truefoundry

Enterprise Outcome Onboarding

-

Agentic AI with TFY Platform

Anant Agarwal

(Principal Applied Scientist)

This crash course on Agentic AI starts from scratch with pre-agentic GenAI, then dives deep into Agentic AI and MCP development & deployment with TrueFoundry AI Gateway.

What You'll Learn

Day 1: Quick Primers on Pre-Agentic GenAI

Day 2: Building Agents

Day 3: Deep-Dive into AI Gateway

Day 4: Model Context Protocol

Day 5: CI/CD and Evaluations

Certification Test

Assessments	**10%**	**15%**	**15%**	**15%**	**15%**	**30%**

Technologies Used

(feel free to experiment)

\+

TrueFoundry Platform

LangChain
LLM Orchestrator Framework

CURSOR
Development IDE

tavily
Agentic Search

Chroma
Vector Database

LangGraph
LangChain's Agentic AI Framework

gradio
Prototyping UI for LLM Apps

uv
Package Manager

groq
API for Accessing Models

Getting Started

Getting Started

Step 1: Clone the Repository

- Navigate to <insert URL here>, and clone the repository <repository name>
- cd <repository name> on your local system

Step 2: Setup Virtual Environment

- Install uv
- Create virtual environment using <uv venv>
- Perform <insert different commands for activating venv for different operating systems>
- Execute command <uv sync>

Step 3: Install Cursor

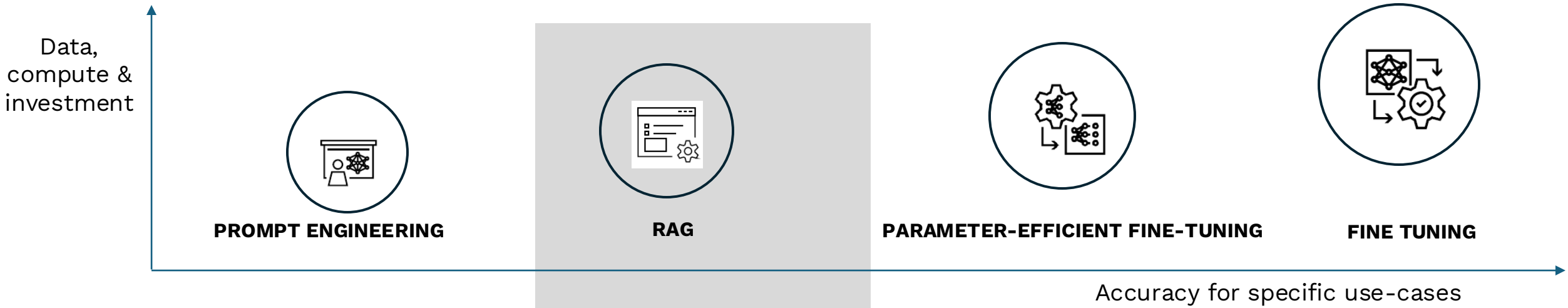
- Install Cursor IDE and open the project
- Select the correct environment

And you’re set!

Hands-On with LangChain and LangGraph

Retrieval Augmented Generation

Customizing* LLMs for Tasks - Overview



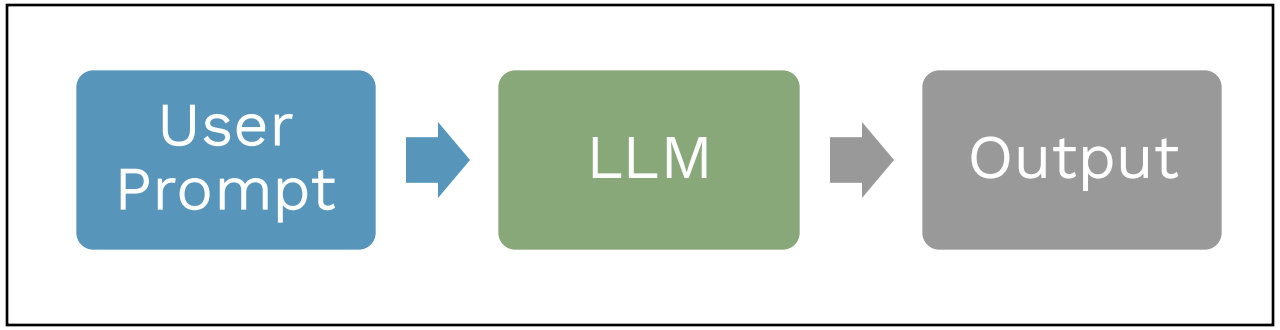
TECHNIQUES	• **Few-shot learning** • **Chain-of-thought reasoning** • **Least-to-most prompting**	• **Retrieval augmented generation (RAG)**	• **Adapters** • **LoRA** • **IA3**	• **SFT** • **RLHF**
HOW	Prompt templates	Using vector databases	Add “custom” layers to LLM	Tune LLM model weights

Adapted from Nvidia Research with a few changes

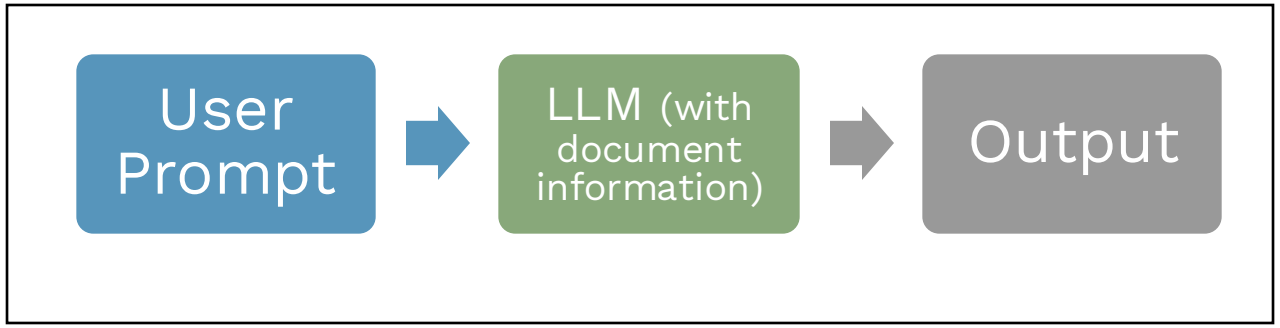
**Note that this is called “finetuning” loosely, and even Prompt Engineering and RAG can be considered as finetuning for your application - a common understanding of the word “finetuning”*

What We Know

This is how you could typically chat with an LLM:



But how do you arm LLMs with the domain-specific / org-specific information you want it to refer to, rather than its own general training data knowledge? **This is the common use case that we'll focus on today.**

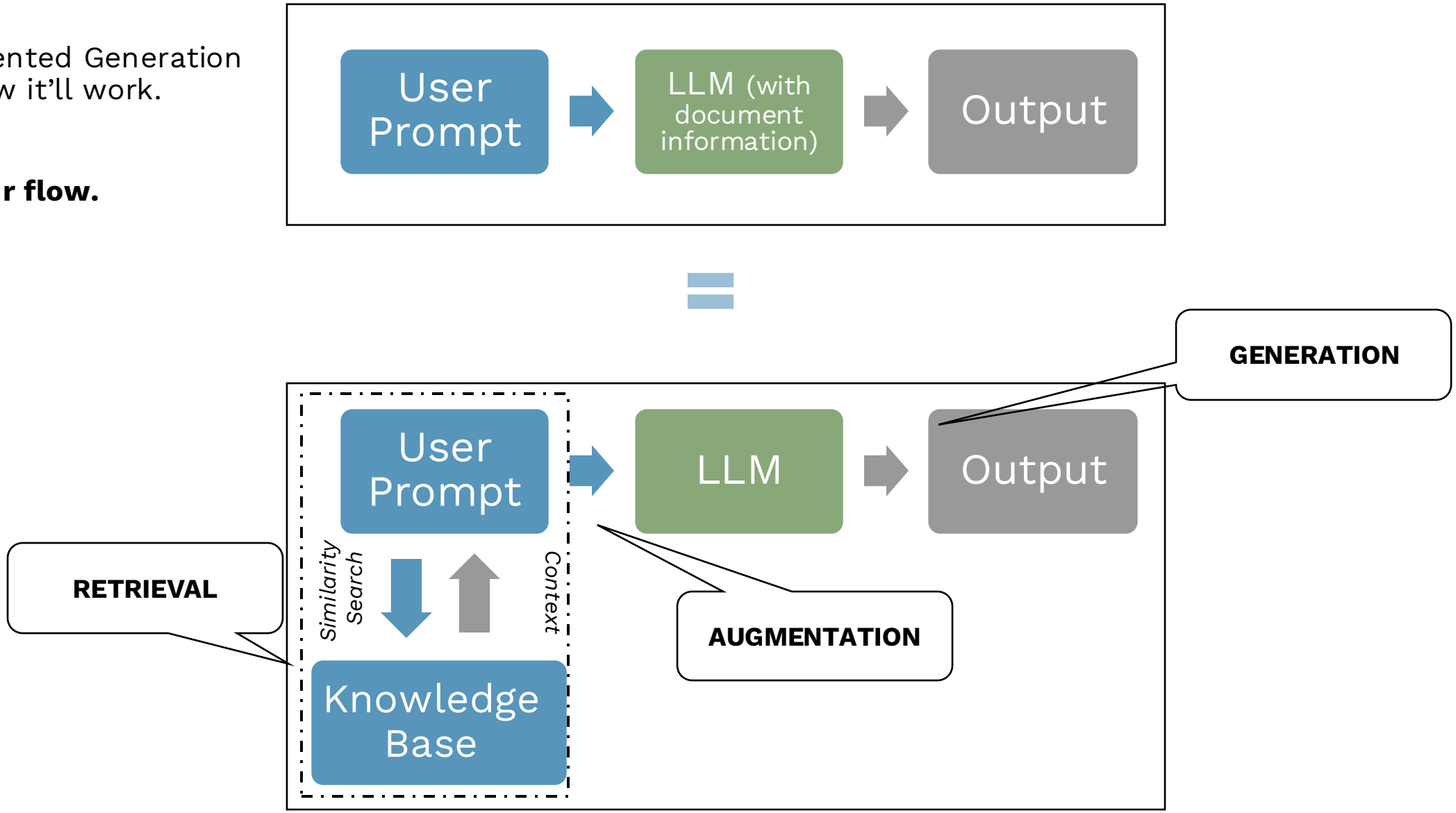


How do you think this would work?

Introduction to RAG

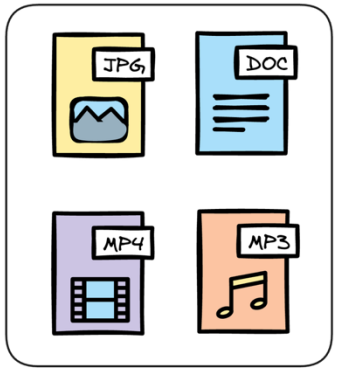
Retrieval Augmented Generation (aka RAG) is how it'll work.

Let's expand our flow.

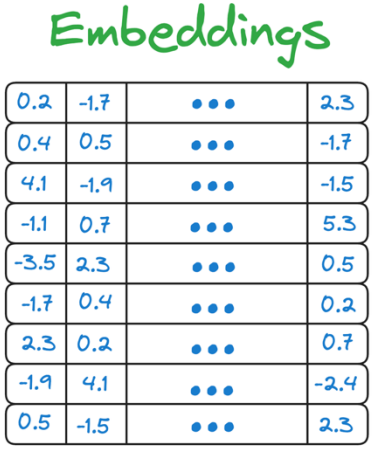


Backbone of RAG: Vector Databases

Creating a Vector DB

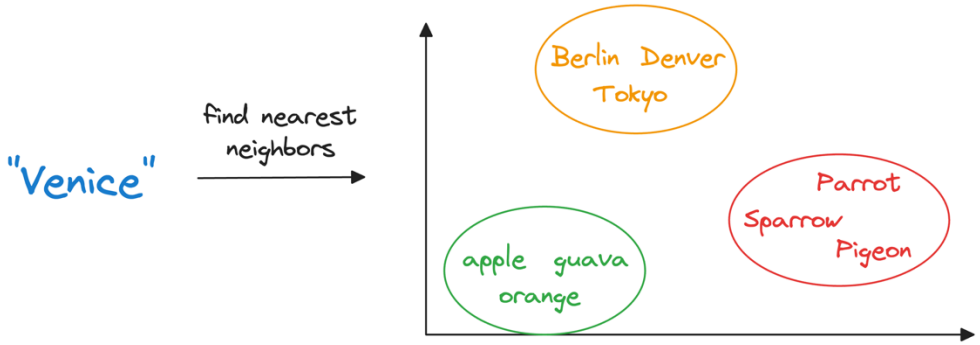


Chunk and embed using an embedding model



*A **vector database** stores unstructured data, in the form of vector embeddings*

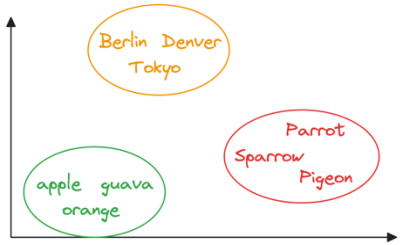
At inference



There are different methods by which similarity search is done

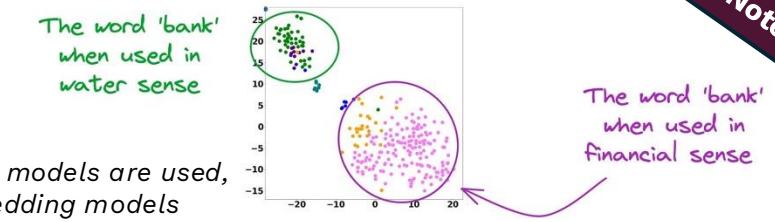
Why embedding works?

Embeddings of similar meaning words are near to each other →



Important Note

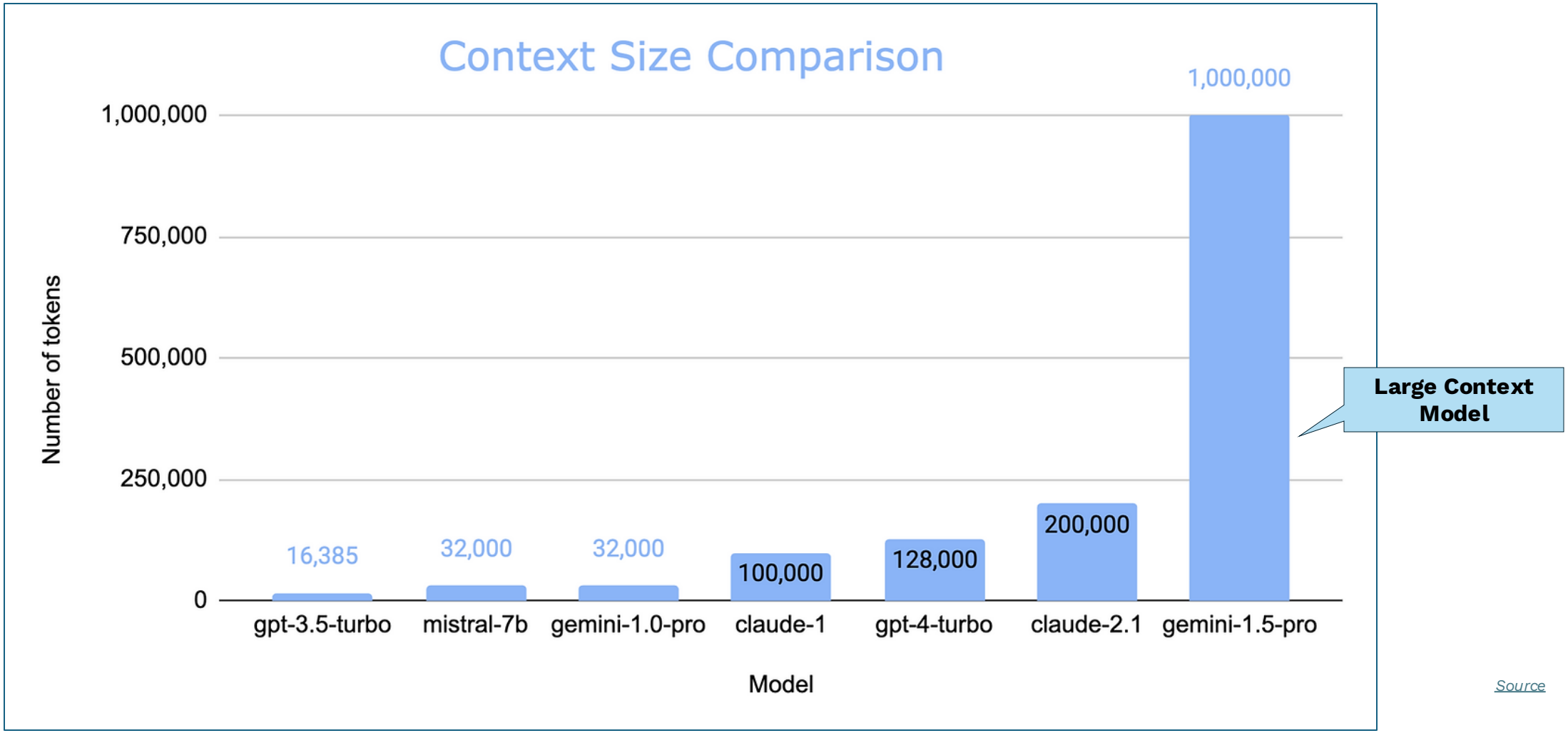
Contextualized embedding models are used, instead of static embedding models



Images source

LCMs vs RAG

Increased Context Length – RAG still relevant?



Source

Increased Context Length – RAG still relevant?

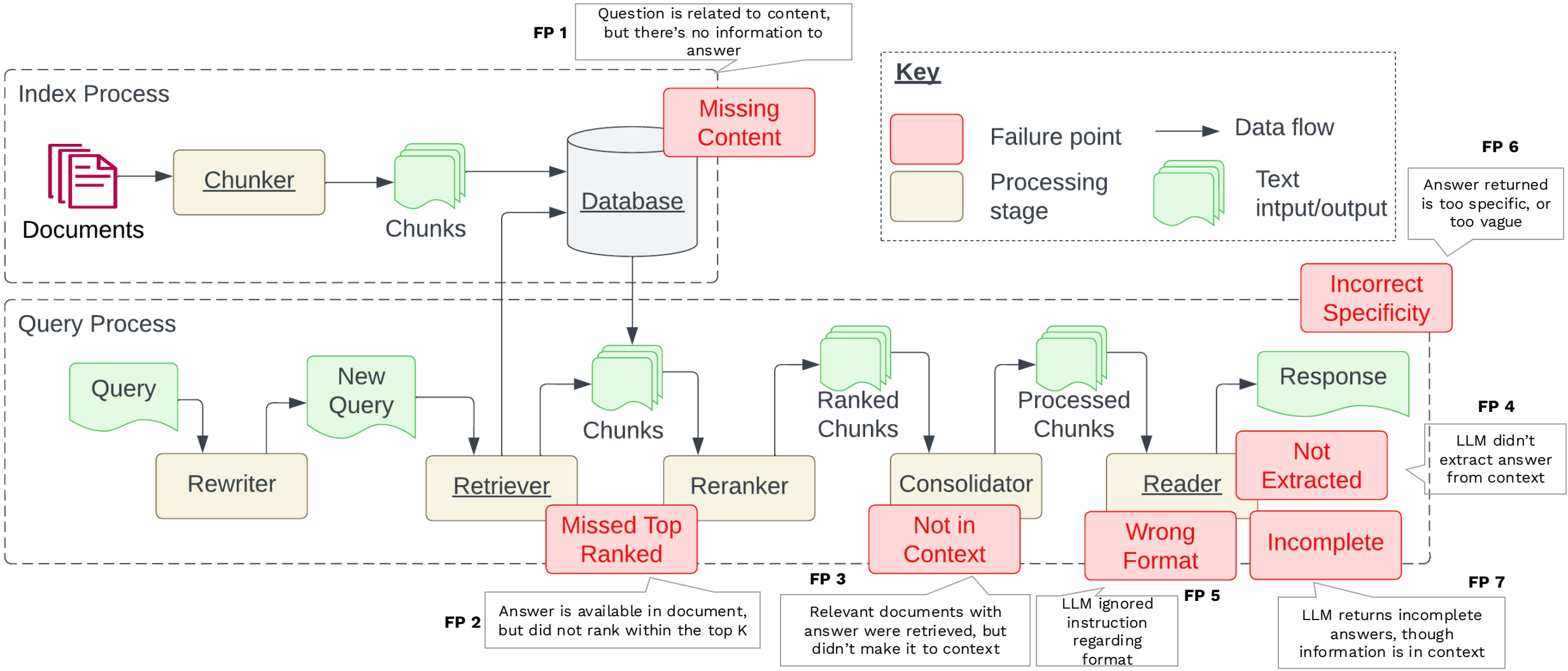
- **Cost:** Large Context Models (LRAG winsCMs) like Gemini 1.5 are expensive per token, making RAG models more cost-effective for most use cases.!
- **Latency:** LCMs take longer to process longer contexts. RAG models can achieve lower latency with smaller contexts. RAG wins!
- **Context Learning:** LCMs excel at complex reasoning and learning from long contexts, demonstrated by reasoning about Apollo 11 transcripts and learning new skills. LCMs win!
- **Long Context Recall:** LCMs may struggle with recalling information in the middle of long contexts. RAG models can curate relevant content before feeding it to the LLM. RAG wins!
- **Flexibility & Customization:** RAG models offer more control over data sources and retrieval strategies, making them more customizable. LCMs are a faster solution but less flexible. RAG wins!
- **Security & Data Privacy:** RAG models provide more control over data privacy, especially for self-hosted deployments. LCMs rely on an external provider's security measures. RAG wins!

Conclusion: RAG is here to stay!

Source

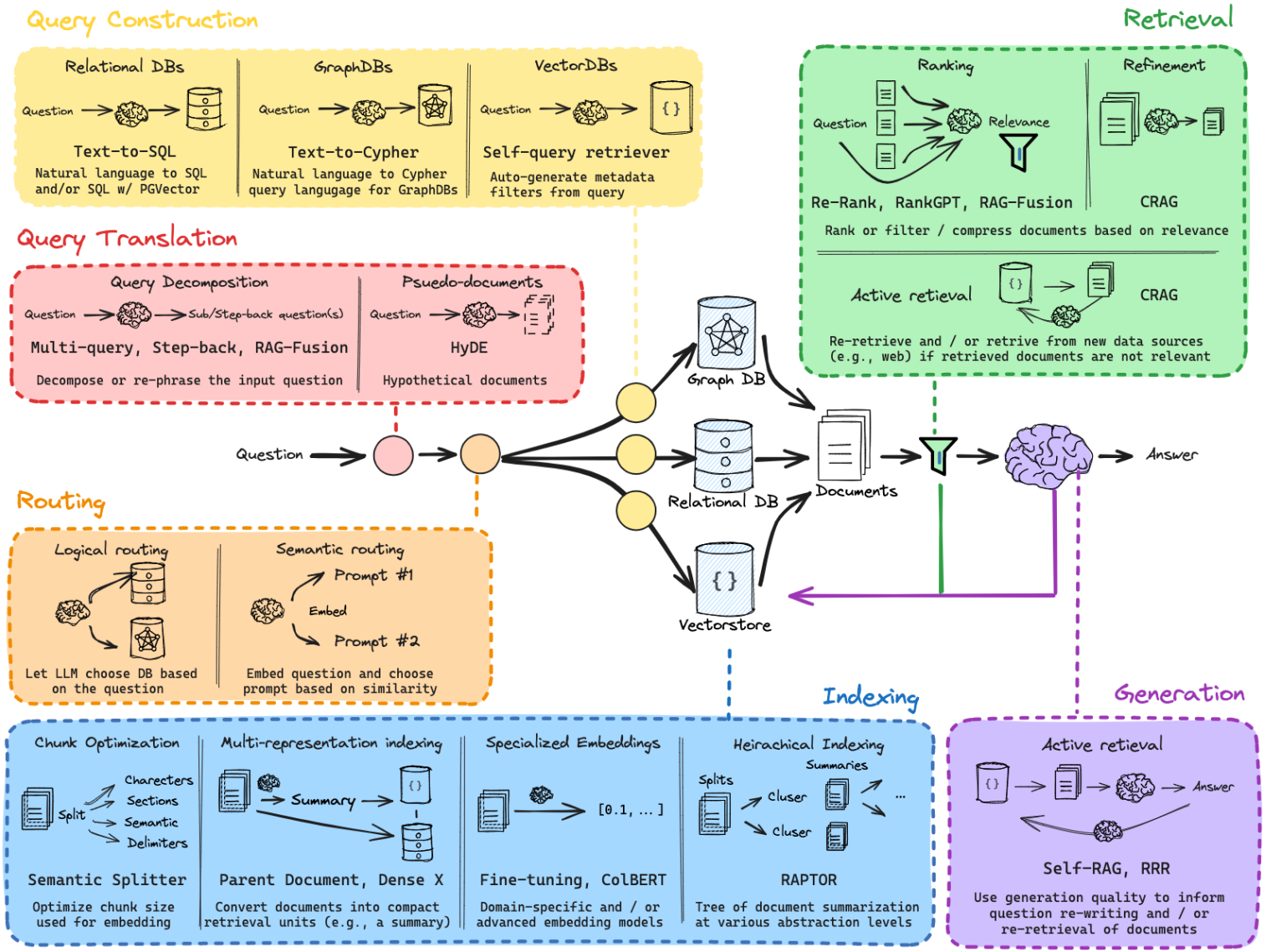
Improving RAG Systems

Key Failure Points of a RAG System



Source: Seven Failure Points When Engineering a RAG System (arXiv)

Methods to Improve RAG



Assuming prompts and choice of LLM are optimized for, we're optimizing for RAG!

Sourced from LangChain AI's GitHub Repo here

Multimodal RAG

Multimodal RAG

Building Retrieval-Augmented Generation (RAG) pipelines for multimodal data presents unique challenges. Here are the main approaches for tackling them, focusing on image and text inputs:

1

Unified Vector Space: Embed all modalities (like images and text) into a single, shared vector space. **(multimodal embedding model needed)**

2

Primary Modality Grounding: Convert or "ground" all modalities into one primary modality (e.g., transforming images into textual descriptions).

(text embedding model needed)

3

Separate Modality Stores: Maintain distinct storage and retrieval systems for each different modality.

(each modality's embedding model needed)

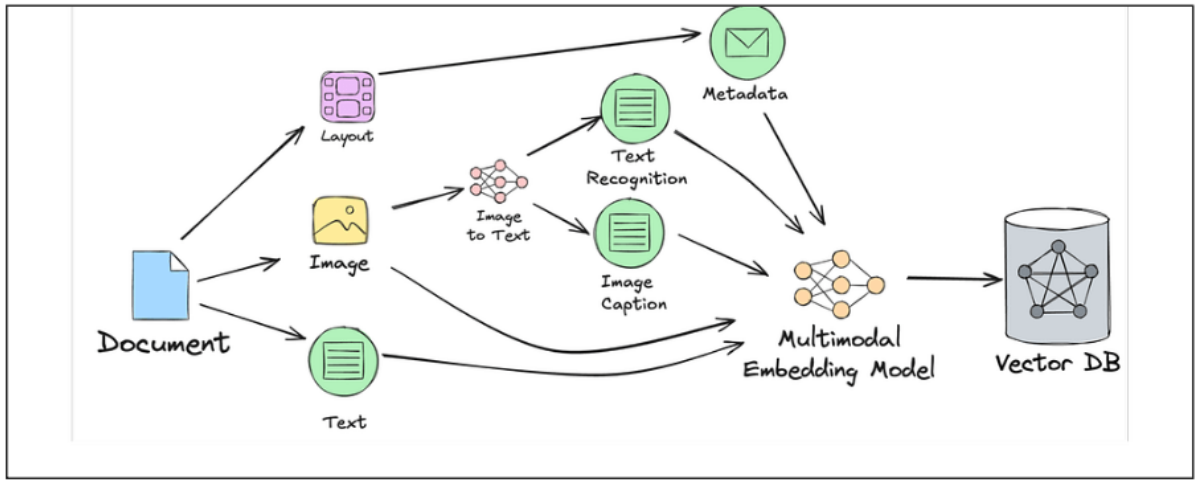


Image source here

Pros/Cons

1. Embedding into single vector space is good, enables cross-modal retrieval, but suffers in latency, cost and scalability
2. Embedding everything into a single modality like text is fast, with wider LLM compatibility, however, rich information of other modalities (like image) is lost and quality of modality translation (from image to text) depends heavily on model used
3. Embedding into multiple vector stores, like a hybrid approach is best of both worlds, but very complex

Hands-On with RAG

Assessment (10%)

Introduction to Agents

Why LLMs Alone Fall Short

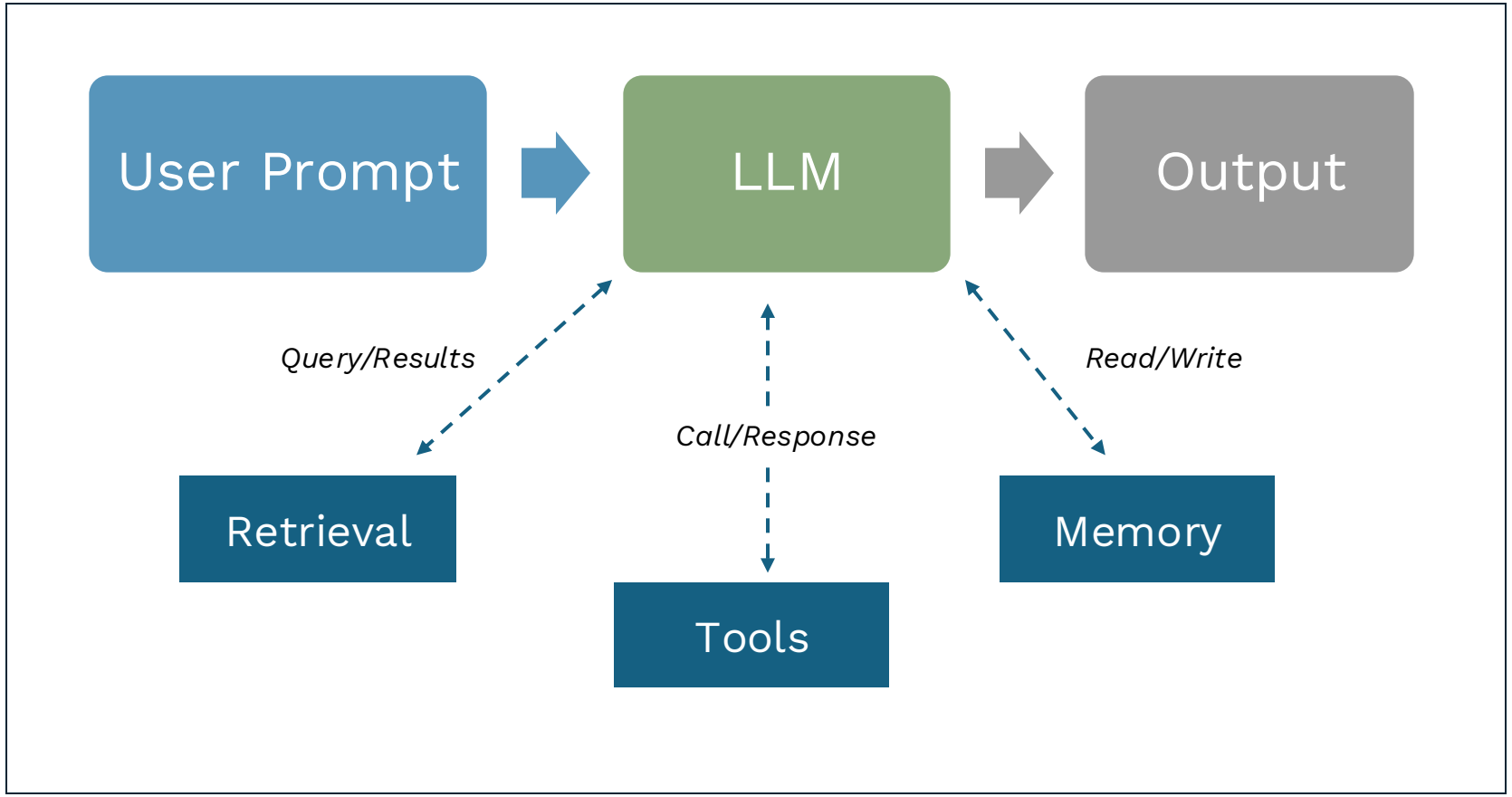
- LLMs can think, but not act
- LLMs are limited by what’s available in their training data
- There is a single inference basis the user query, and no continuous context management (i.e., chat history)

What We Need Instead

- LLMs can think, but not act
  - Need to be able to to perform actions like making an API call
- LLMs are limited by what's available in their training data
  - Need to extend knowledge through connection with external systems such as search tools
- There is a single inference basis the user query, and no continuous context management (i.e., chat history)
  - Need to have a managed session history, which allows for multi-turn conversations

The Augmented LLM

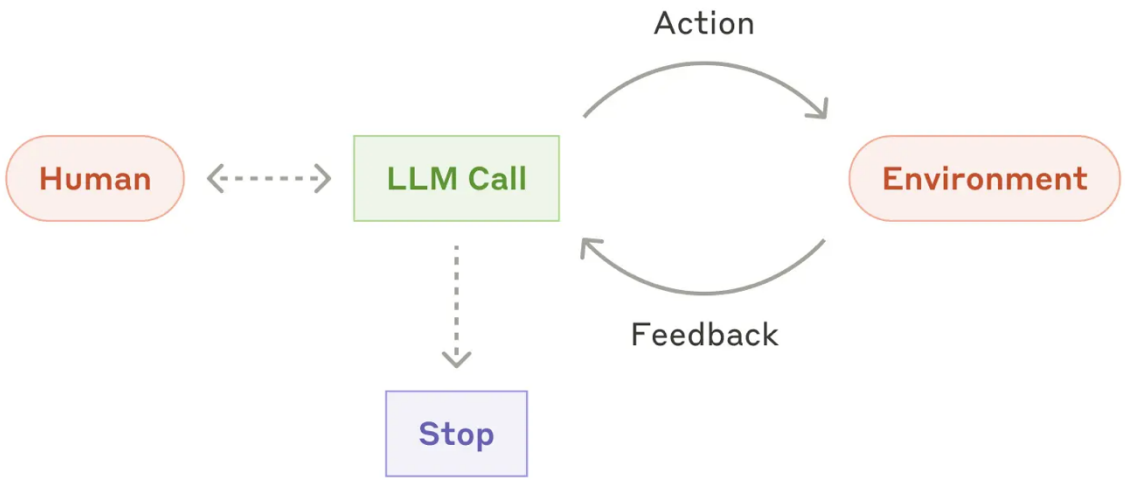
The basic idea is to augment the LLM with retrieval, tools and memory, where they can generate their own search queries, select appropriate tools and determine what information to retain.



The Augmented LLM

Agents

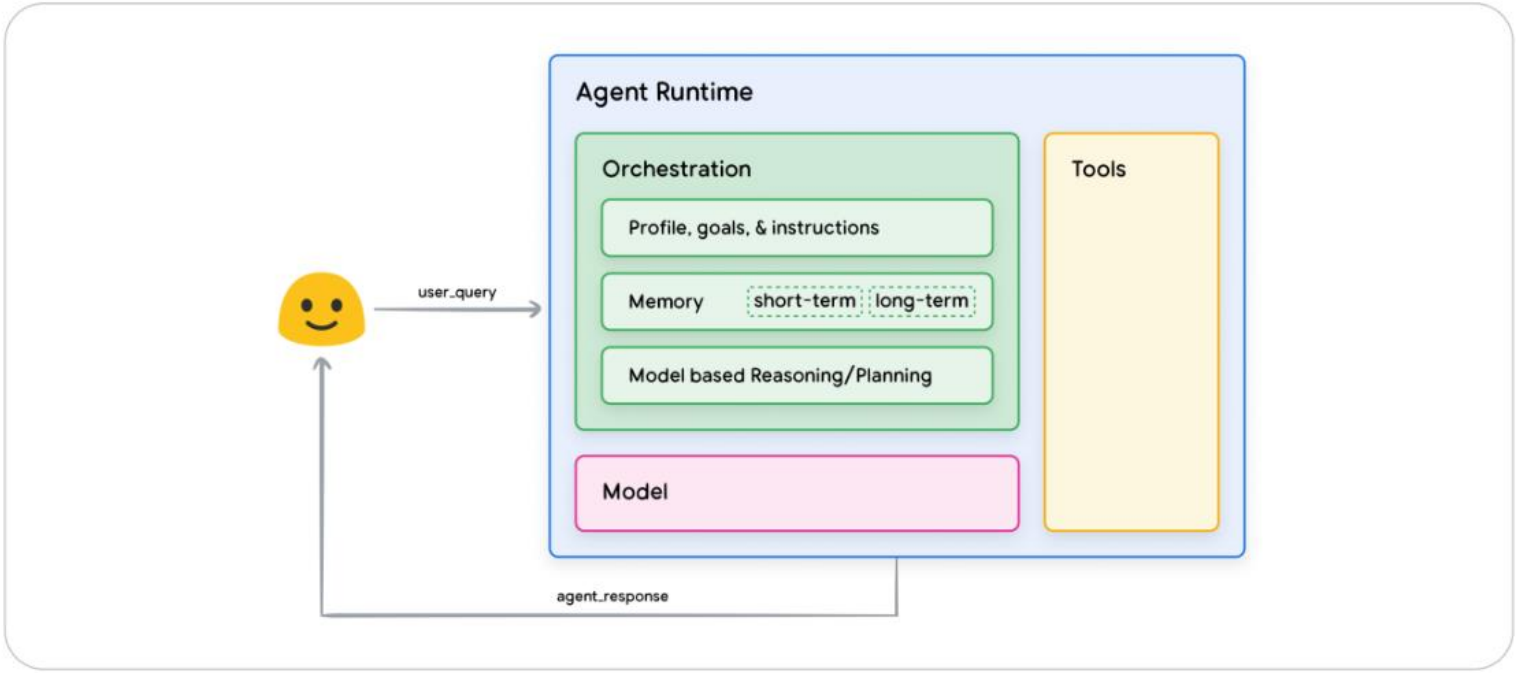
- **Goal:** With more advanced LLM capabilities (understanding complex inputs, engaging in reasoning and planning, using tools reliably, and recovering from errors), plan and operate independently while returning to human for feedback
- **When to Use:** For open-ended problems, where it's difficult to predict required number of steps or find a fixed path
- **Many Use Cases** (complex)
  - Customer support (talk, pull data, act, resolve)
  - Coding agents (design, build, test, measure)



Reference - *Anthropic*

What constitutes an AI Agent

Agents are AI systems that leverage LLMs to reason through a solution, leveraging external tools, planning (like reflection and task decomposition), and having memory (short-term or long-term).



Source: Google's Agents Whitepaper

Deeper into Tools

Tools and Function Calling

- Function calling in Large Language Models (LLMs) enables these models to **interact with external tools and APIs**, thereby extending their capabilities beyond text generation to include real-time data retrieval

Tools and Function Calling

- The function-calling feature is implemented using a combination of **fine-tuning, prompt engineering, and structured output generation** techniques.
  - **Finetuning:** Teach the LLM to call functions, by showing it examples of user query – correct function
  - **Prompt Engineering:** Include the function's name, a description of what it does, and a list of its required parameters and their data types
  - **Structured Output Generation:** LLM should generate a structured output, typically a JSON, containing the function's name and specific arguments needed
  - *Note:* Prompt engineering tools is as important as overall prompt engineering, as agents often fail at their tasks when they don't call tools the right way!

Tools and Function Calling

- Primary use cases that leverage function calling:
  - Agents
    - For example, if you say, "What's the weather like in Boston?", the LLM can decide that it needs a weather API to get the answer. It then generates a structured function call (e.g., get_weather(location="Boston")), which your application executes. The results from that API are then sent back to the LLM, which uses them to formulate a natural language response like, "The weather in Boston is currently sunny and 75°F."
  - Complex multi-turn chatbots
    - By using function calls, the chatbot can "write" to a memory or database and "read" from it in subsequent turns.
  - Batch inference feature extraction
    - For example, you could have millions of customer reviews and use an LLM with a function-calling capability to extract key information like product_name, sentiment (positive/negative), and key_complaint.
  - Structured output generation
    - In general!

Tools and Function Calling

- Not all LLMs are made the same, and LLMs differ in their capabilities to call the right tools

Berkeley Function-Calling Leaderboard

Tools and Function Calling

How the Execution Happens

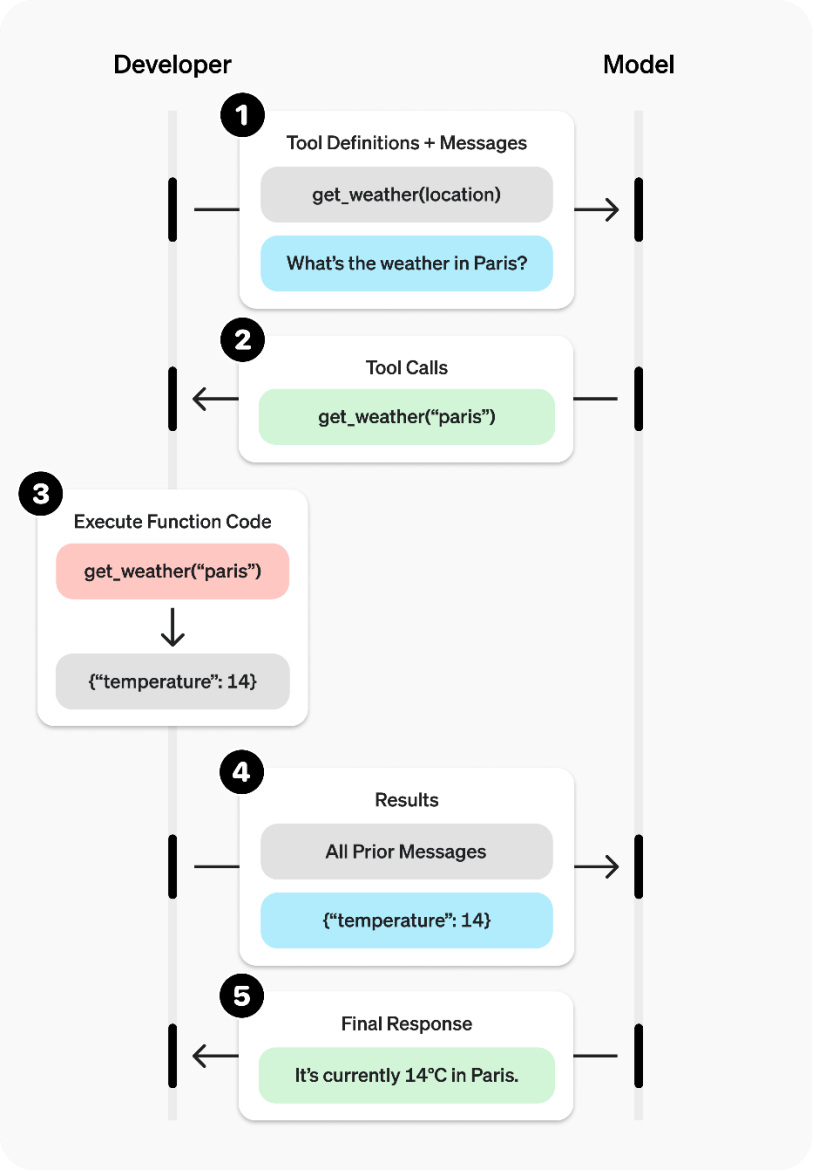


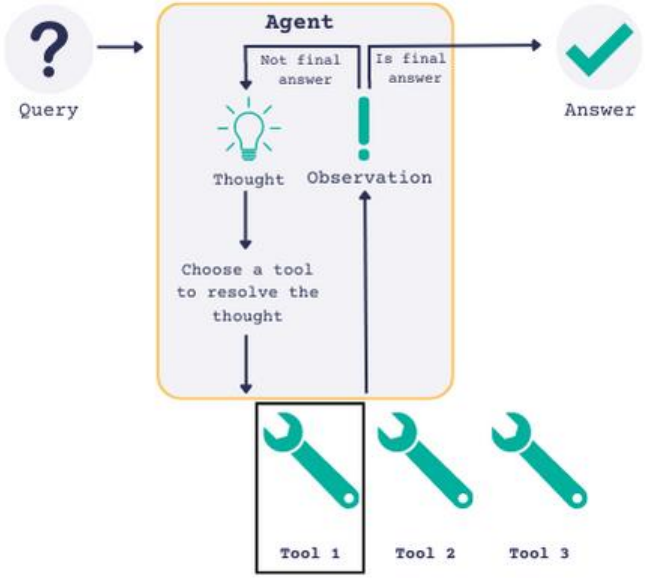
Image Source: OpenAI

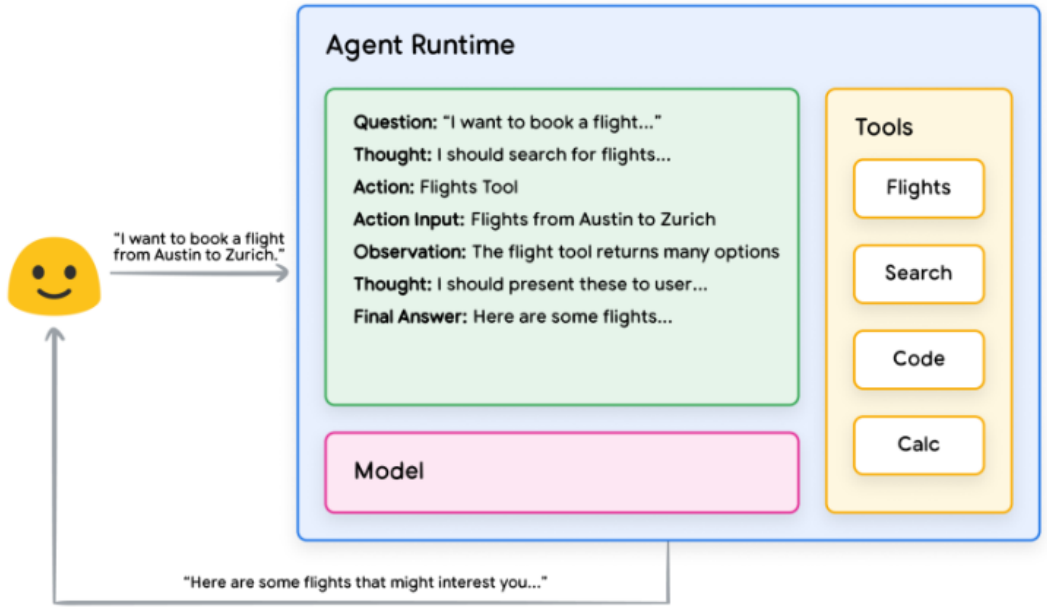
Agentic Design Patterns

The Simplest Form of Agent: ReACT Agent

The ReAct (Reasoning and Action) agent model

- Integrates the reasoning capabilities of large language models (LLMs) with the ability to take actionable steps
- Creates a more sophisticated system that can understand and process information
- Can evaluate situations, take appropriate actions, communicate responses, and track ongoing situations.





ReAct: Synergizing Reasoning and Acting in Language Models

Some Single-Agent Design Patterns

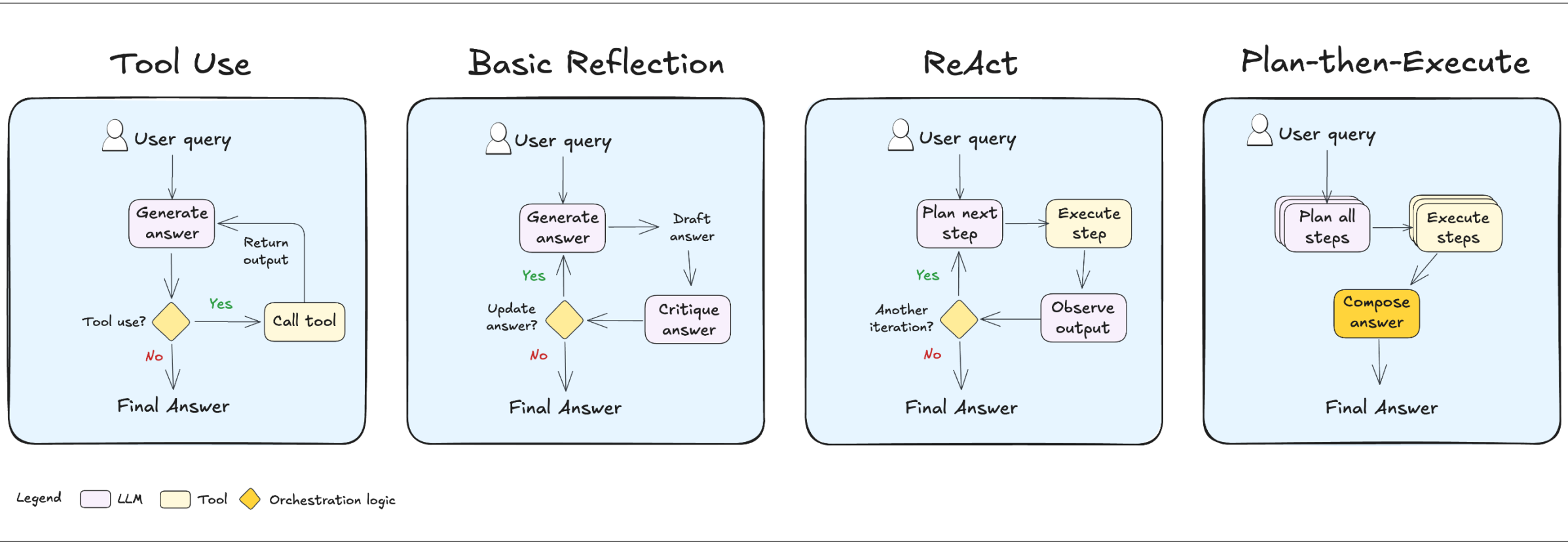
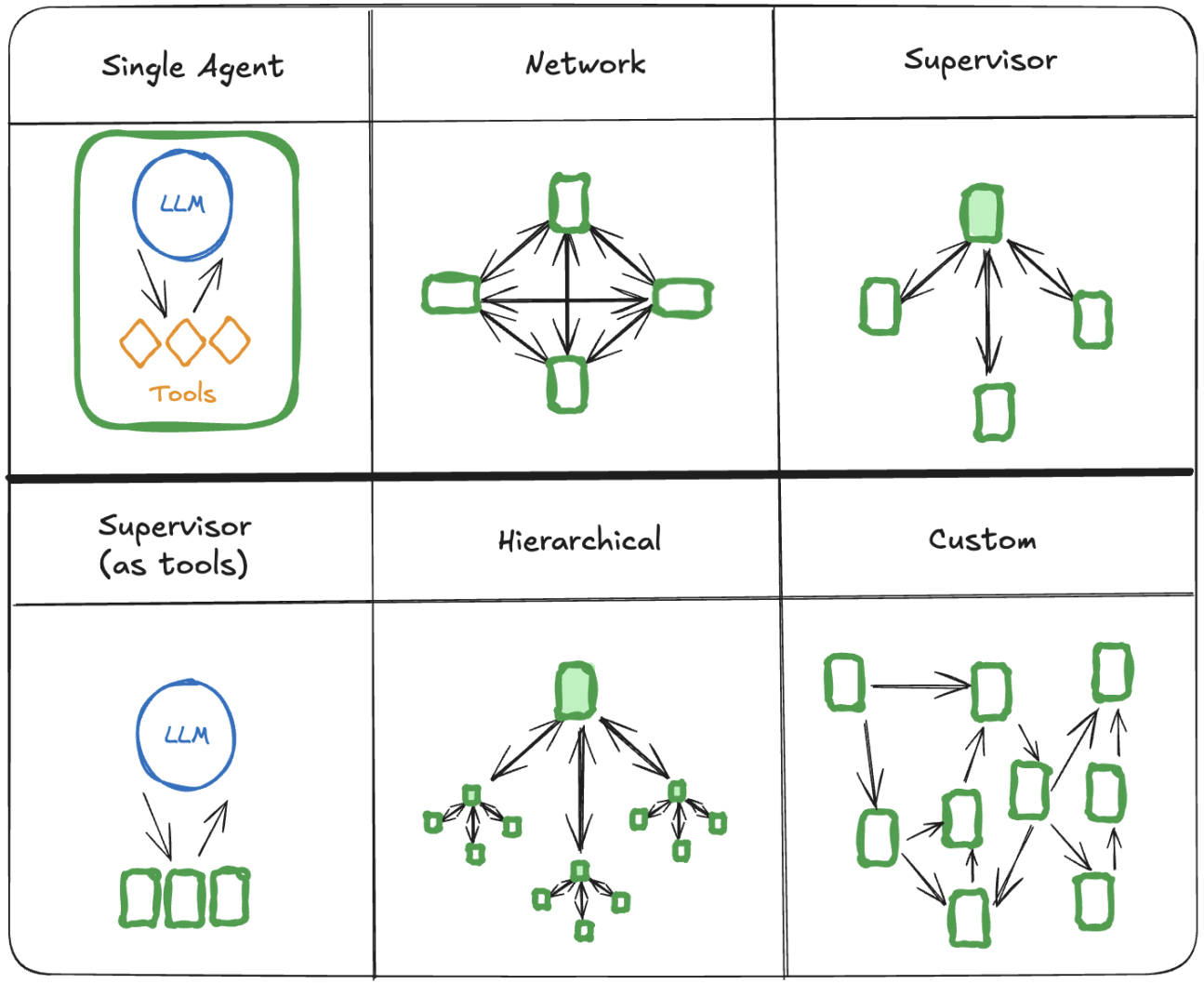


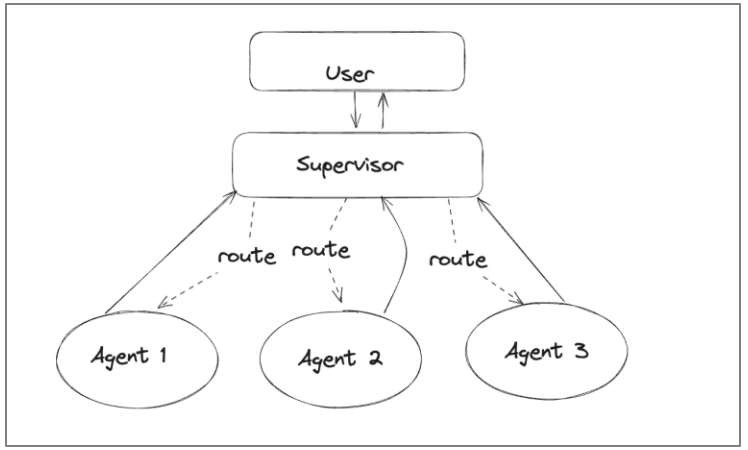
Image Source

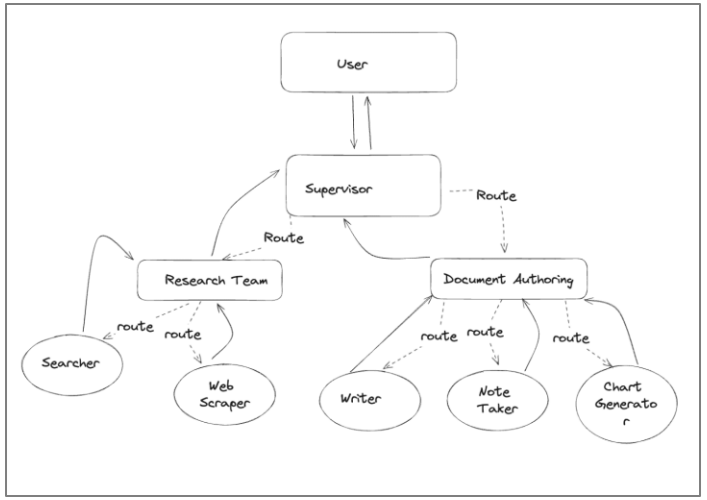
When Single-Agent Systems Fall Short

- **Overloaded Agents:** A single agent can struggle to make good decisions when it has access to too many tools, leading to poor performance.
- **Complex Context:** A single agent can have difficulty managing a large or complex conversational context, causing it to lose track of the user's request.
- **Lack of Specialization:** A single agent isn't equipped to handle a wide range of tasks that require different skills, like planning, research, and mathematical expertise.

Multi-Agent Design Patterns







Reference - LangChain

Guidelines and Considerations

General Guidelines for Building Agents

LangChain

Stage	Define	Design Standard Operating Procedure	Build MVP for Core Prompts	Connect & Orchestrate	Test & Iterate	Deploy, Scale & Refine
Description	*Identify a realistic agent use case and gather 5-10 concrete scenarios the agent should handle*	*Create a step-by-step procedure, breaking down the task(s) into clear steps a human would follow*	*Design agent architecture; Build prompt(s) that handle the core reasoning*	*Connect to live data sources and build orchestration logic*	*Evaluate agent against examples; Identify failures & iterate to improve reliability & performance*	*Deploy agent & monitor how users interact; continuously refine based on real usage patterns.*
Duration	~1-2 week(s)	~1-2 week(s)	~1-2 week(s)	~2-3 week(s)	~1-2 month(s)	Ongoing
Output	Clear agent task scope & concrete use cases	Clear Standard Operating Procedure of step-by-step workflow, incorporating identified scenarios	Agent architecture and core prompt(s) that works with static data	End-to-end agent with live data	Agent with systematically tested performance	Improved agent with expanded capabilities
Stakeholders	Product Owner, Subject Matter Expert	Product Manager, Subject Matter Expert	Product Manager, AI & Prompt Engineer	Engineers	Product Manager, QA Engineer	PM, Engineers, Support
Example: Building an Email Agent	Example tasks: • Categorize email by priority • Schedules meetings • Answer product questions	Step-by-step procedure: • With incoming emails, label with email category & response priority • Checks calendar availability & schedule meeting • Draft response & queue for human review	Email agent architecture design with core prompts, incl. email routing & response generation; Test with manually provided context and sample data	Example integrations: • Gmail & Outlook for email access • Google & Outlook Calendar • CRM database for sender context	Evaluate email responses over success criteria: response quality, accuracy, and professional tone	Monitor traffic, use cases, and feedback. Iterate to improve performance and add additional features

Reference - LangChain

Considerations When Building Agents

Risks

- Unpredictable path
- Unpredictable output
- Unpredictable costs

The Need to Monitor

- Guardrails: they ensure agents behave safely, consistently and within intended boundaries
- Cost, Latency, Tracing etc.
- *Enter AI Gateway as a one-stop solution!*

Recommendation

- Build the right system, not always the most sophisticated one
- Start with simple prompts, optimize with evaluation, then go to multi-step agentic systems

Reference - Anthropic

Hands-On with Agents

Comparison of Popular Agentic AI Frameworks

FEATURE/FRAMEWORK	LANGGRAPH	CREWAI	AUTOGEN
Core Idea	Graph-based workflows, powerful state management for complex cycles.	Role-based multi-agent systems for intuitive collaboration.	Flexible conversation patterns and agent topologies.
Best For	Complex, cyclical processes; human-in-the-loop; production-ready applications.	Team-based workflows; automated customer support, research, or development.	Complex problem-solving; code generation & debugging; diverse agent interactions.
Key Strength	Superior state control & flow; highly customizable; production-grade.	Intuitive collaborative intelligence; clear role delegation.	Flexible conversational dynamics; strong code execution & debugging.
Learning Curve	Steeper (graphs & state)	Medium	Medium
Scalability	High (built for production)	Medium-High	Medium-High
Tooling	Strong via LangChain	Flexible custom tools	Strong (code execution, APIs)

- **Choose LangGraph if:** You need precise control over complex, looping workflows and robust state management for production.

- **Choose CrewAI if:** You want to model collaborative teams of AI agents with distinct roles and intuitive interaction.

- **Choose AutoGen if:** You need highly flexible agent conversations, especially for code-centric problem-solving and iterative tasks.

Assessment (15%)

Deep-Dive into TrueFoundry AI Gateway

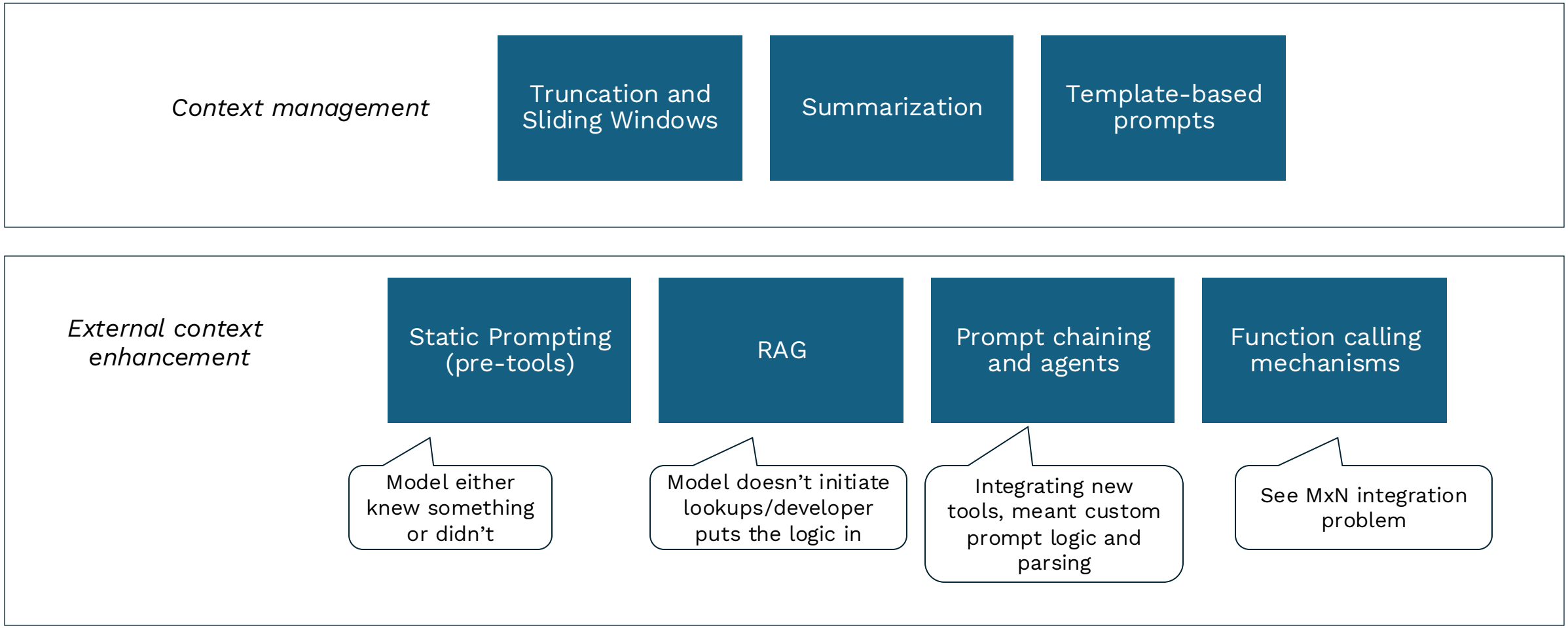
Reuse YouTube Video for an Introduction

Hands-on Demo of TrueFoundry AI Gateway

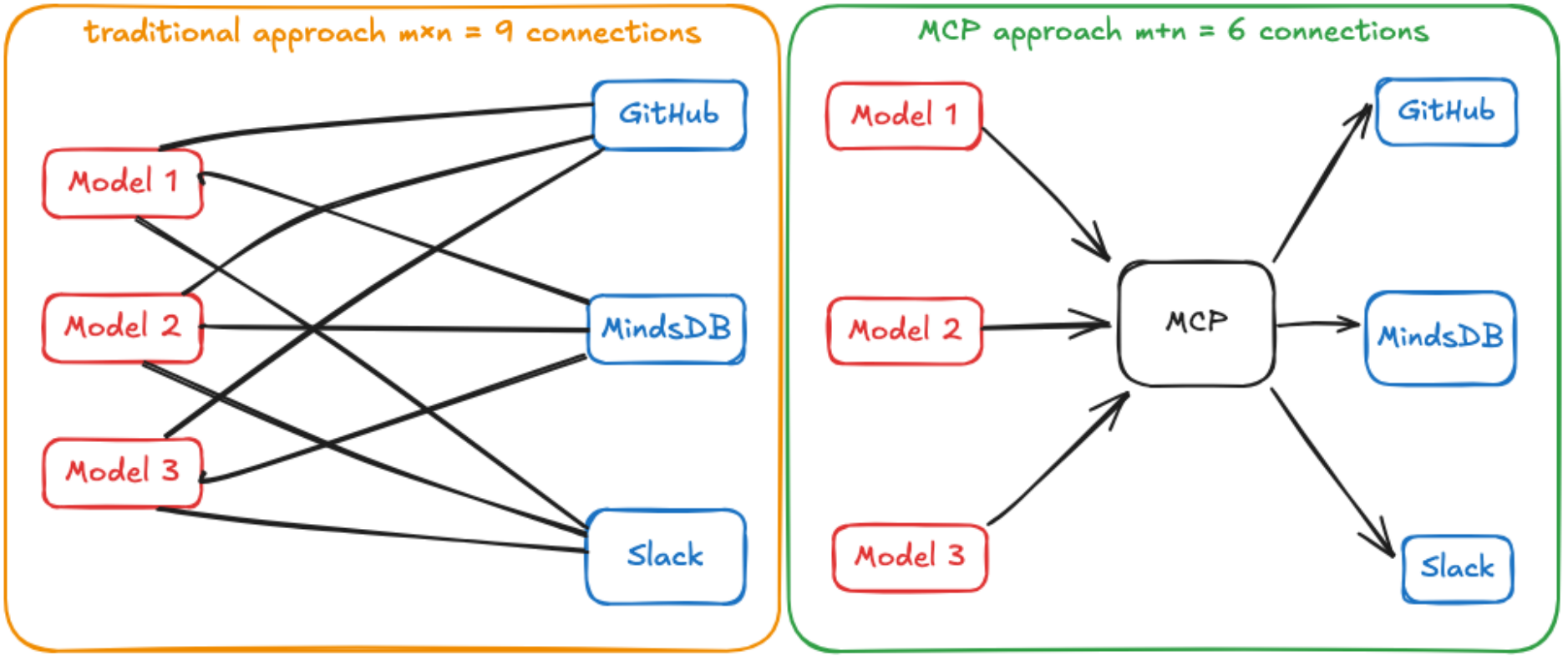
Model Context Protocol

Pre-MCP Context Management

This was a set of one-off solutions that required extensive engineering, and wasn't completely proactive from the model's side.



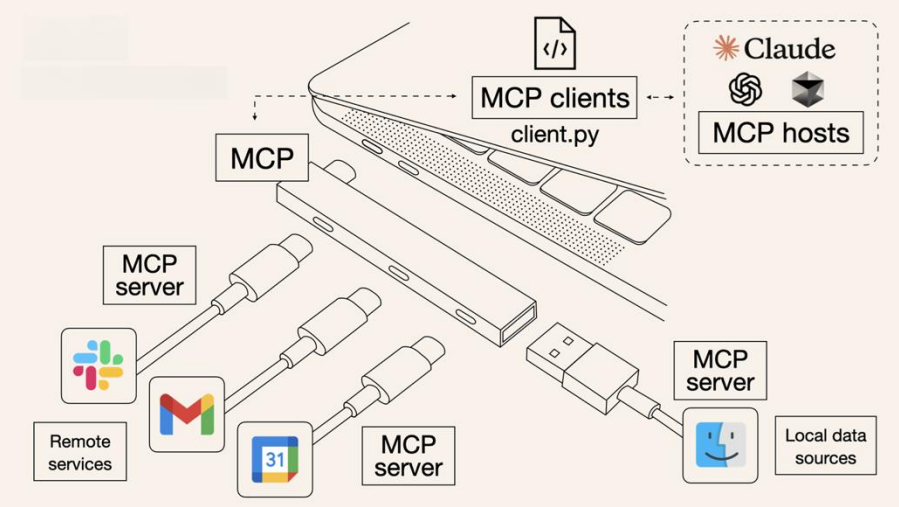
MxN Integration Problem

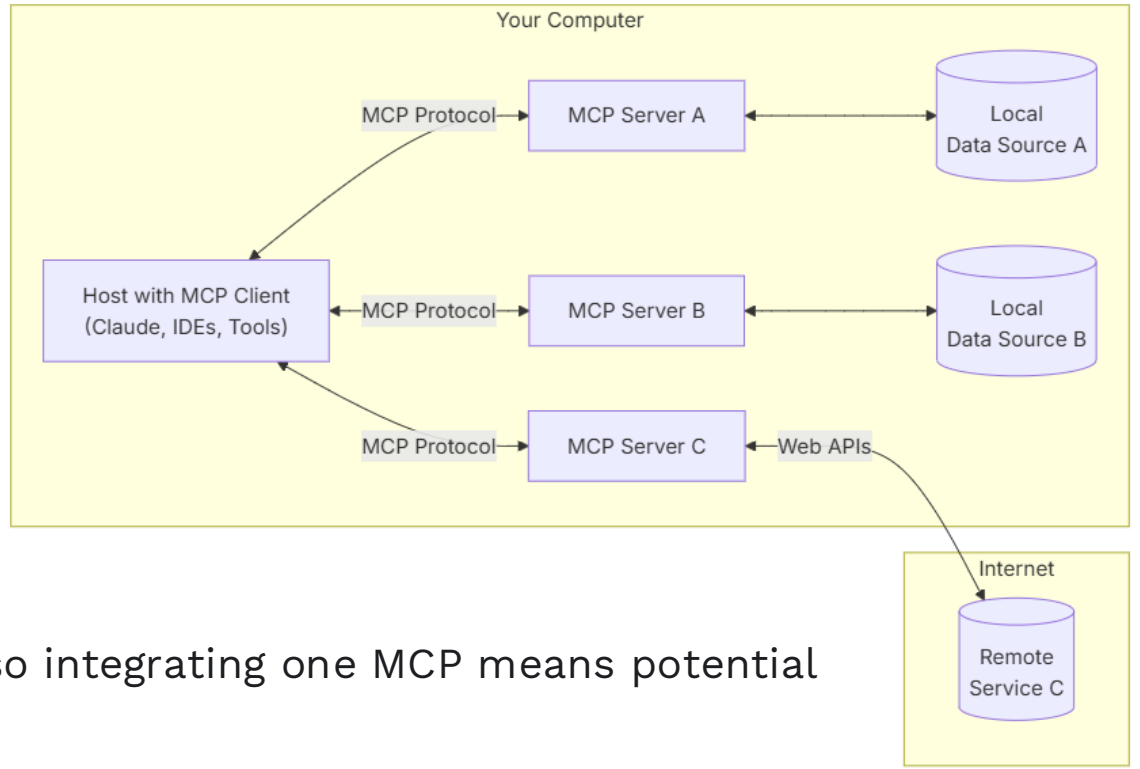


Source

Model Context Protocol (MCP)

MCP is an open protocol that standardizes how applications provide context to LLMs.





Single protocol: MCP acts as a standardized "connector," so integrating one MCP means potential access to multiple tools and services, not just one

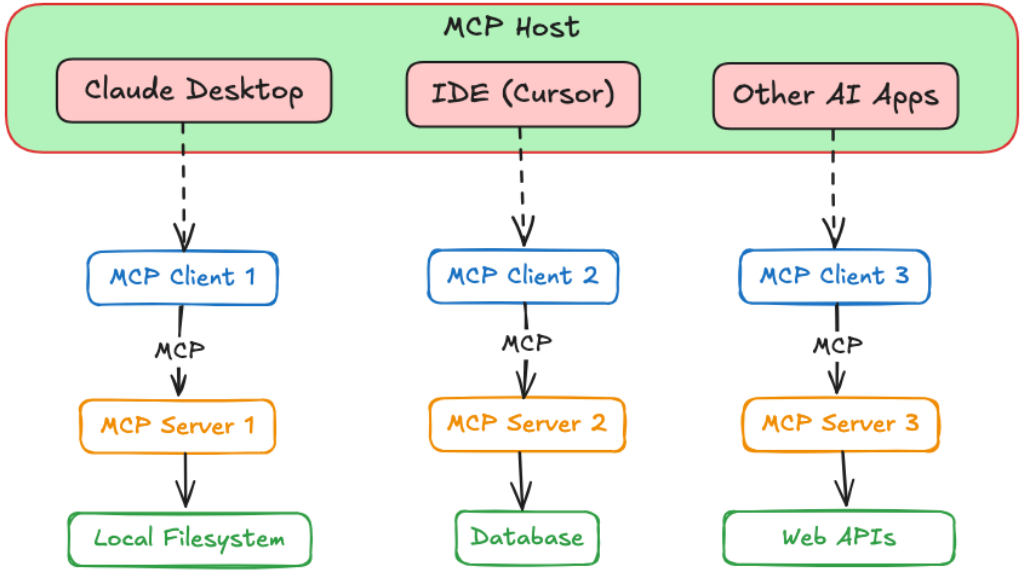
Dynamic discovery: MCP allows AI models to dynamically discover and interact with available tools without hard-coded knowledge of each integration

Two-way communication: MCP supports persistent, real-time two-way communication. The AI model can both retrieve information and trigger actions dynamically

Architecture

MCP (Model Context Protocol) uses a client-server architecture to connect AI models with external resources, built on three key components: the **Host**, **Clients**, and **Servers**.

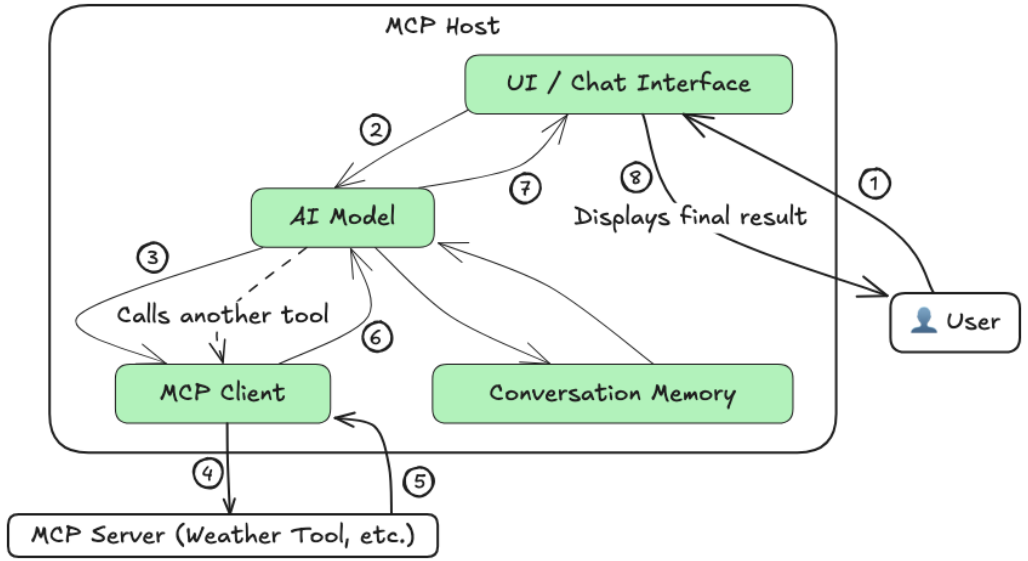
- The **Host** is the AI application or agent the user interacts with (e.g., a desktop app, IDE plugin, or custom LLM app). It can simultaneously connect to multiple MCP Servers.
- An **MCP Client** acts as an intermediary, managing each server connection. Each client handles communication with a single MCP Server, providing security isolation. The Host spawns one client per server, creating a dedicated, one-to-one link.
- The **MCP Server** is an external program implementing the MCP standard. It offers specific capabilities, including tools, data access, and domain-related prompts. Servers can interface with databases, cloud services, and other data sources, with existing examples for Google Drive, Slack, GitHub, various databases, and web browsers.



Source Image

How it Works

1. **Capability Discovery**: The MCP client fetches the list of available tools/resources from the server and shares it with the AI model.
2. **Augmented Prompting**: The user's query is sent to the model along with descriptions of the server's tools/resources, enabling the model to understand what external actions are possible.
3. **Tool/Resource Selection**: The model decides if an external tool is needed and responds with a structured request (per MCP spec) indicating the desired tool (e.g., “Weather API”).
4. **Server Execution**: The client invokes the requested tool on the MCP server, which performs the action (e.g., API call or database query) and returns the result.
5. **Response Generation**: The model receives the result, integrates it with its own knowledge, and generates a final, enriched response for the user.



Source Image

MCP vs. RAG and Vector Search

- **RAG** appends retrieved documents directly into prompts as raw text — useful but unstructured.
- **MCP** offers **structured context management**: retrieved data is slotted into labeled *Resource* blocks (e.g., “Retrieved Documents”), keeping different types of information organized.
- You can **combine RAG with MCP**: fetch via vector search, then pass results as structured context — easier to manage, update, and debug.

MCP vs. Monolithic Prompts and Finetuning

- Traditional methods relied on:
  - **Monolithic prompts** – bulky, hard to edit, expensive (token-wise).
  - **Fine-tuning** – slow, costly, hard to adapt on the fly.
- **MCP** enables **modular prompting**:
  - Define reusable context layers (e.g., “Policy Guidelines”, “User History”).
  - Update individual layers without rewriting or retraining.
- Saves time and tokens while improving flexibility and maintainability.

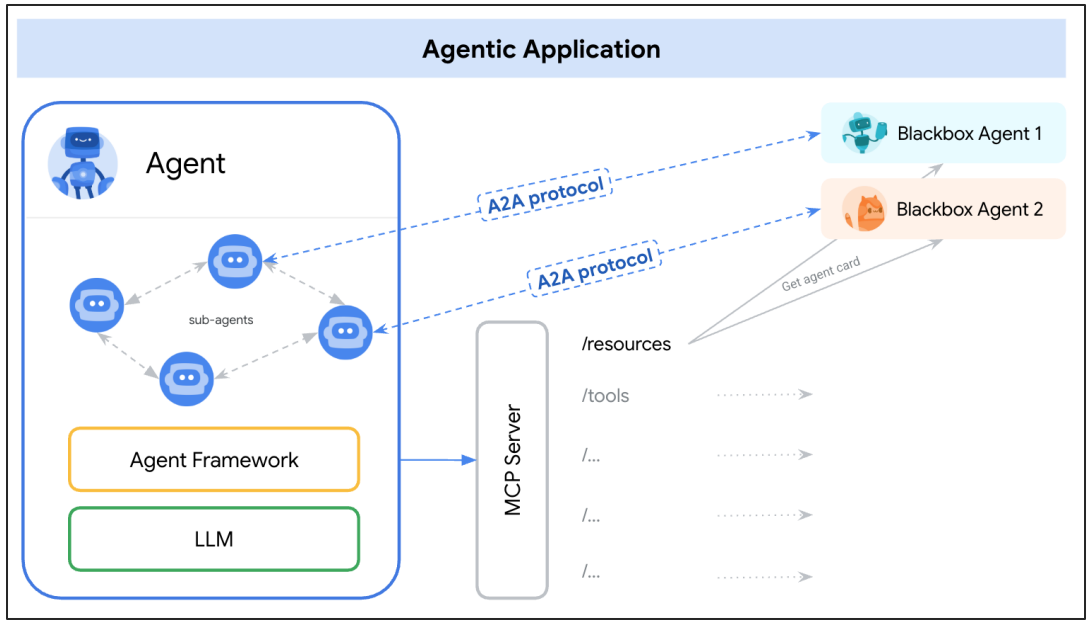
MCP vs. Function Calling and Plugins

- Function calling/tools in LLMs (like OpenAI plugins) are **powerful but fragmented** — each provider has its own setup.
- **MCP standardizes tool use** across models with unified *Tool*, *Prompt*, and *Resource* primitives.
- **Interoperable & extensible**:
  - Write once, work across providers (e.g., OpenAI, Claude, Llama).
  - Covers non-functional context needs (like guidance, templates) beyond API calls.

MCP vs. A2A

A2A and MCP are both complementary protocols

Feature	Agent2Agent Protocol (A2A)	Model Context Protocol (MCP)
Interaction	Between agents (from different frameworks also!)	Between agent and tools
Use Case	Delegating tasks to other agents	Enhancing context with external data
Dependency	Relies on other agents	Relies on tool integrations
Architecture	Multi-agent system	Tool-augmented single-agent system



MCP and A2A Integration

- ✓ By representing **A2A agents as MCP resources**—encapsulated within their **AgentCards**—AI applications facilitate **dynamic agent discovery**.
- ✓ This allows agents on an MCP server to **connect and collaborate** via the A2A protocol.

Image source here

Basics of FastMCP and CrewAI

Important Links for FastMCP

https://gofastmcp.com/tutorials/mcp

https://gofastmcp.com/tutorials/create-mcp-server

https://gofastmcp.com/getting-started/quickstart

What CrewAI Says

How CrewAI Compares

CrewAI's Advantage: CrewAI combines autonomous agent intelligence with precise workflow control through its unique Crews and Flows architecture. The framework excels at both high-level orchestration and low-level customization, enabling complex, production-grade systems with granular control.

- **LangGraph**: While LangGraph provides a foundation for building agent workflows, its approach requires significant boilerplate code and complex state management patterns. The framework's tight coupling with LangChain can limit flexibility when implementing custom agent behaviors or integrating with external systems.

P.S. CrewAI demonstrates significant performance advantages over LangGraph, executing 5.76x faster in certain cases like this QA task example (see comparison) while achieving higher evaluation scores with faster completion times in certain coding tasks, like in this example (detailed analysis).

- **Autogen**: While Autogen excels at creating conversational agents capable of working together, it lacks an inherent concept of process. In Autogen, orchestrating agents' interactions requires additional programming, which can become complex and cumbersome as the scale of tasks grows.
- **ChatDev**: ChatDev introduced the idea of processes into the realm of AI agents, but its implementation is quite rigid. Customizations in ChatDev are limited and not geared towards production environments, which can hinder scalability and flexibility in real-world applications.

https://github.com/crewAIInc/crewAI?tab=readme-ov-file#how-crewai-compares

Deploy Your First MCP Server

CI/CD

What Does It Mean for Agentic AI?

Evaluating LLM Applications

Why Evaluate at All?

Evaluation helps address and identify the bottlenecks related to:

- Hallucinations *(absence of relevant context in final prompt is the most common reason for this, but there could be others)*
- Reduced reasoning capabilities (*if all the relevant context is not provided to LLM, it can lead to this*)
- Unexpected latencies (*what would happen is the retrieval might not be efficient leading to high latencies*)
- Explainability of outputs (*i.e., was performance bad due to the LLM itself or due to other aspects of the application – think agents, RAG, agentic RAG etc.)*

It also helps build trust among users especially for use in high-stakes industries like Finance, Healthcare etc.

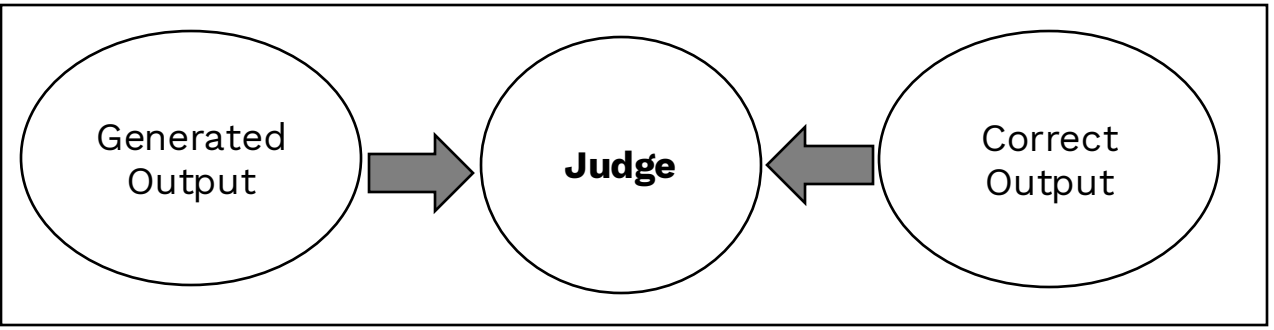
And in general, it is a best practice for building any kind of an “AI” application.

Evaluating a RAG Application

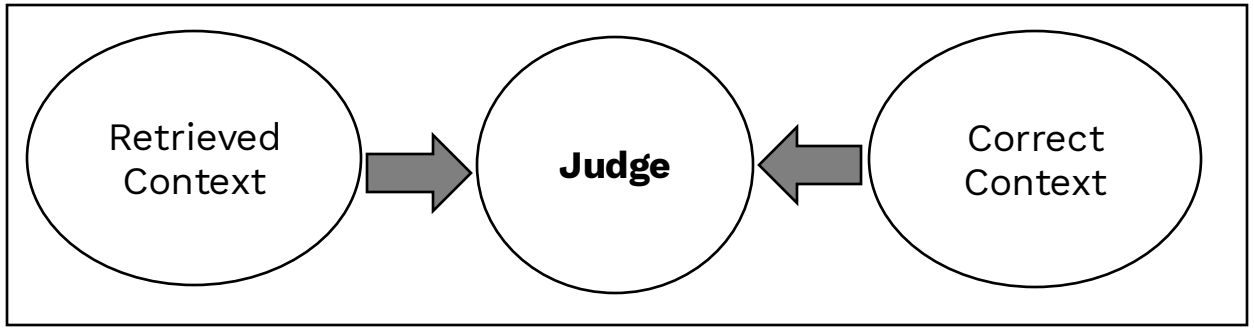
Typically we think of evaluating only the outputs. And if the result is poor, we might attribute it to the LLM itself.

However, remember such an LLM application is based off RAG, and instead of only focusing on "G" = Generation, **we need to also evaluate the "R" = Retrieval which is the other main pillar of such an application.**

Output vs. Output Evaluation



Context vs. Context Evaluation



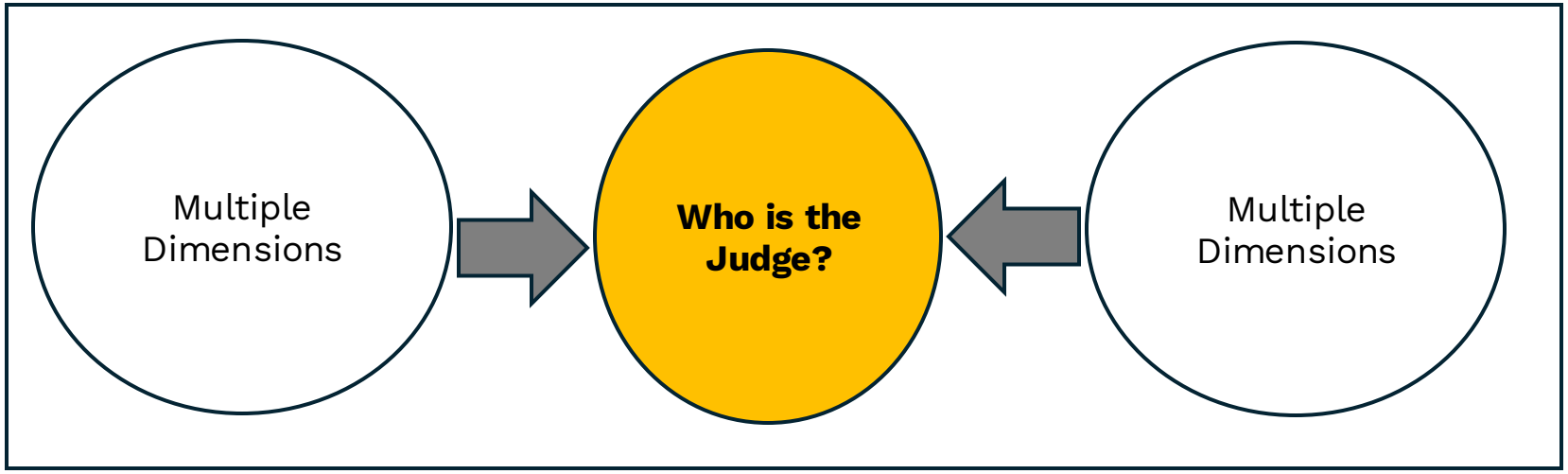
But even then, with the correct context, sometimes the LLM doesn't ground the output in the context leading to hallucinations!

SOLUTION: There are a lot of evaluation metrics that help address these varied issues.

LLM-as-a-Judge Paradigm

As we discussed, there are more dimensions than just *output vs. output evaluation*.

But HOW do we do this?



Judge can be

1) Human (***pros***: *super good;* ***cons***: *time-expensive, not scalable*)
2) a Statistical NLP based method (***pros***: *deterministic, scalable;* ***cons***: *limited in semantic capabilities*)
3) **an LLM itself! = LLM as a Judge!** (***pros***: *good semantic capabilities, scalable;* ***cons***: *scale comes with cost, black box/hallucination could potentially occur*)

Overview of Evaluation Metrics

Now let's see an overview of evaluation metrics, and the dimensions each of these is designed to evaluate.

Dimensions

	S.No.	Metric	User Question	Ground Truth Context	Retrieved Context	Actual Answer	Generated Answer
Retrieval Metrics	1	Context Precision@k	✗	✓	✓	✗(✓)	✗
	2	Context Recall	✗	✓	✓	✗(✓)	✗
Generation Metrics	3	Faithfulness	✗	✗	✓	✗	✓
	4	Answer Relevancy	✓	✗	✗	✗	✓
	5	Prompt Alignment	✓	✗	✗	✗	✓
	6	Toxicity	✗	✗	✗	✗	✓

We'll deep-dive into S.No. 1-6 using the popular frameworks **RAGAs** and **DeepEval** in Python.

Evaluation Metric 1: Context Recall

Key Takeaway

Out of the relevant available context, how much relevant context is retrieved?
(can be done using actual answer also, instead of ground truth context)

User Question	Ground Truth Context	Retrieved Context	Actual Answer	Generated Answer
❌	✅	✅	❌ (✅)	❌

Definition

$$Context\ Recall = \frac{Number\ of\ claims\ in\ the\ reference\ supported\ by\ the\ context}{Total\ number\ of\ claims\ in\ the\ reference}$$

Code Snippet (RAGAs)

```python
sample = SingleTurnSample(
    user_input="What are the interest rates of loans for the new vehicles?",
    response="Interest rates for new vehicle's loans are expected to be around 6.7% on average",
    reference="Interest rates for new vehicle's loans are expected to be around 6.7% on average",
    retrieved_contexts=["Average interest rates for new-vehicle loans are expected to be 6.7%, down 16 basis points from a year ago.",
                        "Average monthly finance payments this month are on pace to be $734, up $11 from January 2024"]
)

await context_recall.single_turn_ascore(sample)
```

Output

1.0

Relative Cost of Evaluation

Small to medium depending on system prompt length

Evaluation Metric 2: Context Precision@k

Key Takeaway

Out of the all context that's retrieved, how much context is relevant and ranked higher than irrelevant ones?
(can be done using actual answer also, instead of ground truth context)

User Question	Ground Truth Context	Retrieved Context	Actual Answer	Generated Answer
❌	✅	✅	❌ (✅)	❌

Definition

$$Context\ Precision@k = \frac{\sum_{k=1}^{K}(Precision@k\ \times v_k)}{Total\ number\ of\ relevant\ items\ in\ the\ top\ K\ results}$$

Code Snippet (RAGAs)

```python
sample = SingleTurnSample(
    user_input="What are the interest rates of loans for the new vehicles?",
    response="Interest rates for new vehicle's loans are expected to be around 6.7% on average",
    retrieved_contexts=["Average monthly finance payments this month are on pace to be $734, up $11 from January 2024",
                        "Average interest rates for new-vehicle loans are expected to be 6.7%, down 16 basis points from a year ago."]
)

await context_precision.single_turn_ascore(sample)
```

Output

0.5

Relative Cost of Evaluation

Small to medium depending on system prompt length

Evaluation Metric 3: Faithfulness

Key Takeaway

How much of the generated answer is attributed to the retrieved context, versus, the model's own knowledge base?

User Question	Ground Truth Context	Retrieved Context	Actual Answer	Generated Answer
❌	❌	✅	❌	✅

Definition

$$Faithfulness\ Score = \frac{Number\ of\ claims\ in\ the\ response\ supported\ by\ the\ retrieved\ context}{Total\ number\ of\ claims\ in\ the\ response}$$

Code Snippet (RAGAs)

```python
sample = SingleTurnSample(
    user_input="What are the interest rates of loans for the new vehicles?",
    response="Interest rates for new vehicle's loans are expected to be around 6.7% on average",
    retrieved_contexts=["Average monthly finance payments this month are on pace to be $734, up $11 from January 2024",
                        "Average interest rates for new-vehicle loans are expected to be 6.7%, down 16 basis points from a year ago."]
)

await faithfulness.single_turn_ascore(sample)
```

Output

1.0

Relative Cost of Evaluation

Small to medium depending on system prompt length

Evaluation Metric 4: Answer Relevancy

Key Takeaway

How relevant is the response generated by the LLM, compared to what the user had originally asked?

User Question	Ground Truth Context	Retrieved Context	Actual Answer	Generated Answer
✅	❌	❌	❌	✅

Definition

$$Answer\ Relevancy = \frac{1}{N}\sum_{i=1}^{N} cosine\ similarity(E_{g_i}, E_a)$$

$$=$$

$$Answer\ Relevancy = \frac{1}{N}\sum_{i=1}^{N} \frac{E_{g_i} \cdot E_a}{||E_{g_i}||||E_a||}$$

by default, 3 questions (i. e. , N = 3) are generated by the LLM but this parameter can be changed

Code Snippet (RAGAs)

```python
sample = SingleTurnSample(
    user_input="What are the interest rates of loans for the new vehicles?",
    response="Interest rates are the rates charged by the financier for the loans bowrrowed by the customer.",
    retrieved_contexts=["Average monthly finance payments this month are on pace to be $734, up $11 from January 2024",
                        "Average interest rates for new-vehicle loans are expected to be 6.7%, down 16 basis points from a year ago."]
)

await relevancy.single_turn_ascore(sample)
```

Output

0.56

Relative Cost of Evaluation

Medium to large as multiple LLM calls are involved

Evaluation Metric 5: Prompt Alignment

Key Takeaway

How well did the generated answer follow the instructions mentioned in the prompt?

User Question	Ground Truth Context	Retrieved Context	Actual Answer	Generated Answer
✅	❌	❌	❌	✅

Definition

$$Prompt\ Alignment = \frac{Number\ of\ instructions\ followed}{Total\ number\ of\ instructions}$$

Code Snippet (DeepEval)

```python
test_case = LLMTestCase(
    input="What are the interest rates of loans for the new vehicles?",
    actual_output="Interest rates for new vehicle's loans are expected to be around 6.7% on average.",
    context=["Average monthly finance payments this month are on pace to be $734, up $11 from January 2024",
             "Average interest rates for new-vehicle loans are expected to be 6.7%, down 16 basis points from a year ago."]
)
metric = PromptAlignmentMetric(prompt_instructions=["You are an AI assistant that helps with answering user's question according to the context given."],
                               model=groq_llm)

output = evaluate([test_case], metrics=[metric], show_indicator=False, print_results=False)
```

Output

1.0

Relative Cost of Evaluation

Medium to large as multiple LLM calls are involved

By default it's 2 (1. Checks the no. of instructions which were followed in the generated answer

2. Generates a reason for the score calculated given the verdict for the opinions for the instructions which were not followed)

Evaluation Metric 6: Toxicity

Key Takeaway

Measures how toxic the generated response is, in terms of number of opinions.

User Question	Ground Truth Context	Retrieved Context	Actual Answer	Generated Answer
❌	❌	❌	❌	✅

Definition

$$Toxicity = \frac{Number\ of\ toxic\ opinions}{Total\ number\ of\ opinions}$$

Code Snippet (DeepEval)

```python
test_case = LLMTestCase(
    input="What are the interest rates of loans for the new vehicles?",
    actual_output="The interest rate is 6.7%, which is down 16 pathetic basis points",
    context=["Average monthly finance payments this month are on pace to be $734, up $11 from January 2024",
             "Average interest rates for new-vehicle loans are expected to be 6.7%, down 16 basis points from a year ago."]
)
metric = ToxicityMetric(threshold=0.1, model=groq_llm)

output = evaluate([test_case], metrics=[metric], show_indicator=False, print_results=False)
```

Output

1.0

Relative Cost of Evaluation

Medium to large as multiple LLM calls are involved

By default it's 3 (1. Call to generate all the opinions in the generated answer

2. Checks which opinions are toxic

3. Generates a reason for the score calculated given the toxic opinions)