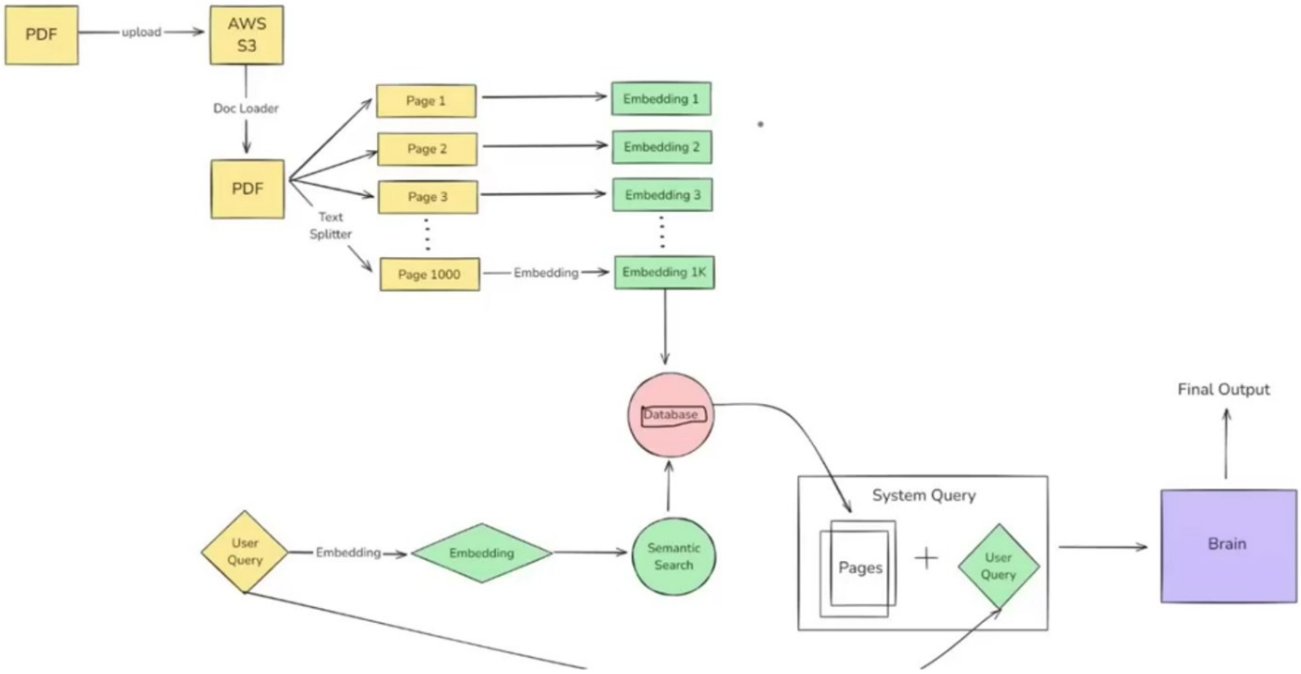
What is LangChain?

LangChain is an open-source framework for developing applications powered by LLMs.



Benefits

Concept of chains

Model Agnostic Development

Complete ecosystem

Memory and state handling

Use cases

Conversational Chatbots

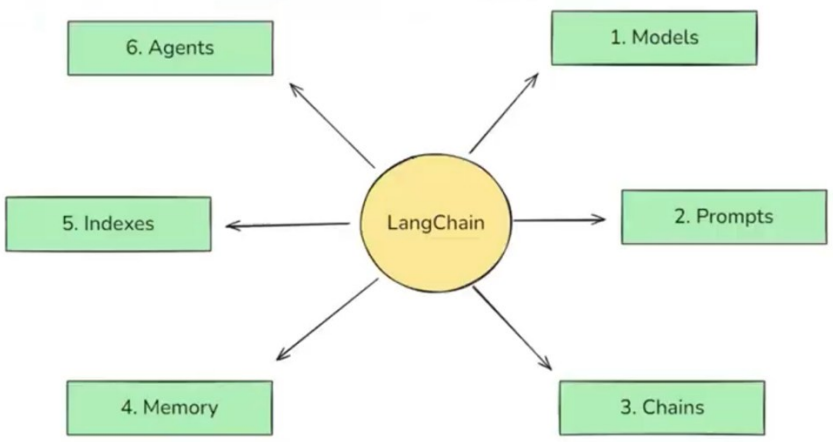
AI Knowledge Assistants

AI Agents

Workflow Automation

Summarization/Research Helpers

Components of LangChain



1. Model
   In LangChain, **models** are the core interfaces through which we can interact with AI models.

Quick starter codes:

```python
from openai import OpenAI
client = OpenAI()

response = client.responses.create(
    model="gpt-4.1",
    input="Write a one-sentence bedtime story about a unicorn."
)

print(response.output_text)
```

```python
from openai import OpenAI
client = OpenAI()

completion = client.chat.completions.create(
  model="gpt-3.5-turbo",
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Hello!"}
  ]
)

print(completion.choices[0].message)
```

```python
import anthropic

client = anthropic.Anthropic()

messages = [
    {"role": "user",
    "content": "Hello!"},
]

response = client.messages.create(
    model="claude-3-haiku-20240307",
    max_tokens=1024,                # MUST
    messages=messages,
    temperature=0.7,
    system="You are a helpful assistant."
)

# print(response.content)
print(next((block.text for block in response.content
            if block.type == "text"), None))
```

After LangChain:

```python
from langchain_openai import ChatOpenAI
from dotenv import load_dotenv

load_dotenv()

model = ChatOpenAI(model='gpt-4', temperature=0)

result = model.invoke("Now divide the result by 1.5")

print(result.content)
```

```python
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)
```

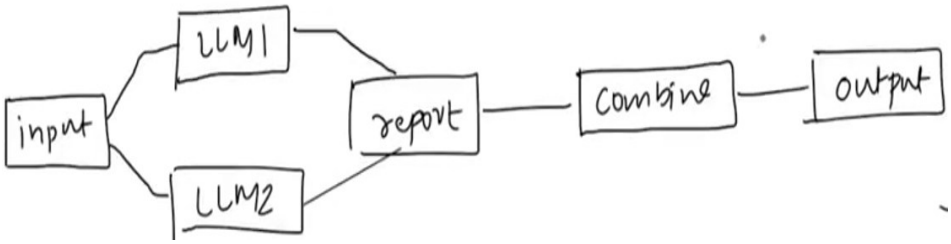
   A. Language Models: (Hugging Face is open-source model provider)
      a. LLMs (from langchain_openai import OpenAI) a str -> a str
      b. Chat Models (from langchain_* import Chat*) sequence of messages -> messages
   B. Embedding Models: (from langchin_* import *Embeddings) text -> numbers as vector embeddings

2. Prompts
   A. Dynamic and Reusable Prompts
   B. Role-Based Prompts
   C. Few-Shot Prompts

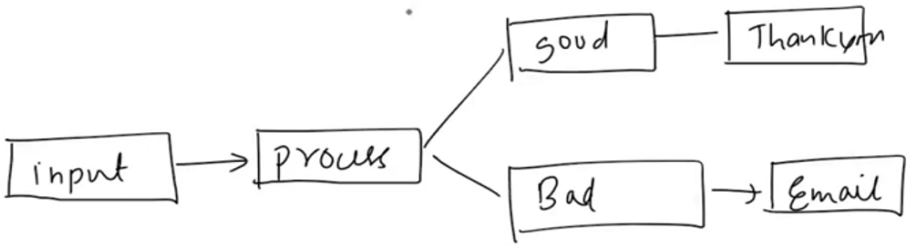
3. Chains

   Chains are complex pipelines, simply by using **|** notation. Like -

   A. Parallel Chain (like In-Depth Research Paper Generation)



   B. Conditional Chain (like AI Agent feedback)



4. Index

   Indexes connects our application to external knowledge sources (like PDF, db, website)

   A. Doc Loader
   B. Text Splitter
   C. Vector Store (db for embeddings)
   D. Retrievers

5. Memory

   Pure LLM calls are stateless (each request is independent)

   A. ConversationBufferMemory: Stores a transcript of recent messages. Great for short chats but can grow large quickly.
   B. ConversationBufferWindowMemory: Only keeps the last N interactions to avoid excessive token usage. Older important information can be missed.
   C. SummarizerBasedMemory: Periodically summarizes older chat segments to keep a condensed memory footprint.
   D. CustomMemory: For advanced use cases, in a custom memory class we can store specialized state. Like – I am a software engineer, give concise python code snippets.

6. AI Agents

   ChatBots / LLMs can only think and talk. Agents can do the work.
   LLMs only have brain and mouth but Agents also have hands.
   Agents have - Reasoning Capability and Tools (like hit api get data)

   Example – Multiply today's Kolkata's temperature with 3
   Process –
   [Available Tools – Calculator , WeatherAPI]
   User asks AI Agent -> reasoning by like - Chain of Thoughts -> hits WeatherAPI -> gets data
   -> checks for tools like Calculator -> calls calculator with params -> returns final output

1. MODEL:

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

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

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

Open Source Models – like from 🤗 Hugging Face , 🦙 Ollama Store

Ways to use OS Models –

A. Using HF Inference API (Free tier available)
B. Running Locally

Disadvantages

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

---

2. PROMPT: (Structured Input)

Example of dynamic prompt

```
Please summarize the research paper titled "{paper_input}" with the following
specifications:
Explanation Style: {style_input}
Explanation Length: {length_input}
1. Mathematical Details:
   - Include relevant mathematical equations if present in the paper.
   - Explain the mathematical concepts using simple, intuitive code snippets
where applicable.
2. Analogies:
   - Use relatable analogies to simplify complex ideas.
If certain information is not available in the paper, respond with: "Insufficient
information available" instead of guessing.
Ensure the summary is clear, accurate, and aligned with the provided style and
length.
```

Prompt Template

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?

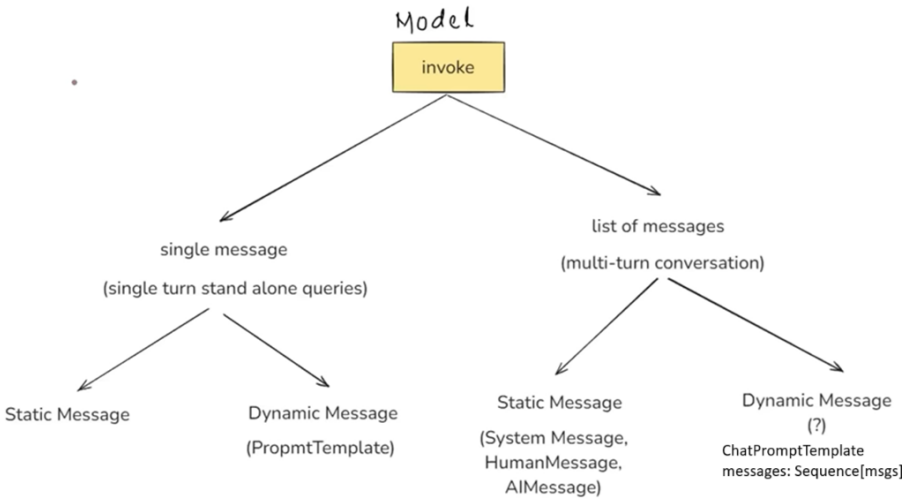
1. Default validation
2. Reusable
3. LangChain Ecosystem

Single-turn dynamic prompting using PromptTemplate (also have params like partial_variables as dict)

```python
from langchain_core.prompts import PromptTemplate, load_prompt
...
product1 = input()   # or choose from List by steamlit dropdown etc

template = PromptTemplate(template="""
I want you to act as a naming consultant for new companies.
What is a good name for a company that makes {product}?""", input_variables=['product'], validate_template= True)

prompt = template.invoke({'product':product1})
result = model.invoke(prompt)
print(result.content)

# OR
chain = template | model
result = chain.invoke({'product':product1})
print(result.content)
```

With above problems are –

(A) Multi-turn Conversation & who told what - ChatPromptTemplate
(B) Can't recall previous convo – fix is MessagesPlaceholder



from langchain_core.prompts import PromptTemplate, load_prompt, ChatPromptTemplate, MessagesPlaceholder

from langchain_core.messages import SystemMessage, HumanMessage, AIMessage

```
chat_template = ChatPromptTemplate([
    ("system", "You are a helpful customer support agent."),
    MessagesPlaceholder(variable_name="chat_history"), # here history is shared
    ("human", "{query}")
])

# load chat history
chat_history = []
with open('chat_history.txt', 'r') as f:
    # chat_history.extend(f.readlines()) # naive way, not recommended
    # below code is verbose and manual work needed because of not-so-mature 🦜🔗 ecosystem
    for line in f:
        if 'HumanMessage' in line:
            content = line.split('content="')[1].split('"')[0]
            chat_history.append(HumanMessage(content=content))
        elif 'AIMessage' in line:
            content = line.split('content="')[1].split('"')[0]
            chat_history.append(AIMessage(content=content))
# print(chat_history)

# create prompt
prompt = chat_template.invoke({'chat_history':chat_history, 'query':'where is my order?'})
print(prompt)

# the next as is
result = model.invoke(prompt)
print(result.text)
```

Structured Output

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.

Why do we need –

(A) Data extraction (from resume fetch must-have infos to dump to database)
(B) Build API (Amazon text review to – topics covered, pros, cons, sentiment)
(C) Build Agents (Agents don't work on textual data, they need specific format as input)

Ways to get structured output – LLMs which

(A) can generate natively (with_structured_output(schema=, method=))
    Schema=
    a. TypedDict (simple and
    b. Pydantic
    c. json_schema

    Method=

    d. json_schema
    e. json_mode – Claude, Gemini
    f. function_calling - OpenAI
(B) can't generate natively (with output parser)- like TinyLlama

TypedDict:

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).

```python
from typing import TypedDict, Annotated

sample_review_input = """
The hardware is great,but the software feels bloated.
There are too many pre-installed apps that I can't remove.
Also, the UI looks outdated compared to other brands.
Hoping for a software update to fix this ."""

class Review(TypedDict):    # output format/schema
    # summary: str          # OR add context specific metadata to a type using Annotated
    summary: Annotated[str, "A brief summary of the review"]
    # sentiment: str
    sentiment: Annotated[str, "Sentiment of the review. Either negative or positive or neutral"]

structured_model = model.with_structured_output(Review) # extra
result = structured_model.invoke(sample_review_input)   # not model.invoke
print(result)                                           # not result.content
print(type(result))                                     # dictionary
print(result['sentiment'])
```

Pydantic:

Pydantic is a third-party library that builds on Python's type hints to provide powerful **data validation, parsing, and serialization** at runtime. It uses Python type annotations to define data schemas, but unlike TypedDict, it **actively enforces these types and rules when you create an instance** of a Pydantic model.

```
# schema
class Review(BaseModel):
    # key_themes: Annotated[list[str], "Write down all the key themes discussed in the review in a list"]
    key_themes: list[str] = Field( description="Write down all the key themes discussed in the review in a list")
    summary: str = Field(description="A brief summary of the review")
    sentiment: Literal["pos", "neg", "none"] = Field(description="Return sentiment of the review either negative,
    pros: Optional[list[str]] = Field(default=None, description= "Write down all the pros inside a list")
    cons: Optional[list[str]] = Field(default=None, description= "Write down all the cons inside a list")
    # name: Optional[str] = Field(default="UNKNOWN", description= "Write the name of the reviewer") # OR
    name: Annotated [ Optional[str], Field(default="UNKNOWN", description= "Write the name of the reviewer") ]


structured_model = model.with_structured_output(Review)
result = structured_model.invoke(prompt)

print(result.name)
# print(result['name']) # will not work by default, need to be converted to dict
# print(result.model_dump()['name'])
```

Json-Schema:

Platform independence needed. like backend with Python, frontend with React etc.

```
import json

# Load from file or define inline
with open("07_json_schema.json") as f:
    json_schema = json.load(f)

structured_model = model.with_structured_output(schema= json_schema, method= "json_schema")
response = structured_model.invoke(prompt)
print(response)
```

When to Use What?

Feature	TypedDict	Pydantic	JSON Schema
Basic structure	✅	✅	✅
Type enforcement	✅	✅	✅
Data validation	❌	✅	✅
Default values	❌	✅	❌
Automatic conversion	❌	✅	❌
Cross-language compatibility	❌	❌	✅

Output Parsers:

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.

```python
from langchain_core.output_parsers import (
    CommaSeparatedListOutputParser,
    ListOutputParser,
    MarkdownListOutputParser,
    NumberedListOutputParser,
    PydanticOutputParser,
    XMLOutputParser,
)
from langchain_core.output_parsers.openai_tools import (
    JsonOutputKeyToolsParser,
    JsonOutputToolsParser,
    PydanticToolsParser,
```

```python
from langchain.output_parsers.boolean import BooleanOutputParser
from langchain.output_parsers.combining import CombiningOutputParser
from langchain.output_parsers.datetime import DatetimeOutputParser
from langchain.output_parsers.enum import EnumOutputParser
from langchain.output_parsers.fix import OutputFixingParser
from langchain.output_parsers.pandas_dataframe import PandasDataFrameOutputParser
from langchain.output_parsers.regex import RegexParser
from langchain.output_parsers.regex_dict import RegexDictParser
from langchain.output_parsers.retry import RetryOutputParser, RetryWithErrorOutputParser
from langchain.output_parsers.structured import ResponseSchema, StructuredOutputParser
from langchain.output_parsers.yaml import YamlOutputParser
```

Paser.parse() or parser.invoke()

StrOutputParser:

Parses only string content from response of ChatModel. No need to extract via `result.content`

It's actually helpful if output of a model needs to be used as input of another model.

```python
parser = StrOutputParser()


# --- Simple, sequential execution ---

# Step 1: Generate the prompt for the detailed report
prompt1_input = template1.invoke({'topic': 'alpha centauri 200'})

# Step 2: Get the detailed report from the model
response1 = model.invoke(prompt1_input)

# Step 3: Parse the detailed report # FYI - Use .content to get the string from AIMessage for ChatModels
detailed_report = parser.parse(response1.content)

# Step 4: Generate the prompt for the summary, using the detailed report
prompt2_input = template2.invoke({'text': detailed_report})

# Step 5: Get the 5-point summary from the model
response2 = model.invoke(prompt2_input)

# Step 6: Parse the 5-point summary
result = parser.invoke(response2.content) # preferred, OR
# result = parser.parse(response2.content)

# OR recommended →
chain = template1 | model | parser | template2 | model | parser
result = chain.invoke({'topic':'alpha centauri 200'})

print(result)
```

JsonOutputParser: Output will be dictionary like

```
parser = JsonOutputParser()
# parser = XMLOutputParser()
# parser -> SimpleJsonOutputParser PydanticOutputParser XMLOutputParser, MarkdownListOutputParser etc etc

template = PromptTemplate(
    template='Give me 3 mind-blowing facts about {topic} \n {format_instruction}',
    input_variables=['topic'],
    partial_variables={'format_instruction': parser.get_format_instructions()} # before runtime
)

chain = template | model | parser

result = chain.invoke({'topic':'UY Scuti'})

print(result)
```

StructuredOutputParser:

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.

Disadvantage:

No data validation. In the schema we can define datatype

```
schema = [ResponseSchema(name=…, description=… type="string"), … ]
parser = StructuredOutputParser.from_response_schemas(schema)
...
```

but does not guarantee to be returned in that exact format.

PydanticOutputParser:

🔹 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.

🚀 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.

```python
class Person(BaseModel):

    name: str = Field(description='Name of the person')
    age: int = Field(gt=18, description='Age of the person')
    city: str = Field(description='Name of the city the person belongs to')

parser = PydanticOutputParser(pydantic_object=Person) # pass pydantic schema class-name
```

...

For more flexibility we can use RunnablePassthrough (check notebook)

3. Chains:

View complete chain by - `chain.get_graph().print_ascii()`

a) Simple Chain:

```python
chain = prompt | model | parser
result = chain.invoke({'topic':'cricket'}) # previously it was .run()
print(result)
```

b) Sequential

```python
chain = prompt1 | model | parser | prompt2 | model | parser
result = chain.invoke({'topic': 'Unemployment in India'})
```

c) Parallel

```python
from langchain.schema.runnable import RunnableParallel
parallel_chain = RunnableParallel({
    'notes': prompt1 | model1 | parser,
# these keys => labels/keys for respective output when invoked
    'quiz': prompt2 | model2 | parser
})
merge_chain = prompt3 | model1 | parser
chain = parallel_chain | merge_chain
```

d) Conditional

```python
from langchain.schema.runnable import RunnableParallel, RunnableBranch,
RunnableLambda
sentiment_chain = template_initial_ip | model_openai | parser_for_sentiment
                        (here output is either positive or negative)



# below is like
# if condition1: chain1
# elif condition2: chain2
```

```
# else default_runnable
branch_chain = RunnableBranch(
    (lambda x:x.sentiment == 'positive', template_pos_ip | model_huggingface | str_parser),
    (lambda x:x.sentiment == 'negative', template_neg_ip | model_anthropic   | str_parser),
    RunnableLambda(lambda x: "could not find sentiment")              )

chain = sentiment_chain | branch_chain
```

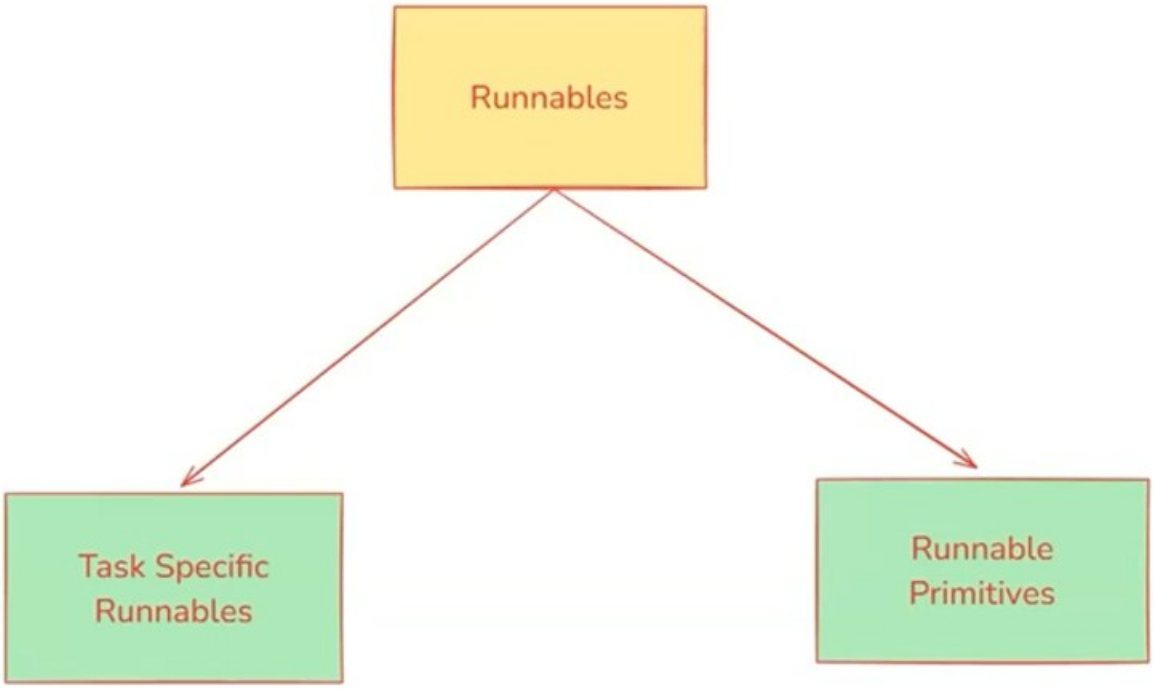
- **Runnables:** (like Lego blocks)

Unit of work - Input, process, output

Common interface – invoke, batch, stream

Connect sequentially, so that one's output, is automatically other's input.

Multiple joined runnable can also act as single runnable.



Task Specific Runnable: These are core LangChain components that have been converted into Runnable 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.

Runnable Primitives: 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** (I 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 function** into Runnable.

RunnablePassthrough - Just forwards input as output (acts as a placeholder).

- **Memory** : In LangGraph Section

RAG (Retrieval Augmented Generation):

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

Components of RAG:

1. Document Loader (from BaseLoader -> like TextLoader, PyPDFLoader, WebBaseLoader, CSVLoader)
2. Text Splitter
3. Vector Database
4. Retriever

Document loader

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. Loads document as a list/generator of Document Object

Each obj looks like ->

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

`load()`

- **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.

`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.

```python
from langchain_community.document_loaders import PyPDFLoader
loader = PyPDFLoader('input_files/sample-pdf.pdf')
docs  = loader.load()
docs2 = loader.lazy_load()
print(docs) # list of p doc objects (p= no of pages)

print(docs[0].page_content) # this can be used while chain/prompt invoking
print(docs[0].metadata) # total_pages etc
```

```
{'producer': 'Skia/PDF m131 Google Docs Renderer', 'creator': 'PyPDF', 'creationdate': ''
, 'title': 'Deep Learning Curriculum', 'source': 'dl-curriculum.pdf', 'total_pages': 23,
'page': 1, 'page_label': '2'}
```

DirectoryLoader:

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

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

`**` = recursive search through subfolders

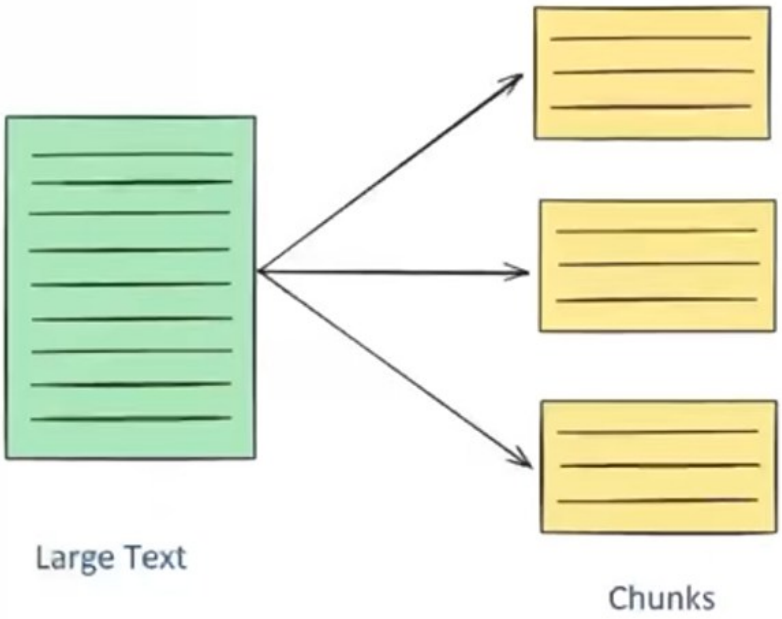
```python
from langchain_community.document_loaders import DirectoryLoader, PyPDFLoader
loader = DirectoryLoader(path=,glob=,loader_cls=PyPDFLoader)
loader.load()
loader.lazy_load()
```

Inside folder every pdf's page will be loaded one after another.

Metadata -> total_page: all combined page no , page: current pdf page no

Text Splitting:

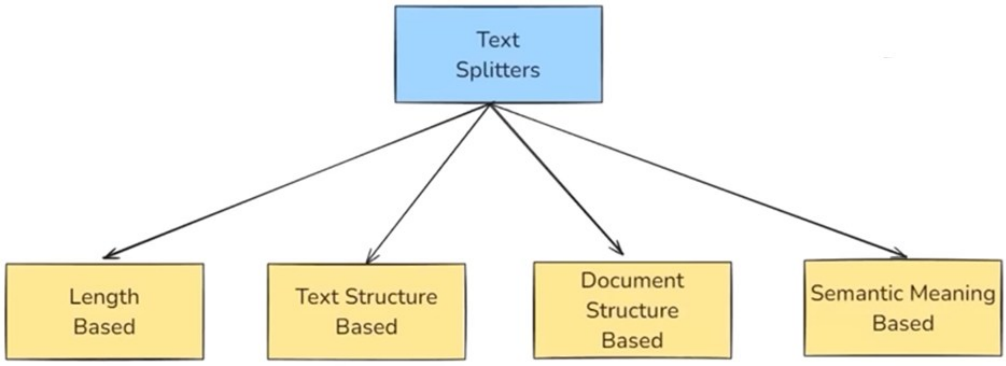
Text Splitting is the process of breaking large chunks of text (like articles, PDFs, HTML pages, or books) into smaller, manageable pieces (chunks) that an LLM can handle effectively.



- **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.



1. Length Based – CharacterTextSplitter:

```
from langchain.text_splitter import CharacterTextSplitter
text ="I am susamay kumbhakar a data scientist with 4 years of experience in python"
splitter= CharacterTextSplitter(chunk_size=25, chunk_overlap=3, separator='') # 15% is a good overlap
chunks = splitter.split_text(text)
print(chunks)

# splitter.split_documents(docs) # for document objs
```

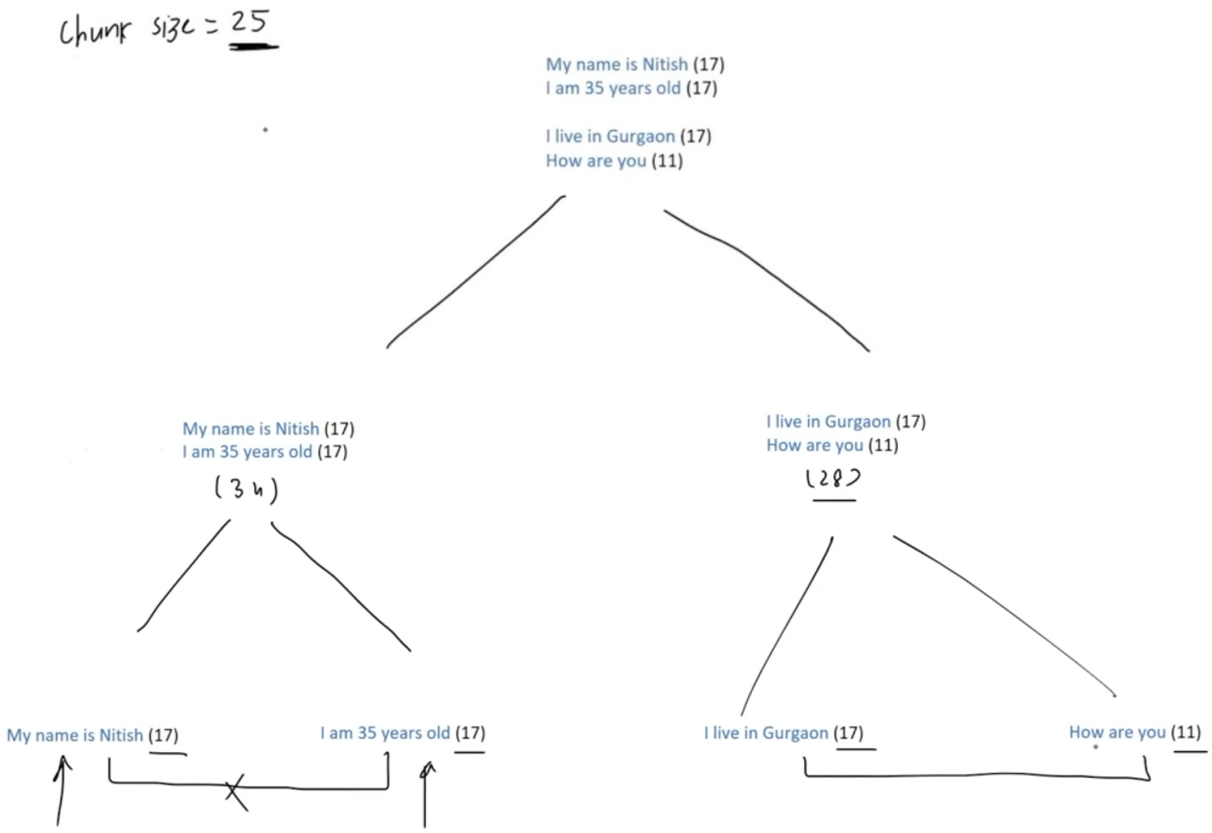
✓ 1.6s

```
['I am susamay kumbhakar a', 'a data scientist with 4', '4 years of experience in', 'in python']
```

---

2. text structure based - RecursiveCharacterTextSplitter:

Tries hierarchical splitting ->like paragraph - \n\n, line - \n, word - ' ', character - ''



```python
from langchain.text_splitter import RecursiveCharacterTextSplitter
splitter = RecursiveCharacterTextSplitter(chunk_size=25, chunk_overlap=3) # max 25 characters, optimize smaller to near 25
```

3. document structure based - Recursive Character Text Splitter:

(Diff format - like markdown or python file) - special case of previous

```python
from langchain.text_splitter import RecursiveCharacterTextSplitter, Language # can also use MarkdownTextSplitter
text= "a python full code"
splitter = RecursiveCharacterTextSplitter.from_language(language=Language.PYTHON, chunk_size=300, chunk_overlap=0)
chunks = splitter.split_text(text)
print(chunks)
```

4. Semantic Text Splitter:

Let's say given 5 sentences: S1, S2, S3, S4, S5

1. Generate embeddings for consecutive pairs: (S1-S2), (S2-S3), (S3-S4), (S4-S5)
2. Compute cosine similarities for each pair → get 4 similarity scores
3. Calculate mean and standard deviation (SD) of these scores
4. If a similarity drops significantly (e.g., below mean − SD), insert a split between those sentences

```python
from langchain_experimental.text_splitter import SemanticChunker
from langchain_openai.embeddings import OpenAIEmbeddings

splitter = SemanticChunker( OpenAIEmbeddings(), breakpoint_threshold_type="standard_deviation",  # percentile/fixed
                            breakpoint_threshold_amount = 1 # allow upto 1 sd, if more than that → split
)
docs = splitter.create_documents([text1]) # the context of text1 and text2 would be treated as different and independent
docs
```

✓ 43.7s

```
[Document(metadata={}, page_content='\nThe ancient forest, a silent guardian of forgotten lore, whispered tales of old. Towe
 Document(metadata={}, page_content='Skyscrapers pierced the clouds, monuments to human ambition and tireless innovation. Ba
```

Every splitter obj has these methods ->

split_text(text): Splits a single string into text chunks.

split_documents(docs): Splits a list of LangChain Document objects (preserving metadata) into smaller Document chunks.

create_documents([sample]): Converts a list of raw strings (with optional metadatas) into Document objects and then splits them into chunks.

Vector Store -

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

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

Vector Stores in LangChain

05 April 2025 17:41

- **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=...)`

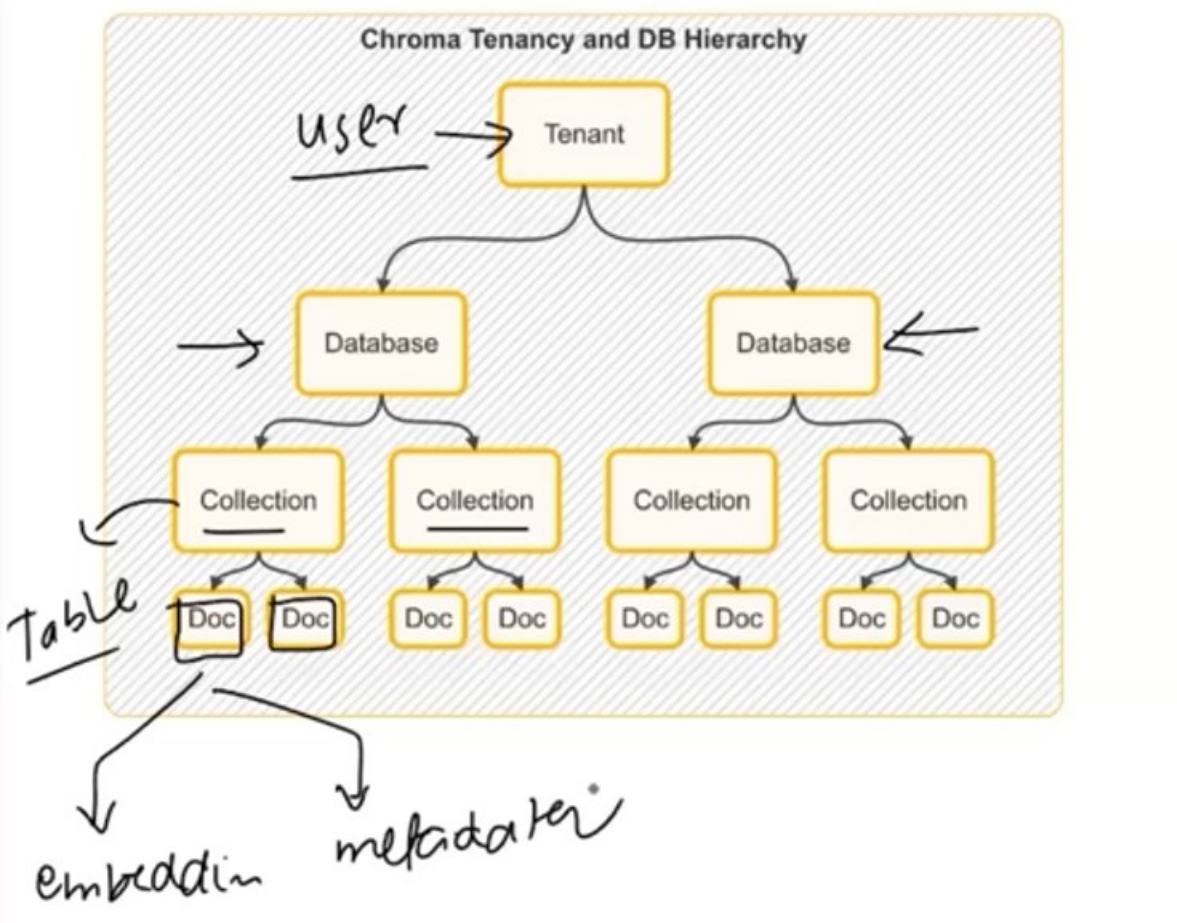
Metadata-Based Filtering

Quick Summary Table

DB	Local	Cloud	Hybrid	Notes
Chroma	✅	🚫	🚫	Simple, LangChain-native
FAISS	✅	🚫	🚫	Fast but lacks persistence
Annoy	✅	🚫	🚫	Good for read-heavy
ScaNN	✅	🚫	🚫	Fast, complex setup
Pinecone	🚫	✅	🚫	Fully managed only
Qdrant	✅	✅	✅	Easy local + cloud
Weaviate	✅	✅	✅	Modular, production-ready
Milvus	✅	✅	✅	Zilliz Cloud or local
Redis Vector	✅	✅	✅	Vector support in Redis
Vald	✅	✅	✅	Kubernetes focused

ChromaDB –

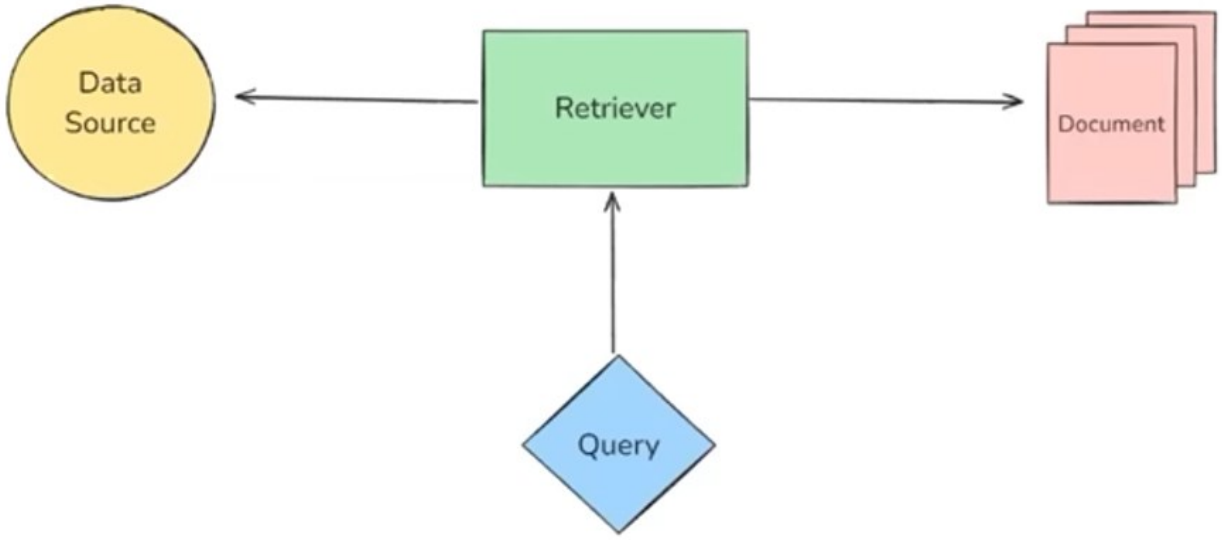
```
from langchain.vectorstores import Chroma -> depricated
from langchain_chroma import Chroma
vector_store = Chroma(
    embedding_function=OpenAIEmbeddings(),
# here shape for each vector is (1,1536), so each plotted in 1536 dim
    persist_directory='my_chroma_db',
    collection_name='sample' )
```




```
vector_store.add_documents(docs)
vector_store.similarity_search(query='Who among these are a bowler?', k=2)
vector_store.similarity_search_with_score(.,,. , filter={"team": "CSK"})

vector_store.update_document(document_id=…, document=updated_doc1)

vector_store.delete(ids=[…])
vector_store.get(include=['embeddings','documents', 'metadatas'])
```

- **Retriever:**

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 runnable.



Categories -

1. Data Source (like Wikipedia retriever, archive retriever, vector store)

2. Search Strategy (MMR - Maximum Marginal Relevance, MultiQuery, Contextual Compression etc)

Wikipedia Retriever (langchain_community.retrievers import WikipediaRetriever)

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, i.e. some logic is written, so it’s not a DocumentLoader

4. It returns them as LangChain Document objects. [Document(metadata=, page_content=), …]

Code:

Vector Store Retriever

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 - `retriever = vectorstore.as_retriever(search_kwargs={"k": 2})`

1. You store your documents in a vector store (like FAISS, Chroma, Weaviate, Qdrant)

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

Other Retrieving Methods:

```
Query (similarity search) # default case
MMR (Maximal Marginal Relevance) Retriever
```

```python
mmr_retriever = vectorstore.as_retriever(
    search_type="mmr", search_kwargs={"k": 2, "lambda_mult": 0.5} # k = top results, lambda_mult = relevance-diversity balance
)
```

Maximal Marginal Relevance (MMR)

10 April 2025 16:24

> "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

Multi-Query Retriever

```
multi_retriever = MultiQueryRetriever.from_llm(
    retriever=retriever,
    llm=llm
)
```

> 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."

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**

Merger Retriever

```
merged_retriever = MergerRetriever(retrievers=[retriever,mmr_retriever,multi_retriever,],
                                   search_kwargs={"k": 2},
                                   scoring_strategy="reciprocal_rank_fusion")  # Better dedup + scoring
```

Contextual Compression Retriever

```
compressor = LLMChainExtractor.from_llm(llm=llm)
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=merged_retriever
)
```

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:

> "What is photosynthesis?"

📄 Retrieved Document (by a traditional retriever):

> *"The Grand Canyon is a famous natural site.*
> *Photosynthesis is how plants convert light into energy.*
> *Many tourists visit every year."*

❌ 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

1. **Base Retriever** (e.g., FAISS, Chroma) retrieves N documents.
2. A **compressor** (usually an LLM) is applied to each document.
3. The compressor keeps **only the parts relevant to the query**.
4. 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

Invoke By –

```
query = "What is Chroma used for?"
results = retriever.invoke(query)
for doc in results:
    print(doc.page_content)
```

Fine-Tuning:

1. Supervised FT – (labelled dataset is provided) like prompt -> desired output. 1k~100k pairs provided.

2. Continued Pretraining – (unsupervised) like subtitle file fed to LLM (like pretraining but smaller)

3. RLHF – Reinforcement Learning on Human Feedback

Process: (supervised)

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.

It's computationally very expensive to FT a model, that's why RAG is used.

RAG:

☑ What Problems Does RAG Solve?

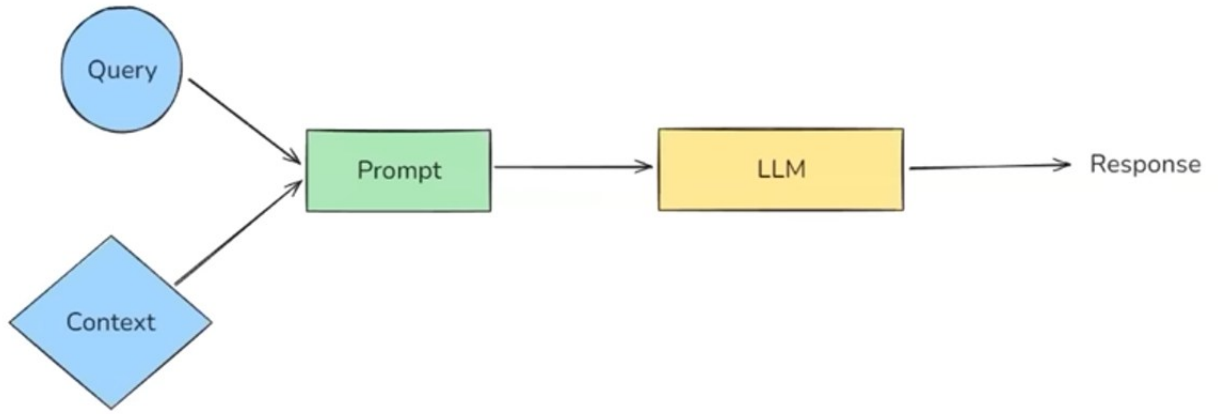
- 🔍 **Lack of knowledge beyond training** → Injects external data at runtime.

- 🧠 **Hallucinations and unreliable responses** → Grounds answers in retrieved documents.

- 💾 **Token/context limits** → Selectively retrieves only relevant chunks.

- 💸 **Expensive LLM fine-tuning** → Avoids retraining by using dynamic context injection.

- 📄 **Domain-specific/private data** → Enables LLMs to answer from your own content (PDFs, databases, wikis).



- Indexing
- Retrieval
- Augmentation
- Generation

Indexing - Indexing is the process of preparing our 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 , GitLoader etc.)

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

Why chunk?

• LLMs have context limits (e.g., 4K—32K tokens)

• Smaller chunks are more focused — better semantic search

Tools: RecursiveCharacterTextSplitter , SemanticChunker

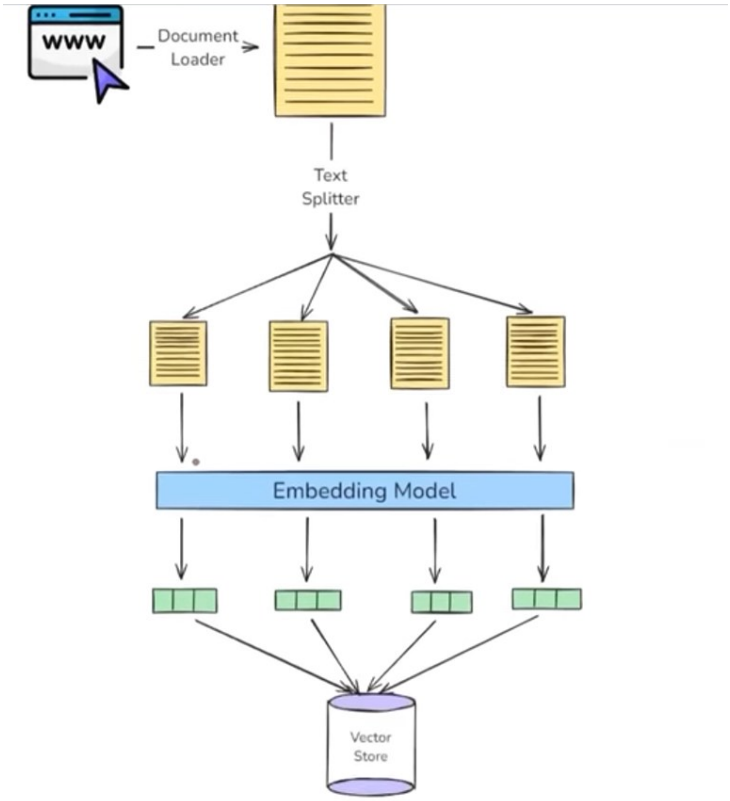
3. Embedding Generation - Convert each chunk into a dense vector (embedding) that captures its meaning.

Q. Why embeddings?

A. Similar ideas land close together in vector space; Allows fast. fuzzy semantic search

Tools: OpenAIEmbeddings, SentenceTransformerEmbeddings, InstructorEmbeddings etc.

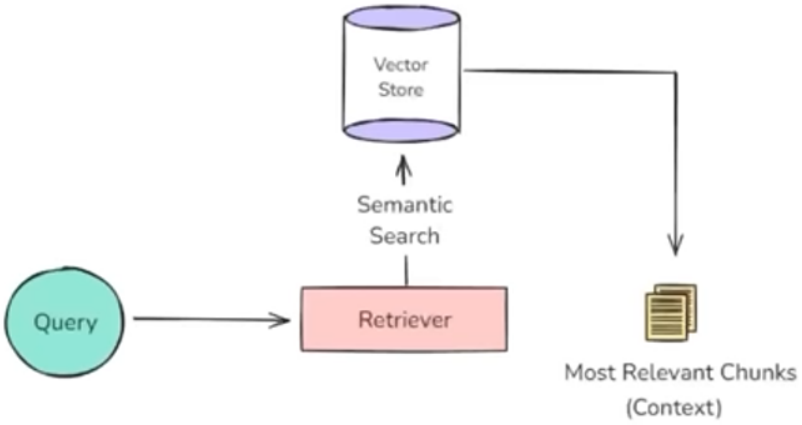
4. Storage in a Vector Store - Store the vectors along with the original chunk text + metadata in a vector database.



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.

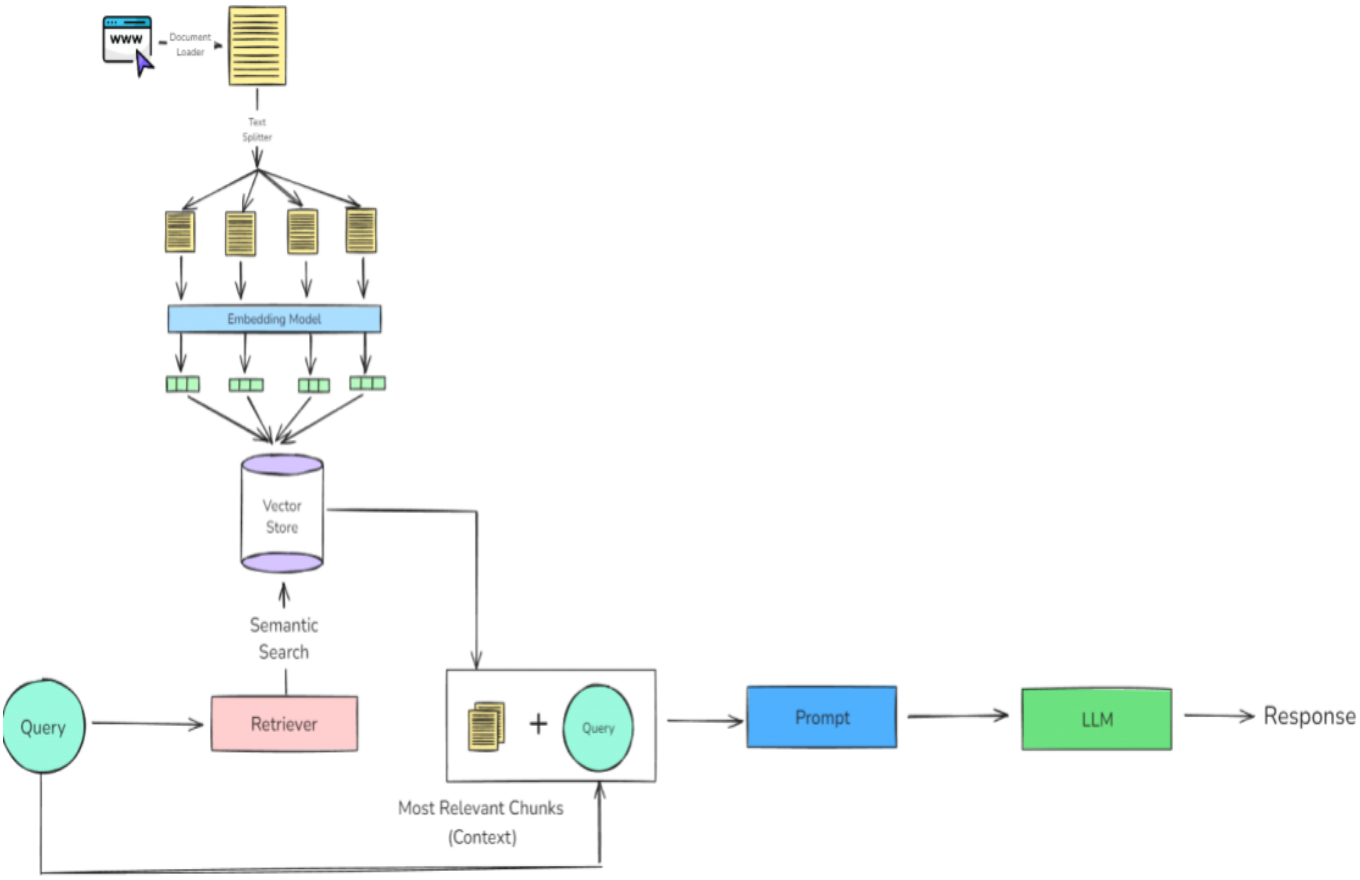
It's like asking:

• 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.

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

{context}
Question: {question}
```



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

The final RAG Chain –

```
retriever = vector_store.as_retriever(search_type="similarity",
search_kwargs={"k": 4})

def format_docs(retrieved_docs):
    context_text = "\n\n".join(doc.page_content for doc in retrieved_docs)
    return context_text

prompt = PromptTemplate(
    template="""
        You are a helpful assistant.
        Answer ONLY from the provided transcript context.
        If the context is insufficient, just say you don't know.
        {context}
        Question: {question}
    """,
    input_variables = ['context', 'question']
)

client = ChatOpenAI(model="gpt-4o-mini", temperature=0.2)

parser = StrOutputParser()

parallel_chain = RunnableParallel({
    'context': retriever | RunnableLambda(format_docs),
    'question': RunnablePassthrough()   })
main_chain = parallel_chain | prompt | client | parser
main_chain.invoke('Describe about the method of cheating for coding
interviews.')
```

Evaluation metrics of RAG based systems –

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

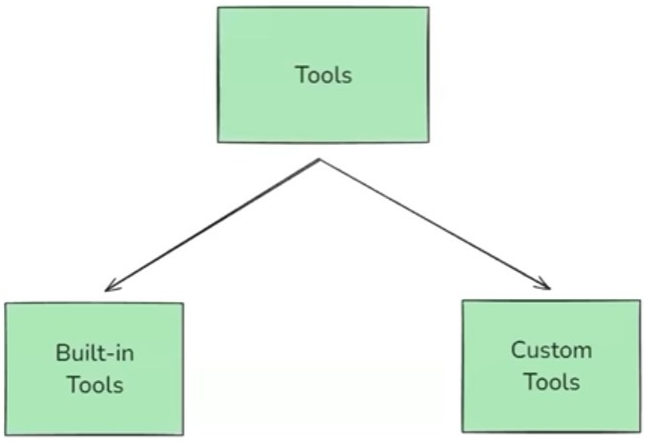
- **Tools:** A tool is just a Python function (or API) that is packaged in a way the LLM can understand and call when needed. They are also runnable (so, invoke etc. is available).

LLMs (like GPT) are great at: Reasoning (brain) & Language generation (mouth)

But they can't do things like:

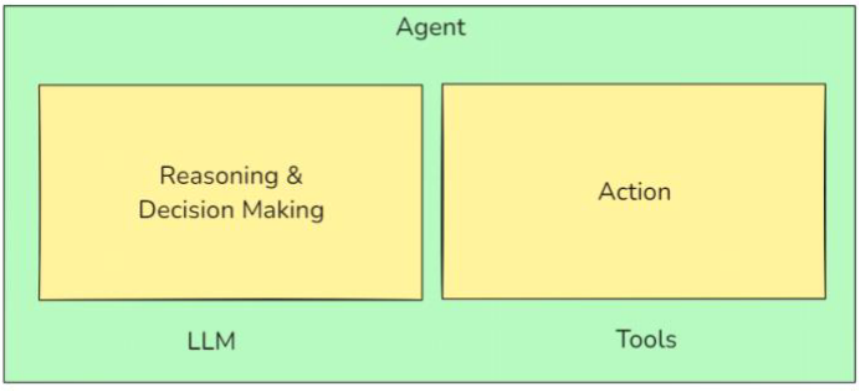
Access live data (weather, news), Do reliable math, Call APIs,

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.



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.**

```python
from langchain_community.tools import DuckDuckGoSearchRun

search_tool = DuckDuckGoSearchRun()

result = search_tool.invoke('top news in india today')

print(result) # full str
```

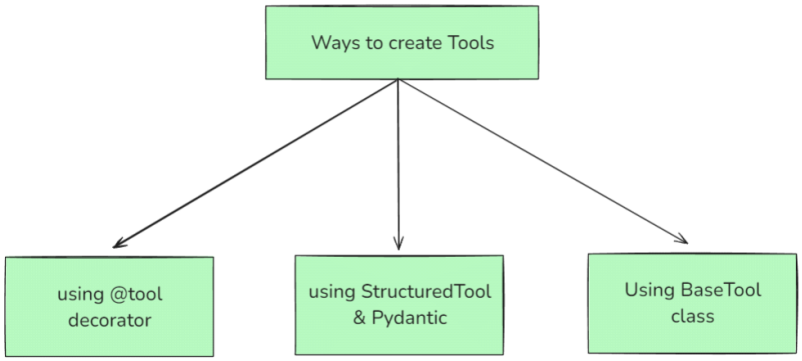
```python
from langchain_community.tools import ShellTool

terminal_tool = ShellTool()

result = terminal_tool.invoke('whoami')

print(result) # run in windows cmd
```

`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

A ***custom tool*** is a tool that you define yourself. It's basically a python function.

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**



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** decorator, StructuredTool are built on top of BaseTool.

```
@tool
def multiply(a: int, b: int) → int:
    """
    For any two given number this tool returns their mathematical multiplication
    """
    return a*b
```

OR

```
class MultiplyInput(BaseModel):
    a : int = Field(..., description="the first number to multiply")    # ... means required
    b : int = Field(..., description="the second number to multiply")

# print(MultiplyInput.model_json_schema())
```

```
def multiply_func(a:int, b:int) → int:
    return a*b
```

Then …

```python
multiply_tool = StructuredTool.from_function(
    func = multiply_func,
    name = "multiply",                          # default - func name
    description = "multiply two numbers",       # default - func docstring
    args_schema=MultiplyInput                   # pydantic class
)
```

OR

```python
class MultiplyTool(BaseTool):
    name: str = "multiply",                     # default - func name
    description: str = "multiply two numbers",  # default - func docstring
    args_schema: Type[BaseModel] = MultiplyInput # pydantic class

    def _run(self, a: int, b:int) -> int:       # only way to implement async -> def _arun
        return a*b
```

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.

All tools have few common attributes, like-

tool.run(str or dict) -> old version,

tool.invoke(,,) -> unified interface to run
tool.name # 'multiply_tool'

tool.description # 'Multiplies two numbers'

tool.args_schema # <class '__main__.MultiplyInput'> (use .model_json_schema())

tool.schema() # JSON schema of input

tool.func # multiply_func

- ❖ **Toolkits:**

A toolkit is just a collection (bundle) of related tools that serve a common purpose —

packaged together for convenience and reusability. Ex -

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

Custom Toolkit:

```
class MathToolKit:
    def get_tools(self): # lc internally calls this
        return [add,multiply]

toolkit = MathToolKit()
tools = toolkit.get_tools() # [StructuredTool(name='add', ), ...]

for tool in tools:
    print(tool.name,'⇒', tool.description)
```

Tool Execution/Use –

query = "tell me the product of 11 with ten"

```
llm_with_tools = llm.bind_tools([multiply]) # RunnableBinding
result = llm_with_tools.invoke(query) # suggest tool_calls

tool_result = multiply.invoke(result.tool_calls[0])
llm_with_tools.invoke([hum(query),aim(result),tool_result])
```

FYI – Runtime data injection –

```
@tool
# def convert(base_currency_value: float, coversion_factor: float) → float:    # concrete but we should pass in runtime
def convert(base_currency_value: float, conversion_factor: Annotated[float, InjectedToolArg]) → float:
    """
    From a given base currency value and conversion factor, this function returns the converted target currency value.
    """
    return base_currency_value * conversion_factor
```

Then injecting the data later using

```
tool_call['args']['conversion_factor'] = conversion_factor
```

AI Agents:

> 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 *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
Reasoning-capable	Makes decisions dynamically (e.g., "what to do next")
Adaptive	Rethinks plan when things change (e.g., API fails, no data)

ReAct –

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.

```
from langchain.agents import create_react_agent, AgentExecutor
from langchain import hub

prompt = hub.pull("hwchase17/react")  # pulls the standard
ReAct agent prompt

search_tool = TavilySearch()
@tool
def get_weather_data(city: str) -> str: api_response_json
llm = ChatOpenAI()
```

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

agent_executor = AgentExecutor(
    agent=agent,
    tools=[search_tool, get_weather_data],
    verbose=True
)
response = agent_executor.invoke({"input": "…"})
response['output']
```

```
Thought: I need to find the capital of France.
Action: search_tool
Action Input: "capital of France"
Observation: Paris
Thought: Now I need the population of Paris.
Action: search_tool
Action Input: "population of Paris"
Observation: 2.1 million
Thought: I now know the final answer.
Final Answer: Paris is the capital of France and has a population of ~2.1 million.
```

ReAct is useful for:

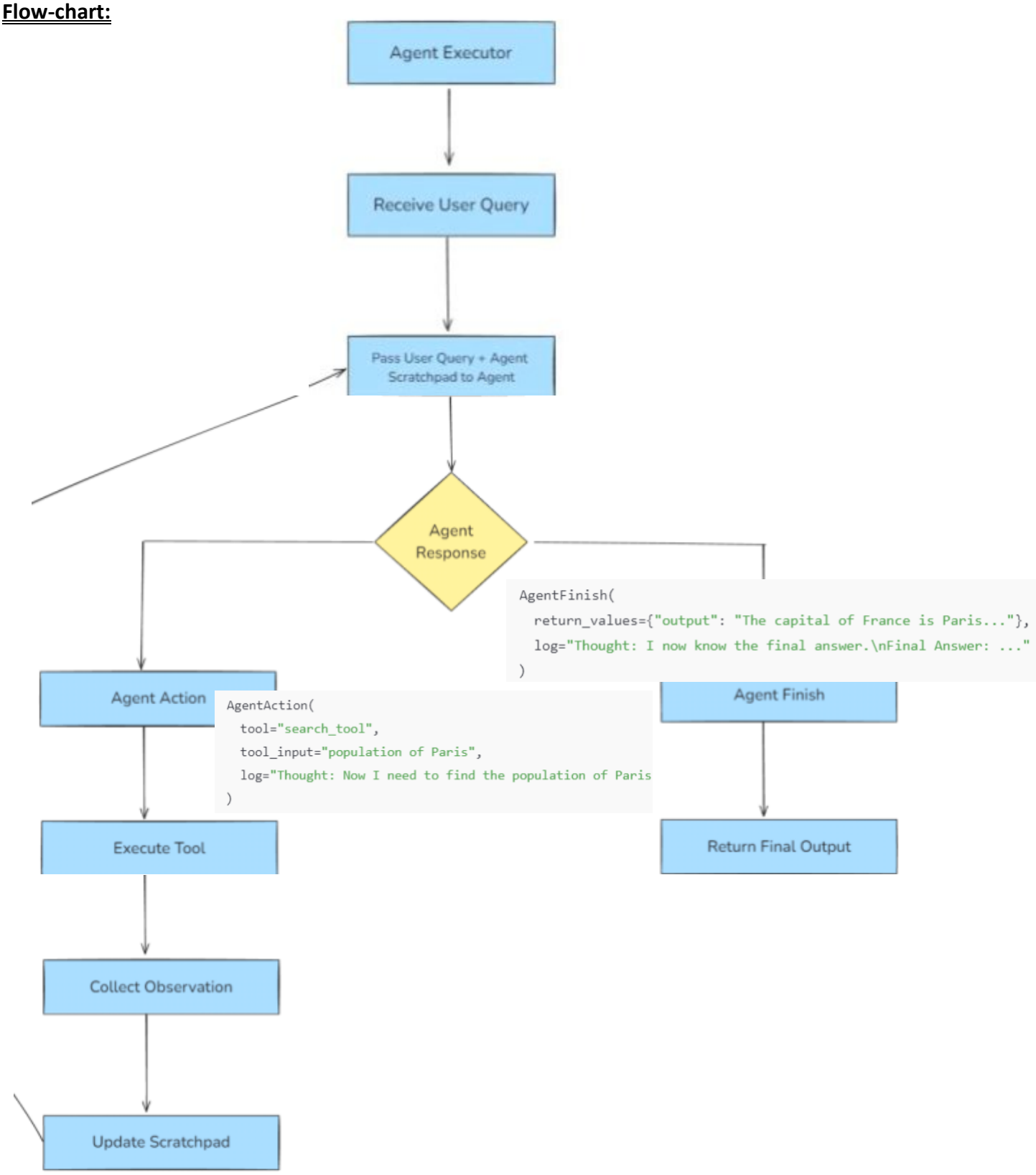
- Multi-step problems
- Tool-augmented tasks (web search, database lookup, etc.)
- Making the agent's reasoning **transparent and auditable**

Agent & Agent Executor:

`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`.

Flow-chart:


Agent Executor
Receive User Query
Pass User Query + Agent Scratchpad to Agent
Agent Response
AgentFinish(
return_values={"output": "The capital of France is Paris..."},
log="Thought: I now know the final answer.\nFinal Answer: ..."
)
Agent Action
AgentAction(
tool="search_tool",
tool_input="population of Paris",
log="Thought: Now I need to find the population of Paris
)
Agent Finish
Execute Tool
Return Final Output
Collect Observation
Update Scratchpad

Example of flow-

🧠 Input Query:

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

```
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:
```

```
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."
```

Agent scratchpad (history) now looks like -

```
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",
  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
```

```
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."
}
```