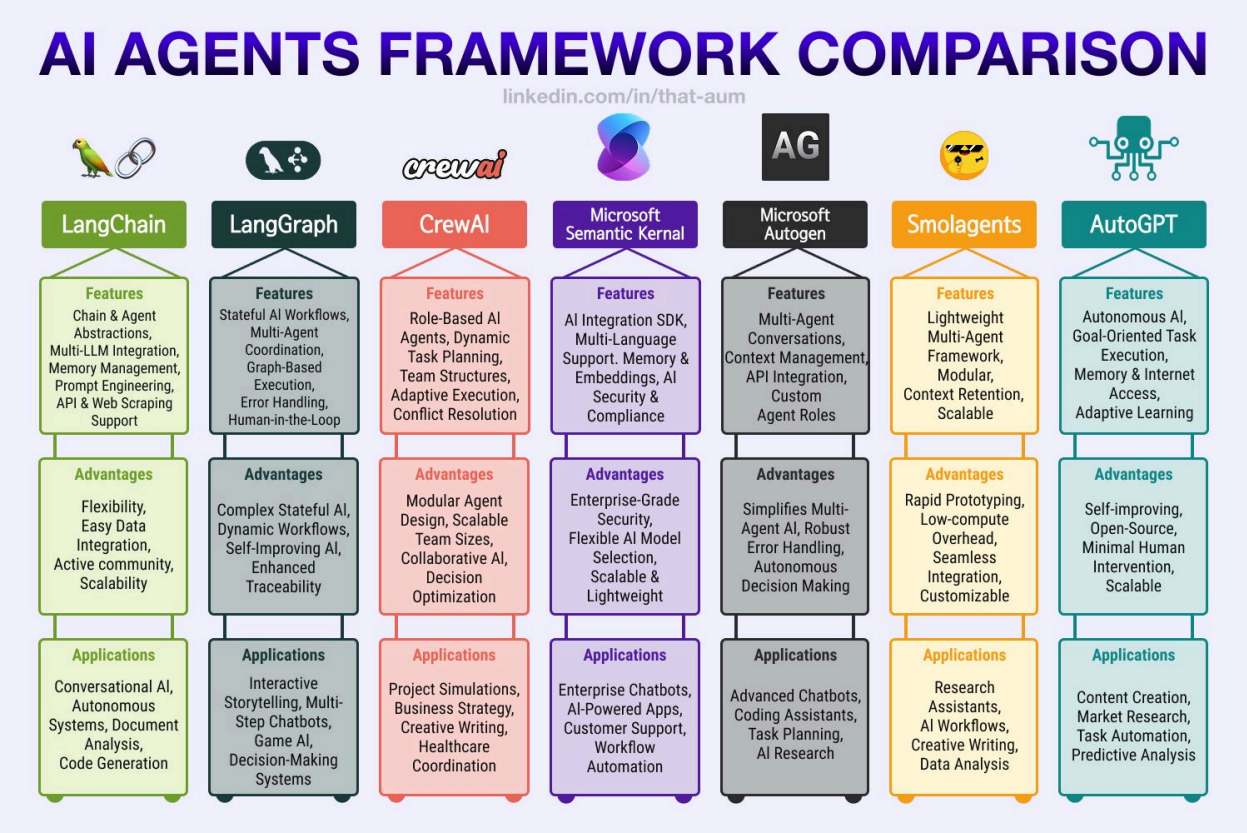
Module 1 : Introduction to LangChain

Module 1 : Introduction to LangChain



Langchain , LlamaIndex and Haystack

Comparison of LlamaIndex, LangChain and Haystack

LlamaIndex	LangChain	Haystack	Which to choose
Purpose Fast, flexible retrieval tool	Chaining tasks together	Production-ready search applications	Choose LlamaIndex if you need a fast and versatile search and retrieval tool
Focus Search and retrieval of complex data	Creating workflows	Building search pipelines	
Use Case Document retrieval system	AI workflows	Question-answering system	

🔍 LlamaIndex vs LangChain vs Haystack: Choosing the Right Framework for Your AI Use Case

---

🦙 What is LlamaIndex?

LlamaIndex (formerly GPT Index) is a framework designed to help you connect large language models (LLMs) with complex datasets like PDFs, websites, and databases. Think of it as a “smart librarian” — it indexes your data so that querying becomes fast, efficient, and LLM-friendly.

🔑 Key Features:

- **Indexing structured, semi-structured, and unstructured data**

- **Natural language data querying**
- **Seamless integration with diverse sources (PDFs, web, SQL, etc.)**

✅ Ideal Use Cases:

- Knowledge bases
- Document retrieval systems
- Semantic search engines for research, healthcare, or legal domains

⚠️ Limitations:

- Focused only on retrieval; not built for complex task orchestration
- May require tuning for enterprise-scale performance

🔗 What is LangChain?

LangChain is designed for chaining together tasks in AI workflows. It allows you to connect LLMs to APIs, tools, databases, and other services, orchestrating them into a step-by-step pipeline. Think of it as the "glue" that binds your LLM-powered application together.

🔑 Key Features:

- **Build chains of tasks (multi-step workflows)**
- **Prompt engineering and optimization**
- **Integrates with APIs, databases, tools, and more**

✅ Ideal Use Cases:

- Autonomous AI agents
- Workflow automation
- Complex task sequencing (e.g., get data → process → respond)

⚠️ Limitations:

- Steeper learning curve

- Requires more engineering effort for setup and debugging

🧠 What is Haystack?

Haystack is a production-ready NLP framework focused on document search, question-answering, and knowledge retrieval at scale. It's the go-to solution for enterprises that need high-performance, multilingual, and real-time search applications.

🔑 Key Features:

- **Retriever-reader pipelines**
- **Integrates with OpenAI, Elasticsearch, and more**
- **Scales for millions of documents and supports multilingual/semantic search**

✅ Ideal Use Cases:

- Enterprise QA systems
- Customer support chatbots
- Legal and financial document search

⚠️ Limitations:

- Complex to set up and operate at small scale
- Best suited for large teams or businesses with infrastructure

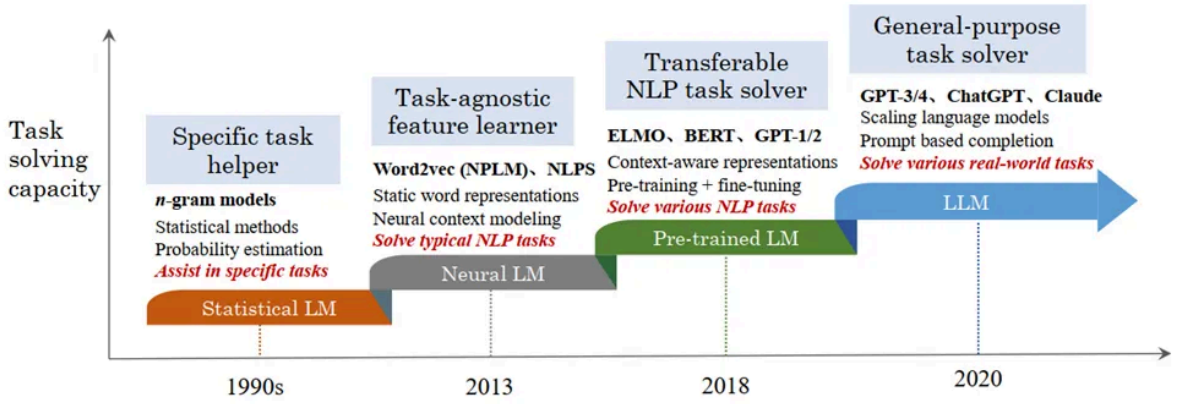
---

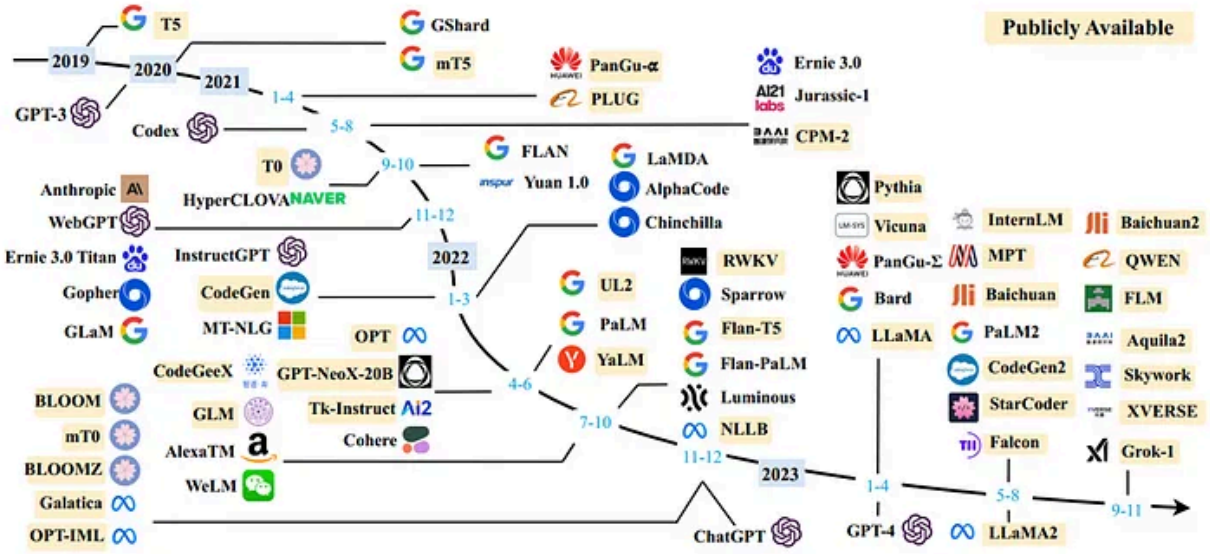
🆚 Summary Comparison Table

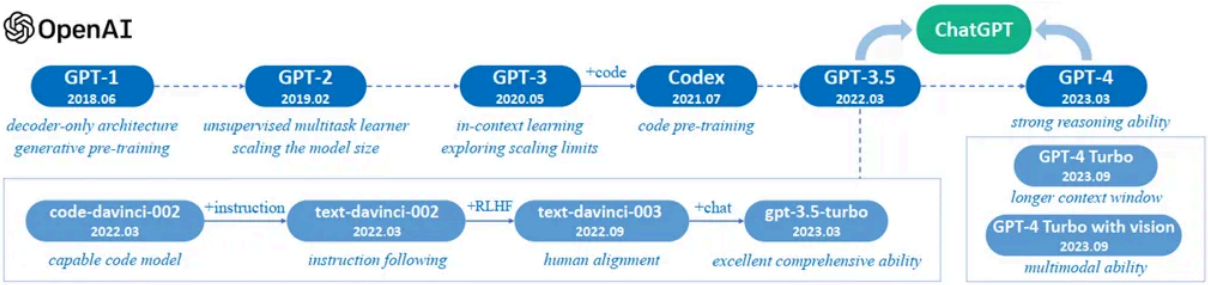
Feature/Criteria	**LlamaIndex**	**LangChain**	**Haystack**
Primary Role	Data indexing and retrieval	AI workflow orchestration	Production-grade document search/QA
Strength	Fast, flexible data querying	Customizable LLM pipelines	Scalable enterprise-grade performance
Use Case	Knowledge bases, semantic search	AI agents, tool chaining, automation	QA systems, legal/enterprise

			search
Ease of Use	Moderate setup, intuitive for devs	Steeper learning curve	Complex, especially for small teams
Integration Focus	PDFs, web, SQL, Notion, etc.	APIs, tools, databases, LLMs	Elasticsearch, OpenAI, cloud infra
Customization	Moderate (indexing, retrieval)	High (custom pipelines, logic flows)	High (retriever/reader tuning, pipelines)
Best For	Developers handling diverse datasets	AI engineers building end-to-end workflows	Enterprises needing search at scale
Community Support	Growing		

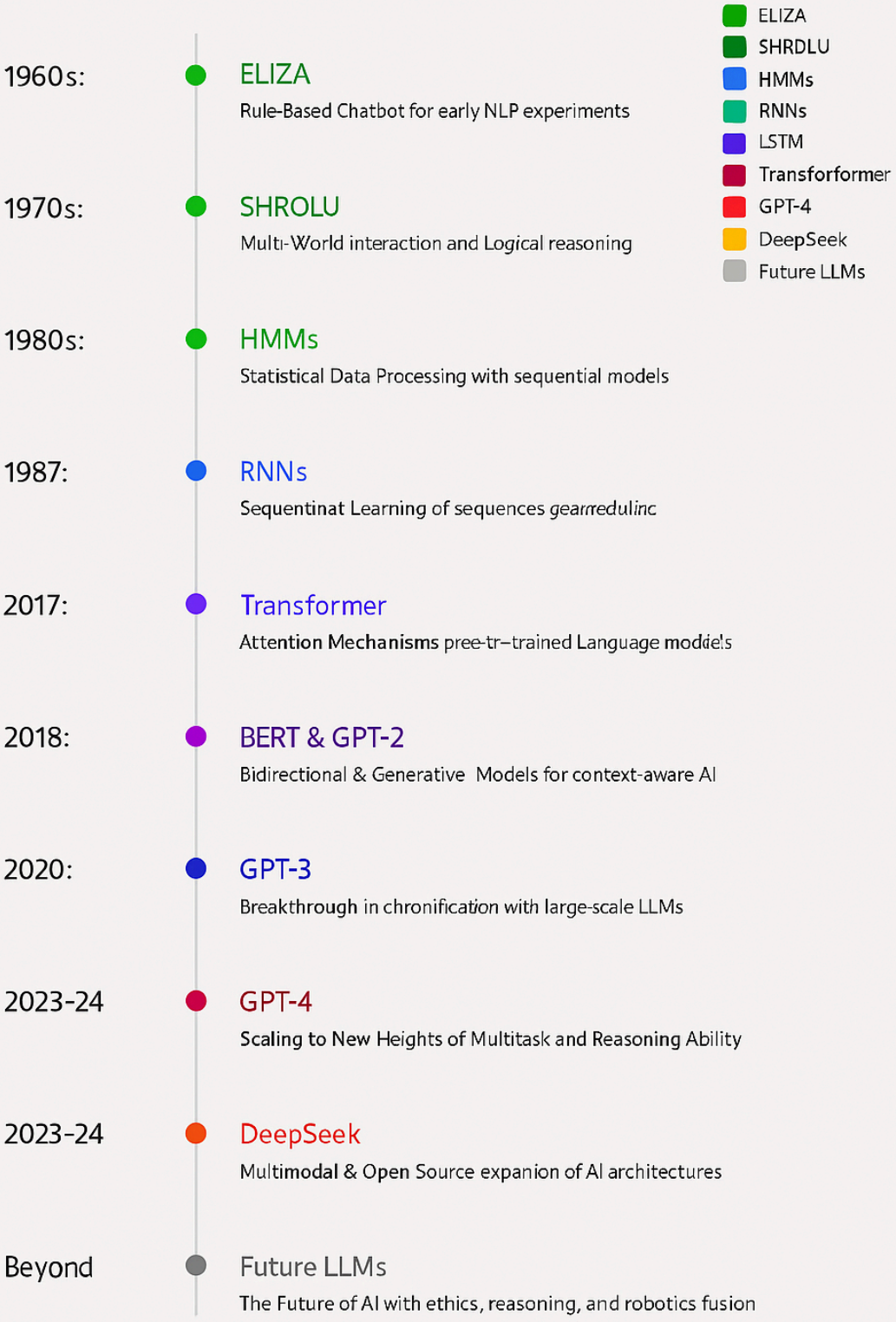
LLM Evaluation :




Publicly Available
2019
2020
2021
2022
2023
T5
GShard
mT5
GPT-3
Codex
PanGu-α
PLUG
Ernie 3.0
Jurassic-1
CPM-2
T0
FLAN
Yuan 1.0
LaMDA
AlphaCode
Chinchilla
Anthropic
HyperCLOVA
WebGPT
Ernie 3.0 Titan
InstructGPT
Gopher
CodeGen
GLaM
MT-NLG
OPT
CodeGeeX
GPT-NeoX-20B
BLOOM
mT0
BLOOMZ
Galatica
OPT-IML
GLM
AlexaTM
WeLM
Tk-Instruct
Cohere
UL2
PaLM
YaLM
RWKV
Sparrow
Flan-T5
Flan-PaLM
Luminous
NLLB
ChatGPT
Pythia
Vicuna
PanGu-Σ
Bard
LLaMA
GPT-4
InternLM
MPT
Baichuan
PaLM2
CodeGen2
StarCoder
Falcon
LLaMA2
Baichuan2
QWEN
FLM
Aquila2
Skywork
XVERSE
Grok-1
1-4
5-8
9-10
11-12
1-3
4-6
7-10
11-12
1-4
5-8
9-11


OpenAI
ChatGPT
GPT-1
2018.06
decoder-only architecture
generative pre-training
GPT-2
2019.02
unsupervised multitask learner
scaling the model size
GPT-3
2020.05
in-context learning
exploring scaling limits
+code
Codex
2021.07
code pre-training
GPT-3.5
2022.03
GPT-4
2023.03
strong reasoning ability
code-davinci-002
2022.03
capable code model
+instruction
text-davinci-002
2022.03
instruction following
+RLHF
text-davinci-003
2022.09
human alignment
+chat
gpt-3.5-turbo
2023.03
excellent comprehensive ability
GPT-4 Turbo
2023.09
longer context window
GPT-4 Turbo with vision
2023.09
multimodal ability

The Evolution of LLMs



🚀 Famous Large Language Models (LLMs

🧠 **OpenAI** (Closed Source, Commercial)

- GPT-3
- GPT-3.5 (used in ChatGPT free tier)
- GPT-4 (used in ChatGPT Plus & API)
- Codex (for code generation, powers GitHub Copilot)

🧠 **Anthropic** (Closed Source, Commercial)

- Claude 1, 2, 3 (as of 2024) is one of the top-performing LLMs for reasoning)

🧠 **Google DeepMind**

- BERT (open-source transformer encoder model, 2018)
- PaLM, PaLM 2 (closed, commercial)
- Gemini 1 & 1.5 (multimodal, cutting-edge)

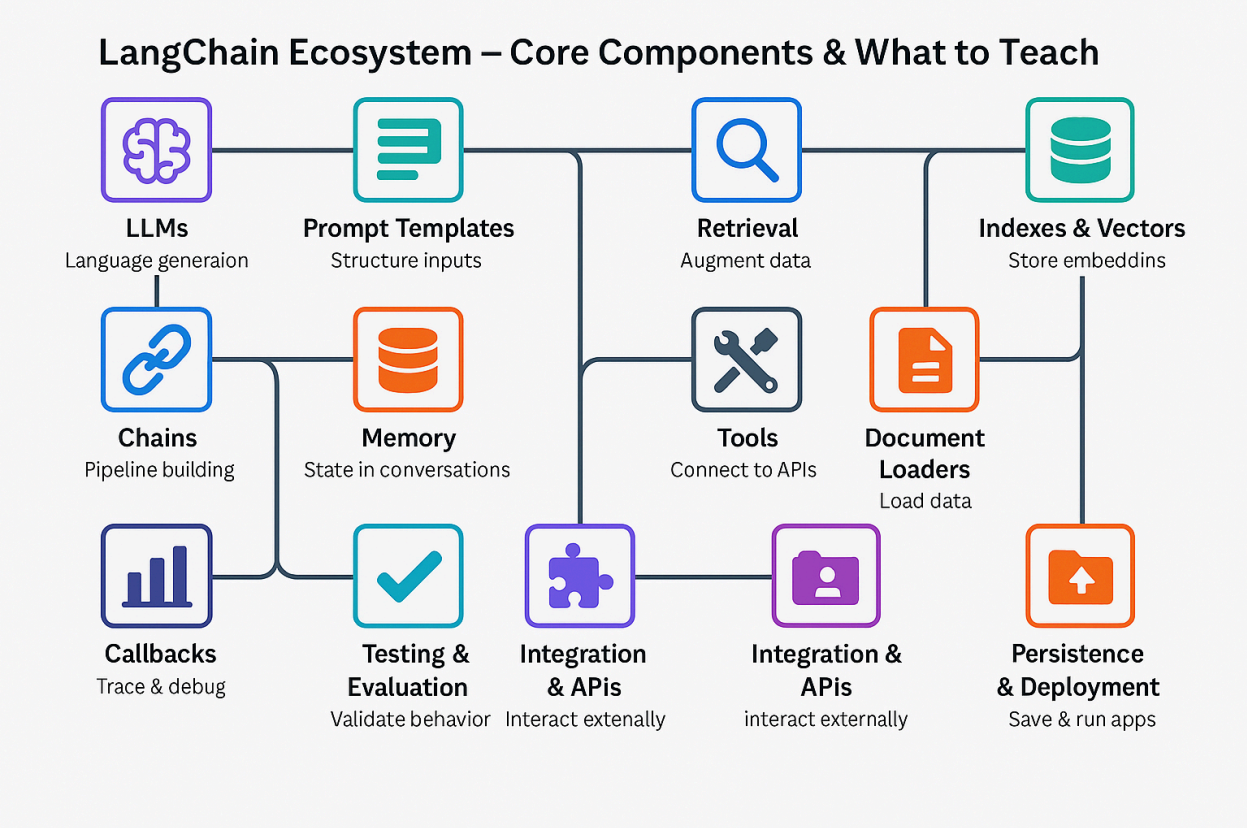
🧠 **Meta AI** (Open Source)

- LLaMA (1, 2, 3)
- OPT (Open Preinture of Experts)

🧠 **Mistral AI** (Open Source)

- Mistral 7B
- Mixtral 8x7B (Mixture of Experts)

🧠 LangChain Ecosystem – Core Components & What to Teach



1. 📦 LLMs (Large Language Models)

- **Purpose**: Core engine that processes and generates language.
- **What to teach**:
  - How to use different LLMs (OpenAI, HuggingFace, Anthropic, Cohere, etc.)
  - Switching providers via wrappers
  - Prompt structuring and generation

---

2. 🧩 Prompt Templates

- **Purpose**: Standardized way to structure inputs for LLMs.
- **What to teach**:
  - `PromptTemplate`, `ChatPromptTemplate`
  - Variables, input/output mapping
  - Few-shot prompting and dynamic prompts

3. 🔗 Chains

- **Purpose**: Combine multiple components into a pipeline.
- **What to teach**:
  - `LLMChain`, `SequentialChain`, `RouterChain`
  - Building custom chains
  - Error handling and branching logic

---

4. 📚 Memory

- **Purpose**: Allows LLMs to maintain state across interactions.
- **What to teach**:
  - `ConversationBufferMemory`, `EntityMemory`, `SummaryMemory`
  - Using memory in agents and chatbots
  - Trade-offs between short-term and long-term memory

---

5. 🔍 Retrieval (RAG - Retrieval-Augmented Generation)

- **Purpose**: Retrieve external data (e.g., from vector DBs) to augment answers.
- **What to teach**:
  - Integration with vector stores (e.g., Pinecone, Chroma, FAISS)
  - `RetrievalQA`, `VectorstoreRetriever`
  - Query transformers, rerankers

---

6. 🛠️ Tools & Toolkits

- **Purpose**: Enable LLMs to use external tools like APIs, calculators, databases.
- **What to teach**:
  - Tool interface
  - Custom tool creation

- Tool usage within agents

---

7. 🤖 Agents

- **Purpose**: Autonomous systems that decide what to do next.
- **What to teach**:
  - Agent types: `ZeroShotAgent`, `ChatAgent`, `ReActAgent`, `Plan-and-Execute`
  - Tool selection via reasoning
  - AgentExecutor, planning, memory use in agents

---

8. 🗃️ Document Loaders

- **Purpose**: Extract and structure data from different sources.
- **What to teach**:
  - Loading PDFs, websites, markdown, CSVs
  - `Unstructured`, `BS4`, `PyMuPDF`, `PyPDF2`
  - Text splitting strategies (recursive, character, token-based)

---

9. 🧠 Indexes & Vector Stores

- **Purpose**: Organize knowledge in searchable form.
- **What to teach**:
  - Vector store integrations
  - Creating indexes (text → embeddings → vectorstore)
  - Using retrievers inside chains/agents

---

10. 📈 Callbacks, Logging & Debugging

- **Purpose**: Observe, trace, and debug LangChain flows.
- **What to teach**:
  - Callback system

- LangSmith (for observability)
- Logging chains and agents

---

11. 🧪 Testing & Evaluation

- **Purpose**: Ensure correctness and reliability.
- **What to teach**:
  - Unit testing with mocked LLMs
  - Prompt testing and output validation
  - LangChainHub and evaluation tools

---

12. 🌐 Integration with APIs and External Services

- **Purpose**: Real-world data interaction.
- **What to teach**:
  - Calling REST APIs via tools
  - Web scraping and API response parsing
  - Authentication and security in agent workflows

---

13. 📂 Persistence & Deployment

- **Purpose**: Productionizing LangChain apps.
- **What to teach**:
  - Deployment to FastAPI, Streamlit, Chainlit, etc.

---

14. 🧰 LangChain Expression Language (LCEL)

- **Purpose**: New expressive syntax for building chains declaratively.
- **What to teach**:
  - LCEL operators (pipe `|`, map, branch)
  - Combining components concisely

---

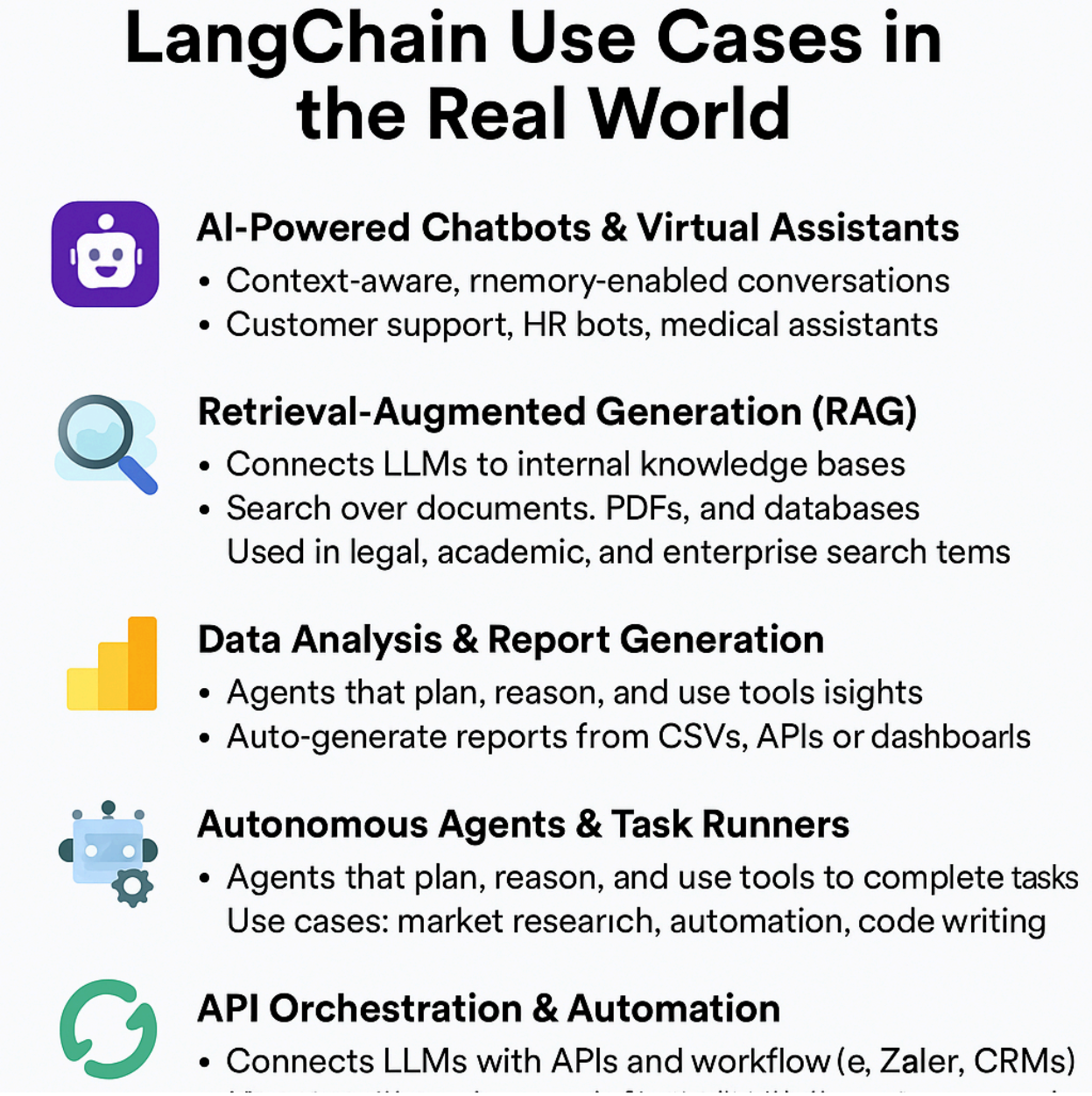
15. 🌍 LangChain Hub

- **Purpose**: Sharing and reusing community chains, agents, and prompts.
- **What to teach**:
  - Searching and publishing components
  - Reuse of chains and templates

✅ Summary Table

Component	Purpose	Key Topic Highlights
LLMs	Core language generation	Providers, APIs, prompt input/output
Prompt Templates	Standardize input	Few-shot, dynamic, chat prompts
Chains	Pipeline building	Sequential logic, control flows
Memory	Maintain context	Chat memory, summarization
Retrieval	Augment responses	Vector DBs, retrievers, query engines
Tools	External capabilities	API calls, calculators, web tools
Agents	Autonomous task execution	Tool use, planning, reasoning
Document Loaders	Ingest data	PDFs, HTML, text, chunking
Vector Stores	Semantic search	FAISS, Pinecone, Chroma
Callbacks	Observability	Logs, LangSmith, debugging
Testing	Validate performance	Output checks, prompt tests
API Integration	Real-world utility	External services, scraping
Persistence	Save & deploy apps	Deployment, caching
LCEL	Code-less chaining	New syntax, operator-based pipelines
LangChain Hub	Community & Reuse	Shared chains, prompts, templates

🌍 Top 5 LangChain Use Cases in the Real World



1. 🤖 **AI-Powered Chatbots & Virtual Assistants**
    - Context-aware, memory-enabled conversations
    - Customer support, HR bots, medical assistants
2. 🔎 **Retrieval-Augmented Generation (RAG)**
    - Connects LLMs to internal knowledge bases
    - Search over documents, PDFs, and databases
    - Used in legal, academic, and enterprise search systems
3. 📊 **Data Analysis & Report Generation**
    - Converts natural language queries to insights
    - Auto-generates reports from CSVs, APIs, or dashboards
4. 🧠 **Autonomous Agents & Task Runners**
    - Agents that plan, reason, and use tools to complete tasks
    - Use cases: market research, automation, code writing
5. 🌐 **API Orchestration & Automation**
    - Connects LLMs with APIs and workflows (Zapier, CRMs)
    - Used for AI-powered workflow automation

🏗️ LangChain Architecture & Modular Design

LangChain is designed as a **modular framework** with interchangeable components that can be assembled into custom workflows. The architecture is typically broken down into the following layers/modules:

🔷 1. Language Model Layer (LLM Layer)

- **Core Engine**: Connects to LLMs (OpenAI, Anthropic, HuggingFace, etc.)

- **Abstractions**:
  - `BaseLanguageModel`
  - `ChatModel`, `LLM`

🔷 2. Prompt Management Layer

- **Prompt Templates**: Format inputs to LLMs
- **Key Components**:
  - `PromptTemplate`
  - `ChatPromptTemplate`
  - `FewShotPromptTemplate`

🔷 3. Chain Layer

- **Execution Logic**: Manages flow of tasks
- **Types**:
  - `LLMChain`, `SequentialChain`, `RouterChain`
- **Features**: Supports nesting, branching, and conditional logic

---

🔷 4. Memory Layer

- **Stores Context**: Keeps past inputs/outputs
- **Types**:
  - `ConversationBufferMemory`
  - `SummaryMemory`
  - `EntityMemory`

🔷 5. Tooling Layer

- **External Tools**: Allow LLMs to use APIs, calculators, etc.
- **Agents use tools to perform tasks**

- **Tool Abstractions:** `Tool`, `MultiInputTool`

🔷 6. Agent Layer

- **Autonomous Reasoning**: LLM chooses tools based on goals
- **Agent Types**:
  - `ZeroShotAgent`, `ReActAgent`, `PlanAndExecuteAgent`

🔷 7. Document Loaders & Text Splitters

- **Data Ingestion**:
  - Load from PDFs, websites, CSVs
  - Chunk data for embedding

🔷 8. Vector Store & Embeddings

- **For Retrieval & RAG**:
  - Stores and searches data as vectors
  - Interfaces: FAISS, Pinecone, Chroma, Weaviate

🔷 9. Retrieval QA / RAG Layer

- **Combines Vector Search + LLM**
- **Key Class:** `RetrievalQA`, `ConversationalRetrievalChain`

🔷 10. Output Parsers & Evaluation

- **Handles outputs**:
  - Parses JSON, lists, or structured formats
- **Supports evaluation and feedback**

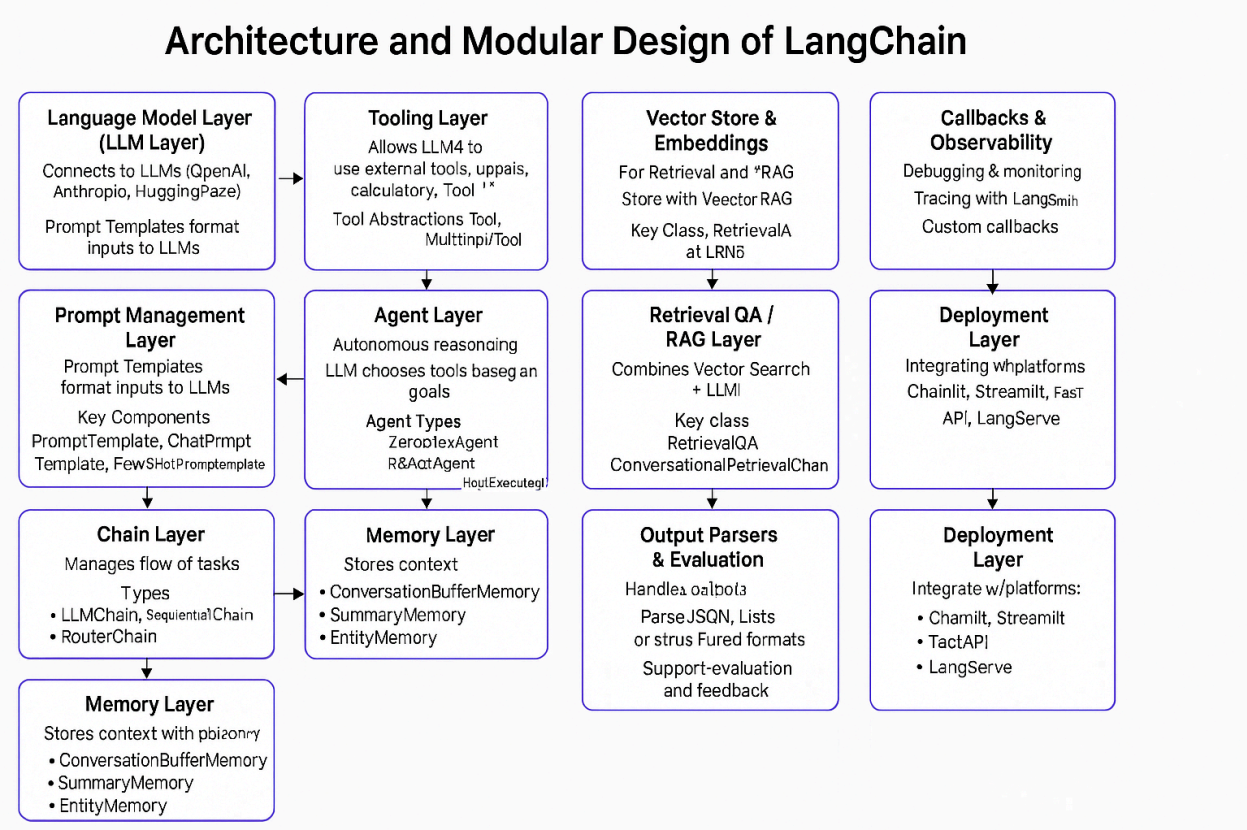
🔷 11. Callbacks & Observability

- **Debugging & Monitoring**:

- Tracing with `LangSmith`, custom callbacks

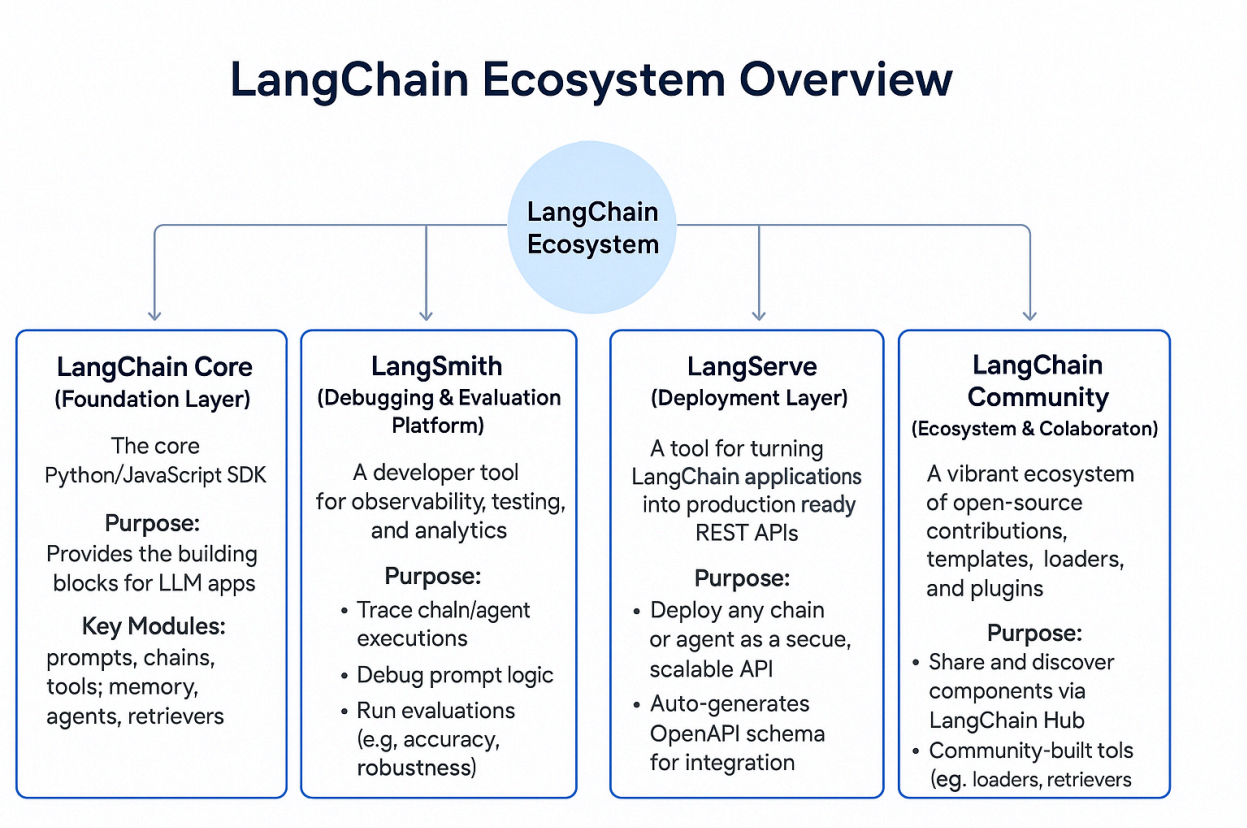
🔷 12. Deployment Layer

- **Integration with platforms**:
  - Chainlit, Streamlit, FastAPI, LangServe
- **Persistence**: Caching and saving chains



🌐 LangChain Ecosystem Overview

LangChain is more than just a framework — it's a complete ecosystem designed to build, test, deploy, and collaborate on LLM-based applications.



🔷 1. LangChain Core (Foundation Layer)

- **What it is**: The core Python/JavaScript SDK
- **Purpose**: Provides the building blocks for LLM apps — prompts, chains, tools, memory, agents, retrievers
- **Key Modules**:
  - `langchain-core`, `langchain-community`, `langchain-openai`, etc.
- **Use Case**: Rapid prototyping, building modular LLM pipelines

🔷 2. LangSmith (Debugging & Evaluation Platform)

- **What it is**: A developer tool for **observability, testing, and analytics** of chains, agents, and prompts
- **Purpose**:
  - Trace chain/agent executions
  - Debug prompt logic
  - Run evaluations (e.g., accuracy, robustness)
- **Use Case**: Ensures reliability and performance of LLM workflows before and after deployment

---

🔷 3. LangServe (Deployment Layer)

- **What it is**: A tool for **turning LangChain applications into production-ready REST APIs**
- **Purpose**:
  - Deploy any chain or agent as a secure, scalable API
  - Auto-generates OpenAPI schema for integration
- **Use Case**: Build SaaS products, backend services, and LLM-powered APIs

---

🔷 4. LangChain Community (Ecosystem & Collaboration)

- **What it is**: A vibrant ecosystem of open-source contributions, templates, loaders, and plugins
- **Purpose**:
  - Share and discover components via LangChain Hub
  - Community-built tools (e.g., loaders, retrievers, vector store wrappers)
- **Use Case**: Jumpstart development, share best practices, extend LangChain's reach

Installation Guide :

Step 1: Create a Dedicated Conda Environment

```
conda create -n langchain-env python=3.11
```

Step 2: Activate the Environment

```
conda activate langchain-env
```

```
# LangChain Setup Guide


"""
1. Installation with Version Compatibility
-----------------------------------
Always install packages in this order to ensure compatibility:
"""

# Core LangChain packages (must be installed first)
"""
pip install langchain

pip install --upgrade "langchain>=0.3,<0.4"
pip install --upgrade "langchain-community>=0.3,<0.4"
pip install --upgrade "langchain-text-splitters>=0.3,<0.4"
pip install --upgrade "langchain-core>=0.3,<0.4"
```

```
"""

# LLM Provider-specific packages
"""
# For OpenAI
pip install --upgrade "langchain-openai>=0.3,<0.4"

# For Groq
pip install --upgrade "langchain-groq>=0.2,<0.3"


#gemini

pip install google-generativeai


# For other providers (if needed)
pip install --upgrade "langchain-anthropic>=0.3,<0.4"
pip install --upgrade "langchain-mistral>=0.3,<0.4"
"""

# Additional tools and utilities
"""
# For graph-based workflows
pip install --upgrade "langgraph>=0.2.20,<0.3"

# For vector stores and embeddings
pip install --upgrade "langchain-vectorstores>=0.3,<0.4"
"""

"""
2. Environment Setup
-----------------
Create a .env file with your API keys:
"""
```

```
# .env file content
"""
OPENAI_API_KEY=your-openai-api-key
GROQ_API_KEY=your-groq-api-key
"""

"""
3. Basic Usage with OpenAI
-----------------------
Here's how to use LangChain with OpenAI:
"""



# openai code :

import os
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI

load_dotenv()

llm = ChatOpenAI(
   model = "gpt-3.5-turbo",
   temperature = 0.2,
   api_key= os.getenv("OPENAI_API_KEY")

)

prompt = "TELL ME A JOKE ABOUT PROGRAMMER "

response= llm.invoke(prompt)
print(response)

## llm chain
```

```
import os
from dotenv import load_dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain.chains import LLMChain


# Load .env variables
load_dotenv()

# Function to set up the chain
def setup_openai_chain():
    """Setup a basic chain using OpenAI"""
    llm = ChatOpenAI(
        model="gpt-3.5-turbo",
        temperature=0.7,
        api_key=os.getenv("OPENAI_API_KEY")
    )

    prompt = ChatPromptTemplate.from_template(
        "Tell me a {adjective} joke about {topic}."
    )

    chain = LLMChain(llm=llm, prompt=prompt)
    return chain

# 🔁 Initialize the chain
chain = setup_openai_chain()

# ✅ Run the chain with inputs
response = chain.invoke({"adjective": "funny", "topic": "programmers"})

# 📤 Print the output
print(response)
```

```
"""
4. Basic Usage with Groq

3 Groq :

----------------------

Here's how to use LangChain with Groq:
"""


 # groq chatmodel :


import os
from dotenv import load_dotenv
from langchain_groq import ChatGroq


load_dotenv()

ChatGroq(
    model = "llama3-8b-8192",
    temperature = 0.2,
    api_key= os.getenv("GROQ_API_KEY")

)
prompt = "TELL ME A JOKE ABOUT PROGRAMMER "

response= llm.invoke(prompt)
print(response)
```

```
 # groq chain


# ✅ Imports
import os
from dotenv import load_dotenv
from langchain_groq import ChatGroq
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain

# ✅ Load environment variables from .env file
load_dotenv()

# ✅ Define the chain setup function
def setup_groq_chain():
    """Setup a basic chain using Groq"""
    llm = ChatGroq(
        model="llama3-8b-8192",
        temperature=0.7,
        api_key=os.getenv("GROQ_API_KEY")  # Ensure .env contains this
    )

    prompt = ChatPromptTemplate.from_template(
        "Tell me a {adjective} joke about {topic}."
    )

    chain = LLMChain(llm=llm, prompt=prompt)
    return chain

# ✅ Set up the chain and run it with inputs
chain = setup_groq_chain()
response = chain.run({"adjective": "funny", "topic": "Python developers"})

# ✅ Print the output
print("🤣 Joke Response:")
```

```
print(response)



# Gemeni Basic


import os
from dotenv import load_dotenv
from langchain_google_genai import ChatGoogleGenerativeAI

load_dotenv()


llm = ChatGoogleGenerativeAI(
    model = "models/gemini-1.5-pro-latest",
    temperature = 0.2,
    api_key= os.getenv("GEMINI_API_KEY")

)


prompt = "TELL ME A JOKE ABOUT PROGRAMMER "

response= llm.invoke(prompt)
print(response)







# genemi

pip install google-generativeai
```

```
import os
from dotenv import load_dotenv
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# Load .env with GOOGLE_API_KEY
load_dotenv()

# Setup Gemini 1.5 chain using LangChain
def setup_gemini_15_chain():
    llm = ChatGoogleGenerativeAI(
        model="models/gemini-1.5-pro-latest",
        temperature=0.7,
        google_api_key=os.getenv("GEMINI_API_KEY"),
    )

    # Define the prompt
    prompt = ChatPromptTemplate.from_template("Tell me a {adjective} joke abou

    # Output parser
    parser = StrOutputParser()

    # Build the full chain
    chain = prompt | llm | parser
    return chain

# Run the chain
if __name__ == "__main__":
    chain = setup_gemini_15_chain()
    response = chain.invoke({"adjective": "funny", "topic": "Python developers"})
    print("🤣 Gemini 1.5 Joke:")
    print(response)
```

Task :

"We explored models like Gemini, Groq, and OpenAI in class. Now it’s your turn! Try accessing **5 more AI/LLM models** (like Claude, Mistral, Perplexity, etc.). Write a **short summary or interesting finding** about each model. Share your experience **with code or screenshots** in our WhatsApp group or send it directly to me."