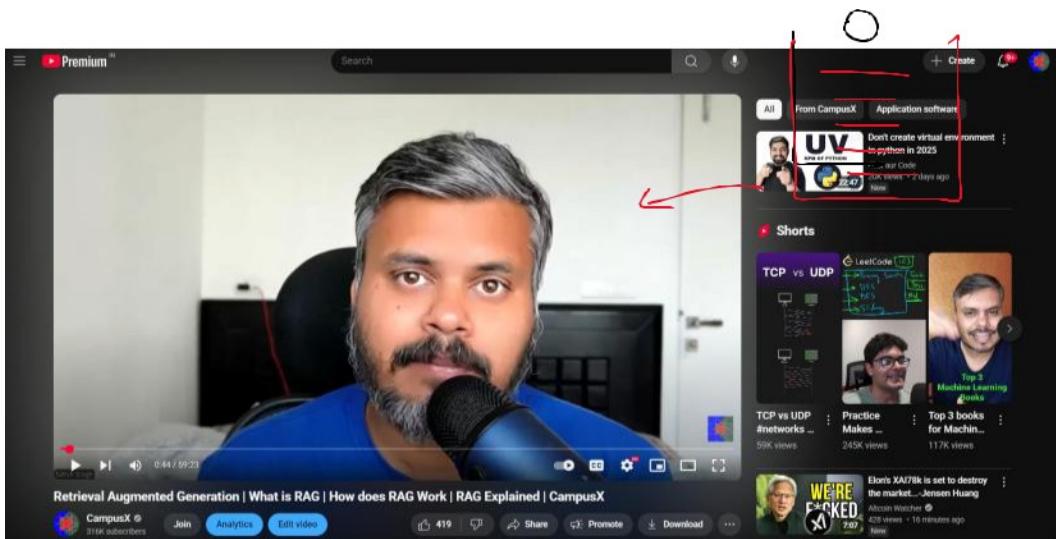


Generation - Generation is the final step where a Large Language Model (LLM) uses the user's query and the retrieved & augmented context to generate a response.

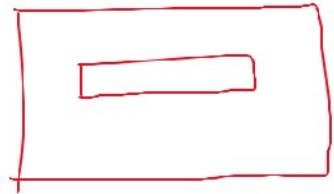


Problem Statement

21 April 08:55
2025



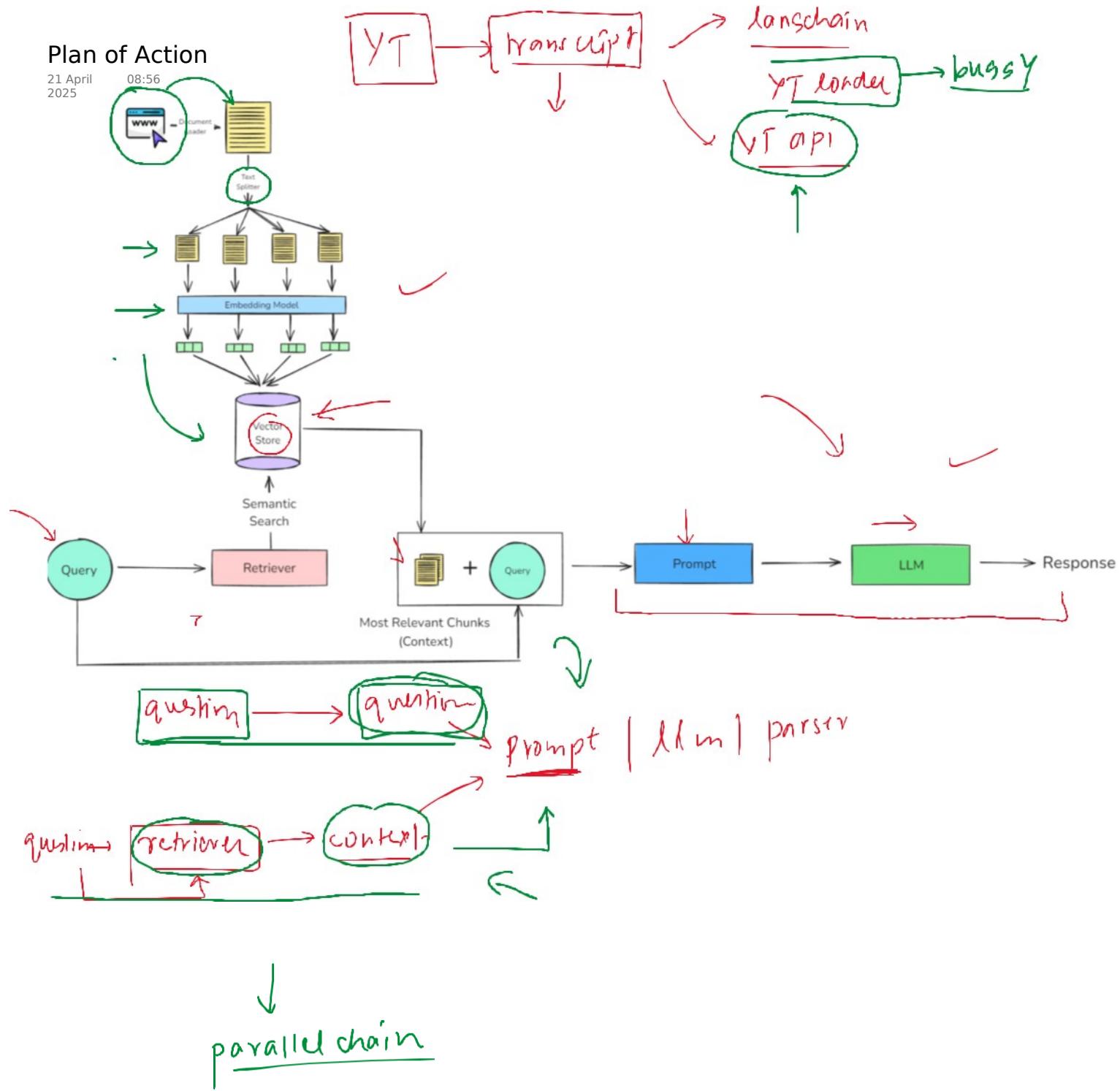
→ Chrome plugin
HTML/CSS/JS



Streamlit → []

Plan of Action

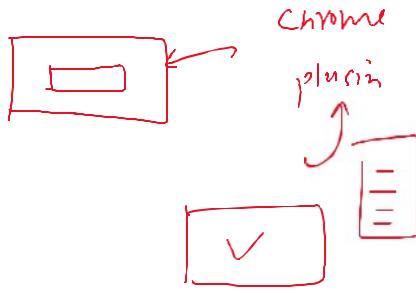
21 April 2025
08:56



Improvements

22 April 2025 08:50

→ 1. UI based enhancements



→ 2. Evaluation

- a. Ragas
- b. LangSmith

→ 3. Indexing

- a. Document Ingestion →
- b. Text Splitting →
- c. Vector Store → Pinecone

Metric	What It Measures
<u>faithfulness</u>	Is the answer grounded in the retrieved context?
<u>answer_relevancy</u>	Is the answer relevant to the user's question?
<u>context_precision</u>	How much of the retrieved context is actually useful?
<u>context_recall</u>	Did we retrieve all necessary information?

4. Retrieval

- a. Pre-Retrieval
 - i. Query rewriting using LLM
 - ii. Multi-query generation
 - iii. Domain aware routing

- b. During Retrieval
 - i. MMR
 - ii. Hybrid Retrieval
 - iii. Reranking

- c. Post-Retrieval
 - i. Contextual Compression

5. Augmentation

- a. Prompt Templating
- b. Answer grounding
- c. Context window optimization

6. Generation

- a. Answer with Citation
- b. Guard railing

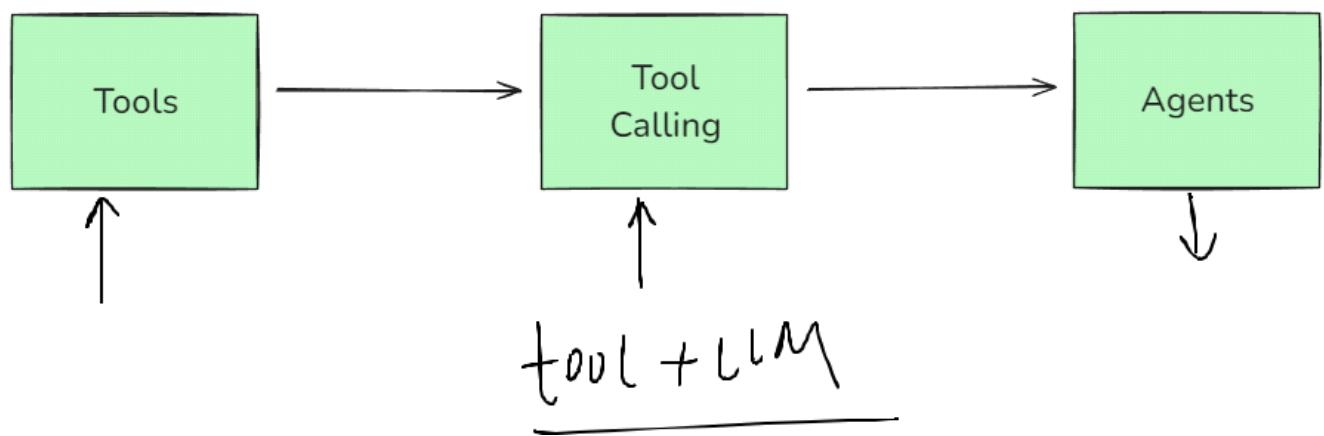
7. System Design

- a. Multimodal →
- b. Agentic →
- c. Memory based →

Tools in Langchian:

Overview:

23 April 15:51
2025



16. Tools Page 118

What is a Tool?

23 April 2025 15:45

A tool is just a Python function (or API) that is packaged in a way the LLM can understand and call when needed.

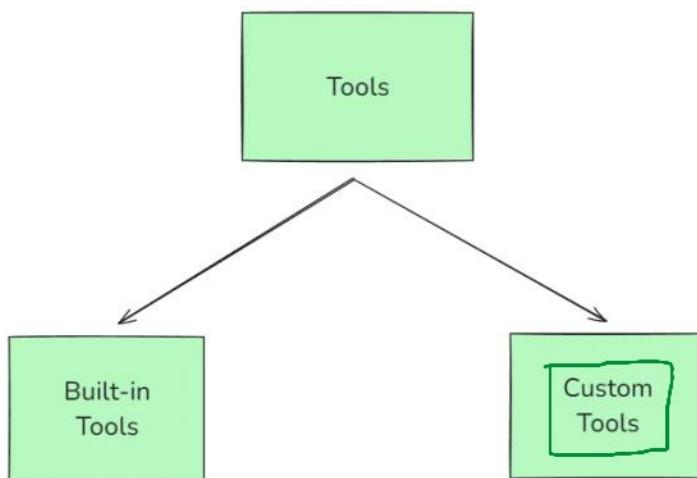
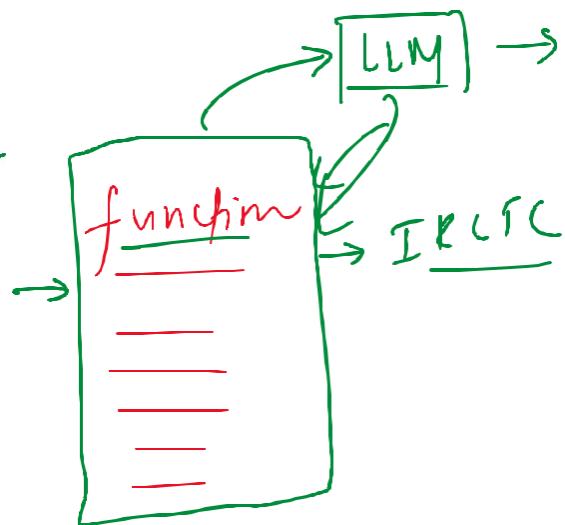
LLMs (like GPT) are great at:

- Reasoning (think)
- Language generation

(speak)

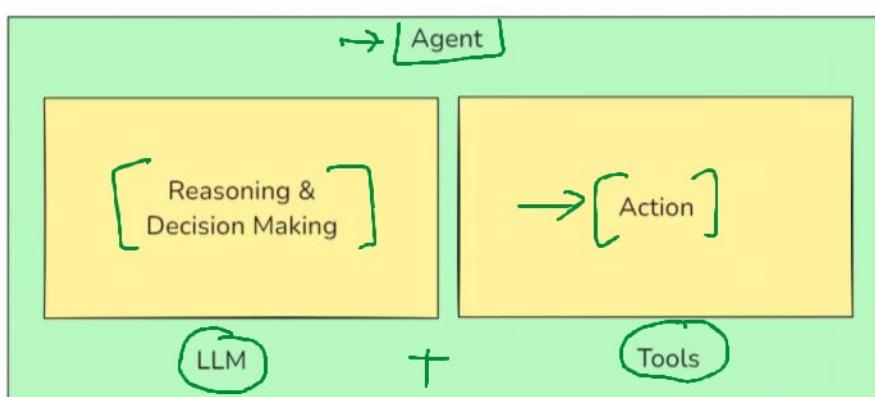
But they can't do things like:

- Access live data (weather, news)
- Do reliable math
- Call APIs
- Run code
- Interact with a database



How Tools fits into the Agent ecosystem

An AI agent is an LLM-powered system that can autonomously think, decide, and take actions using external tools or APIs to achieve a goal.



Built-in Tools

23 April 15:
2025 46

A built-in tool is a tool that LangChain already provides for you — it's pre-built, production-ready, and requires minimal or no setup.

You don't have to write the function logic yourself — you just import and use it.

DuckDuckGoSearchRun	Web search via DuckDuckGo
WikipediaQueryRun	Wikipedia summary
PythonREPLTool	Run raw Python code
ShellTool	Run shell commands
RequestsGetTool	Make HTTP GET requests
GmailSendMessageTool	Send emails via Gmail
SlackSendMessageTool	Post message to Slack
SQLDatabaseQueryTool	Run SQL queries

16. Tools Page 120

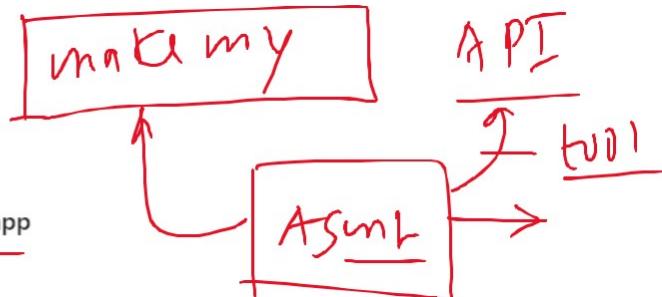
Custom Tools

23 April 15:46
2025

A custom tool is a tool that you define yourself.

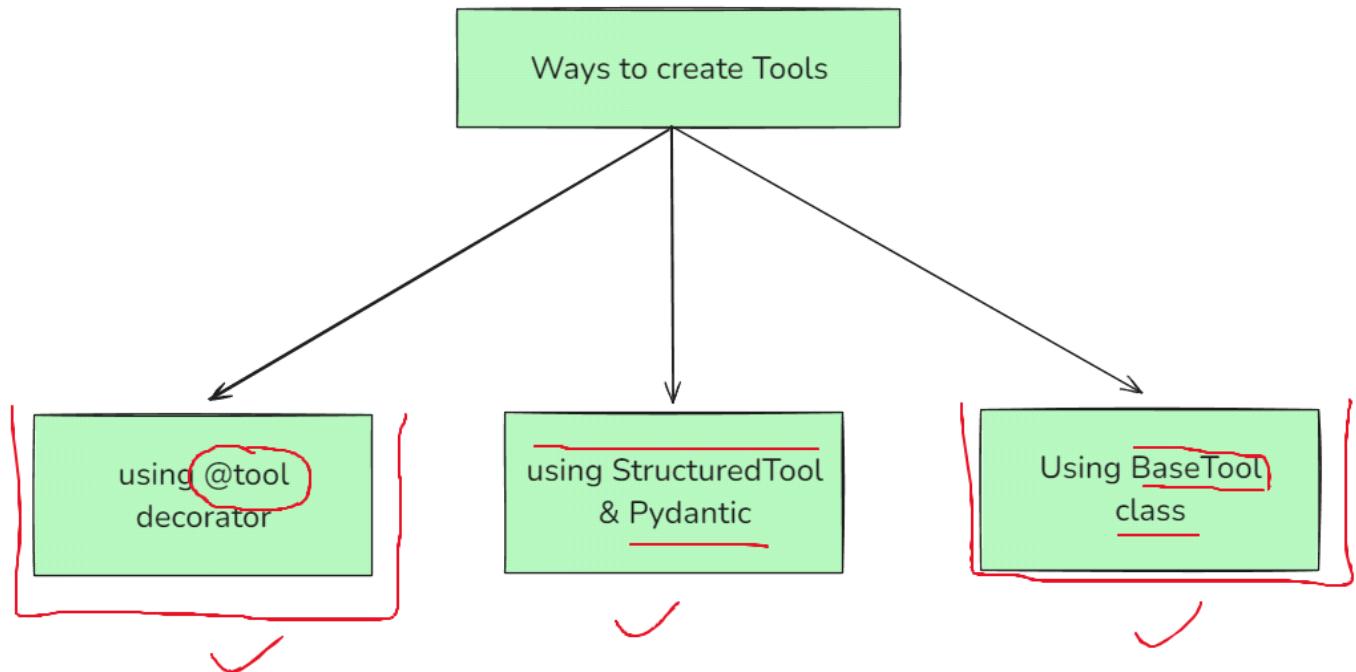
Use them when:

- You want to call your own APIs
- You want to encapsulate business logic
- You want the LLM to interact with your database, product, or app



Ways to create Custom Tools

23 April 15:46
2025



A **Structured Tool** in LangChain is a special type of tool where the input to the tool follows a structured schema, typically defined using a Pydantic model.

BaseTool is the abstract base class for all tools in LangChain. It defines the core structure and interface that any tool must follow, whether it's a simple one-liner or a fully customized function.

All other tool types like **@tool**, **StructuredTool** are built on top of **BaseTool**

16. Tools Page 122

Toolkits

23 April 15:
2025 47

A toolkit is just a collection (bundle) of related tools that serve a common purpose — packaged together for convenience and **reusability**.

In LangChain:

- A toolkit might be: GoogleDriveToolKit
- And it can contain the following tools

- GoogleDriveCreateFileTool : Upload a file
- GoogleDriveSearchTool : Search for a file by name/content
- GoogleDriveReadFileTool : Read contents of a file

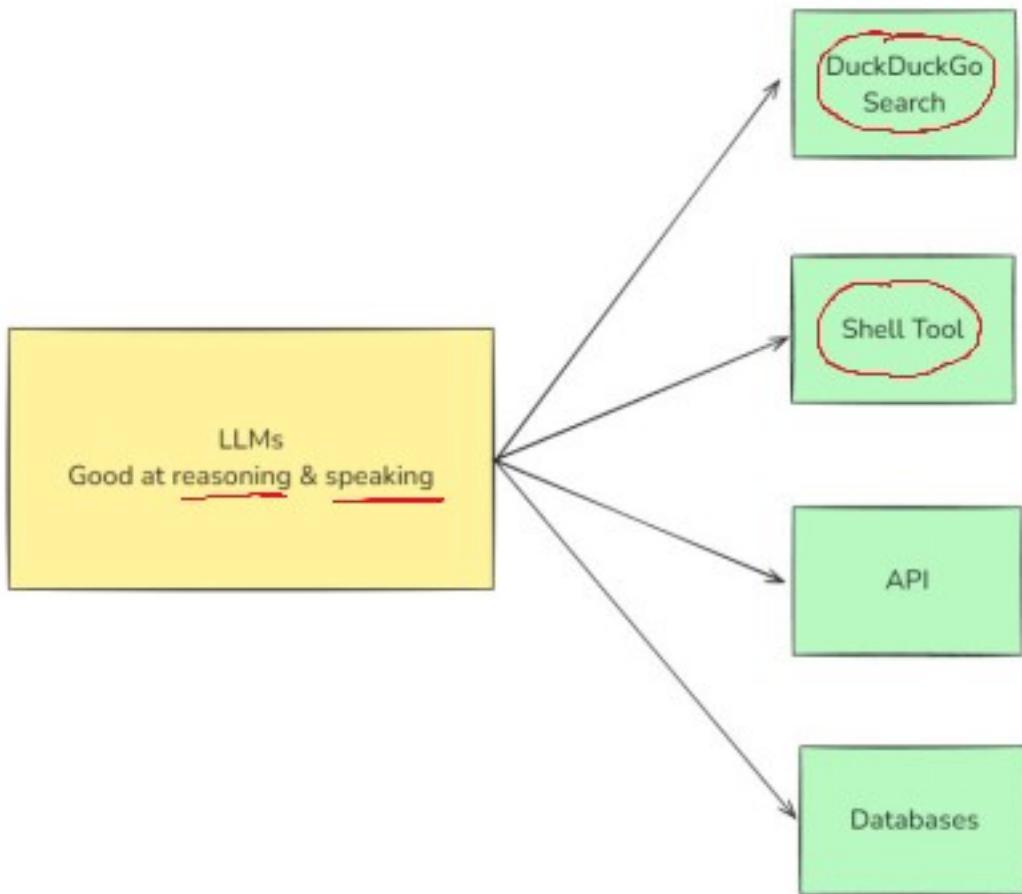
Heading 2

16. Tools Page 123

Quick Revision

25 April 16:18
2025

TOOLS



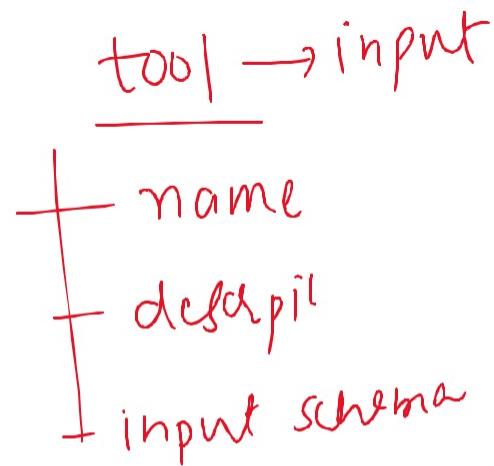
17. Tool Calling Page 124

Tool Binding

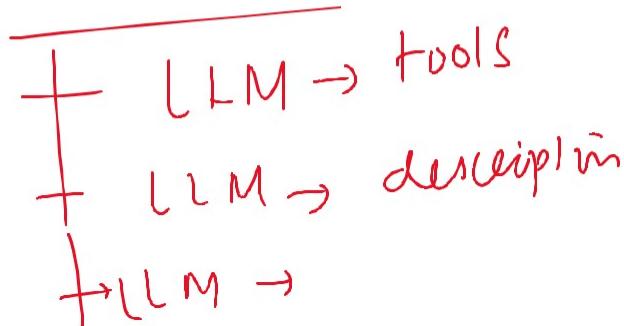
25 April 16:45
2025

Tool Binding is the step where you register tools with a Language Model (LLM) so that:

1. The LLM knows what tools are available
2. It knows what each tool does (via description)
3. It knows what input format to use (via schema)



TOOL BINDING



17. Tool Calling Page 125

Tool Calling

25 April 16:54
2025

Tool Calling is the process where the LLM (language model) decides, during a conversation or task, that it needs to use a specific tool (function) — and generates a structured output with:

- the name of the tool
- and the arguments to call it with

⚠ The LLM does not actually run the tool — it just suggests the tool and the input arguments. The actual execution is handled by LangChain or you.

Tool creation
Tool bindings

What's 8 multiplied by 7?

The LLM responds with a tool call:

json

```
{  
  "tool": "multiply",  
  "args": { "a": 8, "b": 7 }  
}
```

name: multiply
schema

17. Tool Calling Page 126

Tool Execution

25 April 17:12
2025

Tool Execution is the step where the actual Python function (tool) is run using the input arguments that the LLM suggested during tool calling.

In simpler words:

💡 The LLM says:

"Hey, call the `multiply` tool with a=8 and b=7."

⚙️ **Tool Execution** is when you or LangChain actually run:

`multiply(a=8, b=7)`

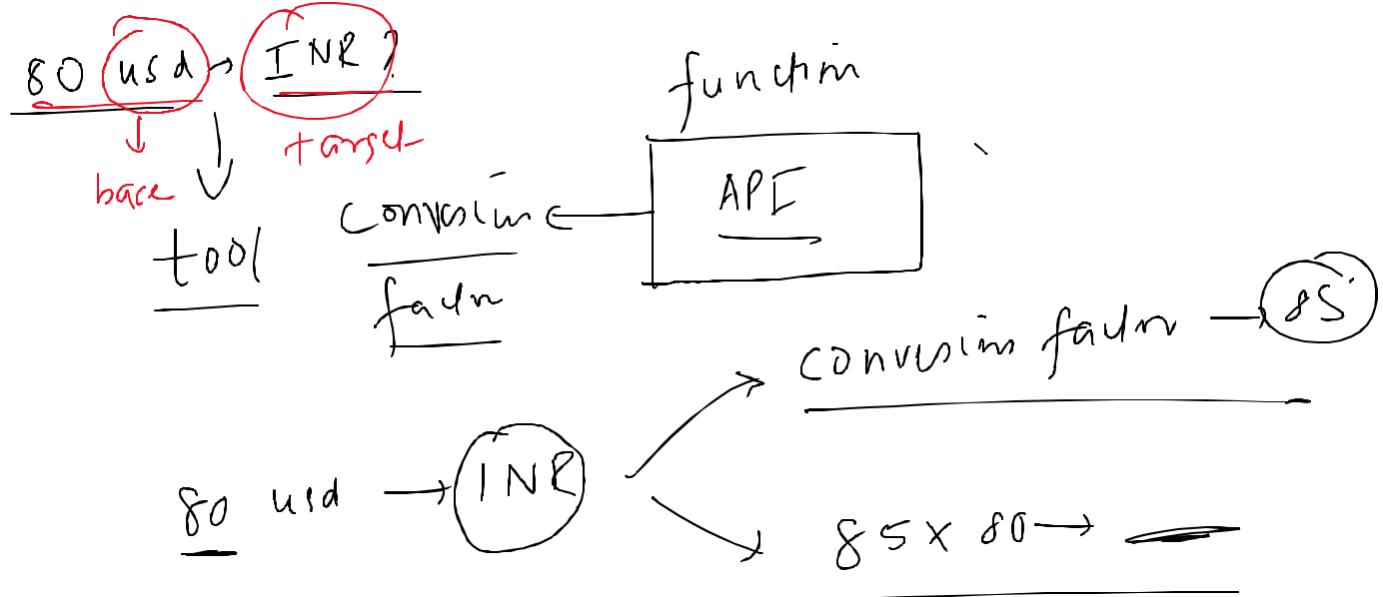
→ and get the result: 56

17. Tool Calling Page 127

Currency Conversion Tool

25 April
2025

18:02



"LLM, do not try to fill this argument." "I (the developer/runtime) will inject this value after running earlier tools."

AI Agent → tools / tool calling (X)

1. User says: "Convert 10 USD to INR."
2. LLM thinks: "I don't know the rate. First, let me call `get_conversion_factor`."
3. Tool result comes: 85.3415
4. LLM looks at result, THINKS again: "Now I know the rate, next I should call `convert` with 10 and 85.3415."
5. Tool result comes: 853.415 INR
6. LLM summarizes: "10 USD is 853.415 INR at current rate."

17. Tool Calling Page 128

What are AI Agents

29 April 2025 22:53



make my trip 25 YEARS CELEBRATION

List Your Property Grow your business! myBiz Introducing myBiz Business Travel Solution My Trips Manage your bookings

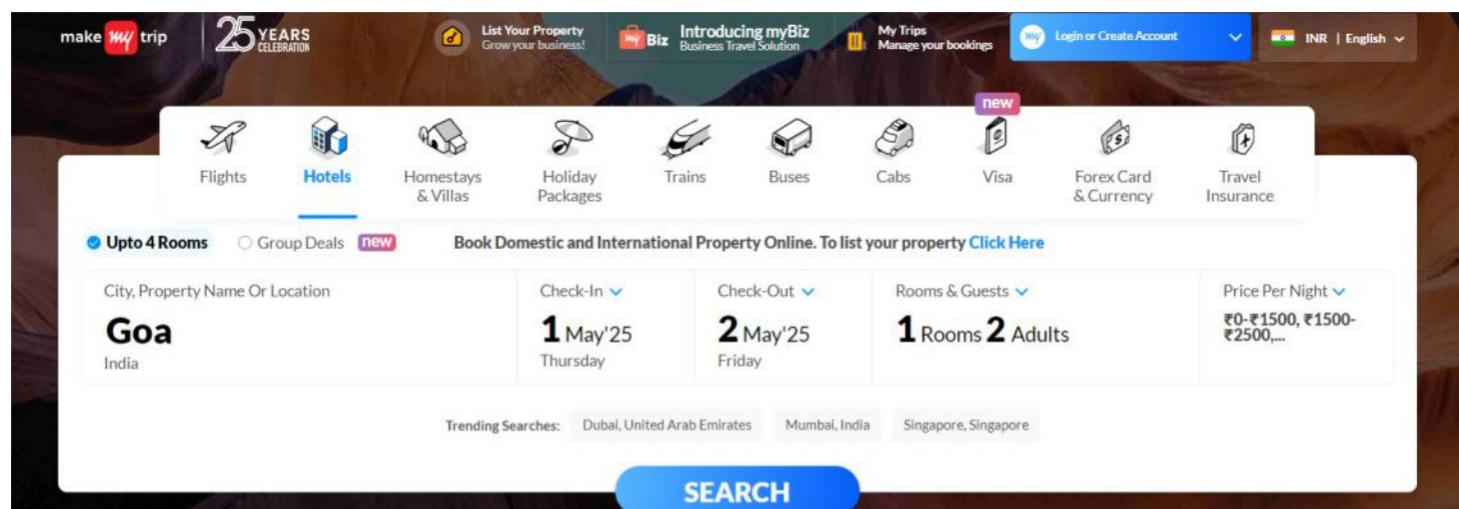
Login or Create Account INR | English

Flights Hotels Homestays & Villas Holiday Packages Trains Buses Cabs Visa Forex Card & Currency Travel Insurance

Book Train Tickets Check PNR Status Live Train Status Train Ticket Booking IRCTC Authorized e-ticketing

From: New Delhi NDLS, New Delhi Railway Station To: Kanpur CNB, Kanpur Central Travel Date: 1 May' 25 Thursday Class: ALL All Class

SEARCH



make my trip 25 YEARS CELEBRATION

List Your Property Grow your business! myBiz Introducing myBiz Business Travel Solution My Trips Manage your bookings

Login or Create Account INR | English

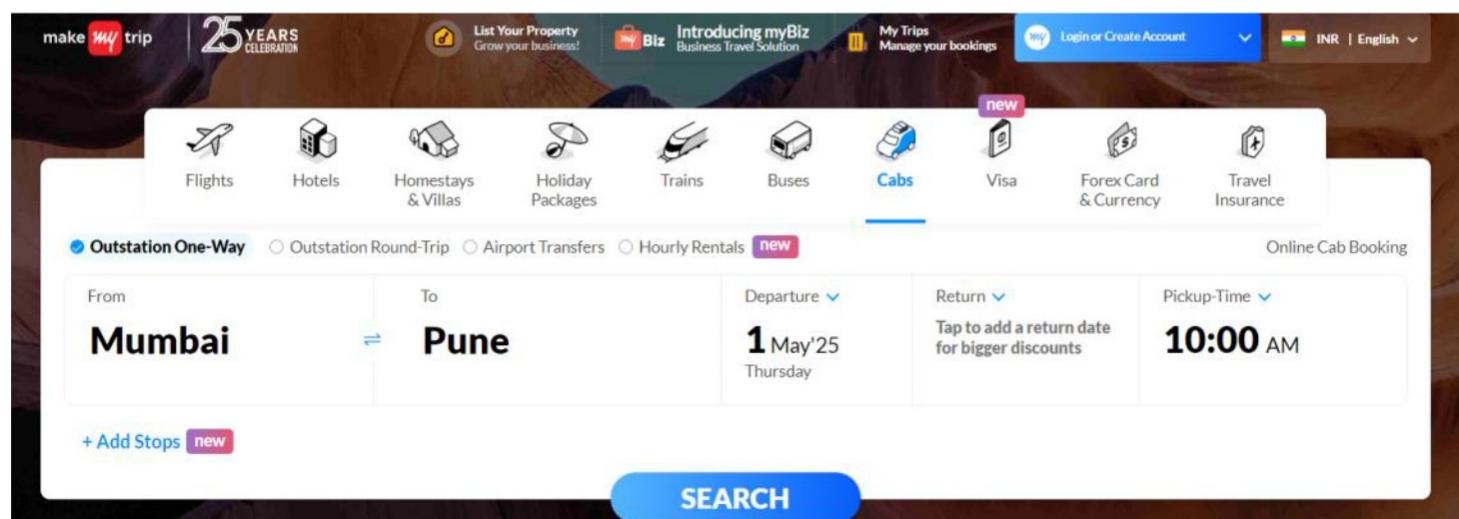
Flights Hotels Homestays & Villas Holiday Packages Trains Buses Cabs Visa Forex Card & Currency Travel Insurance

Upto 4 Rooms Group Deals Book Domestic and International Property Online. To list your property [Click Here](#)

City, Property Name Or Location: Goa, India Check-In: 1 May'25 Thursday Check-Out: 2 May'25 Friday Rooms & Guests: 1 Room 2 Adults Price Per Night: ₹0-₹1500, ₹1500-₹2500,...

Trending Searches: Dubai, United Arab Emirates Mumbai, India Singapore, Singapore

SEARCH



make my trip 25 YEARS CELEBRATION

List Your Property Grow your business! myBiz Introducing myBiz Business Travel Solution My Trips Manage your bookings

Login or Create Account INR | English

Flights Hotels Homestays & Villas Holiday Packages Trains Buses Cabs Visa Forex Card & Currency Travel Insurance

Outstation One-Way Outstation Round-Trip Airport Transfers Hourly Rentals Online Cab Booking

From: Mumbai To: Pune Departure: 1 May'25 Thursday Return: Tap to add a return date for bigger discounts Pickup-Time: 10:00 AM

+ Add Stops

SEARCH

User enters query:

"Can you create a budget travel itinerary from Delhi to Goa from 1st to 7th May?"

1. Understanding your intent

The agent internally interprets:

- Origin: Delhi
- Destination: Goa
- Dates: May 1–7
- Preference: Budget travel
- Goal: Plan complete itinerary + optimize cost

Agent thinks: *"I need to plan affordable travel, stay, local movement, and activities over 7 days."*

2. Research transport options

Agent (internally):

- Calls IRCTC/train API and Flight APIs
- Compares price, duration, and availability

Agent (to user):

"The cheapest and most available option is the Goa Express from Delhi on April 30th night (sleeper class ₹800 or 3AC ₹1500).
It arrives in Goa on May 1st evening. Would you like to go ahead?"

User selects "3AC".

3. Search for affordable stay

Agent (internally):

- Uses hotel API or scraping tools
- Filters by:
 - Price < ₹700/night
 - Close to Baga or Anjuna beach
 - User reviews

Agent (to user):

"I found a dorm room at The Hosteller, Baga for ₹650/night with good ratings. Shall I book this for 6 nights (May 1–7)?"

User agrees.

4. Plan local travel

Agent (to user):

"Scooter rentals are the most cost-effective in Goa.
Should I pre-book a scooty for ₹300/day? (~₹1800 for 6 days)"

User says yes.

5. Generate activity plan

Agent (internally):

- Calls sightseeing API / uses static local dataset
- Maps popular attractions across North & South Goa
- Builds a balanced plan:
 - Culture, relaxation, nightlife, shopping

Agent (to user):

"Here's a draft plan:

- May 2: Baga, Calangute, Aguada Fort
- May 3: Old Goa churches, Fontainhas, local lunch
- May 4: South Goa day trip: Palolem, Cabo de Rama
- May 5: Anjuna flea market + sunset at Chapora
- May 6: Free day + beach cafe dinner"

User loves it.

6. Plan return

Agent (to user):

"Return options on May 7:

1. Train (₹800–₹1500, overnight)
2. Early morning flight (₹2800, 6:00 AM via SpiceJet)

Choose your preference."

User picks train.

7. Budget summary

Agent (to user):

Item	Cost
Train (round trip)	₹3,000
Stay (6 nights)	₹3,900
Scooter + fuel	₹2,200
Food (local + budget)	₹2,500
Sightseeing & entry	₹1,000
Shopping/misc	₹1,500
Total	₹14,100

"Total trip cost: ₹14,100. Would you like to finalize the bookings?"

 User confirms.

8. Final Output

Agent (to user):

- Bookings confirmed.
 - Receipt sent to your email.
 - Calendar invite added for train departures.
 - You'll get daily reminders and local recommendations during your stay.
 - Need help anytime? Just ask!

An AI agent is an intelligent system that receives a high-level goal from a user, and autonomously plans, decides, and executes a sequence of actions by using external tools, APIs, or knowledge sources — all while maintaining context, reasoning over multiple steps, adapting to new information, and optimizing for the intended outcome.

Goal-driven	You tell the agent <i>what you want</i> , not <i>how to do it</i>
Autonomous planning	Agent breaks down the problem and sequences tasks on its own
Tool-using	Agent calls APIs, calculators, search tools, etc.
Context-aware	Maintains memory across steps to inform future actions
Reasoning-capable	Makes decisions dynamically (e.g., "what to do next")
Adaptive	Rethinks plan when things change (e.g., API fails, no data)

Building an agent in LangChain

29 April 22:

2025 53

AI Agent



Goal-driven

You tell the agent what you want, not how to do it



Autonomous planning

Agent breaks down the problem and sequences tasks on its own



Tool-using

Agent calls APIs, calculators, search tools, etc.



Context-aware

Maintains memory across steps to inform future actions



Adaptive

Rethinks plan when things change (e.g., API fails, no data)



18. Agents Page 133

Explanation

29 April 22:
2025 53

18. Agents Page 134

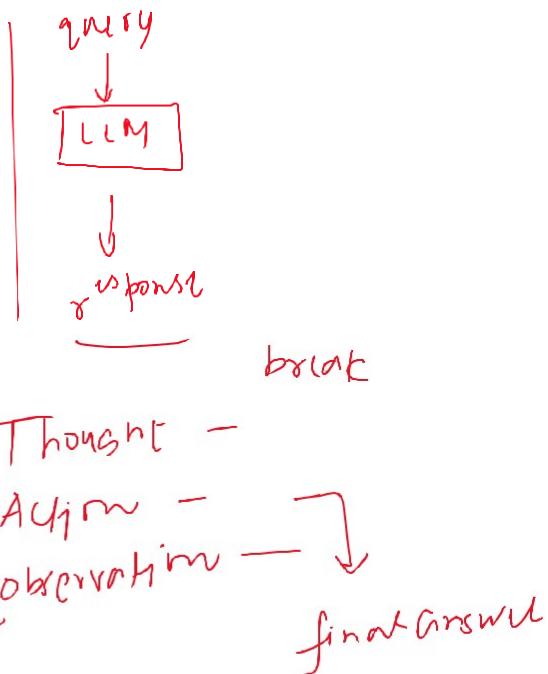
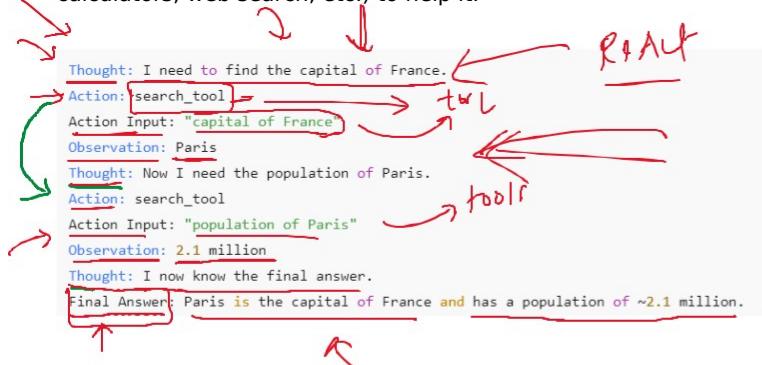
ReAct

1. ReAct

01 May 2025 14:17

ReAct is a design pattern used in AI agents that stands for Reasoning + Acting. It allows a language model (LLM) to interleave internal reasoning (Thought) with external actions (like tool use) in a structured, multi-step process.

Instead of generating an answer in one go, the model thinks step by step, deciding what it needs to do next and optionally calling tools (APIs, calculators, web search, etc.) to help it.



ReAct is useful for:

- Multi-step problems
- Tool-augmented tasks (web search, database lookup, etc.)
- Making the agent's reasoning transparent and auditable

It was first introduced in the paper:

ReAct: Synergizing Reasoning and Acting in Language Models (Yao et al., 2022)

REACT: SYNERGIZING REASONING AND ACTING IN LANGUAGE MODELS

Shunyu Yao^{*,1}, Jeffrey Zhao², Dian Yu², Nan Du², Izhak Shafran², Karthik Narasimhan¹, Yuan Cao²

¹Department of Computer Science, Princeton University

²Google Research, Brain team

¹{shunyuy, karthikn}@princeton.edu

²{jeffreyzhao, dianyu, dunan, izhak, yuancao}@google.com

ABSTRACT

While large language models (LLMs) have demonstrated impressive performance across tasks in language understanding and interactive decision making, their abilities for reasoning (e.g. chain-of-thought prompting) and acting (e.g. action plan generation) have primarily been studied as separate topics. In this paper, we explore the use of LLMs to generate both reasoning traces and task-specific actions in an interleaved manner, allowing for greater synergy between the two: reasoning traces help the model induce, track, and update action plans as well as handle exceptions, while actions allow it to interface with and gather additional information from external sources such as knowledge bases or environments. We apply our approach, named ReAct, to a diverse set of language and decision making tasks and demonstrate its effectiveness over state-of-the-art baselines in addition to improved human interpretability and trustworthiness. Concretely, on question answering (HotpotQA) and fact verification (Fever), ReAct overcomes prevalent issues of hallucination and error propagation in chain-of-thought reasoning by interacting with a simple Wikipedia API, and generating human-like task-solving trajectories that are more interpretable than baselines without reasoning traces. Furthermore, on two interactive decision making benchmarks (ALFWorld and WebShop), ReAct outperforms imitation and reinforcement learning methods by an absolute success rate of 34% and 10% respectively, while being prompted with only one or two in-context examples.

2. Agent & Agent Executor

01 May 2025 14:49



AgentExecutor **orchestrates the entire loop:**

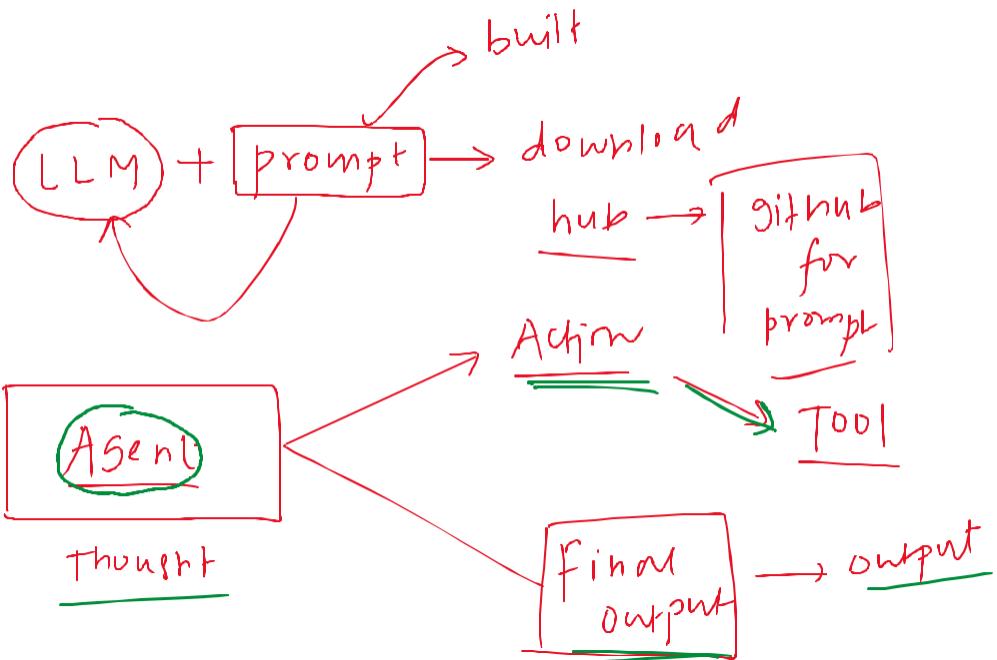
1. Sends inputs and previous messages to the agent ✓
2. Gets the next **action** from agent ✓
3. Executes that tool with provided input ✓
4. Adds the tool's **observation** back into the history ✓
5. Loops again with updated history until the agent says **Final Answer**.

3. Creating an Agent

01 May 2025 16:29

```
agent = create_react_agent(  
    llm=llm,  
    tools=[search_tool],  
    prompt=prompt  
)
```

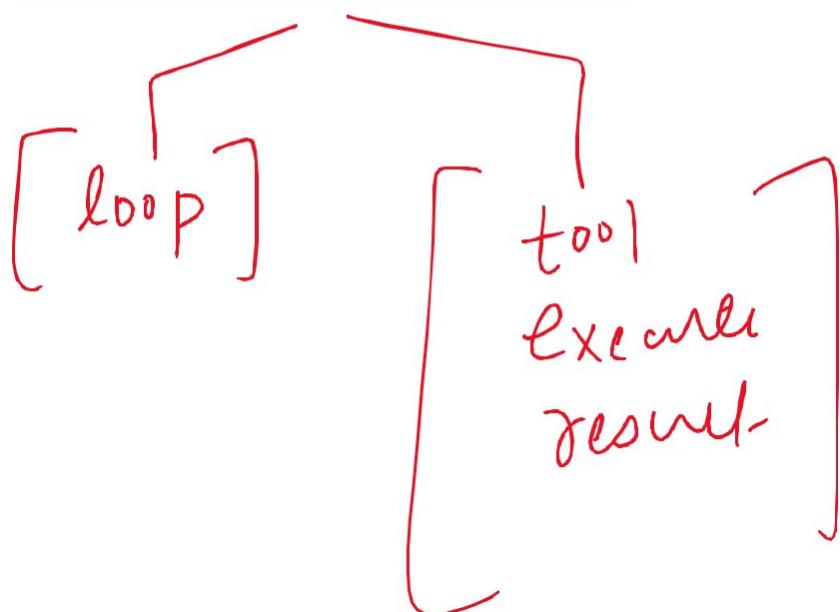
User query →
Thought →
Tool



4. Creating an Agent Executor

01 May 2025 16:30

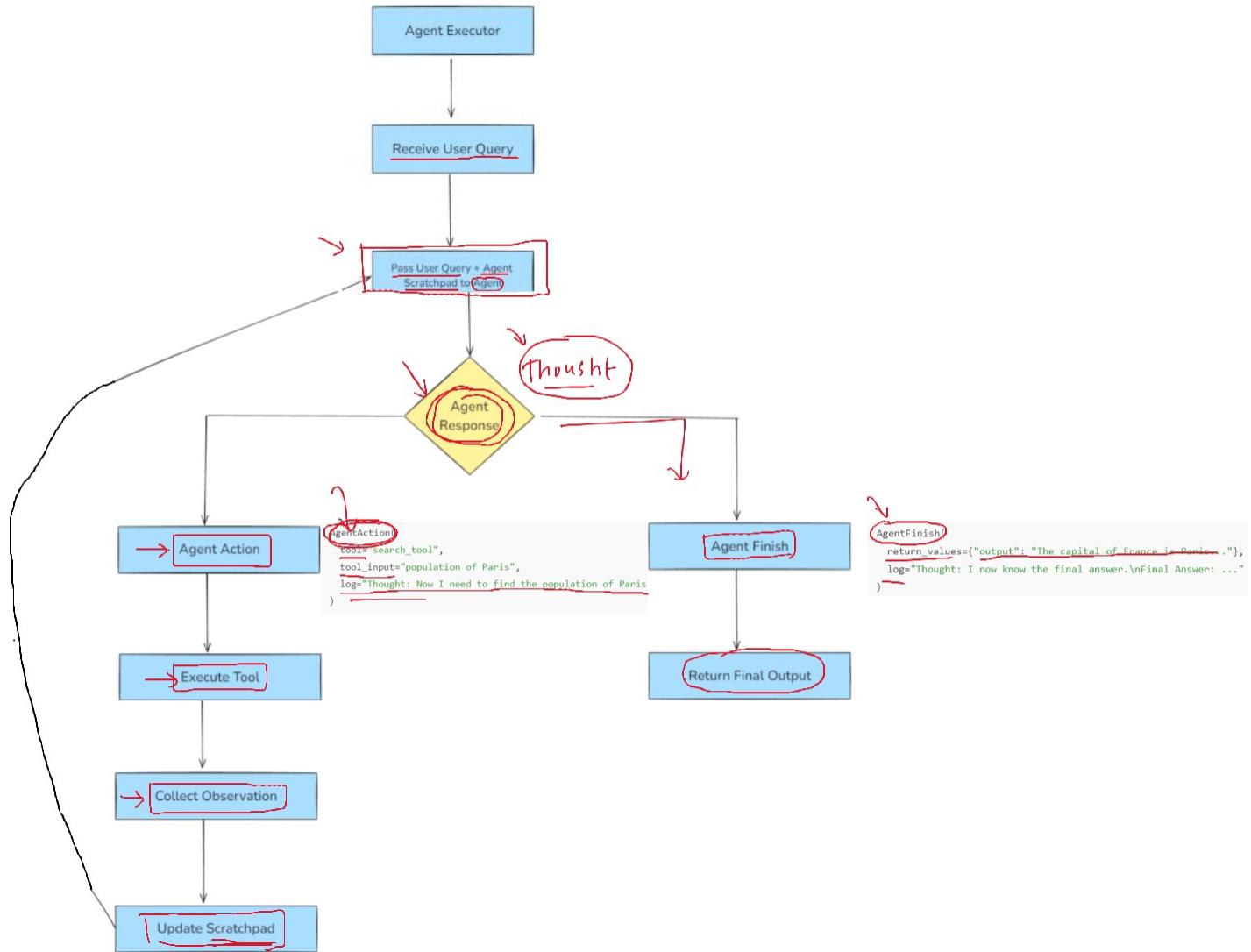
```
agent_executor = AgentExecutor(  
    agent=agent,  
    tools=[search_tool],  
    verbose=True  
)
```



18. Agents Page 138

5. Flow Chart

01 May 2025 16:30

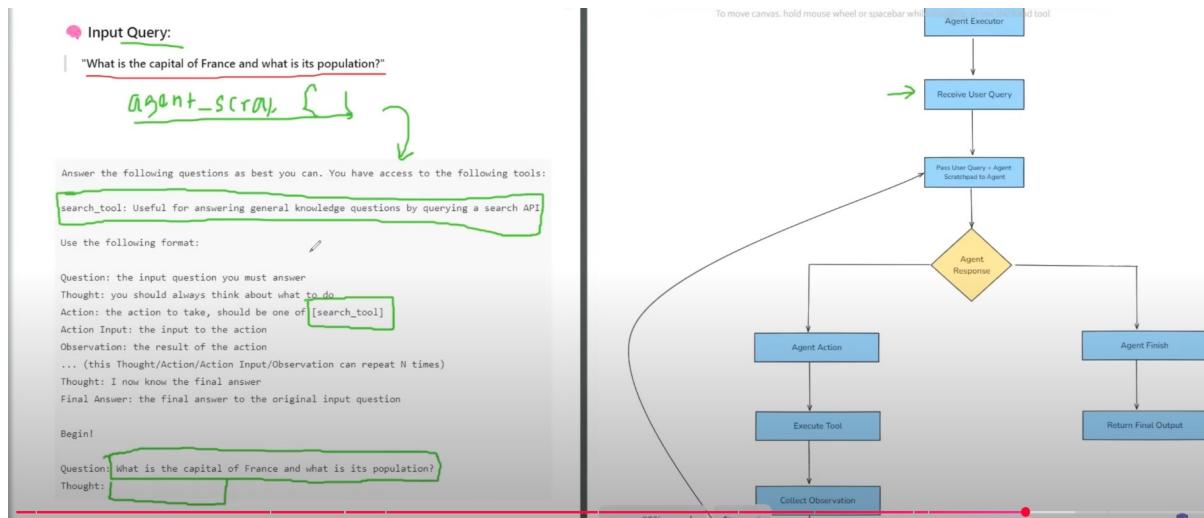


6. Example

01 May 2025 17:23

Input Query:

"What is the capital of France and what is its population?"



Agent executor receives the user query

- makes into proper query as the above
- and gives to the agent
 - agent now generates the thought
 - if there is action it creates the agent action object and executes the that action

Thought: I need to find the capital of France first.

Action: search_tool

Action Input: "capital of France"

```
AgentAction(  
    tool="search_tool",  
    tool_input="capital of France",  
    log="Thought: I need to find the capital of France first."  
)
```

observation = search_tool("capital of France")

"Paris is the capital of France."

18. Agents Page 140

Thought: I need to find the capital of France first.
Action: search_tool
Action Input: "capital of France"
Observation: Paris is the capital of France.

📌 Current state so far:

- User Input:

→ What is the capital of France and what is its population?

- Agent Scratchpad after Step 2:

text

→ Thought: I need to find the capital of France first.

Action: search_tool

Action Input: "capital of France"

Observation: Paris is the capital of France.

Answer the following questions as best you can. You have access to the following tools

search_tool: Useful for answering general knowledge questions by querying a search API.

Use the following format:

Question: the input question you must answer

Thought: you should always think about what to do

Action: the action to take, should be one of [search_tool]

Action Input: the input to the action

Observation: the result of the action

... (this Thought/Action/Action Input/Observation can repeat N times)

Thought: I now know the final answer

Final Answer: the final answer to the original input question

Begin!

Question: What is the capital of France and what is its population?

Thought: I need to find the capital of France first.

Action: search_tool

Action Input: "capital of France"

Observation: Paris is the capital of France.

Thought:



Thought: Now I need to find the population of Paris.
Action: search_tool
Action Input: "population of Paris"

AgentAction
tool="search_tool",
tool_input="population of Paris",

```
AgentAction
  tool="search_tool",
  tool_input="population of Paris",
  log="Thought: Now I need to find the population of Paris.
)
```

observation = search_tool("population of Paris")

"Paris has a population of approximately 2.1 million."

```
Thought: I need to find the capital of France first.
Action: search_tool
Action Input: "capital of France"
Observation: Paris is the capital of France.
Thought: Now I need to find the population of Paris.
Action: search_tool
Action Input: "population of Paris"
Observation: Paris has a population of approximately 2.1 million.
```

Answer the following questions as best you can. You have access to the following tools:

search_tool: Useful for answering general knowledge questions by querying a search API.

Use the following format:

```
Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [search_tool]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question
```

Begin!

```
Question: What is the capital of France and what is its population?
Thought: I need to find the capital of France first.
Action: search_tool
Action Input: "capital of France"
Observation: Paris is the capital of France.
Thought: Now I need to find the population of Paris.
Action: search_tool
Action Input: "population of Paris"
Observation: Paris has a population of approximately 2.1 million.
Thought:
```

Thought: I now know the final answer.

Final Answer: Paris is the capital of France and has a population of approximately 2.1 million.

Thought: I now know the final answer.

Final Answer: Paris is the capital of France and has a population of approximately 2.1 million.

```
AgentFinish()  
    return_values={"output": "The capital of France is Paris..."},  
    log="Thought: I now know the final answer.\nFinal Answer: ..."  
)
```

```
{  
    "output": "Paris is the capital of France and has a population of approximately 2.1 million."  
}
```

