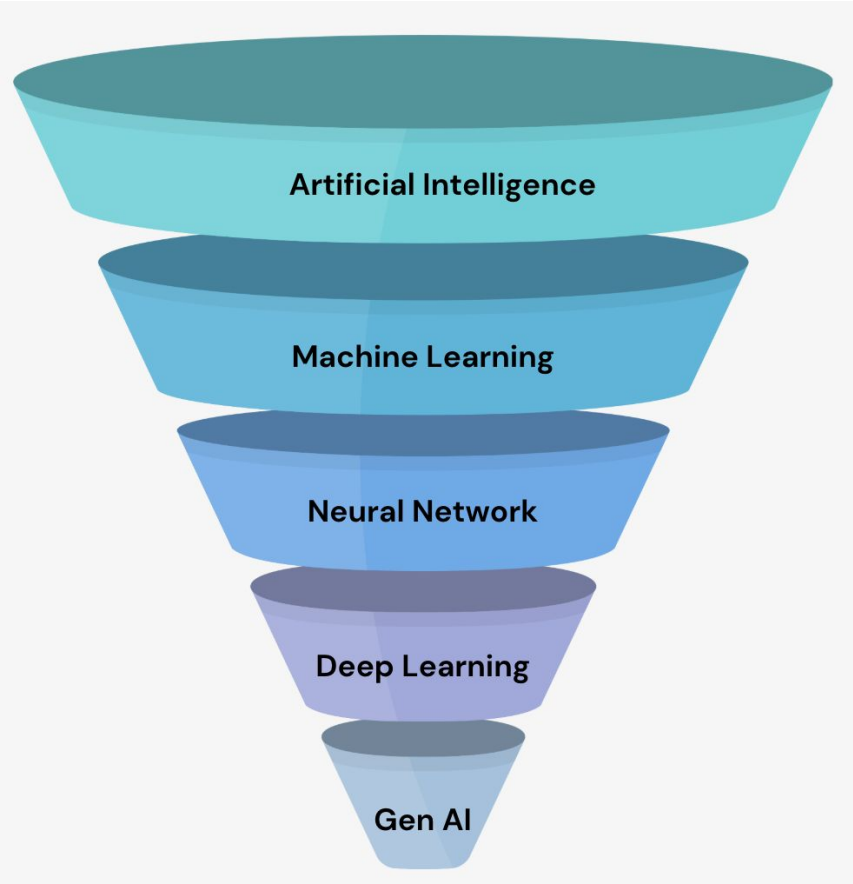
Agentic AI Theory

Agenda

1. What is GenAI ? What is Agentic AI ? What are AI Agents?
2. How are GenAI Models trained?
3. How to design Agentic Systems ?
4. Prompt Based Agents
5. RAG
6. Dynamic Agents
7. Demo - Langgraph
8. Demo - No Code tool

What is Generative AI ?



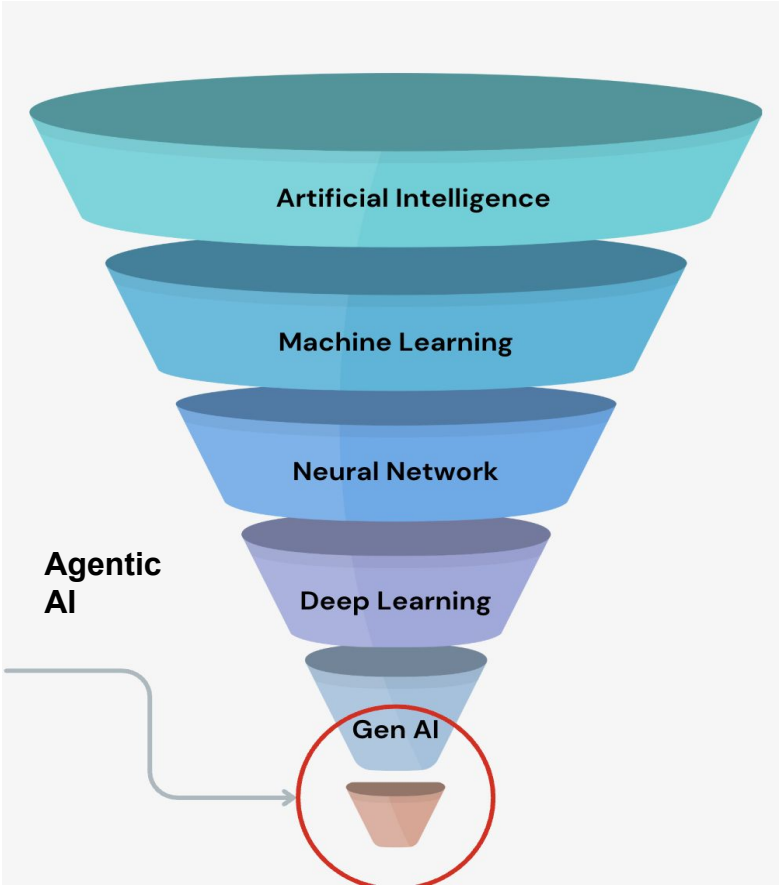
What is Generative AI?

Usually neural networks that can both understand and **generate** new data

- Generative AI is essentially **generative deep learning**, though we don't explicitly call it that.
- It follows the same principles as previous AI models
- While earlier models predicted from a limited set of categories (e.g., dog vs. cat),
- Gen AI predicts from a much larger and more open-ended set, such as words in a vocabulary, pixels in an image, or actions on a webpage.

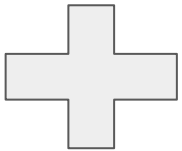
What is Agentic AI ?

Agentic AI is an application layer built on top of Generative AI



Agentic AI decoded

Generative AI



Tools

Gen AI Models can call APIs or Tools
(Ex - Code executer , HTTP requests etc)

Memory

AI Models can retrieve external memory as required for a task

Plans

AI Models can generate a plan first which involves a set of actions

What are AI Agents?

*An Agent is a system that **leverages an AI model** to interact with its environment in order to achieve a user-defined objective.*

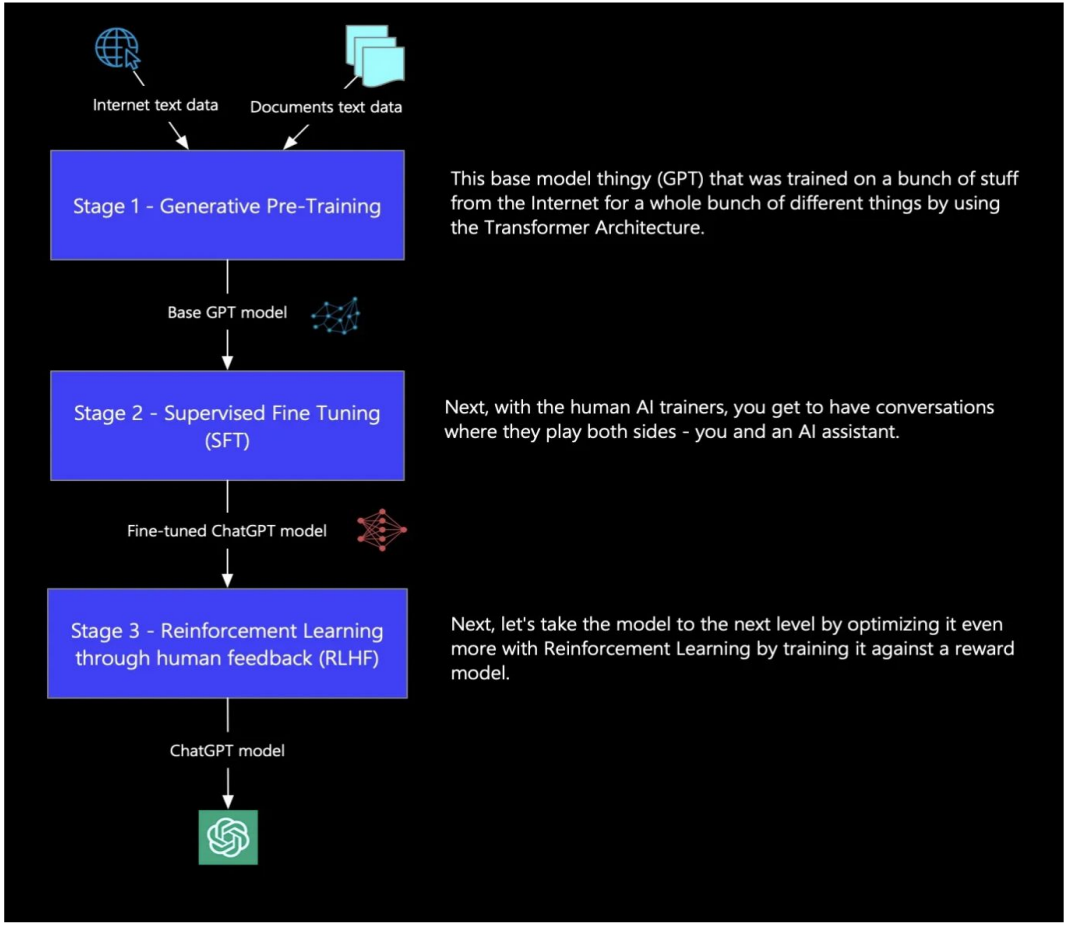
*It combines **reasoning, planning, and the execution of actions (often via external tools)** to fulfill tasks.*

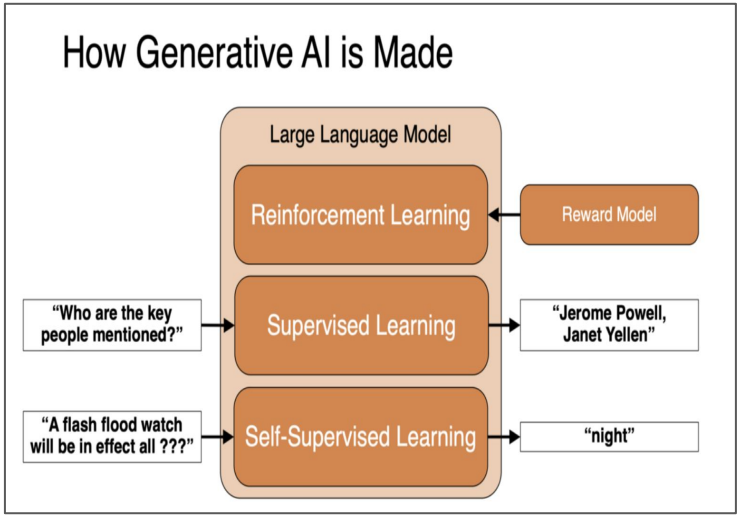
Hugging Face (Link)

In our discussion :
Agentic AI = AI Systems that use AI agents capable of acting towards an end goal . Think about AI Agents doing their parts and as a system they achieve a broader goal

This definition aligns with how openai , anthropic defines Agentic Ai

How today's Generative AI models are built





Link and Link

Note : All LLMs have a fixed context length (though increasing)

1. Scaling Limits: Compute, Memory, and Time
Think of an LLM (Large Language Model) like a **giant classroom** where every student (word/token) has to listen to every other student at the same time.

- If there are **10 students**, it's fine—they can all talk and listen.
- If there are **1,000 students**, suddenly every student is trying to listen to 999 others at once.
- As the classroom grows, it becomes **noisy, crowded, and super hard to manage**.

That's why very long inputs (lots of words) take too much time, energy, and memory for the model to handle efficiently.

2. Positional Encoding: Tracking Word Order

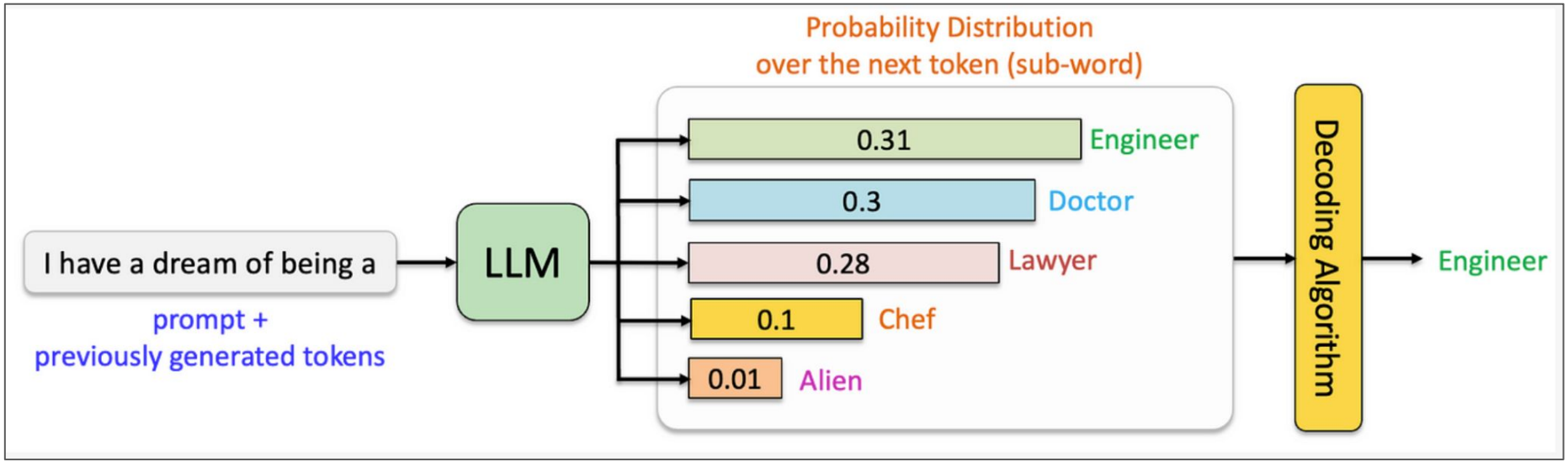
Imagine reading a sentence cut into flashcards. The model doesn't naturally know if "dog bites man" is different from "man bites dog."

- To fix this, we number the flashcards (like page numbers in a book).
- These numbers tell the model the order of words, even though it looks at all the words at once.

Older models used fixed numbering (like fixed seats in a theater).
Newer models use a rotating system (RoPE), which is like a carousel where positions keep shifting smoothly. But after too many rotations, the system gets confused and the accuracy drops.

Temperature

Most LLMs are **autoregressive** in nature, meaning they generate text token by token, with each token depending on the previously generated ones. They achieve this by assigning probabilities to different possible next words.

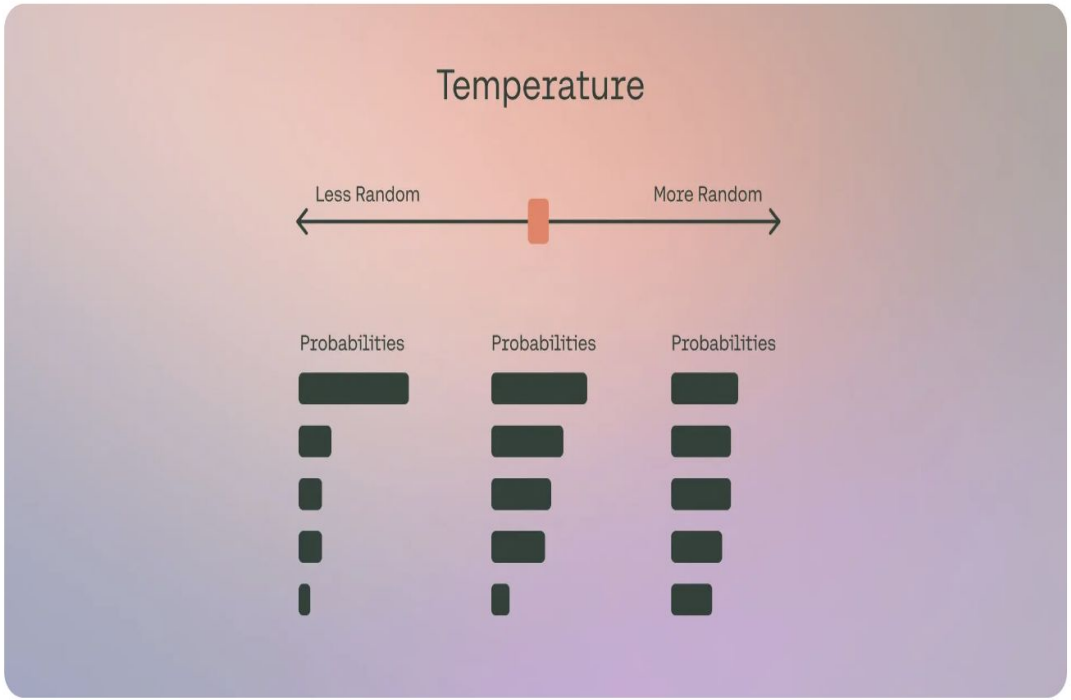


Temperature

Temperature controls this process by scaling these probabilities—

T < 1: Sharpening them for more deterministic output

T > 1: Flattening them to introduce randomness.



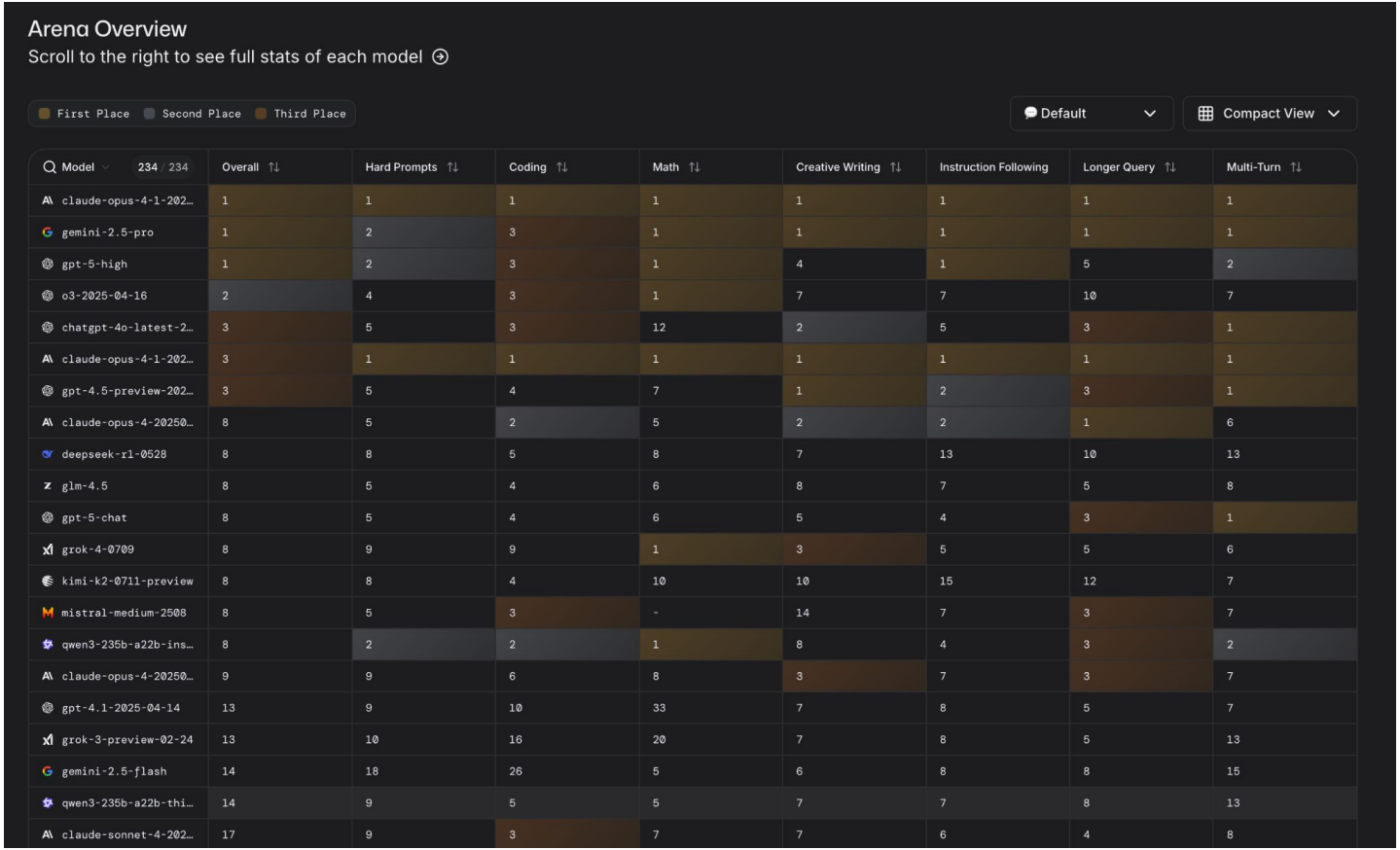
Link

LLM Evaluation or Benchmarkings

- **Static, Ground–Truth–Based:** Most common method due to **low cost** and reproducibility.
- **Static, Human Preference–Based:** Uses fixed questions but evaluates based on human feedback.
- **Live, Ground–Truth–Based:** Continuously updated questions paired with definitive answers.
- **Live, Human Preference–Based:** Incorporates real-time questions and evaluates using ongoing human feedbacks.

		Question Source	
		Static	Live
Evaluation Metric	Ground Truth	MMLU, HellaSwag, GSM-8K	Codeforces Weekly Contests
	Human Preference	MT-Bench, AlpacaEval	**Chatbot Arena**

Chatbot Arena Leaderboard



Arena Overview

Scroll to the right to see full stats of each model

First Place | Second Place | Third Place

Default | Compact View

Model (234 / 234)	Overall	Hard Prompts	Coding	Math	Creative Writing	Instruction Following	Longer Query	Multi-Turn
claude-opus-4-1-202…	1	1	1	1	1	1	1	1
gemini-2.5-pro	1	2	3	1	1	1	1	1
gpt-5-high	1	2	3	1	4	1	5	2
o3-2025-04-16	2	4	3	1	7	7	10	7
chatgpt-4o-latest-2…	3	5	3	12	2	5	3	1
claude-opus-4-1-202…	3	1	1	1	1	1	1	1
gpt-4.5-preview-202…	3	5	4	7	1	2	3	1
claude-opus-4-20250…	8	5	2	5	2	2	1	6
deepseek-r1-0528	8	8	5	8	7	13	10	13
glm-4.5	8	5	4	6	8	7	5	8
gpt-5-chat	8	5	4	6	5	4	3	1
grok-4-0709	8	9	9	1	3	5	5	6
kimi-k2-0711-preview	8	8	4	10	10	15	12	7
mistral-medium-2508	8	5	3	-	14	7	3	7
qwen3-235b-a22b-ins…	8	2	2	1	8	4	3	2
claude-opus-4-20250…	9	9	6	8	3	7	3	7
gpt-4.1-2025-04-14	13	9	10	33	7	8	5	7
grok-3-preview-02-24	13	10	16	20	7	8	5	13
gemini-2.5-flash	14	18	26	5	6	8	8	15
qwen3-235b-a22b-thi…	14	9	5	5	7	7	8	13
claude-sonnet-4-202…	17	9	3	7	7	6	4	8

Link

What are the key design questions to think about in designing Gen AI solutions

What’s unique about Generative AI System Design

Biggest challenge of GenAI systems are that it will give non deterministic outputs . But enterprises prefer very high predictability of outcome for seamless experience and automation. **How to get deterministic outcome from a non deterministic technology is unique?** Apart from that add - Hallucination, Sycophancy, Prompt sensitivity , Latency etc

Does the image represent a cat or a dog?

All these answers are right (but might not be useful for all use cases and workflows)

- dog
- yes, this is a cute brown dog
- well, this isn’t a cat for sure, it’s a dog

Key Design Questions:

- Choosing the right **AI model** for a given use-case
- **Choosing the right setup/application format**
- Designing your AI application:
    - Workflow Agents
    - RAG systems
    - Agentic Systems (Dynamic Agents - mostly multi agent)

Choosing the right AI model

Why Closed Source/Hosted models are Better for Early Use Cases

- Quick Deployment: Ready-to-use with minimal setup.
- Fully Managed: No need for in-house maintenance or support.
- Seamless Updates: Changes happen at the API level, avoiding infrastructure overhauls.
- Operational Benefits: Prompt caching etc.

When to Use Open Source

- High Security/Privacy Needs: Avoid sharing sensitive data with external providers.
- In-House Expertise: Skilled teams for model infrastructure and development.
- Mature Use Cases: Familiarity with LLM limitations allows fine-tuning.
- Niche Applications: Tailored use cases requiring domain-specific customization

Key Considerations: Choosing the right AI model

- **Performance:** Find a benchmark closer to your task, if not, it's always okay to go with overall benchmarks. (Link1 and Link2)
- **Security/Guardrails:** *Anthropic* emphasizes strong guardrails for safer outputs, suitable for sensitive applications.
- **Long Context Support:** *Gemini* supports up to 10M tokens, ideal for handling lengthy documents or complex workflows.
- **Integration Overhead/Ecosystem Compatibility:** *OpenAI* models integrate seamlessly with Azure workloads, using existing cloud partnerships for easier deployment.
- **Customization Capabilities:** *Amazon Nova* offer cheap and easy custom fine-tuning services, enabling tailored performance for domain-specific tasks.
- **Global Language Support:** *Gemini* models perform best on multi-lingual tasks.

TL & DR from experience

Always start off with **medium-sized models** (50-70B parameters) or **mid-priced proprietary models** like GPT-4o, Gemini 2.0 Flash, and Sonnet 3.5/3.7

You can move up/down as you build

Don't get lost in **model-selection conundrum**

Choosing the right setup / Application format

DATA : Biggest indicator of what kind of application u wanna build . Always start with ~100 sample data

What data to collect :

- **Input** Data : Say customer takes help of a customer support exec
- **Output** Data : Customer support exec gave some solution
- **Understanding reasoning** : Same question if multiple answer try to understand the whys ?

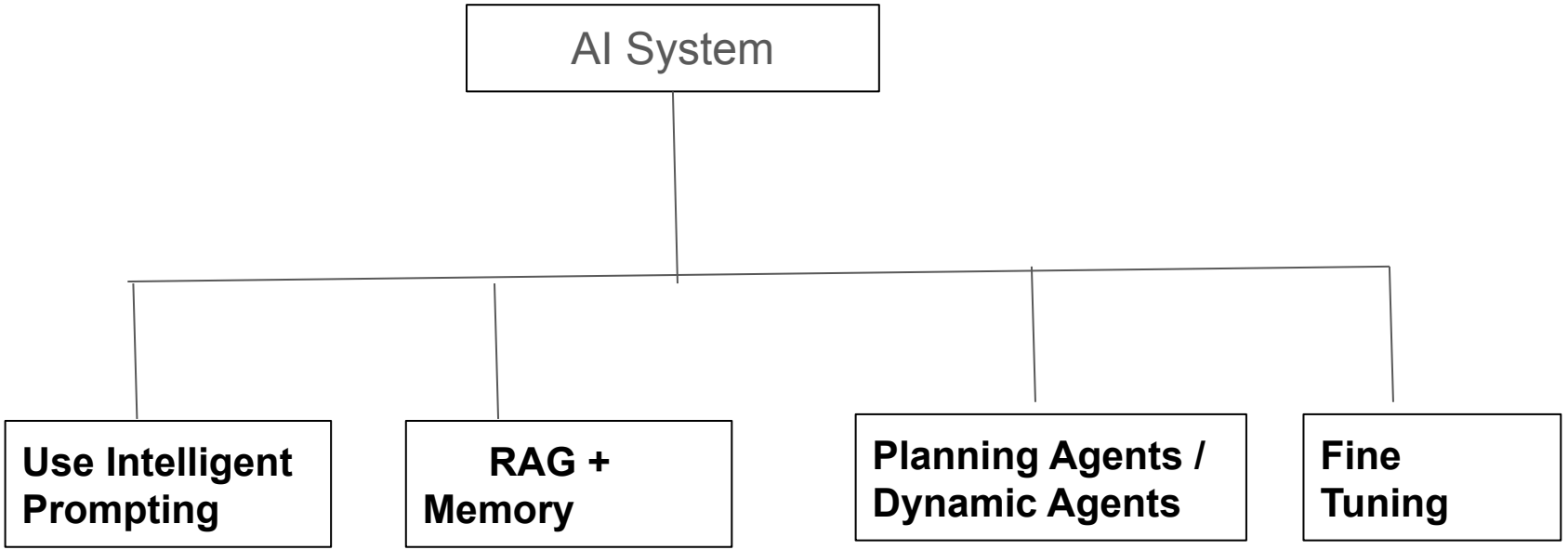
This step is most crucial because data will guide what kind of application design you want to have

Prototype first , optimization later

Design Principles (Step by Step)

1. Prioritize **effort**: Get a working prototype quickly, understand issues
2. Optimize for **performance**: Improve accuracy and reliability.
3. Focus on **cost and latency**: Try cheaper and faster models, now that you understand the problem.

Choosing or designing the right AI Application



Why Prompt engineering is easiest way to build AI systems

Latency: You only incur the latency of the LLM itself, and in most cases, you can precisely estimate it—especially time to first token.

Cost: No additional components are required, making costs predictable since they scale with the number of tokens.

Skill/Effort: Prompt engineering is becoming increasingly automatable, reducing the expertise barrier.

Performance: The only real reason to consider a different approach is if prompt-based methods don't meet your performance needs.

Prompt Engineering

- **Skill Based Prompting** 2023 and Early 2024 Users need to acquire prompt engineering techniques like **Chain of Thought** and **ReAct** to guide the model effectively.

- **Automated Prompt Optimization** Late 2024 Application or Model developers add an optimization layer that refines and enhances prompts before they reach the model.

- **Self-Optimizing Models** >2025 The model automatically recognizes the nature of the query (e.g., reasoning tasks) and selects the best prompting strategy internally, without user intervention.

Skill based prompting - Simple rules

- **Zero-Shot:** Ask LLM to answer without examples
- **Few-Shot:** Including examples in your prompt reliably guides the model.
- **Role-Based Prompts:** Define a role (e.g., "You are a...") to set context.
- **Keep It Simple:** Use clear, straightforward English.
- **Comprehensive Explanation:** Share all relevant information to reduce ambiguity.
- **Natural Tone:** Instruct the model to respond as it would in everyday conversation (tonality control)

Skill based prompting - Complex rules

1. Chain of Thought (CoT)

- **Concept:** Breaks down complex problems into a series of logical steps.
- **How it Works:** Instruct the LLM to "think step by step" to show its reasoning.
- **Example:** For a math problem, the LLM will calculate intermediate steps before the final answer.
- **Use When:** The query requires complex reasoning or multi-step logic.

2. Decomposition Prompting

- **Concept:** Divides a large task into smaller, solvable sub-tasks.
- **How it Works:** You prompt the LLM to complete one sub-task at a time, then combine the results.
- **Example:** To write a full report, first ask the LLM for an outline, then for content for each section, and finally, for a combined draft.
- **Use When:** The task is too big or complex for a single prompt.

3. Ensembling Prompting

- **Concept:** Generates multiple answers to a single query and selects the best one.
- **How it Works:** Use different prompts or models to get varied responses. A "judge" model or human then picks the most accurate answer.
- **Example:** Get three different explanations for "black holes" and choose the clearest one.
- **Use When:** You need highly reliable and accurate answers, as it reduces the risk of a single bad response.

Automated prompting - Meta Prompting

Instead of requiring humans to learn and apply optimization techniques, what if you could integrate these techniques directly into a **model** and train it to generate useful prompts automatically?

Meta prompts are specifically designed to help the model generate prompts by leveraging best practices in prompt generation.

For example:

- https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/prompt-generator
- https://platform.openai.com/docs/advanced-usage

Automated prompting - DSPy

Link

Prompt based agents

LLM calling tools based on user prompts - rule based (?)

link link link

RAG

Fun fact

What's the coolest thing about AI Agents ?

- You can talk to them in natural language

What's the most painful thing about agents?

- They can take as much information as their **context length** (in fact much lower - needle in haystack funda) and they are stateless i.e they **forget** beyond their training data (ok …they used to forget a lot now it is getting better)

What are the implications and what can be done?

Implications :

- **Memory:** Any improvements or feedback provided through the prompt are short-lived and don't support continuous learning.
- **External Context:** In enterprise settings, there's GBs of data to learn from

Solutions:

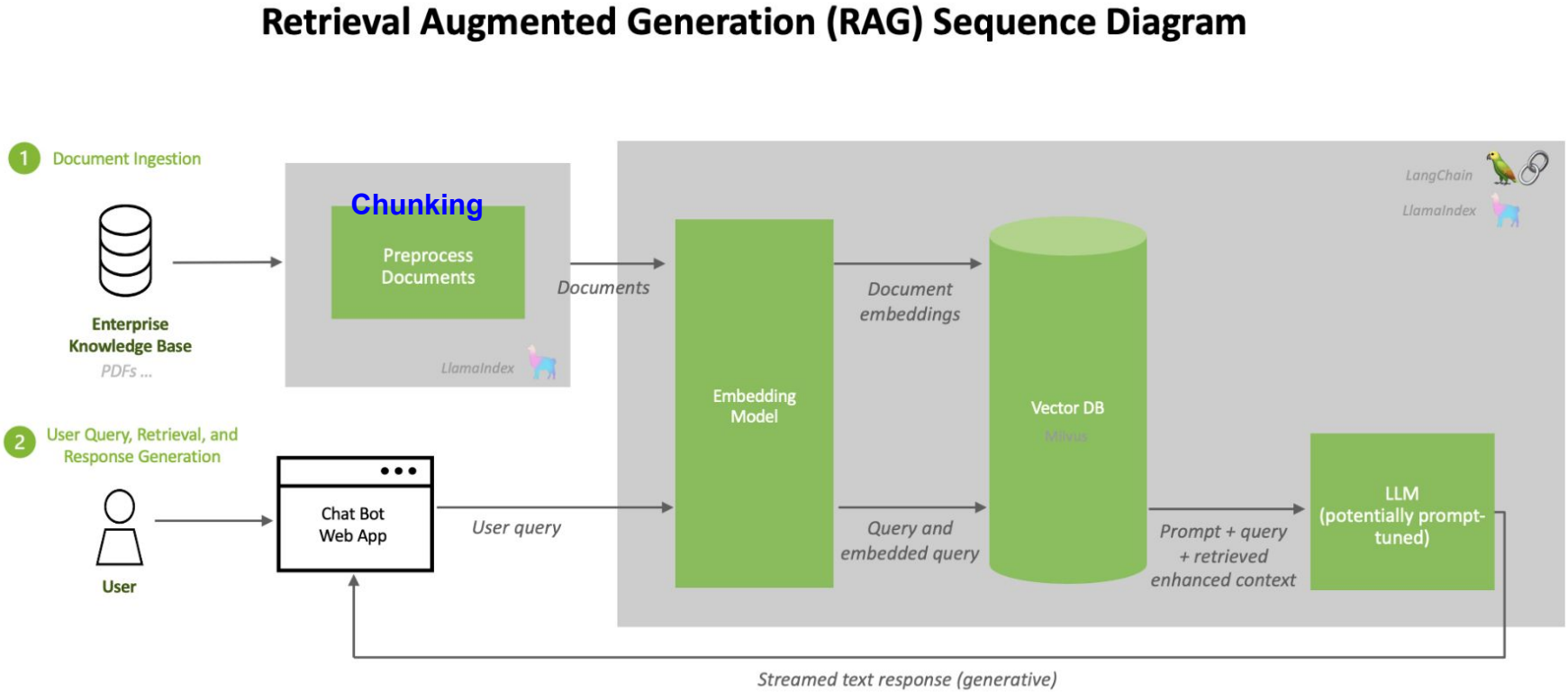
How can we enrich agents with contextual information as needed to:

- Provide enterprise-specific context
- Store and update memory of past conversations
- Dynamically incorporate relevant information from large datasets that don't fit within the LLM's context length

This is exactly where **Retrieval-Augmented Generation (RAG)** plays a crucial role.

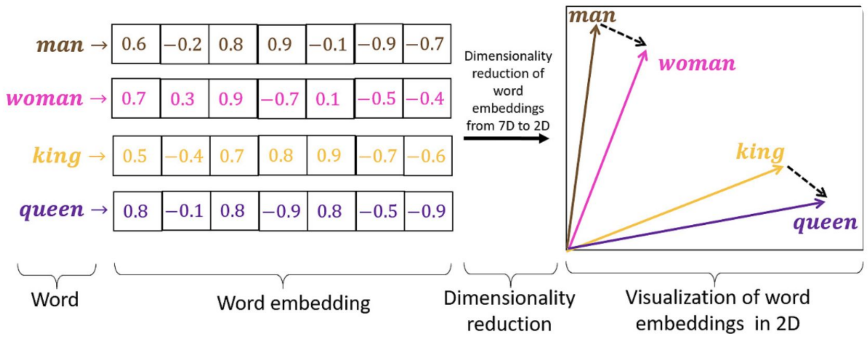
Enter RAG

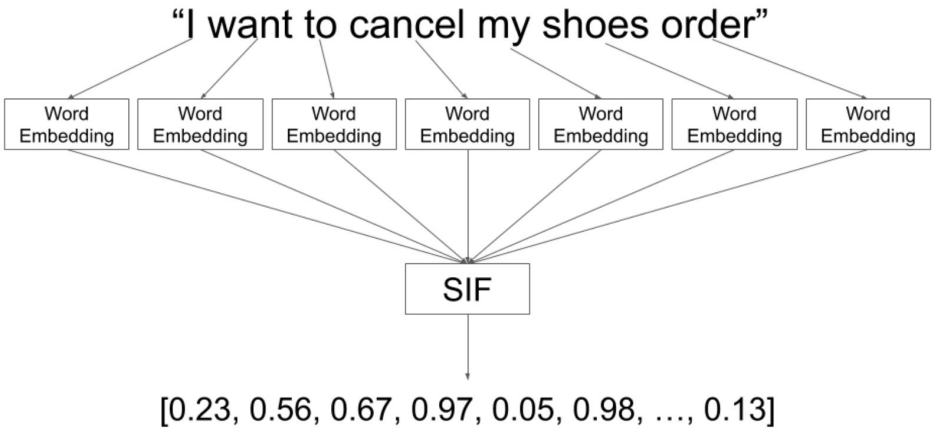
RAG is a technique that helps process large volume of data by efficiently retrieving and optimizing the most relevant information before passing it to LLM



link

What is Embedding ?





Word embedding represents a fundamental technique that transforms words into dense numerical vectors within high-dimensional space, where geometric relationships reflect semantic similarities between corresponding terms

Sentence embeddings extend the vector representation paradigm **from individual words** to encoding entire sentences, paragraphs, or documents
The fundamental distinction between sentence embedding vs word embedding lies in their scope and **contextual integration.**

Commonly used Embeddings:
General RAG → OpenAI (`text-embedding-3-large`) or `all-MiniLM-L6-v2`.

Cost-sensitive → `all-MiniLM-L6-v2`.

Accuracy-critical → `e5-large` or OpenAI `3-large`.

Multilingual → Cohere or LaBSE.

Domain-specific → BioBERT, FinBERT, LegalBERT.

Link

Calculating Similarity

Semantic similarity or **Vector Similarity** measures how close two pieces of text (words, phrases, or sentences) are in meaning by calculating the distance between their embeddings (vector representations).

- High semantic similarity: The two texts have nearly the same meaning.
- Low semantic similarity: The texts are very different in meaning.

For example: semantic_similarity(vector of sentence 1, vector of sentence 2) = 0.58

- 0.0 → No similarity (completely different meanings)
- 0.5 → Somewhat related (partial overlap in meaning)
- 1.0 → Identical meaning (very close in context)

Vector DB

A vector database is a specialized database designed to store, index, and query high-dimensional data as **vectors**, or numerical representations called **embeddings**

Feature	Traditional Relational Databases (e.g., MySQL, PostgreSQL)	Vector Databases (e.g., Pinecone, Weaviate)
Data Structure	Organized in tables with rows and columns, enforcing a strict schema.	Stores data as high-dimensional vectors (arrays of numbers).
Data Type	Optimized for structured data.	Designed for unstructured and semi-structured data.
Query Mechanism	Uses SQL for exact matches, joins, and filters based on predefined criteria.	Uses similarity search (vector search) to find data points with a similar meaning.
Search Type	Keyword-based search; finds exact matches.	Semantic or vector similarity search; finds data based on contextual meaning.
Primary Use Case	Transactional systems, data analysis, and applications requiring data integrity.	AI/ML applications, semantic search, recommendation engines, and chatbots.

RAG Evaluation

Retrieval Metrics (Assess how well relevant documents are retrieved)

- **Context Recall** – Measures how often the correct document appears in the top K retrieved results.
- **Context Precision** – Measures the proportion of relevant documents among the top K retrieved.
- **MRR (Mean Reciprocal Rank)** – Evaluates how high the first relevant document appears in the ranked list.
- **NDCG (Normalized Discounted Cumulative Gain)** – Weighs the relevance of retrieved documents, giving more importance to higher-ranked ones.

Generation Metrics (Can be semantic match or LLM Judges):

- **Faithfulness /Hallucination Rate** – Measures how well the generated response aligns with retrieved documents, avoiding hallucinations (Also called hallucination score)
- **Relevance** – Evaluates how useful and contextually appropriate the response is.
- **Fluency** – Checks whether the output is coherent and well-structured.
- **Factuality** – Ensures the generated response is factually accurate.

Link and Link

Enterprise RAG Design considerations

Decision Factors

- Document Parsing problem ?
- How should documents be chunked?
- What models can I use for generating vectors?
- How to setup my retrieval pipeline?

Retrieval Problems:

- What if chunks retrieved do not contain the right answers?

Generation Problems:

- What if the model cannot access information from context?

Optimization:

- Cost/Latency
- Performance

Document Parsing related problems

Traditional OCR Functions

- Amazon Textract – Ideal for structured documents.
- EasyOCR – Open-source and supports multiple languages.
- Docling – A powerful open-source document parsing tool.

Handling Images and Charts: Use Multimodal Embeddings (should be handled separately)

When PDFs are Clumsy: Use Multimodal Models

- Models like GPT-4o can directly process images, extract text, and even interpret tables, charts, and handwriting.
- These models understand context, making them useful for messy or unconventional document formats.
- Latest, Mistral: https://mistral.ai/news/mistral-ocr
- Use with caution: https://huggingface.co/spaces/echo840/ocrbench-leaderboard

Chunking related Issues

Different Chunking Strategies	Best for	How it works
Fixed-size Chunking	Simple documents, quick prototyping	Breaks text into chunks of a predefined size, often with a slight overlap to maintain context.
Sentence Splitting	Documents where sentence boundaries are clear	Splits text into chunks based on sentence endings (periods, question marks, etc.).
Recursive Splitting	Complex documents with varying structures	Iteratively splits text using different delimiters (e.g., paragraphs, then sentences, then words) until chunks are small enough.
Semantic Splitting	Documents where topics change frequently	Uses an LLM to identify changes in topic or meaning and splits the document at these semantic breaks.
Parent-Child Chunking	Maintaining context for questions that need detail	Creates small, detailed chunks for retrieval and pairs them with larger, parent chunks that provide broader context.

Loss of Context: When text is split at arbitrary points, sentences or logical units can be broken, scattering relevant information across multiple chunks.

Irrelevant Information: If chunks are too large, they can contain multiple topics, diluting the main semantic meaning and leading to less precise search results.

Incomplete Answers: The complete answer to a user's question might be spread across several different chunks, and if only one is retrieved, the LLM will provide an incomplete response.

Out-of-Order Chunks: Naive chunking can lead to adjacent chunks being presented to the LLM out of their original order, which causes confusion and can lead to hallucinations.

Popular Vector DB providers and when to use?

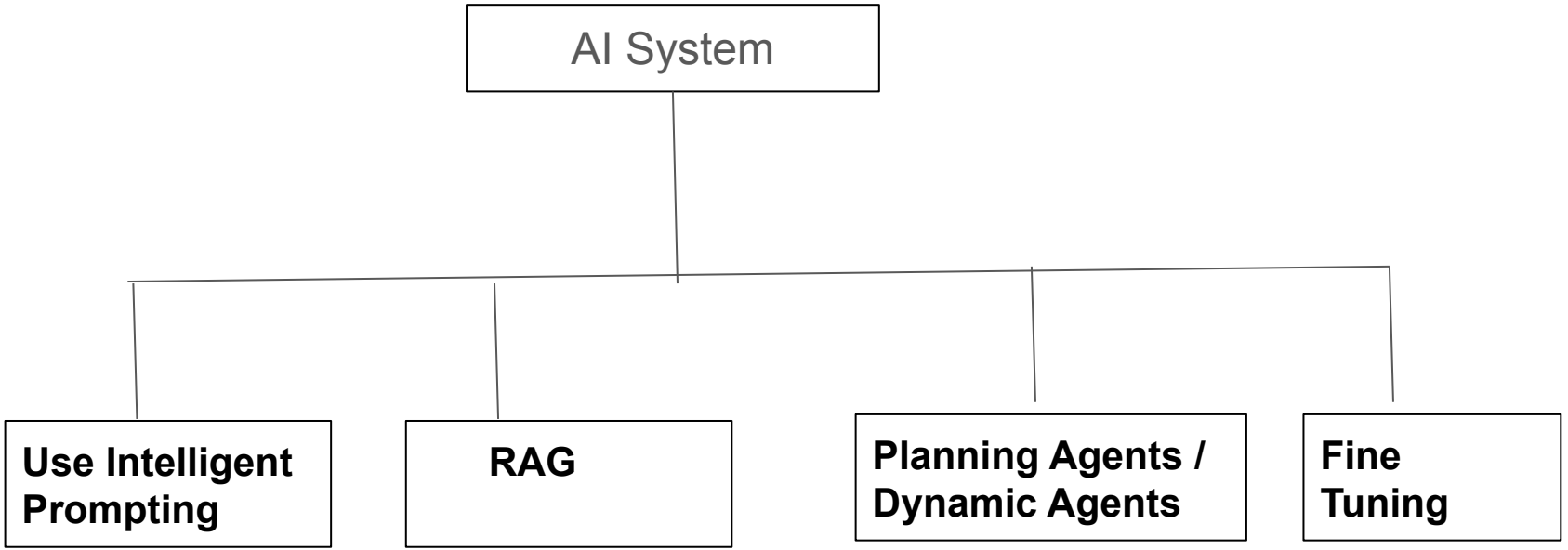
Vector Database	Type/Description	When to Use
ChromaDB	Open-source, in-memory	**Small-scale projects**, **prototyping**, and local development. It's fast and easy to set up, but data doesn't persist.
Pinecone	Fully-managed, cloud-native	**Large-scale, production-grade RAG applications,** especially for enterprises needing high performance and scalability with minimal management overhead.
Weaviate	Open-source (self-hosted or managed)	Projects that require a **production-ready solution with flexibility and control**. It has built-in vectorization and strong metadata filtering capabilities.
Qdrant	Open-source, built in Rust	High-performance, scalable, self-hosted solutions where developers need granular control, advanced filtering, and cost efficiency.
Milvus	Open-source, cloud-native	**Applications with massive datasets** (billions of vectors) and requirements for extreme performance and real-time similarity search.

Choice of Search algorithm

Algorithm/Method	Description	Key Features & Use Case	When to Use
BM25	A keyword-based ranking function that scores documents based on the frequency of query terms, their rarity across the document corpus (Inverse Document Frequency or IDF), and document length.	✅ Efficient for exact keyword matches and well-defined queries. ❌ Lacks semantic understanding; can fail with synonyms or paraphrased queries.	When the user query contains specific, well-defined keywords or entities, and the documents are less conceptually complex. For example, searching for a specific product ID, a person's name, or a technical term in a structured knowledge base.
TF-IDF	A statistical measure that evaluates the importance of a word in a document relative to a corpus. It's a precursor to BM25.	✅ Simple and effective for basic keyword retrieval. ❌ Similar to BM25, it does not capture semantic meaning and can be less effective than BM25 in some cases.	In scenarios where simplicity and computational efficiency are the top priorities and the dataset is small or the queries are strictly keyword-based.
Cosine Similarity	Measures the cosine of the angle between two non-zero vectors. A score close to 1 indicates high similarity, and a score close to 0 indicates low similarity.	✅ Excellent for semantic search with vector embeddings, as it's not affected by vector magnitude. ❌ Requires pre-processing to create vector embeddings.	When the user query is more conversational or conceptual, and you need to find documents with similar meaning, even if they don't share keywords. For example, finding articles about "climate change mitigation strategies" when the query is "how to reduce global warming".
Dot Product	A measure of vector similarity that takes both the angle and the magnitude (length) of the vectors into account.	✅ Can be faster to compute than cosine similarity. ❌ Highly influenced by vector length, which can skew results if embeddings are not normalized.	In specialized applications where vector embeddings are normalized or where the magnitude of the vector is a meaningful feature, such as in certain recommendation systems or when speed is critical.
Hybrid Search	Combines a keyword-based method (like BM25) and a vector-based method (like Cosine Similarity) to create a single, combined score.	✅ Balances precision (from keywords) and recall (from semantic meaning). ❌ More complex to implement and tune than a single method.	As a general, robust solution for most RAG applications. Use it when queries can be a mix of specific keywords and broad, conceptual questions to ensure high-quality retrieval.
Re-ranking	An additional step after the initial retrieval to re-order the top documents. A separate model (often a cross-encoder) scores the query-document pairs to produce a more refined ranking.	✅ Significantly improves the relevance of the retrieved documents. ❌ Adds latency and computational cost to the retrieval process.	When the initial retrieval set is large or contains many potentially relevant but not perfect documents. Use it to refine the results of a hybrid or semantic search to ensure the best documents are presented to the LLM.

80% problems in RAG development are in document parsing and retrieval

Choosing or designing the right AI Application



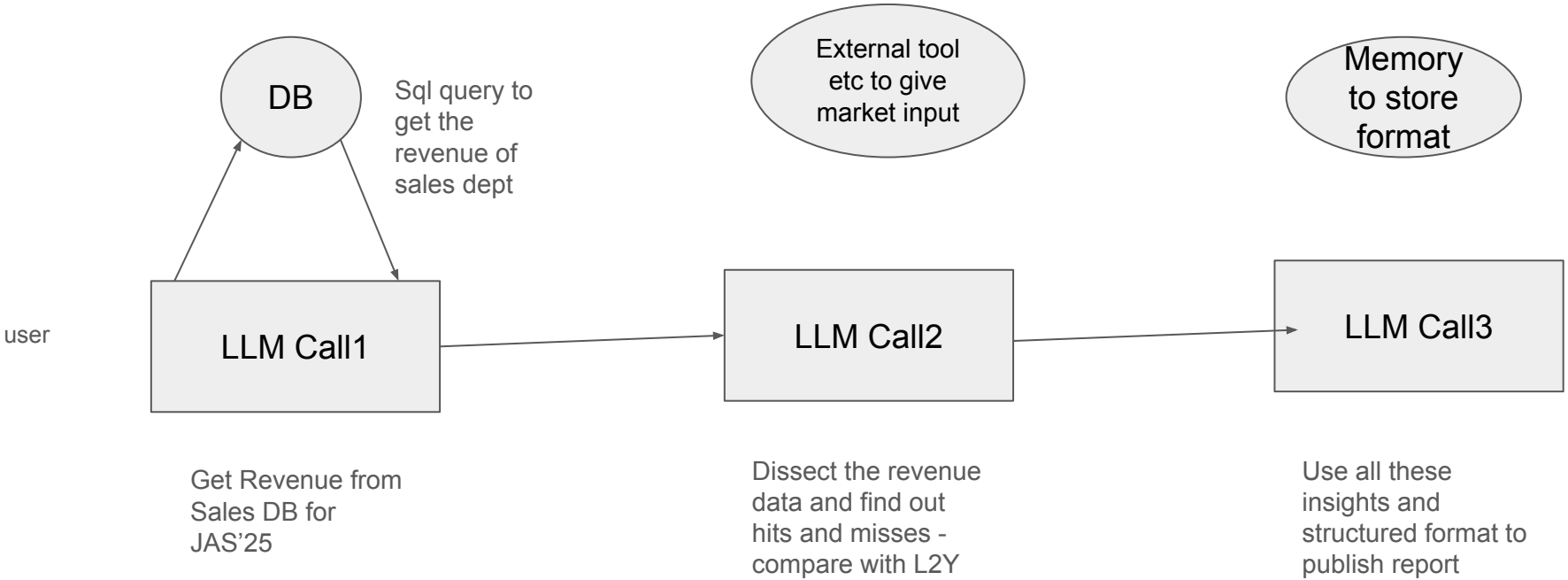
Difference between Prompt based agents vs Dynamic Agents

In **Prompt based workflow agents** , Al models focus on understanding and generating content, while planning and orchestration are typically handled by humans (Through pre-defined code). (Only Action Autonomy)

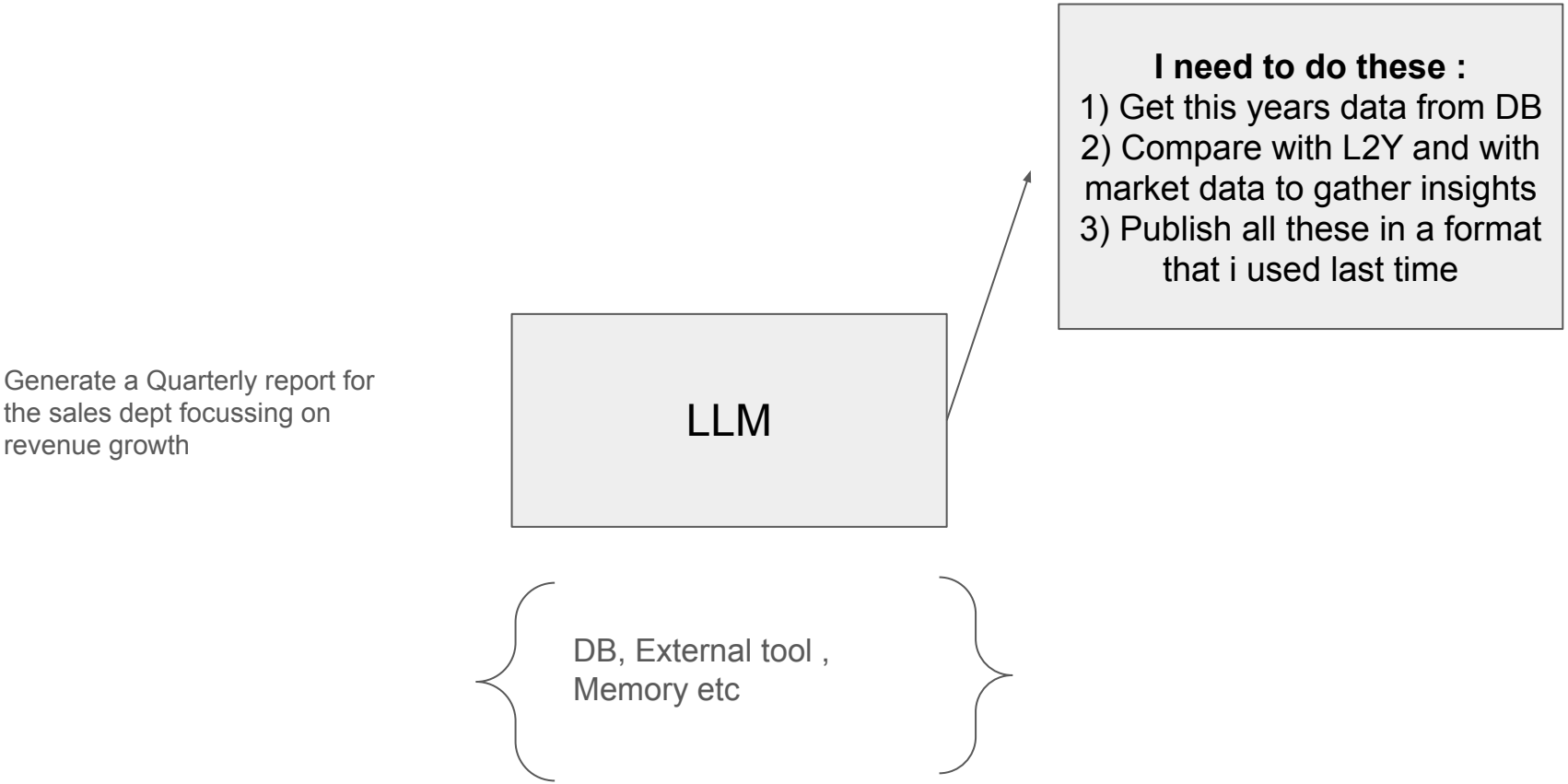
The key factor that sets **Dynamic or planning agents** apart from other agents—is the kind of planning autonomy they exhibit. (Action Autonomy + Planning Autonomy)

Prompt Based Agents - Example

GOAL : Generate a quarterly performance report for our sales department focussing on revenue growth



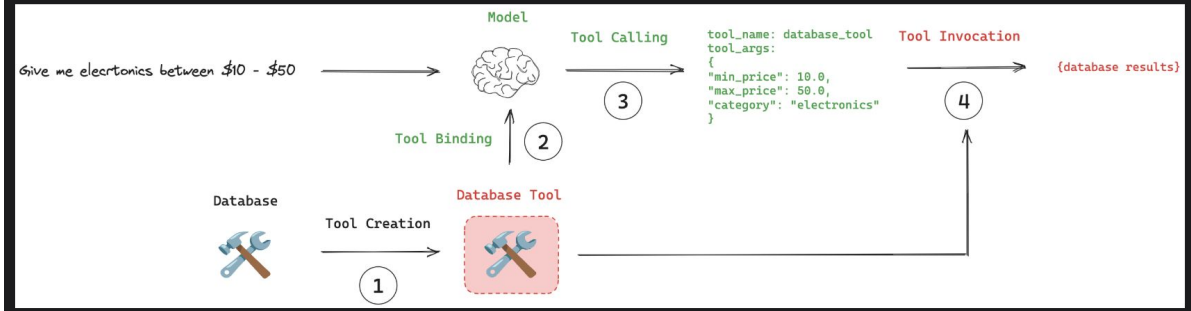
Autonomous / Planning Agents



How Tool calling works

Key concepts

1. **Tool Creation:** Use the @tool decorator to create a tool. A tool is an association between a function and its schema.
2. **Tool Binding:** The tool needs to be connected to a model that supports tool calling. This gives the model awareness of the tool and the associated input schema required by the tool.
3. **Tool Calling:** When appropriate, the model can decide to call a tool and ensure its response conforms to the tool's input schema.
4. **Tool Execution:** The tool can be executed using the arguments provided by the model.



Tool calling steps:
- Agent is given a JSON description of tools.
- Agent **decides** the right tool to call with arguments - doesn't call the tool.
- **User executes the tool** and returns the result.
- Repeat above two steps until the plan is completed.
- Agent finally returns the answer.

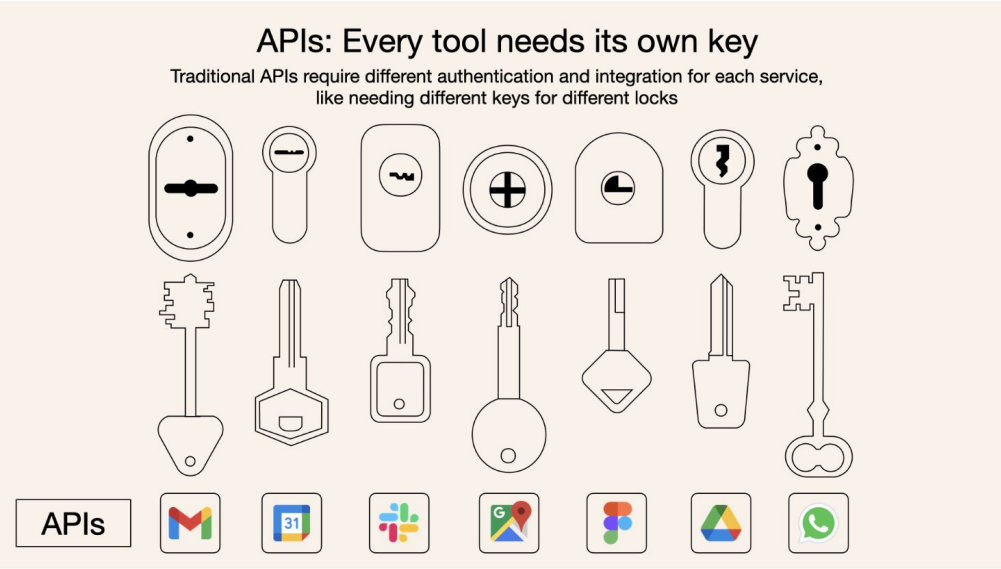
Tools can provide both read/write functionality. E.g. Check for flights, book a flight.

Link

Problem with tool calling

Every LLM provider and every tool has different custom glueing

For M LLM provider and N tools possible combinations are like MxN - there is no standardization and very difficult to scale



Enter MCP

Link

What do you mean by Agentic Frameworks ?

- **Langchain** (misses orchestration - it is not suited for agentic application ; it only able to support applications with sequential /chain like LLM calls. No dynamic / planning agents can be written using Langchain)
- **Langgraph** (Agents and tools are nodes connected via graphs ; inspired by statemachine) . **Coding heavy** ; its extremely flexible and low leve agentic framework
- **Autogen** (They focus a lot on different types of multi agent conversations - 1x1 , group , sequential , nested) Very strong orchestration ; very resilient in enterprise setup **(Mid Code) debate heavy agents**
- **CrewAI** (Team philosophy , good for multi agent workflows ; each team member is an agent. They have specialized roles assigned and they either collaborate among themselves or they talk to their manager) - **Low Code**
- **Agno (Phidata)** - very similar to crewAI ; it is like writing a automation script (lagging behind , but like crewai user friendly)

All these frameworks abstracts the process of building , orchestrating and managing AI Agents doable

- State management , memory management , Communication protocol between agents
-

Agenda :

Goal : Knowing and learning Agentic Frameworks and building Agentic App

- Langflow (Low no code)
- Langgraph
- CrewAI
- Flowwise / n8n (this is not this week)

Github link : https://github.com/agenticgogol/Classcode_B2

Question Link: Link

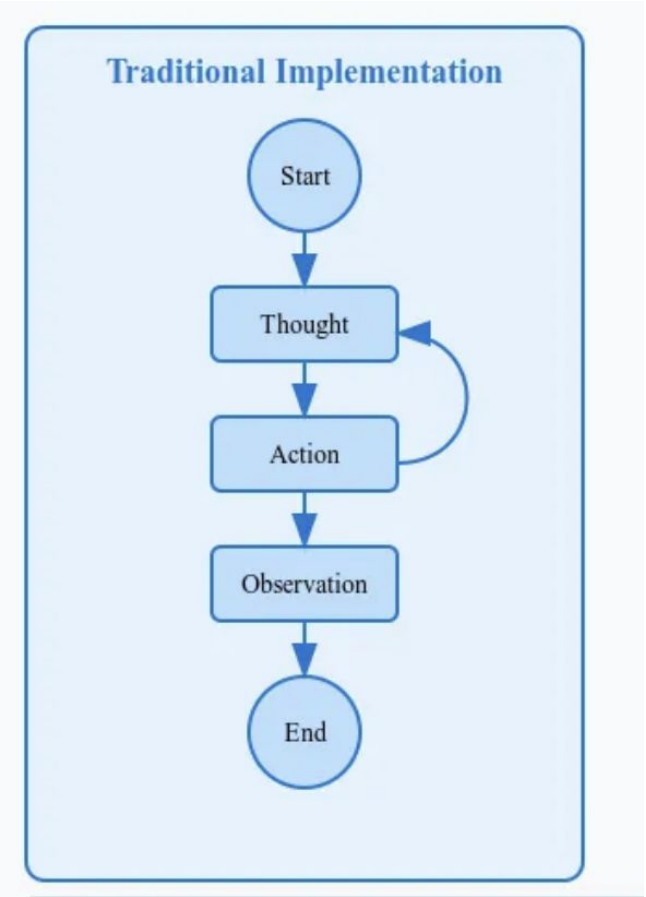
Langflow (No / Low Code) installation

1. Create a directory in your local system and go there from terminal
2. Create a virtual env and activate the same
   - uv venv --python 3.11.8
   - uv python pin 3.11.8
   - source .venv/bin/activate
3. Install Langflow
   - uv pip install langflow==v1.2.0
4. Run Langflow
   - python -m langflow run
5. Open the langflow on browser

Langgraph

1. Nodes , edges , graph that we have to build
2. Nodes - functions , agents (with LLM)
3. Edges - fixed edges , conditional edges
4. Tool definition custom (with descr string) and tool binding with LLM
5. How to call standard built in langchain tools
6. Multiple tools that are binded with LLM (LLM based on question figures out how to call the right tool) - description string

ReACt Agentic Pattern (Think - Act -> Observe in loop)



ReAcT pattern is especially helpful :
When in order to achieve a business goal , we have to call multiple tools and very often outcome of one tool is input to the other ones

Multi step processes - we cannot just call one tool and finish …based on the output of one tool , LLM decides what is the next action to be taken

Agentic RAG

1. Just like what we saw when there are multiple tools , bind with LLM - the agent at the runtime figures out which tool to call based on user query

2. Similarly , in Agentic RAG we can have multiple external data sources bind with LLMs - the agent at the runtime figures out which tool to call based on user query.

Ideally show the code for basic rag —> agentic rag

Basic rag theory discussed

Agenda D3 (in the process of covering D2)

1. Basic RAG
2. Multimodal RAG
3. Agentic RAG with multiagents
4. Agentic RAG with multiple DBs
5. Adaptive RAG
6. RAG Eval
7. RAG Deployment

Basic RAG Steps :

Within MultiModalRAG Class :

__init__

1. Gemini Setup (Which LLM to use)
2. Embedding Initialization (Which Embedding to use)
3. ChromaDB Initialization (vectorDB) (Which VectorDB to use + Collection)
4. Text Splitter (Chunking) (Which Splitting mechanism to use)
5. Session Memory

Basic RAG Steps :

Within MultiModalRAG Class :

Extract_text_from_documents - We will extracting text from PDF, PPT , TXT DOCX all these formats

Process image with gemini : Image data to Gemini Vision for OCR and content understanding

Add Documents to Knowledge Base (vectorDB) :

- We will call Extract_text_from_documents and get the text
- Split text into chunks
- Generate embedding for each chunks
- Store embedding or vector representation vectordb

Basic RAG Steps :

Within MultiModalRAG Class :

Search Knowledge Base :

- Converts user query into embedding
- Searches text and image collection separately
- Combines the results and sorts by importance and returns top k documents

Generate Answer :

- Use retrieved document from above function as context
- Prompts the LLM to generate a comprehensive answer
- Retruns the answer with source document name

Get Session info:

- Returns details of all uploaded documents in the current session
- Tells us chunk count for all the documents

Basic RAG Steps :

Outside of MultiModalRAG Class :

Initialize system:

- Initializes the multimodal class and passes the LLM Keys

Gradio / UI Related coding:

Agentic RAG - Multi DB

This Agentic RAG System handles Multiple Vector Databases each specialized for specific domain / topic. Few components :

1. Gemini LLM : Used for reasoning and generating answers
2. OpenAI Embedding : for semantic search across all the databases
3. Chroma DB : store embeddings
4. VectorDBRegistry: Dynamically chooses the most relevant DB for a given query
5. Gradio UI : web interface

Agentic RAG - Multi DB

1. Class VectorDBRegistry:
- Manages multiple vector databases and dynamically selects DB for a query
- register_db()
  - Registers a database with its domain keywords
  - Computes the embedding for keywords to represent DB topic vectors
- choose_db(query)
  - Choose top -K DBs based on cosine similarity with query embedding
  - (When user asks a query that query is embedded and compared with the topic vectors To identify which database RAG should refer to)
- get_db

Agentic RAG - Multi DB

1. Class AgenticRAGMultipleDB:
   - __init__ (initializes LLM , Embeddings, VectorDBRegistry, TextSplitter)
   - register_chroma_db (creates a chroma db instance and registers it with the keywords in the registry. It enables dynamic selection based on query)
   - **Ingest_to_db** (get text , split text, embed chunks, metadata, stored)
   - **Retrive_for_query** (compute the query embedding , call the vectordDBregistery class to find relevant database and within the database relevant chunks , pass the retrieved context to Gemini LLM for answer generation)
   - Ask_query (inside this function we call **Retrive_for_query to get the context and then call the LLM directly)**

Agentic RAG - Multi DB (Implementation approach-1)

Ingestion :

- We will upload documents
- Based on the document name : we will convert the document name to embedding
- We will map the domain keywords mapped to database into embedding
- We will compare the domain keyword embedding with filename to decide which DB the document should be ingested

Retrieval :

- User asks a query . That query is converted to embedding
- That query embedding is matched to domain keyword embedding to finalize which DB to call
- Then within that DB again a semantic search happens between query embedding and vector store
- Retrieved context is pushed to LLM for answer generation

Agentic RAG - Multi DB (Implementation approach-2)

Ingestion :

- We are getting documents related to topic1 and related to topic2
- Separately all the documents are converted to vectors and stored in 2 different databases after going through chunking and embedding
- After creating the database we are adding a descriptions for each of the databases

Agent Code :

- Inside this we are instantiating an agent (LLM + Tools) and we are binding the agent with tools (in this case the vector databases)
- Remember , the agent as expected has access to query the LLM

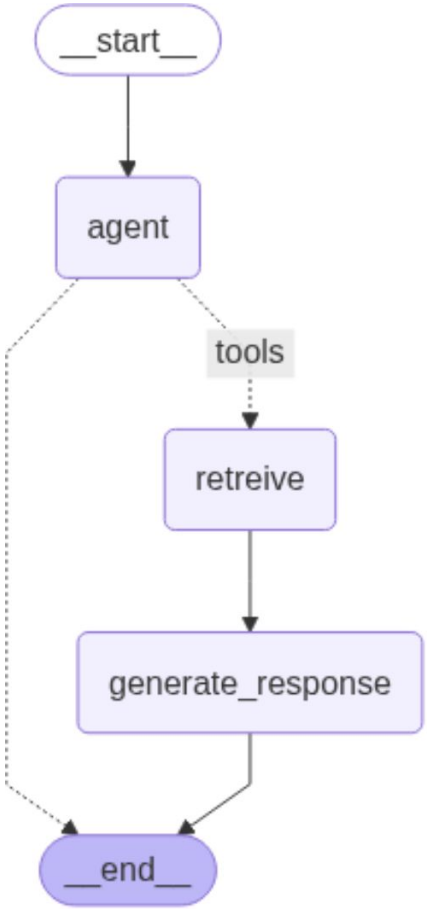
Agentic RAG - Multi DB (Implementation approach-2)

Function generate_response :

- Context + query + prompt to send it to LLM
- Inside this I have used prompt = hub.pull("rlm/rag-prompt")

Langgraph Workflow:

- Agent decides which tool to call or it directly says I don’t know
- Agent sends the user query and the VectorDB descriptions - both to LLM
- LLM tells us which DB to use
- And the answer from that DB is then again passed to LLM and answer is generated



Difference between two approaches

1. Matching of Query to DB is done inside our system (match the embeddings ourselves) **[more control)**

2. Whereas when we bind the vectordb + descriptions with the LLM in Langgraph this choice of which DB to use is done by the LLM **(more autonomy)**

Multi Agent RAG

Multi Agent Workflow:

- Researcher Agent : Searches the knowledge base i.e Vector DB for relevant documents and from that calls LLM to extract key facts and gaps
- Synthesis Agent : Combines the research findings into coherent answer (takes help of LLM)
- Fact Checker Agent : Validates the synthesized answer for consistency and accuracy (again takes help from LLM)
- Follow-up Agent : Generates contextual follow up question for further exploration to the user
- Coordinator Agent : Orchestrates all agents and formats the final answer

RAG Evals

Retriever Evaluation (top K semantic similar documents that are retrieved) :

Recall@K: % of queries where at least one relevant document is in the top K retrieved result (You ask 100 questions to a RAG and 70 of them in the top 3 document you have a relevant document … the recall is 70%)
Precision@K: % of top K documents that are relevant (you have 3 retrieved docs for a given question - and 3 of them are relevant to the question - P = 100%)

- **Manual labels of query to relevant document**
- Synthetic query and relevant document generation (you give LLM the responsibility to label)

Generator Evaluation :

- **Faithfulness :** Does the answer of generation is using the retrieved data (faithful to retrieval)
- **Completeness ;** Did the answe cover all the relevant points
- **Factual correctness :** is the answer factually correct
- **Grammatical / language correctness :** is the language similar to how human talk

- **LLM as a judge** to check the faithfull , correctness and language coherence (use more powerful LLM that that is used for generation)

- Cosine Similarity or semantic similarity between retrieved and generated documents

<HUMAN EVALUATION> in RAG is very important for end to end performance measurement
However, industry is now relying more on LLM based automated evaluation

Agenda D4 (in the process of covering D2)

1. Langgraph - Single Agent , Multiple Agent , Cost / Latency, Eval
2. CrewAI - Single Agent , Multiple Agent , Cost / Latency, Eval
3. Adaptive RAG
4. RAG Eval
5. RAG Deployment

Recap of Langgraph whatever we have seen so far :

1. Step1:
   a. In Langgraph , we will have **nodes that can be functions** (and…)
   b. All the nodes are connected in a graphical format (Start -> next node -> next node)
   c. The connection can be fixed edge or conditional edge
2. Step 2:
   a. We have langchain_openai, langchain_grow , langchain_gemini ->ChatOpenAI(), ChatGroq()...using these we connect with LLM . We will have to mention keys and model name . Keys can be user input or can be read from env file
   b. We also defined **Agents** here : (Functionality + LLM + optional tool calling capability is agent)
   c. We also said agents / nodes pass information among themselves via "**State**" . State is a class which can have any number of variables . Every agent has access to this state object .
   d. Nodes can be agents also

Recap of Langgraph whatever we have seen so far :

1. Step3:
   a. We can define our custom tool (i.e function) . These tools should have a description string about what they do
   b. llms_need to be bound to the tools for it to be used - bind_tools
   c. In a workflow , if you have multiple agents each of them can have different tools available (in your code A1 → LLM1 with t1,t2 and Agent A2-> LLM2 with t3,t4)
   d. **In Langgraph tools are also nodes**
   e. Tools are always called from an Agent (agent has access to LLM and LLM has binded itself with tools)
   f. Conditional edges from agent to tools is possible

Recap of Langgraph whatever we have seen so far :

1. Step4:
    a. There are inbuilt tools offered by langchain (Link). These inbuilt tools have already description strings inside itself
    b. Rest of the process is similar …binding and then llm deciding which tool is call when is similar
    c. **ReACT agent :** Here we learnt that tools do not need to give the final answer. The preferred or more common approach in enterprise setting is tools will give the response back to Agent and Agent will then decide the next step . (THINK -> ACT -> OBSERVE).
    Think -> Agent decides what tools to call or what to do ?
    Act —> tool does the action
    Observe —> Once tool gives the result back to Agent , based on that the next step is taken

Single Agent and MultiAgent Workflow using langgraph

We defined :

```
class AgentState(TypedDict):

    messages: list
    topic: str
    research: str
    synthesized: str
    fact_checked: str
    edited: str
    summary: str
    metrics: dict
```

- Structured state object passed between agents
- The beauty of this structure is every agent can modify certain fields and the other agent can read from it

What are the differences between Single Agent and MultiAgent architecture ?

	Single Agent	Multi Agent
Definition	One Agent handles entire task end to end	Multiple agents will collaborate or compete to solve a specific task (It means every agent will likely to do subtasks)
How it works	Query --> Agent will call LLM to plan & reason ---> tool usage (if needed) ---> final output	Orchestrator / Manager assigns subtasks --> other agents will work ---> results are exchanged --> final aggregation
Complexity	Sinpler to design , implement and debig and manage	More complex due to coordination and role assignment
cost / latency	Lower cost (lower latency ?)	Higher Cost + Higher latency (because multiple agents will call LLMs multiple times and there is orchestration and collaboration overhead)
When to use?	Repetitive , straightfwd usecase (Chatbot Q&A)	Multi Step process , Workflow is complex with multiple decision points , and there is a need of specialized skills

A sample agent in Langgraph

```python
def ResearchAgent(state: AgentState) -> AgentState:
    prompt = f"""
    You are a Research Agent.

    Task: Gather relevant, high-quality information on the topic below.
    - Cover definitions, background, current trends, and key debates.
    - Include real-world applications or case studies if relevant.
    - Present findings in a structured way (not an essay, not bullet-only).
    - Keep it neutral and fact-rich.

    Topic:
    {state['topic']}
    """
    state["research"] = call_llm_with_metrics("ResearchAgent", prompt, state)
    state["messages"].append("research_done")
    return state
```

1. AgentState is input and AgentState is output
2. Every Agent enhances the prompt by using a placeholder that it fills from state object
3. Every Agent as expected calls the LLM
4. And then output of the agent is stored in another variable within state object to be used by some other agent

Multi Agent langgraph workflow after defining the agents

```python
def build_sequential_agent_graph(llm_param):
    wf = StateGraph(AgentState)
    wf.add_node("research", ResearchAgent)
    wf.add_node("synthesize", SynthesizeAgent)
    wf.add_node("fact", FactCheckAgent)
    wf.add_node("edit", EditAgent)
    wf.add_node("summary", SummarizeAgent)
    wf.set_entry_point("research")
    wf.add_edge("research", "synthesize")
    wf.add_edge("synthesize", "fact")
    wf.add_edge("fact", "edit")
    wf.add_edge("edit", "summary")
    wf.add_edge("summary", END)
    return wf.compile()
```

1. For each agent u r defining a node
2. And then connecting them through edges
3. These edges in this case are sequential

Multi Agent langgraph workflow after defining the agents

```python
def build_parallel_agent_graph(llm_param):
    wf = StateGraph(AgentState)
    def parallel_stage(state: AgentState) -> AgentState:
        synth_state, fact_state = state.copy(), state.copy()
        with ThreadPoolExecutor(max_workers=2) as ex:
            futures = {
                ex.submit(SynthesizeAgent, synth_state): "synth",
                ex.submit(FactCheckAgent, fact_state): "fact"
            }
            for fut in as_completed(futures):
                res = fut.result()
                if futures[fut] == "synth":
                    state["synthesized"] = res["synthesized"]
                else:
                    state["fact_checked"] = res["fact_checked"]
        return state
    wf.add_node("research", ResearchAgent)
    wf.add_node("parallel", parallel_stage)
    wf.add_node("edit", EditAgent)
    wf.add_node("summary", SummarizeAgent)
    wf.set_entry_point("research")
    wf.add_edge("research", "parallel")
    wf.add_edge("parallel", "edit")
    wf.add_edge("edit", "summary")
    wf.add_edge("summary", END)
    return wf.compile()
```

1. See the nodes first … there is a new node named as **parallel**..which is not any specific agent :

2. This is a design call on which tasks/ which agents can operate parallely

3. Parallel is a node / combined agent that combines synthesize and fact check agent and it is run on parallel threads to save time

Multi Agent langgraph workflow after defining the agents

```python
def build_single_agent_graph(llm_param):
    workflow = StateGraph(AgentState)
    def single_node(state: AgentState) -> AgentState:
        start = time.time()
        resp = llm_param.invoke(f"Do research, synthesize, fact-check, edit, and summarize:\n{state['topic']}")
        edited = _extract_content(resp)
        summary_resp = llm_param.invoke("Summarize in 3 sentences:\n" + edited)
        summary = _extract_content(summary_resp)
        end = time.time()
        in_tokens = estimate_tokens(state["topic"])
        out_tokens = estimate_tokens(edited + summary)
        model = getattr(llm_param, "model_name", "gpt-4o-mini")
        pricing = PRICES.get(model, PRICES["gpt-4o-mini"])
        cost = in_tokens * pricing["input"] + out_tokens * pricing["output"]
        state["edited"], state["summary"] = edited, summary
        state.setdefault("metrics", {})["SingleAgent"] = {
            "latency": round(end - start, 2),
            "in_tokens": in_tokens,
            "out_tokens": out_tokens,
            "cost": round(cost, 6),
        }
        return state
    workflow.add_node("single", single_node)
    workflow.set_entry_point("single")
    workflow.add_edge("single", END)
    return workflow.compile()
```

Evaluation function

```
def evaluate_outputs(single_out, seq_out, par_out):
    global llm
    judge_prompt = f"""
You are an impartial judge tasked with evaluating three different outputs.
Evaluate each output against the following metrics on a scale from 1 (very
poor) to 10 (excellent).

### Metrics
1. **Accuracy**: How factually correct and relevant is the response to the
intended question or task?
2. **Coherence**: How logically consistent, well-structured, and easy to follow
is the output?
3. **Conciseness**: Does the output avoid unnecessary repetition or verbosity
while still being complete?

### Instructions
- Compare the three outputs independently, not relative to each other.
- Assign integer scores from 1 to 10 for each metric.
- Provide a one-sentence justification for each score.
- Respond in strict JSON format.

### Outputs to Evaluate
"single": {single_out}

"sequential": {seq_out}
```

Evaluation can be integrated within the agentic application or outside the application via observability tools .

The beauty of using observability tools is it helps you to debug and track all historical evaluations. So there is storage of eval and automated scoring that is available

Different kind of patterns in Langgraph for Agentic workflow creation

Link

1. **Prompt chaining** : Example :
2. **Parallelization:**
   Example : Product GTM Planning agent
   Run MarketResearch , Marketing plan creation , Finance budgeting , initial pilot result analysis
3. **Router :**
   Example : IT Support Assistant
   Based on the user query → it will route it to cloud team , security team , hardware team , FAQ bot
4. **Orchestrator worker**
   Example : User asks a question Plan a 5 day trip to Japan
   Orchestrator: Makes a plan to answer this question by breaking it into subtasks and based on number of subtasks it will dynamically create n number of agents (Worker 1 - find flight , Worker 2 - find hotel Worker 3 - find sightseeing)
5. **Evaluator Optimizer:**
   Example : Code Writing Agent :
   Generator agent generates the code
   Evaluator checks the code with respect to requirement doc , coding practice , guidelines… etc
   Generator now revises the code based on evaluator findings
6. **Human in the loop :**
   Example : Medical Disease diagnosis System , Bug fixing agent
   AI Suggests likely issues from patient data
   Doctor verifies before making the final call (AI is improving the human productivity)

Single class vs Multiple Class difference

1. Multiple class is used when each agents or some number of agents maintains its own state class specific to his role
2. Agent1 : a state class called researchstate
3. Agent2 : a state class called analysisstate
4. Agent3: a state class called as factcheckstate
5. Orchestrator or a coordinator agent who will take a megastateclass {researchstate, analysis, factcheckstate}
6. Pros : Clear boundaries on states mean there is lesser chance of overwriting states , modular workflows , scaling on enterprise system
7. Cons: Overhead because of orchestration

—

The single class will become very big if you add too many sub variables / sub tasks as a result of that it will be harder to manage in enterprise setup when we scale to many subsystems …
(linear usecases this is very good …easy to build)

Different runnable in Langgraph

LLM as a judge :

Where another LLM model is given the user query and the agentic output to evaluate on certain parameters

LLM as a synthetic data generator :
You have created synthetic ground truth data using better LLMs and then evaluate your agentic solution vs that

Human based evaluation :
If you have ability to capture end user feedback - able to evaluate your performance → user feedback is generating more label data

Rule based evaluation:
Bunch of hard coded rules (cosine similarity > sth , upto date , how many steps

Supervisor , Swarm pattern

CrewAI (v low code)

1. **CrewAI is open-source (orchestration engine is released for everyone)**
2. It is free to use under MIT license
3. However, there is something called as CrewAI+ or enterprise suite - this includes lots of features like no code / dashboards for monitoring / security control , scaling options
4. Pricing tiers – > go and check 1000$/month
5. Free to use , paid subscriptions allow us to use production grade scaling options (any enterprise case , u will have to pay)

Crew AI Framework

3 important concepts to know :
Agent :

Task :

Crew

Agent definition

```
def get_travel_researcher():
  return Agent(
      role = "Travel Researcher",
      goal = "Find best attractions, accomodations, travel tips",
      backstory = "Expert in finding travel destinations , must visit spots",
      tools = [scrape_tool,search_tool],
      verbose = True
  )
```

- Whatever u are defining here will be used internally by crewAI to formulate system prompt
- Backstory : must
- Goal : task in hand
- Tools : binding

Task definition

```python
# Tasks
research_task = Task(
    description = f""" Use search and scraping tools to find
    - top attractions
    - family/ kid friendly places
    - best hotels / best hotel areas
    - travel tips
    - local food suggestions for {user_input['location']}
    """,
    expected_output = "A bullet point summary of attractions, hotels or hotel areas and travel tips along with food suggestions or food places",
    agent = researcher
)
```

1. Description of the task in a very detailed way + any placeholders coming from user query or other agent
2. Expected output :
3. Assign each tasks to an agent

Crew definition

```
#Crew
crew = Crew(
    agents = [researcher,planner,budget_analyst],
    tasks = [research_task,planning_task,budget_optimization_task],
    verbose = True
)
```

You are creating a crew by combing each of the agents and tasks
And also defining how the task is going to happen (Sequential , Parallel ..)
process = how the collaboration is going to happen

Agenda

Langraph , CrewAI in this week
Autogen and Phidata(agno)

- More about Multi Agent Systems
  - **Different Multi Agent Patterns**
  - **Theory on when to use multi agent systems**
- More about RAG
  - Reranking
  - Query transformation
  - Semantic Caching
  - Adaptive RAG , Corrective RAG
  - Theory of how a video can be embedded into vector DB
- Deployment + LLMOps
- AI/Agentic Evals + Observability (Langfuse, Langsmith, Phoenix , Arize.ai)
- **N8n class - next sunday**
- Whatever is remaining to cover on crewAI

Multi Agent Systems

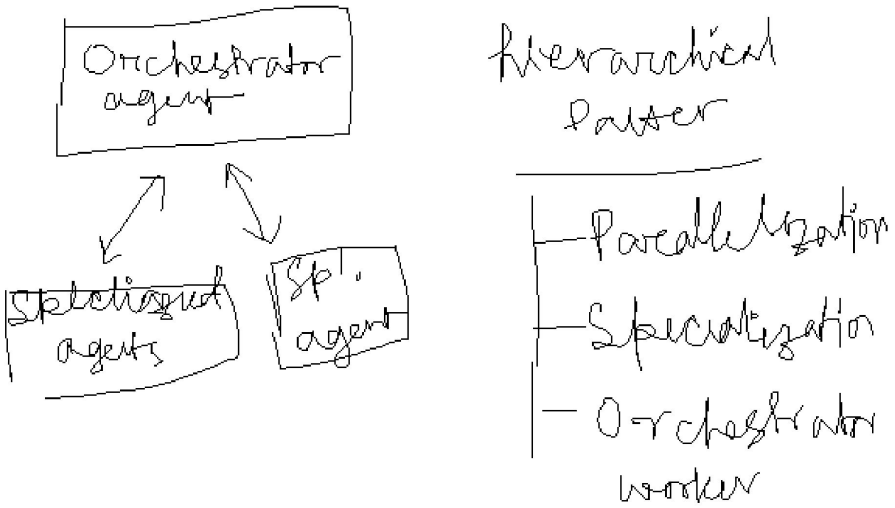
Why use MAS?

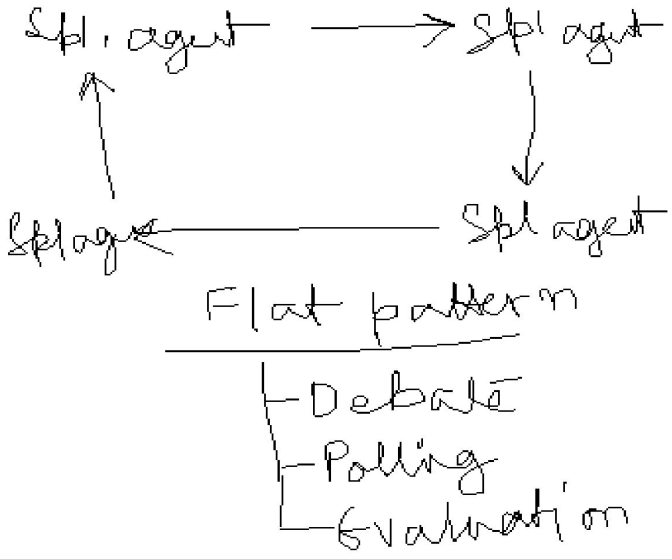
- **Too Slow :** Single agent systems may process tasks sequentially or requires multiple LLM calls, causing bottlenecks

- **Too overloaded:** Then SAS has too many calls , databases to manage and memory dependencies . Making it difficult to scale and often leading to failures

- **Lacks oversight:** You need an evaluation mechanism or second opinion to verify the agent outputs . And then that makes the system MAS (evaluation can be done just by an LLM , but there can be cases where it needs an agent)

Multi Agent System Frameworks (Hierarchical vs Flat)





More Controllable by Human
Lesser Costly and Lesser Latency

Less Controllable , More dynamic and creative
Higher Cost and higher latency

Building Multi Agent systems robustly in enterprise settings is challenging. Period.

Our general recommendation is to maximize a single agent's capabilities first. More agents can provide intuitive separation of concepts, but can introduce additional complexity and overhead, so often a single agent with tools is sufficient. Link

> Success in the LLM space isn't about building the most sophisticated system. It's about building the *right* system for your needs. Start with simple prompts, optimize them with comprehensive evaluation, and add multi-step agentic systems only when simpler solutions fall short.

Link

Prompt Based Agents → RAG → Tools with Single Agent → Multi Agent → Finetuning

What are the problems of MAS:

1. **Coordination Complexity :** Maintaining seamless communication and synchronization among the agents require well defined rules from the developers and that cane be challenging and time consuming

2. **State and Memory Management** : Deciding how much memory each agent should share , what states should be common , what should be unique to agents, how to evaluate their interactions is very difficult to quantify

3. **AI Collusion:** Errors from one agent can ripple through the system compounding the failures rather than resolving

Issues on coordination :

- Compliance agent runs Background check once HR is done. If HR raises any flag , compliance agent needs ti additional steps to be doubly sure
- IT agent needs to wait for compliance clearance and info from HR on which team at which role he will be there
- Not all agents need to know everything about the employee because in that case the agent context will run out of memory . So what info each agent should hold is also a concern
- Handoffs between the agents needs to be handled clearly

Other issues around data security . For an example IT agent should not know the salary of the employee. **Manager agent** needs to probably get all the feedback from all agents . But what happens if one agent fails / raises flag

In a single agent system ,

- Maintains a single state and sees the full picture of execution - progress, dependencies
- It can use internal function calls to handle HR, IT , Compliance etc tasks
- Manage error handling inside itself without being bothered about different possible scenarios in a MAS

Observability in MAS for improving the MAS :

Token Usage : Track the number of tokens used per query per agent . Exceeding the token limit could suggest inefficient prompting and verbose planning

Latency : Measure the response times of LLM calls. Measure the communication overhead between agents. High latency can indicate inefficient queries , data retrieval or communication patterns.

Failure Rate : How frequently an agent fails to produce valid response , due to different issues : - API timeout , tool error , hallucination etc

Retrieval Delay : Determine teh time spent by agent in retrieving data from database / VectorDB

Tool Execution Time and failure : Track external tool or API call - time and failure rate and reasons so as to address them

Agentic Framework Selection

Consider all the agentic frameworks on two axis : **Controllability** and **Autonomy**

- Frameworks like **Langchain, Llamaindex and Langflow** offer limited autonomy , primarily supporting **workflow agents** (i.e where only actions are automated , planning is not mostly)

- Frameworks like **Langgraph , CrewAI and Autogen** provide higher autonomy . (Planning is more done by agents , than by human - dynamic agents)
  (for dynamic agents human controllability is lesser)
  (Better frameworks should provide as much flexibility to control some of these)

  As autonomy increases , controllability overhead also increases accordingly for these tools . Controllability is around State and memory management , Conflighuration flexibility , Communication frameworks etc

Llama Index Lanchain Phidata DSPy	**High Controllability**	**LangGraph** **CrewAI**
Lower Planning Autonomy , More of Action Autonomy		**Higher Planning Autonomy**
Zapier Hubspot Lyzr (Pureplay workflow agents)	**Low Controllability**	OpenAI Swarm (Creative outputs Langflow Autogen

Different kind of patterns

1. **Prompt chaining** sequential
2. **Parallelization** : if parallel tasks can be done then it saves latency
3. **Router :** VVI in real enterprise setting . This helps us to route the query to the right agent when you are typically addressing multiple classes of queries.
4. **Orchestrator Worker :** orchestrator does the planning and dynamically forks the # of worker agents . every worker agent has his own state. They report back with output to the state maintained globally.

   (it takes away the determinism in output and gives agent complete freedom for generation of output - TRUE AUTONOMOUS AGENT)

5. **Evaluator Optimizer :** iteration improves the quality of output . Evaluator needs to be providing structured output ideally for consistent consumption by the generator

Homework

Framework : Langgraph or CrewAI ; Complexity level - intermediate

RAG Application:

Build a **food delivery customer support bot** that answers users questions about food delivery companies policies around account settings, refund, cancellation policies , pricing offers , commission , qualities etc.

- Create 2-3 pdf documents with food delivery policies ; ingest these documents into a vector Db chroma db , implement a retriever + LLM pipeline , wrap the pipeline into a single agent that can answer questions like - “What is the refund policy in company XYZ ; Why is the pricing offer not getting applied?”

(test cases u can generate via LLM)

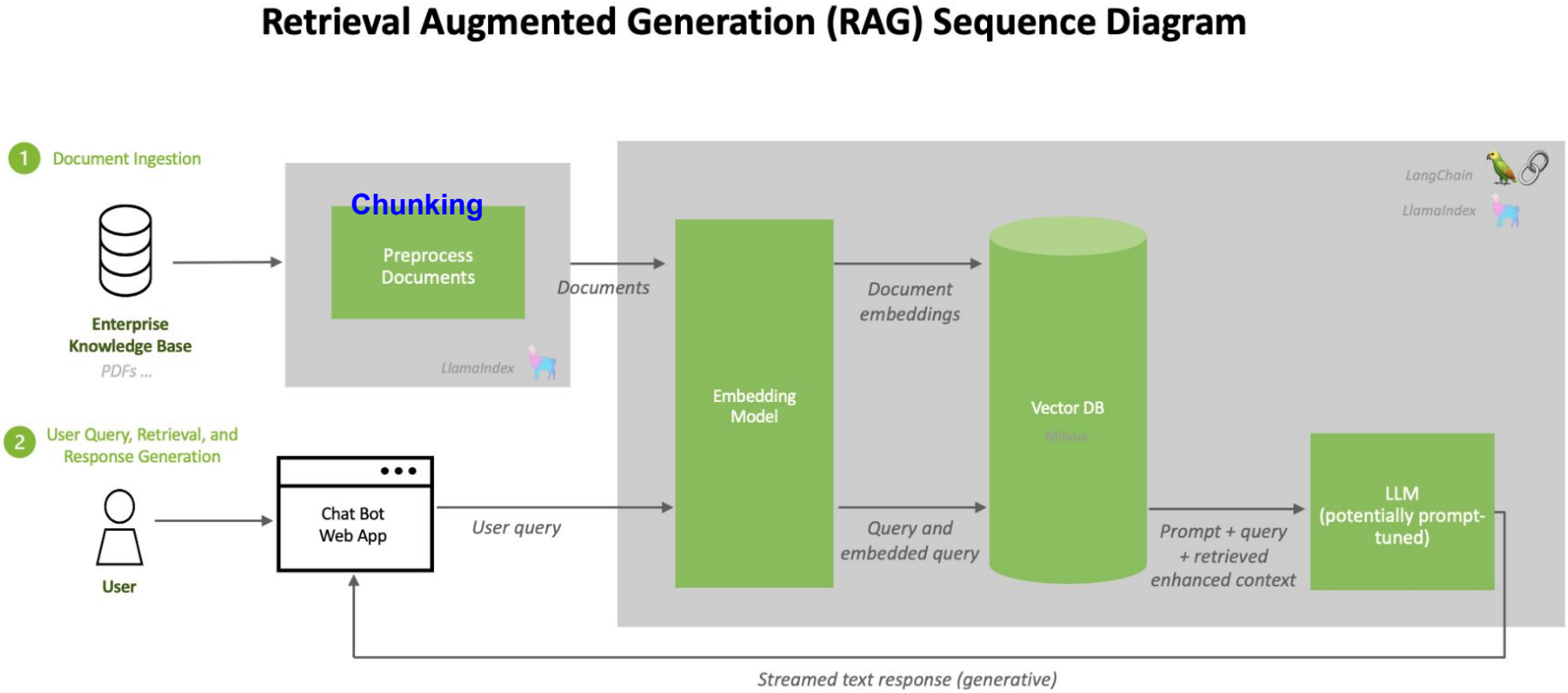
Single agent application

Multi Agent Application : Enhance the food delivery customer support bot with multi agent functionality . There can be a) Order related issue : Where is my order ? Can I cancel my order b) Refund & payment related issues : My order was poor quality , I want refund …, my money is debited , not showing successful payment c) Restaurant related issues : Clarification on restaurant and menu , restaurant sent me wrong item…

- RAG
- Multi Agent Roles
- And interact with tools (for getting order status you may have to connect database)

Enter RAG

RAG is a technique that helps process large volume of data by efficiently retrieving and optimizing the most relevant information before passing it to LLM



link

Practical RAG Hacks in Enterprises

Hacks :

1. **Re ranking :** Vector Similarity between query and chunks look for keyword overlap and hence retrieved documents may have **incorrect rank orders** i.e generic documents may come up at the top. This is **fixed by Re-ranking** where we use cross-encoder (sentence similarity) to find the relevance of the retrieved documents with the query and rerank the results . Remember LLM always gives more attention at the top part of the context …so reranking improves the quality of LLM output
2. **Hybrid Search:** Combining vector search (keyword similarity based search) with exact keyword search to improve the retrieval quality . We combine by weighted score on both of these and then rank order based on hybrid score to generate rankings . And as a result the RAg retrieval quality significantly improves
3. **Query Segmentation :** For complex queries , vector search results will not be optimal if we use a single query to search. Recommended is we break the query into multiple subqueries and those subqueries to embedding & search pipeline to generate more optimal search results that cover different aspects of the original query

Practical RAG Hacks in Enterprises

4. Query Routing: When you want to have a single frontend interface and in the backend you have multiple knowledge bases , You want to classify the queries into different categories to cater to different knowledge based in order to more efficient and better retrieval . Mechanism used - LLM based, Embedding based vector similarity of knowledge base description and query , rule based, hybrid based.

5. **Query Transformation :** Change vague / ambiguous / informal/ bad english user queries to more structured / contextual / formal / grammatically correct user queries so that the chance of similarity and keyword match increase i.e retrieval becomes better

6. Semantic Caching to control cost : In order to call the LLM for same kind of queries again and again, it makes send to store the query and results in an embedding format in a cache (vector store) and then when user asks a query we first look at cache, if we do not find the match then will we go ahead with rest of the RAG

7. Corrective RAG

8. Adaptive RAG

9. Creating your own embedding with little data : Most of the embeddings are general-purpose . They will miss domain specific terms . Training embedding from scratch is out of question . Pretrained embeddings : OpenAI / Hugging model + Fine tune embeddings with your data . Pairs of Query → Relevant result you embed (positive data) and query ->irrelevant result(negative data) . Using these pairs you can fine tune a pretrained embedding models and create custom embedding . Custom embedding can improve your search quality significantly

Text to SQL - can you share the document

Recommender system agentic - share the document

When to decide we need reranking ?

a) **Why the RAG Cost is high ? How to reduce the cost ?**
- Vector Similarity results are yielding many document chunks increasing the input context size to the LLM . As a result of that the cost is increasing .

b) **Retrieval metric of NDCG is low , but recall is very high** - that means your most relevant documents are coming not at the top . This will reflect to poor accuracy in RAG final output

c) **Just in case you are using bigger chunk size** , it is often seen that embedding similarity will give you poorer result . If your chunk size is high / you are using semantic chunking - then it is recommended you put a reranker after it

Deployment of any agentic application

1. **Containerized Applications :**
   a. What : Package the entire app and dependencies in container
   b. Example : Docker + Kubernetes , AWS ECS , Google Cloud Run
   c. Pros : Portability , Scalability
2. **PaaS**:
   a. What : Deploy code directly , platform handles the infrastructure
   b. When: Fast deployment , prototypes , minimal ops overhead
   c. Examples : Heroku , Railway , Render, Vercel
3. **VM / Server based deployment :**
   a. What : Traditionally self hosted servers will have code deployed artifacts
   b. Pros: Full control is with you , legacy setup
   c. Examples : AWS EC2, Azure VMs, Google CE
   d. Cons: Manual Scaling framework
4. **Serverless Deployment :**
   a. Conde runs on-demand without any management of servers
   b. When : **auto scaling functionality** , pay per use, zero maintenance
   c. Examples: AWS Lambda, Azure Functions ,

Infrastructure deployment modes:

- Cloud Based:
  - Public Clouds
  - Hybrid cloud
- Self hosted / On prem :
  - Dedicated servers your own hardware
  - Private cloud on your infra
  - Deploying on CDN / edge functions

Cost Models :

- Payperuse (serverless functions , api calls - variable traffic)
- AlwaysOn (fixed rates for running instances - consistent traffic)
- Resource Based (pay for cpu , memory , storage)

Deployment - When to use which one

B	C
Usage	**Recommended Deployment**
Quick Demo	Railway , Heroku, Vercel , Render - PaaS
Production MVP	Google Cloud Run , AWS App runner
Product High Scale	Kubernetes + Autoscaling
Cost Sensitive	Serverless (lambda , functions)
Full Control	Self hosted
Compliance + Scale	Private Cloud

Railway + Dev / Prod

1. Method 1: Mapping to the github branch (dev / main)
2. Method 2: Environment settings deploy different triggers (environment creation is itself handled within heroku, railway , render)

CrewAI

1. Will define agents:
    a. Every agent needs to have **role, goal and backstory** defined
    b. Optional - mention LLM (default openai) , allow_delegates, tools
2. Will define task:
    a. Description of the task - detailed
    b. Expected_output = <> (descr + expected output used by CrewAI to create prompts internally)
    c. Agent = which agent will do the task
    d. Context = [write the dependent task] OR async ?
    e. In task definition , will assign tasks to agents (OR…)
    f. In task definition , we can mention dependent tasks as well
3. Then we create a crew having agents and tasks + communication method (default - sequential
4. Run the crew

Eval in Agentic System

What is Eval in Agentic System ?

- How well your agent / system performs against a set of defined criterias is called eval

What are the different ways to set up eval ?

- **Streaming eval:** evaluate your agent at the run time
- **Offline eval :** run scripted evaluations on stored run or datasets

What are the common eval metrics:

- Correctness (did the agent achieve the assigned task)
- Efficiency (did it achieve as per the expected set of steps)
- Robustness (does it handle edge cases)
- User Satisfaction (quality of final output)

Setup

- pip install langsmith (or langfuse or arize.ai)
- Configure API Keys :
  - LANGSMITH_API_KEY
  - LANGSMITH_TRACING

Creating dataset for evals - Offline evaluation

```
from langsmith import Client

client = Client()

dataset = client.create_dataset(
        Dataset_name = “dataset for testing app XYZ”,
        Description = “eval set for testing app XYZ
)

# add examples - user provided
Client.create_example (
Input = {}
Output ={}

)
```

Write Eval scripts - offline

In langsmith evals are python scripts where you define how to call your agent , how to compare the agent output with expected output and how to score the result

from langsmith.evaluation import evaluate

client = Client() # langsmith client

```
def correctness_metric(run, example):
    # Metric to evaluate did agent answer correctly
    Expected = example[“expected output”]
    Actually_got = run.outputs[“output”]
    return  {“correctness”: int(expected in actually_got)}
```

Run the agent over all the dataset that you have

Write Eval scripts - Streaming data

You will need callbacks for streaming evals

def streaming_data_eval(run, example):

Agent output at runtime

You will find out comparable example to evaluate from ground truth data

Evaluate the score

Using LLM as a judge in the eval script

- Define LLM
- Pass the LLM criteria

From langsmith.evaluation.evaluator import LangChainStringEvaluator

Llm = ChatOpenAI()

Judge = LangChainStringEvaluator(

Criteria = { correctness : is the answe correct as per the given expected responses,
Robustness : is the answer directly addresses the user question },

Llm ,

Name = llm_judge

)

Evaluate (evaluator = Judge, data , agent output)

Streaming eval

- There is no fixed dataset to run prediction and then compare final output
-
- Agent runs in real time (with .stream()) trace the intermediate tokens / events through **observability** platforms and evaluate the behavior during streaming (did it stay on topic, did it halluciante , did it take too much time)

- Langsmith supports this by letting us define a **predict** function that streams the outputs but at the same time it gives a callback of the result to evaluate

N8n theory

- Open Source **Workflow automation** tool. Similar to zapier but more advanced
- It can be self hosted (VPC)
- It can accommodate relational database queries very easily, HTTP requests very easily , it can have custom JS as well
- Supports almost 500+ integration out of the box (slack, telegram, whatsapp, google sheet, openai)
- It is node based design - workflows are built using interconnected nodes

N8n components

- **Trigger Node:** starts the workflow (manual , scheduled, instantaneous)
- **Filtering** : allows or block certain type of data to flow via the workflow
  (Somebody submits a CV - there is a field that asks you have skillset X or not – based on the skillset the cv will go for further processing or it will stop)
  <expressions using variable $json.X != sth>
- **Apps** : (notion , slack , telegram , gdrive…)
- **Actions**: updating a row in googlesheet , responding to an email , querying a db etc
- **Node** (each building block)
- **Routing**
- **Data flow** : between the nodes in n8n the data flows as json object

Usecases

No AI :
- Auto post content from my repository to linkedin
- Pulling some data from db and update a dashboard
- Creating a jira ticket when a new incident email comes

AI :
- chat -> Call OpenAI to generate content -> edited -> post on linkedin

Cannot be done :
- Anything where you want LLM to do planning cannot be done