AI for App Development

Presented by Brent Laster &

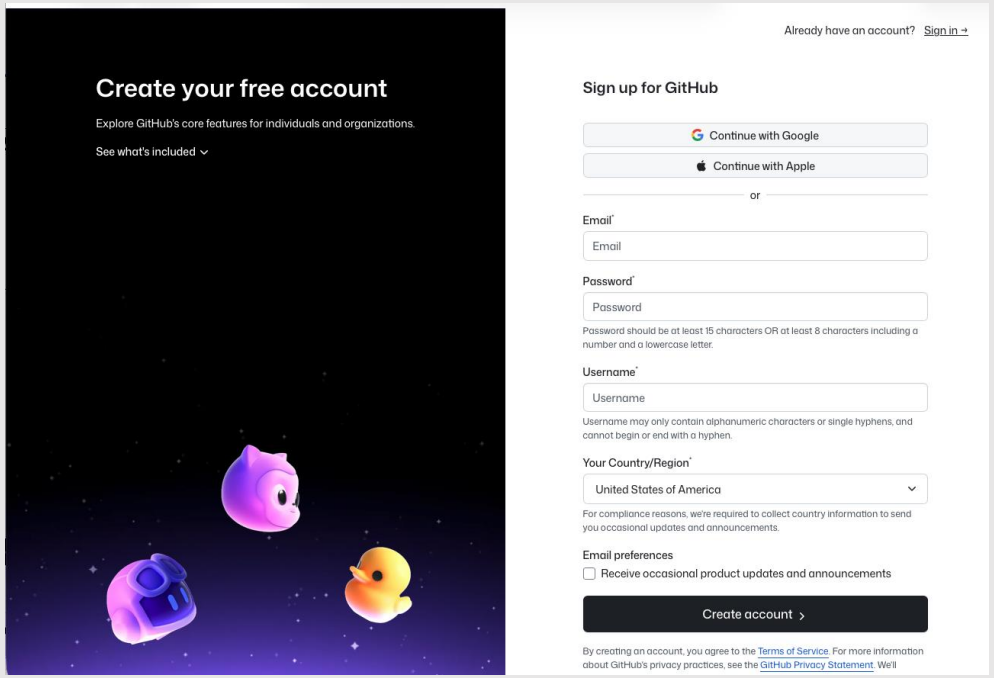
Tech Skills Transformations LLC

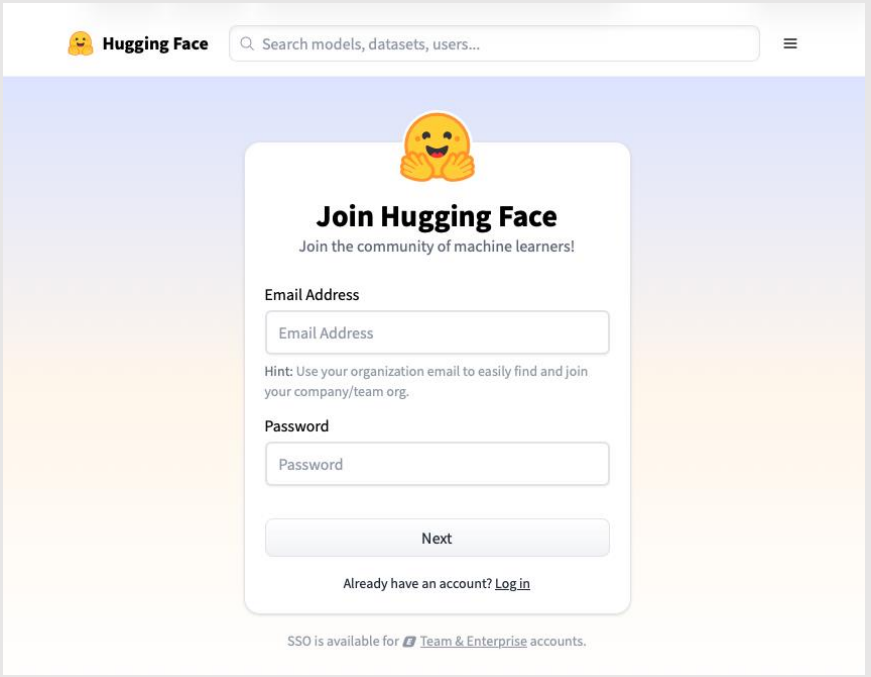
TECHUPSKILLS ®

Prerequisites

- Account on the public GitHub.com (free tier is fine)
  - https://github.com/signup

- Account on huggingface.co (free tier is fine)
  - https://huggingface.co/join



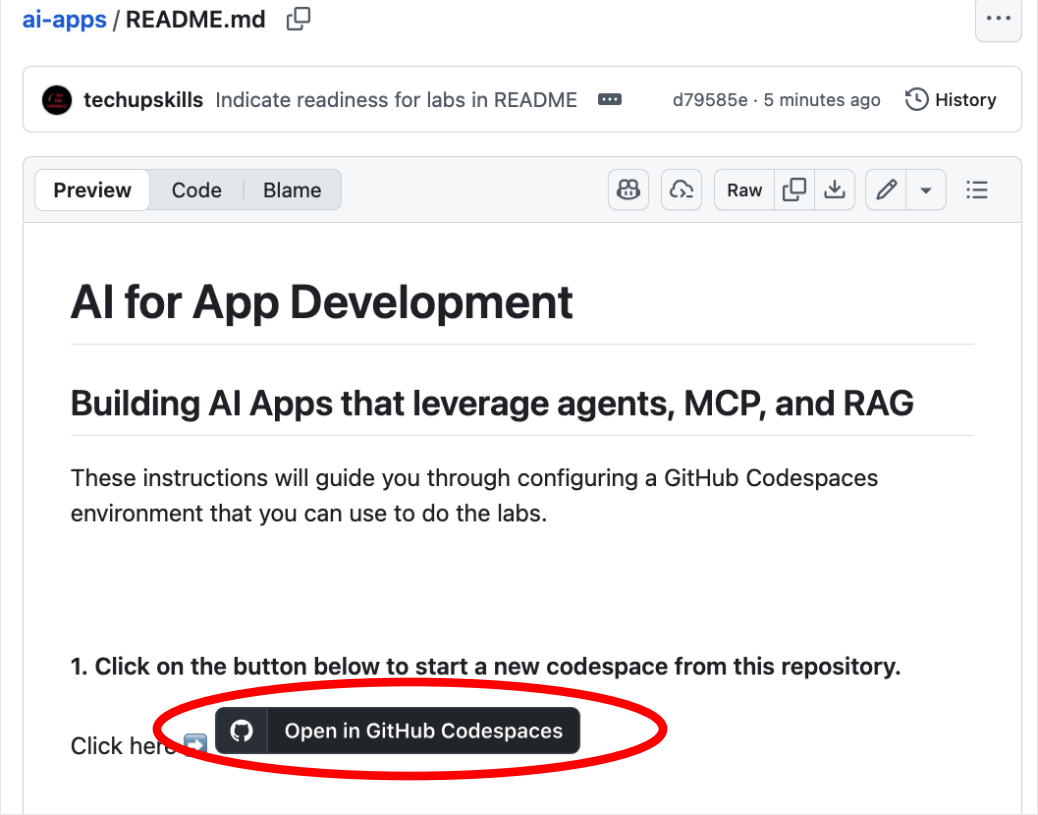


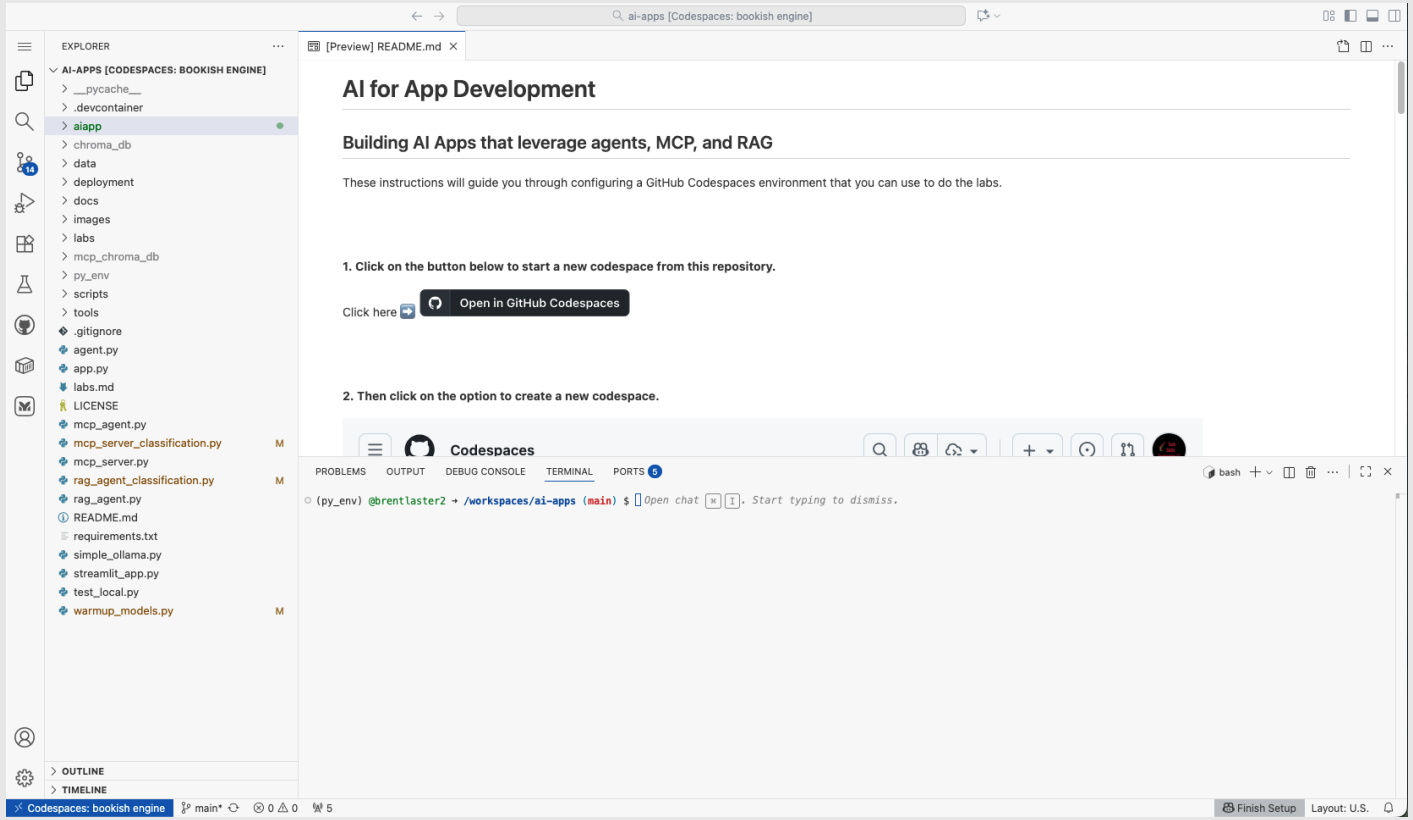
Logistics

- Scheduled 8:45 – 5
- Workshop is lecture and labs
- Will have breaks to do labs
- Lunch break (time TBD)

Lab setup

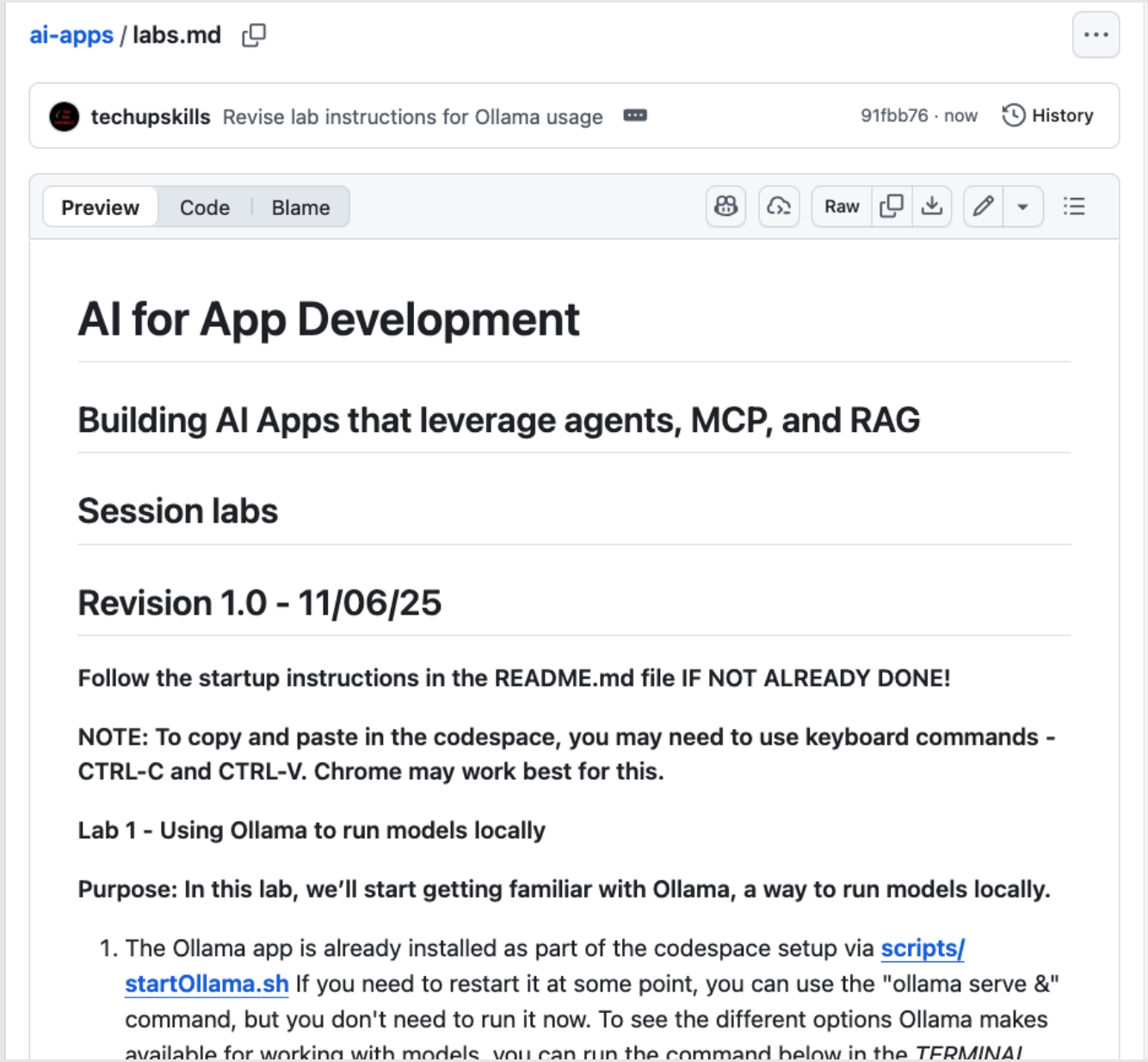
- Repository is https://github.com/skillrepos/ai-apps
- Labs recommended to be run in codespace environment
- Follow steps in README.md file
- Creates GitHub Codespace (free with github account) – vscode-like interface





Workshop Labs

- In labs.md file in repo (https://github.com/skillrepos/ai-apps/blob/main/labs.md)
- Follow numbered steps
- Gray boxes contain content to put in terminal or in files (can copy and paste)
- Screenshots included
- "code" command opens new file or existing file

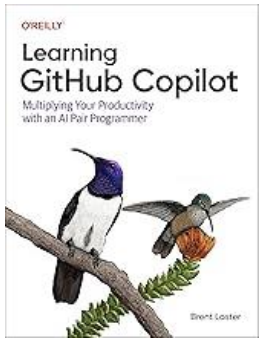


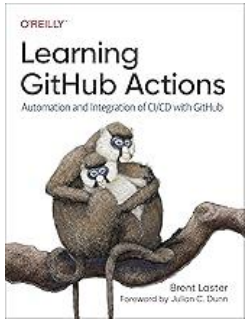
About me

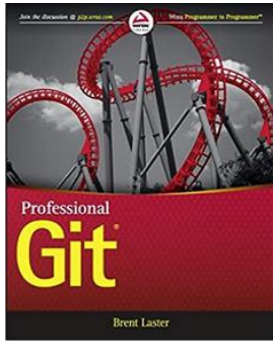
- LinkedIn: brentlaster
- X: @BrentCLaster
- Bluesky: brentclaster.bsky.social
- GitHub: brentlaster

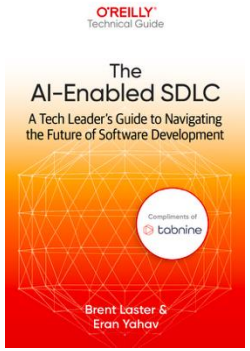
Long career in corporate:

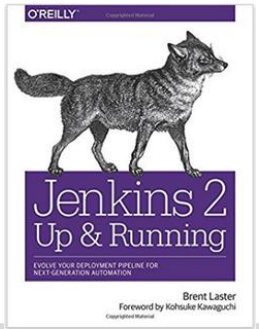
- *Principal Dev*
- *Manager/Senior Manager*
- *Director*











- Founder, Tech Skills Transformations LLC
- https://getskillsnow.com
- info@getskillsnow.com

Our Agenda

Large Language Models (LLMs)

- A type of AI model trained on massive amounts of text and data, usually from sources like internet content
- Uses deep learning (NLP + ML) to understand content and record it as numeric relationships
- Perform tasks like content summarization and generation, sentiment analysis, and translation
- Make predictions for "next word" based on their input and training
- Reasoning models think "step by step"

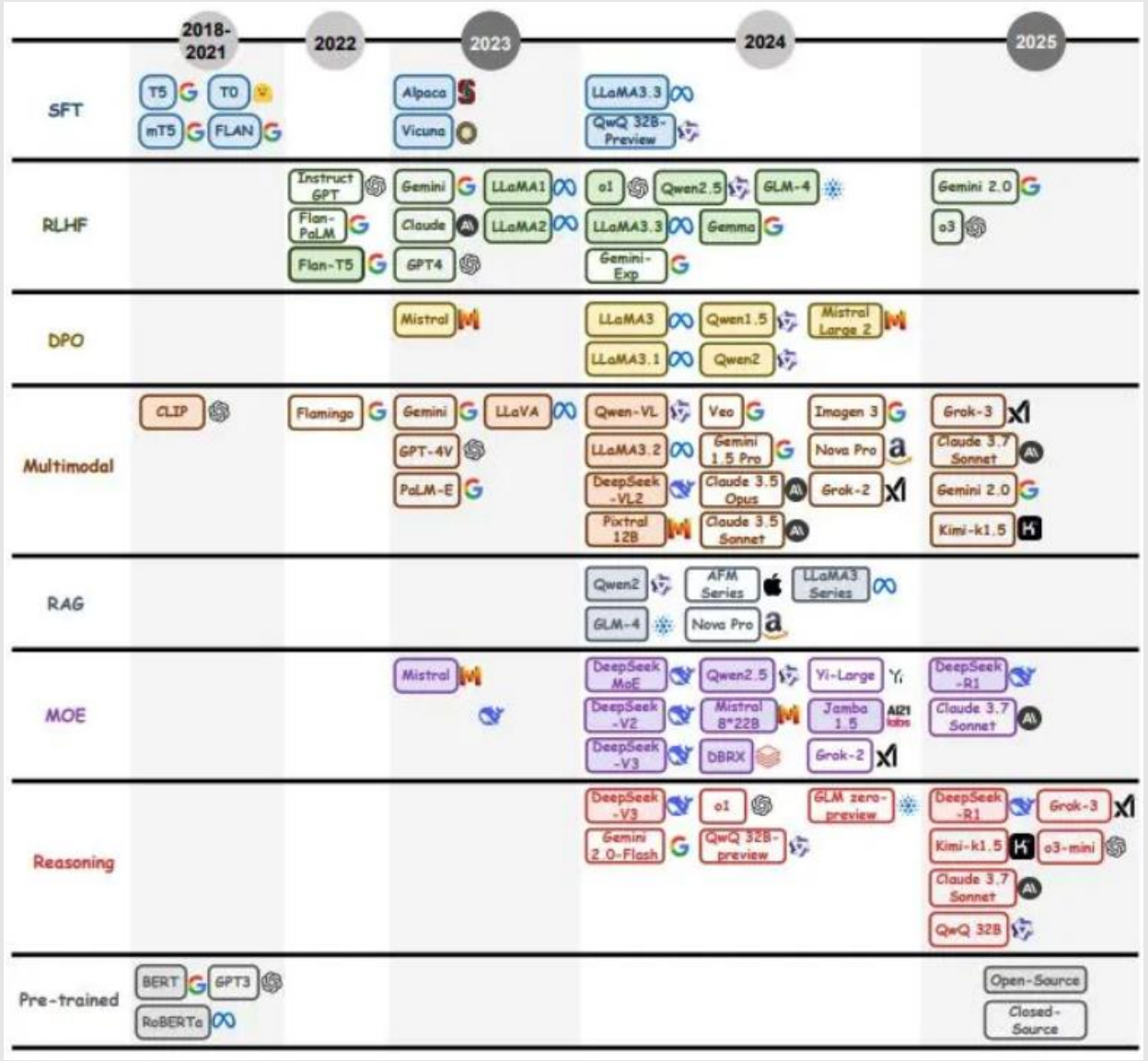


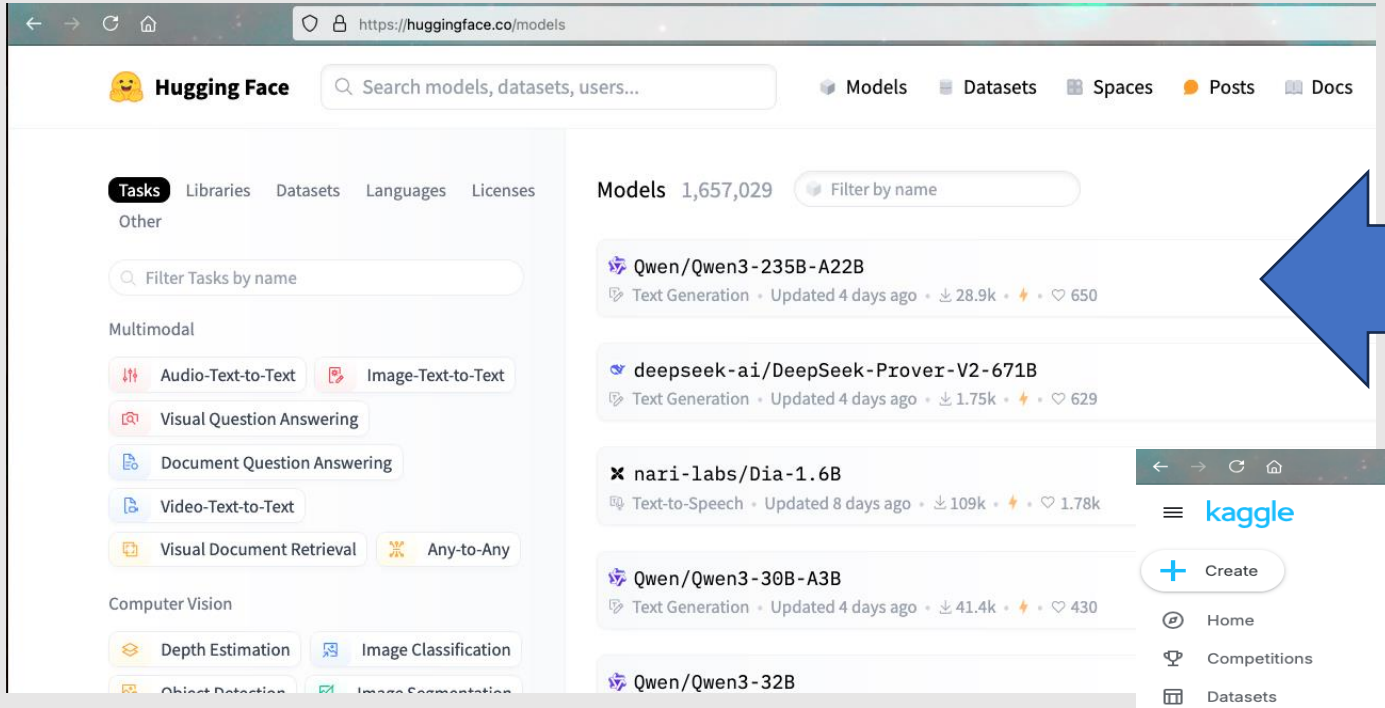
image source: https://medium.com/@genai.works/the-evolution-of-generative-ai-2018-2025-timeline-revealed-ff1cf4d46fcc

Running models locally

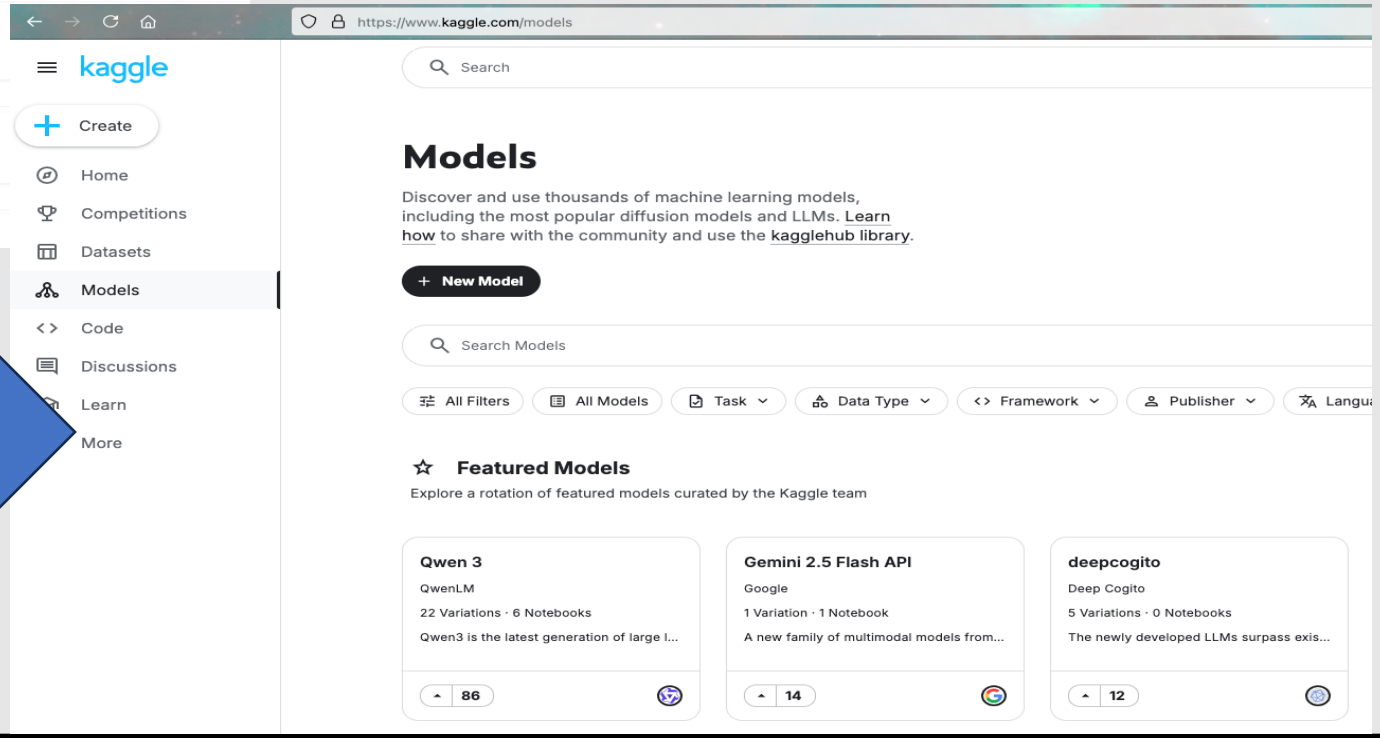
Why run models locally?

- Privacy - no need to share data
- Gives you control over setup, configuration, and customization options
  - Can tailor LLM to your needs, experiment with settings, integrate into your infra
- Can easily swap between different models for different tasks
- Work in offline mode
- No censoring of results
- Cost savings
  - No charges for subscriptions or API calls
- Useful for prototyping

Where to get models +



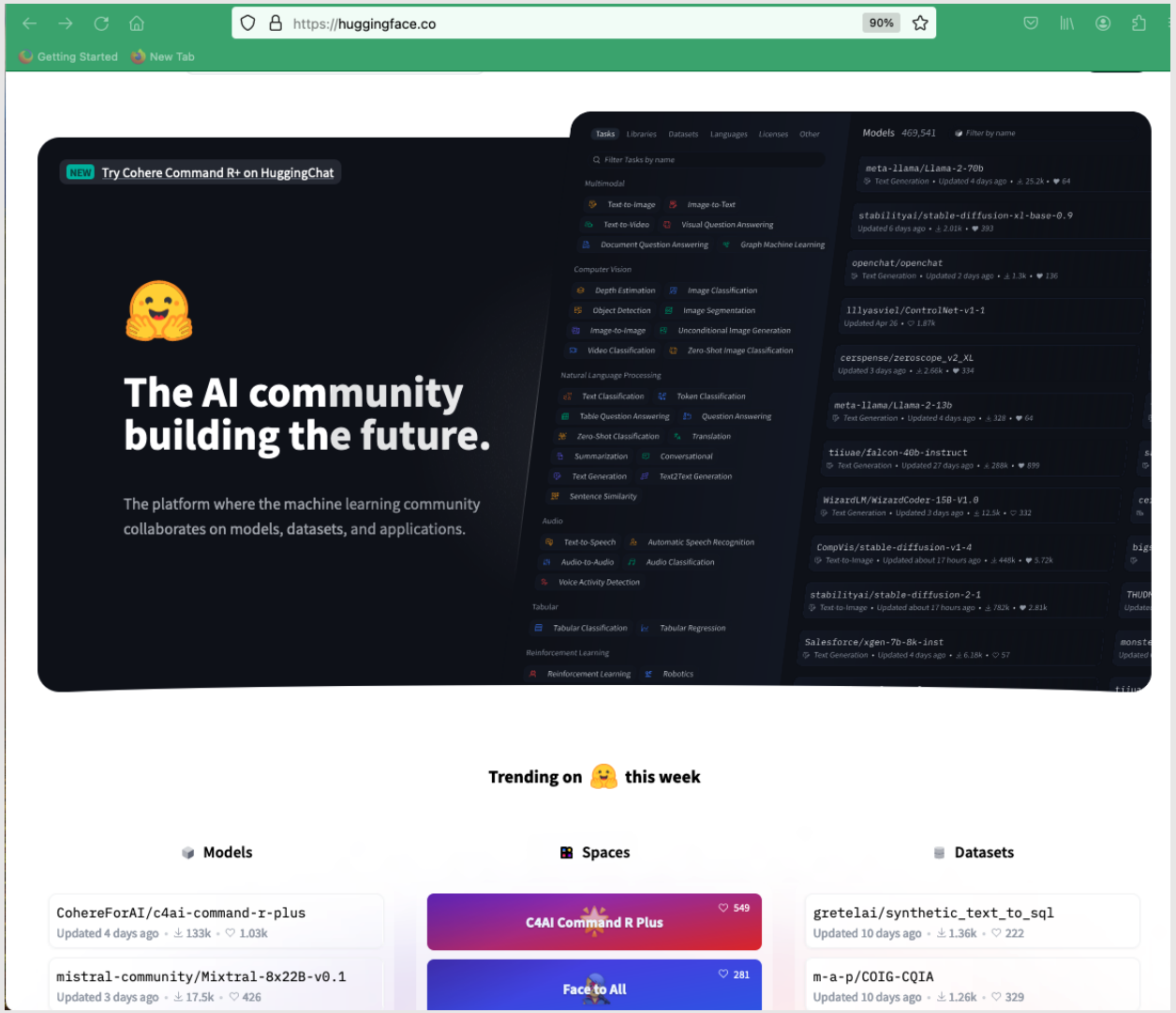
http://huggingface.co/models



http://kaggle.com/models

Hugging Face

- https://huggingface.co
- Community focused on creating and sharing AI models
- Many free and open for your use and pre-trained
- Like Docker Hub for ML and AI
- Offers lots of open-source models
- Provides transformers - Python library that streamlines running LLM locally



Options for running LLMs locally

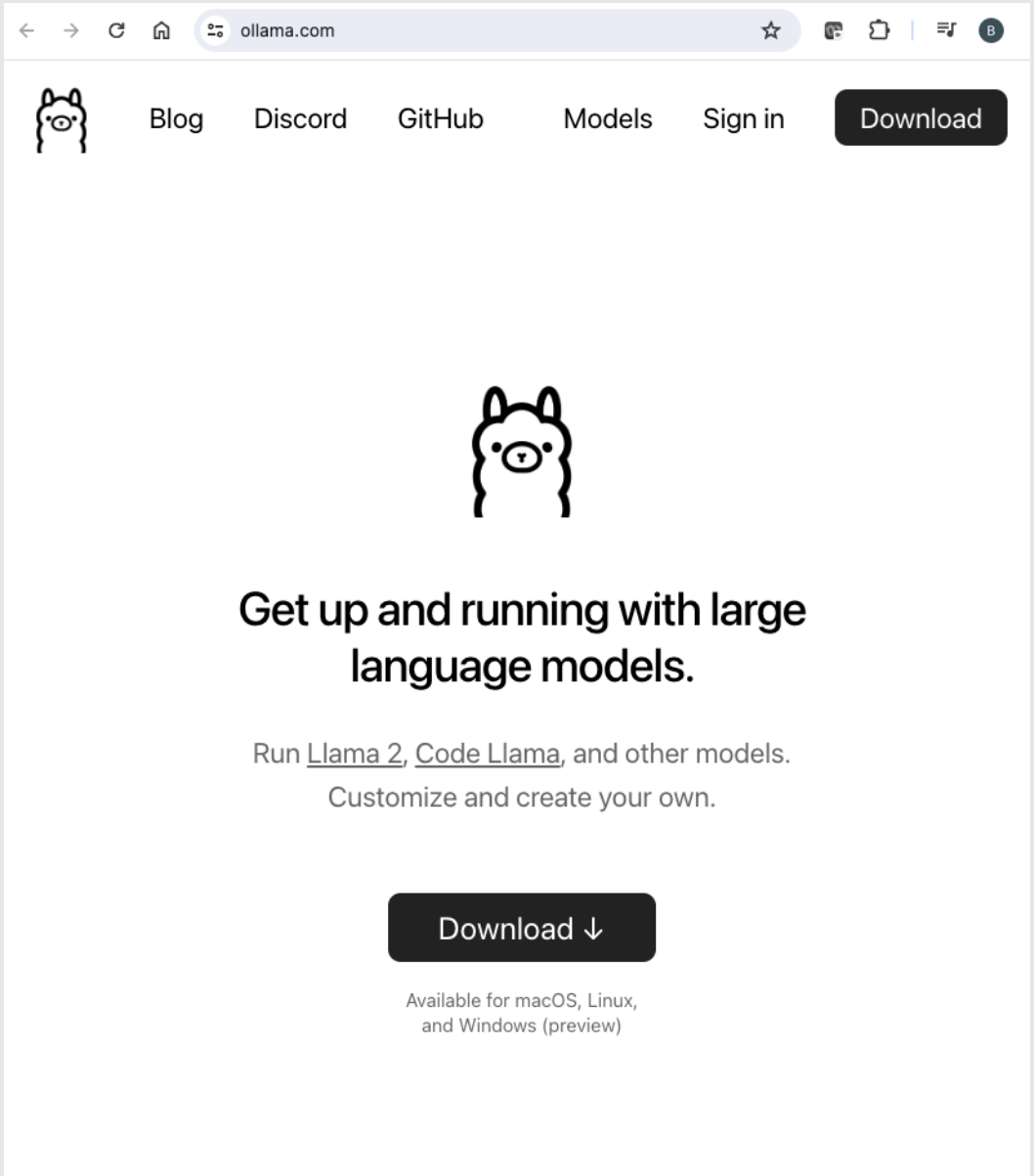
- GPT4All - https://github.com/nomic-ai/gpt4all
- LM Studio - https://lmstudio.ai
- Jan AI - https://jan.ai
- llama.cpp - https://github.com/ggerganov/llama.cpp
- LlamaFile - https://github.com/Mozilla-Ocho/llamafile
- Ollama - https://ollama.com/
- HuggingFace Transformers - https://huggingface.co/docs/transformers
- More!

LlamaFile
Local Executable GPT

Transformers

Ollama

- Command line tool for downloading, exploring and using LLMs on local machine
- open source
- supports most of Hugging Face's popular models
- allows uploading new ones
- Links:
  - `main site:` https://ollama.com
  - `GitHub:` https://github.com/ollama/
- Advantages
  - speeds up and simplifies
    - » model selection and download
    - » configuring endpoints
    - » integration with Python or JavaScript codebase



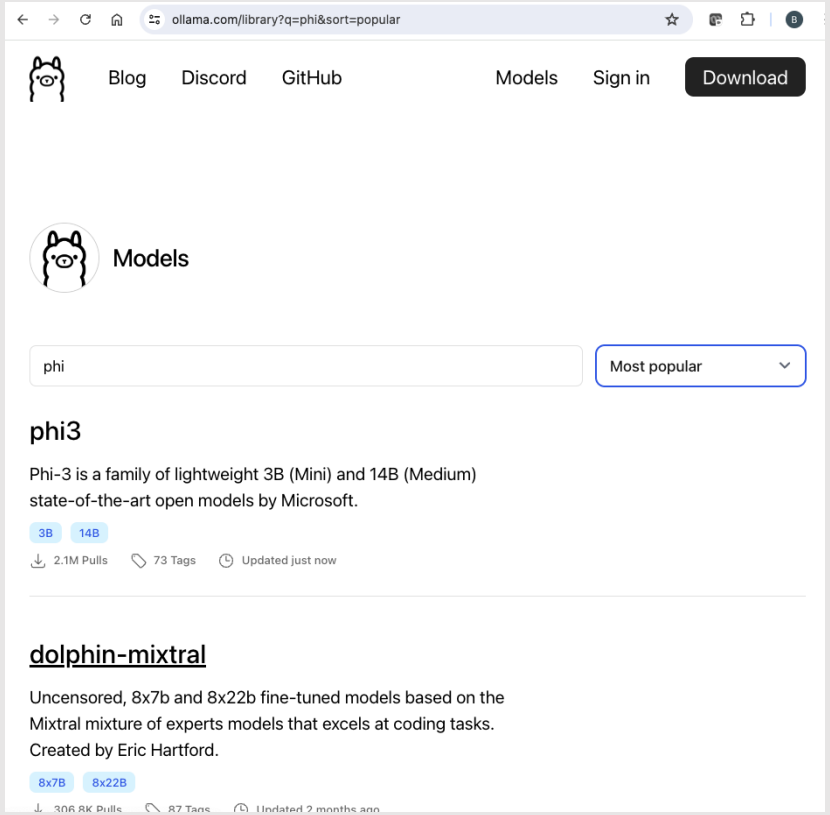
Functions of Ollama

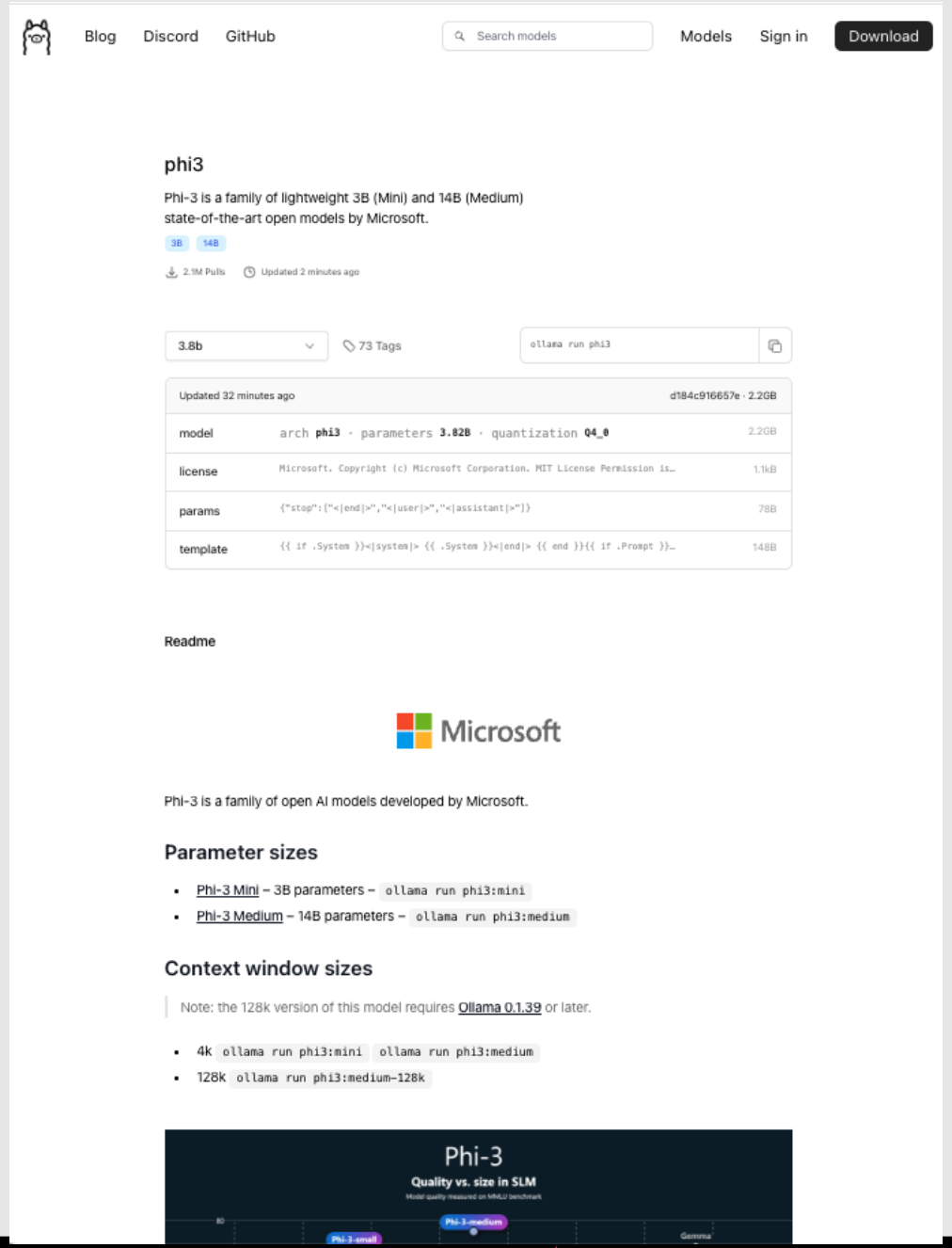
- Typically used functions
  - *serve:* starts ollama
  - *show:* shows info about a specific model
  - *run:* allows you to run a previously downloaded model
    - If model is not present, ollama will download it
  - *pull:* downloads a model, without running it once finished
  - list: prints the list of models available on the machine on the screen
  - *rm:* removes the model

```
● @techupskills → /workspaces/diy-gen-ai (main) $ ollama
Usage:
  ollama [flags]
  ollama [command]

Available Commands:
  serve       Start ollama
  create      Create a model from a Modelfile
  show        Show information for a model
  run         Run a model
  pull        Pull a model from a registry
  push        Push a model to a registry
  list        List models
  ps          List running models
  cp          Copy a model
  rm          Remove a model
  help        Help about any command

Flags:
  -h, --help      help for ollama
  -v, --version   Show version information

Use "ollama [command] --help" for more information about a command.
```

Ollama Models

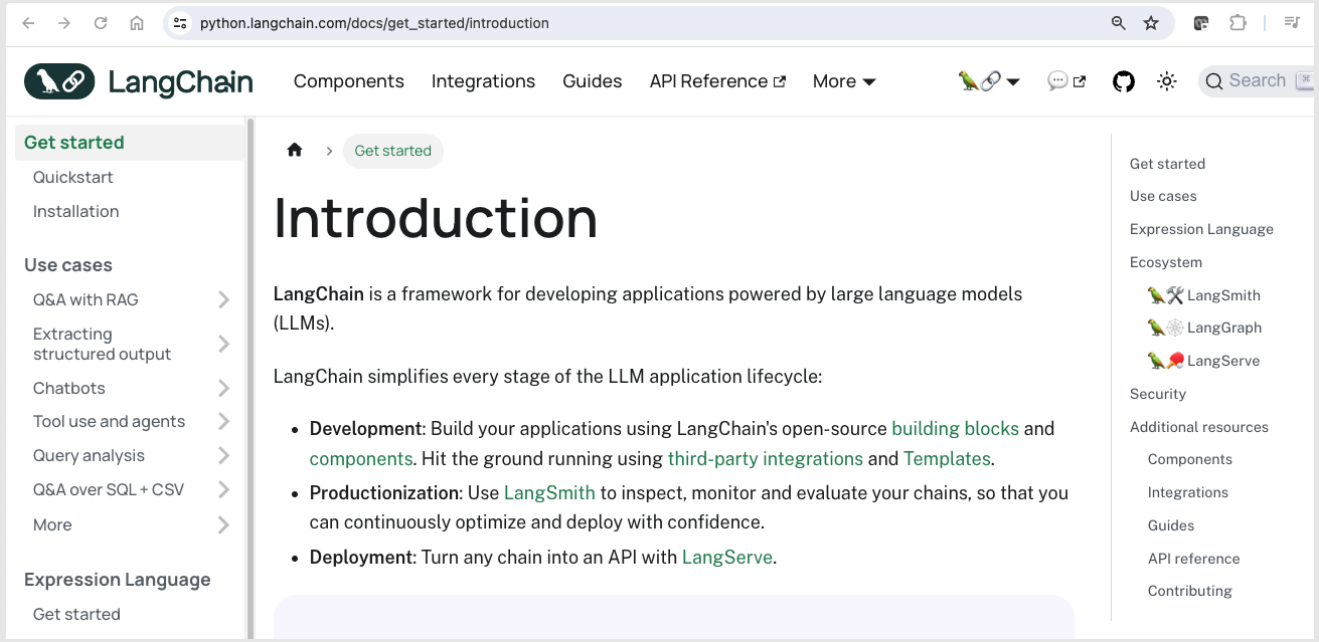
- Not all models are supported
- To find supported ones, go to https://ollama.com/library
- Click on model link to learn more

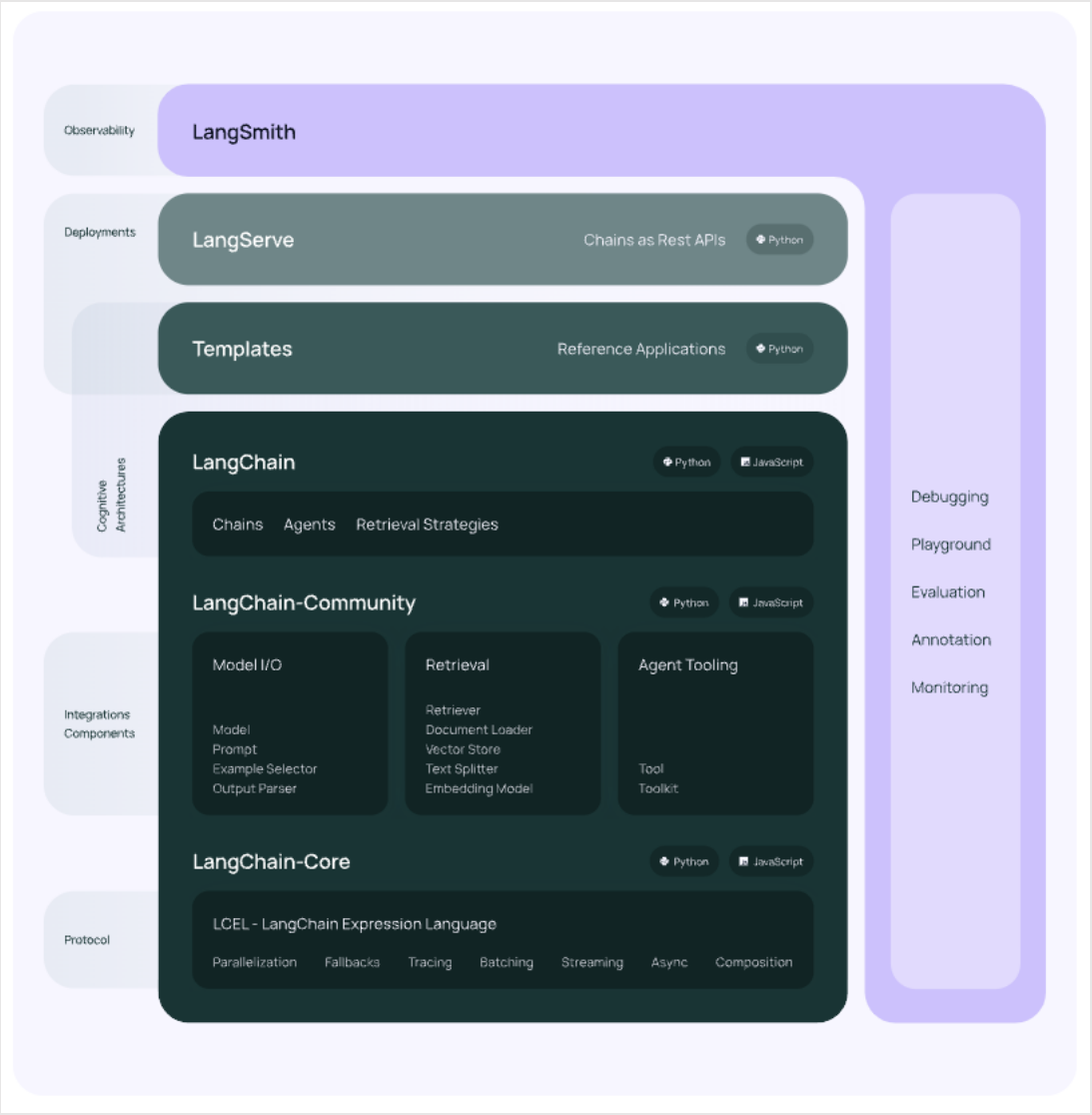




LangChain

- Framework for building applications with Large Language Models (LLMs).
- Provides middleware and abstractions to build AI app on one of its supported models
- **Provides:**
  - **Abstractions**: Common interfaces for different LLM providers
  - **Chains**: Connect multiple LLM calls together
  - **Memory**: Maintain conversation context
  - **Tools**: Integrate LLMs with external functions



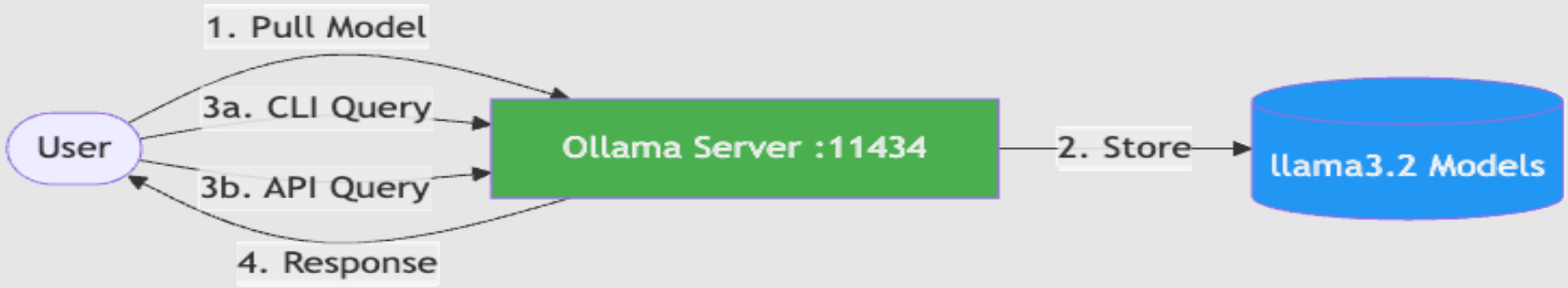


Lab 1 – Using Ollama to run models locally

Purpose: In this lab, we'll start getting familiar with Ollama, a way to run models locally.

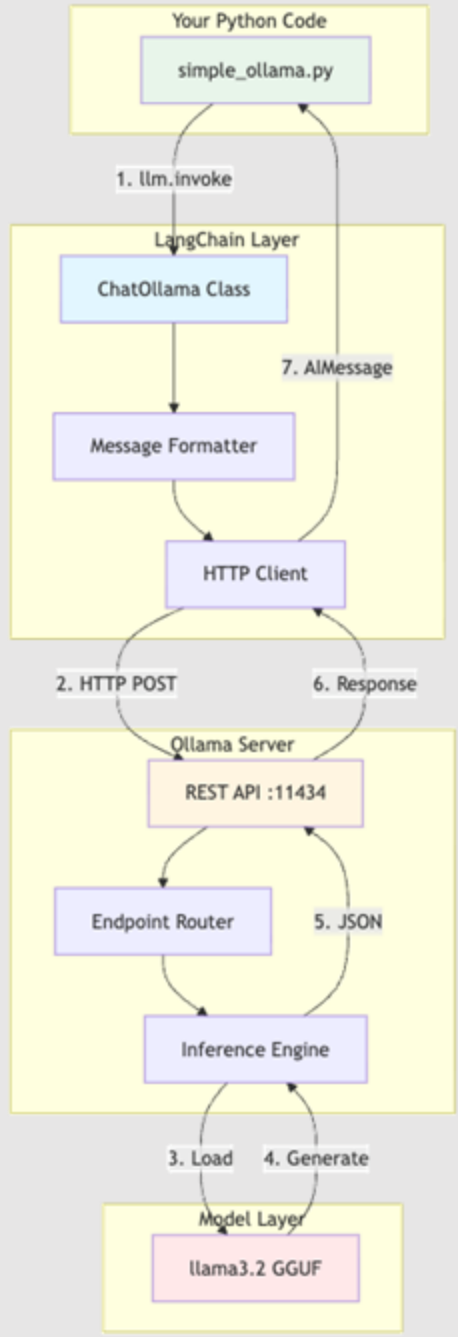
Lab 1 Recap

- Models themselves don't have access to real-time data
- Some models can be run locally for testing
- You can test out prompts with apps like Ollama via:
  - Direct inference (run)
  - API calls (curl)
  - Programmatic (python)
- Can be useful to try out prompts for AI Apps locally first to verify expected results



ChatOllama

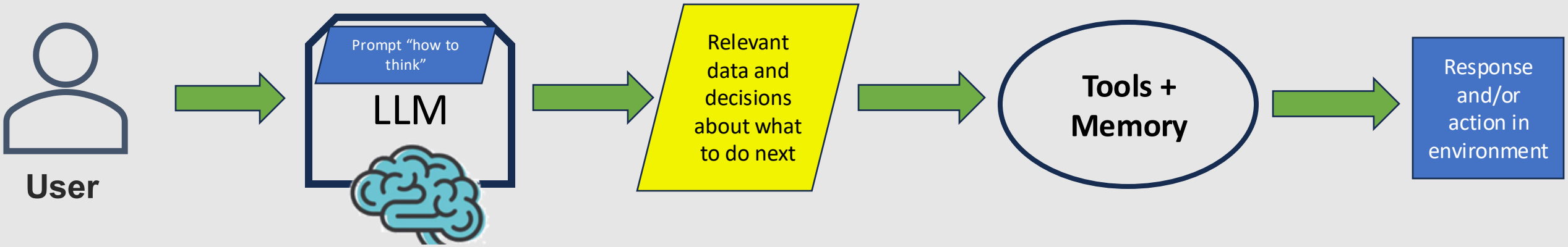
- LangChain class that wraps the Ollama HTTP API
- Provides:
  - **Simplified API**: Clean Python interface instead of raw HTTP
  - **Message Formatting**: Automatically formats prompts as chat messages
  - **Error Handling**: Manages connection issues gracefully
  - **Type Safety**: Returns structured Python objects
- Part of langchain-ollama package
- **Under the hood**, ChatOllama:
  - Converts your prompt into Ollama's expected JSON format
  - Makes HTTP POST requests to http://localhost:11434/api/chat
  - Parses JSON responses into Python objects
  - Handles streaming and non-streaming responses



Agents

What is an AI Agent?

- An AI agent is a system that:
  - **Observes** its surroundings (with sensors),
  - **Thinks** about what it sees (makes decisions), and
  - **Acts** to change or respond to the environment (with actions).
- This interaction enables the agent to achieve specific goals autonomously while continuously learning and adapting over time
- Agents use LLMs to identify key data, drive decisions, and communicate naturally



Ag

system_message=“””You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.

You have access to the following tools:

Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string

You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.

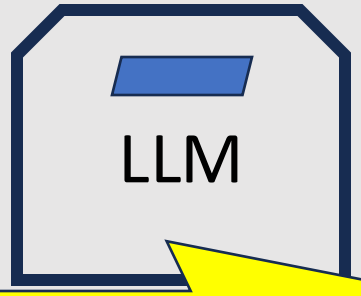
You should first reflect with “Thought: {your_thoughts}” on the current query, then (if necessary), call a tool with the proper JSON formatting “Action: {JSON_BLOB}”, or else print your final answer starting with the prefix “Final Answer:”“””

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""

User

What's the weather in Paris?



Chain of Thought – Step 1: Interpret User Query
Thought: "The user is asking about the weather in Paris. I need to extract 'Paris' as the location.
Action: Extracted location = "Paris"

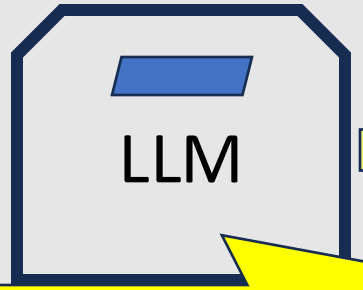
AI Agent

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""

User

What's the weather in Paris?




```
AIResponse(
    tool_calls=[{
        name:
"find_weather"
        parameters:  {
            latitude:
"48.8566",
            longitude:
"2.3522",
        },
        id: "call_tool123",
        type: "tool_invoke"
    }]
)
```

Chain of Thought – Step 2: Decide to use tool
Thought: "I need real-time data, so I will call the '**find_weather**' tool. First, I need to get the latitude and longitude for the tool call.

Weather Search Tool

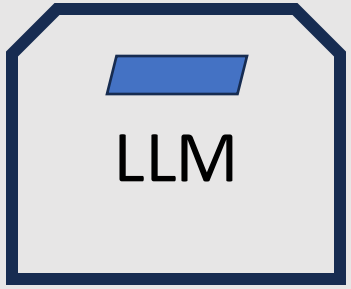
AI Agent

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""

User

What's the weather in Paris?




```
AIResponse(
    tool_calls=[{
        name: "find_weather"
        parameters: {
            latitude: "48.8566",
            longitude: "2.3522",
        },
        id: "call_tool123",
        type: "tool_invoke"
    }]
)
```

Agent parses LLM output identifies JSON tool call, parses it, forms it into actual tool call

```
{
    name: "find_weather"
    parameters: {
        latitude: "48.8566",
        longitude: "2.3522",
    },
    id: "call_tool123",
    type: "tool_invoke"
}
```

Weather Search Tool

AI Agent

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""

User

What's the weather in Paris?

LLM

```
AIResponse(
    tool_calls=[{
        name:
"find_weather"
        parameters:  {
            latitude:
"48.8566",
            longitude:
"2.3522",
        },
        id: "call_tool123"
    }]
)
```

Agent executes tool call

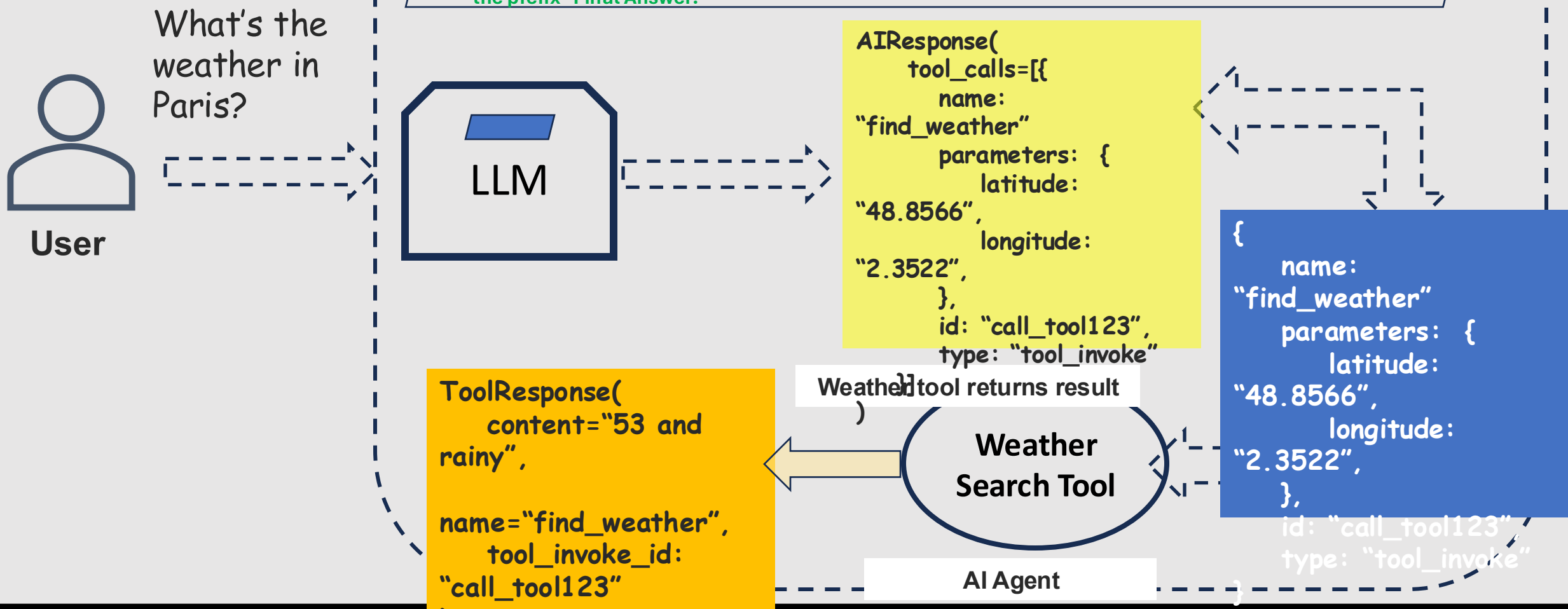
Weather Search Tool

```
{
    name:
"find_weather"
    parameters:  {
        latitude:
"48.8566",
        longitude:
"2.3522",
    },
    id: "call_tool123",
    type: "tool_invoke"
}
```

AI Agent

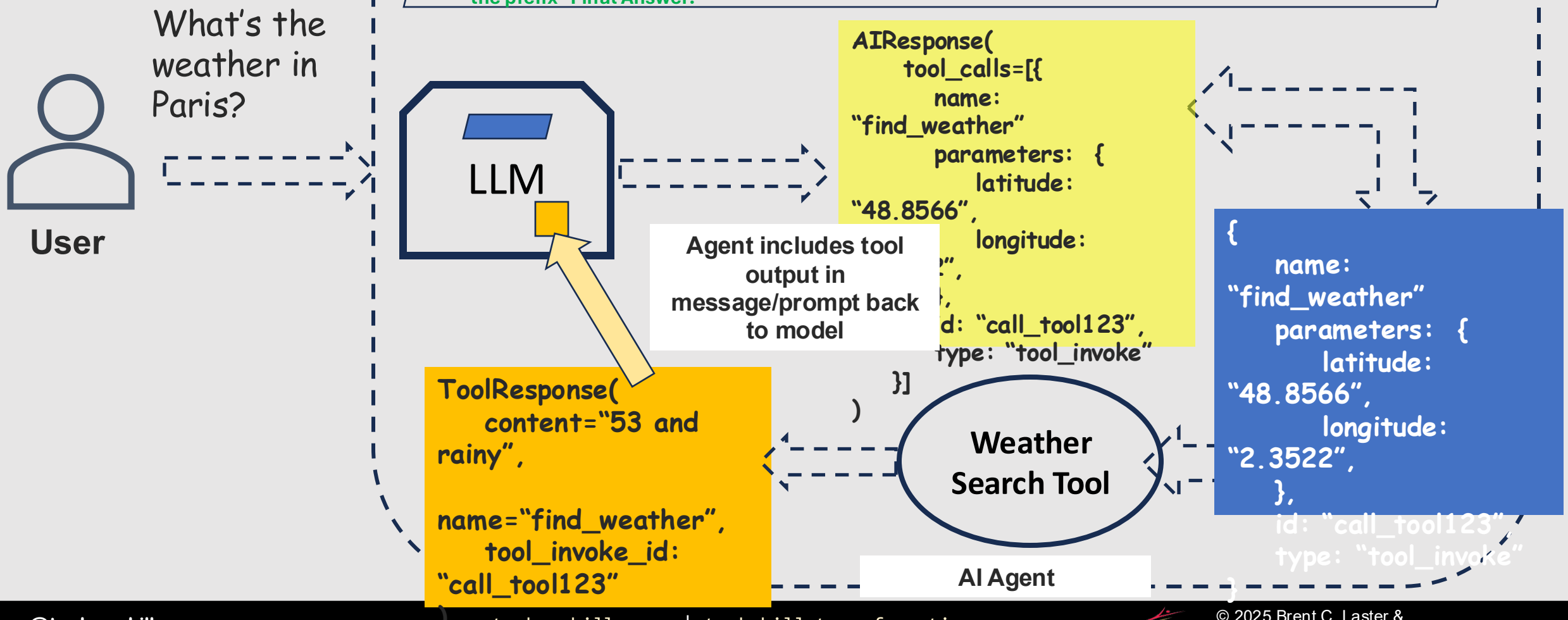
Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""



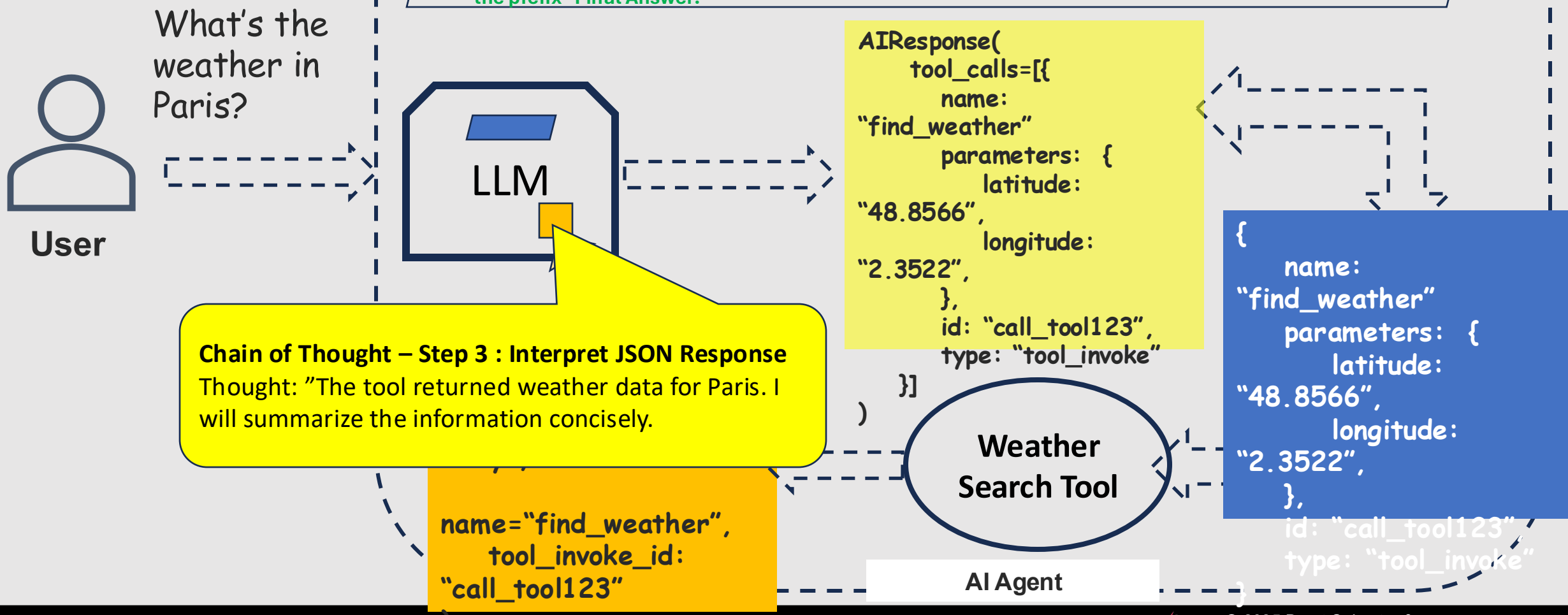
Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""



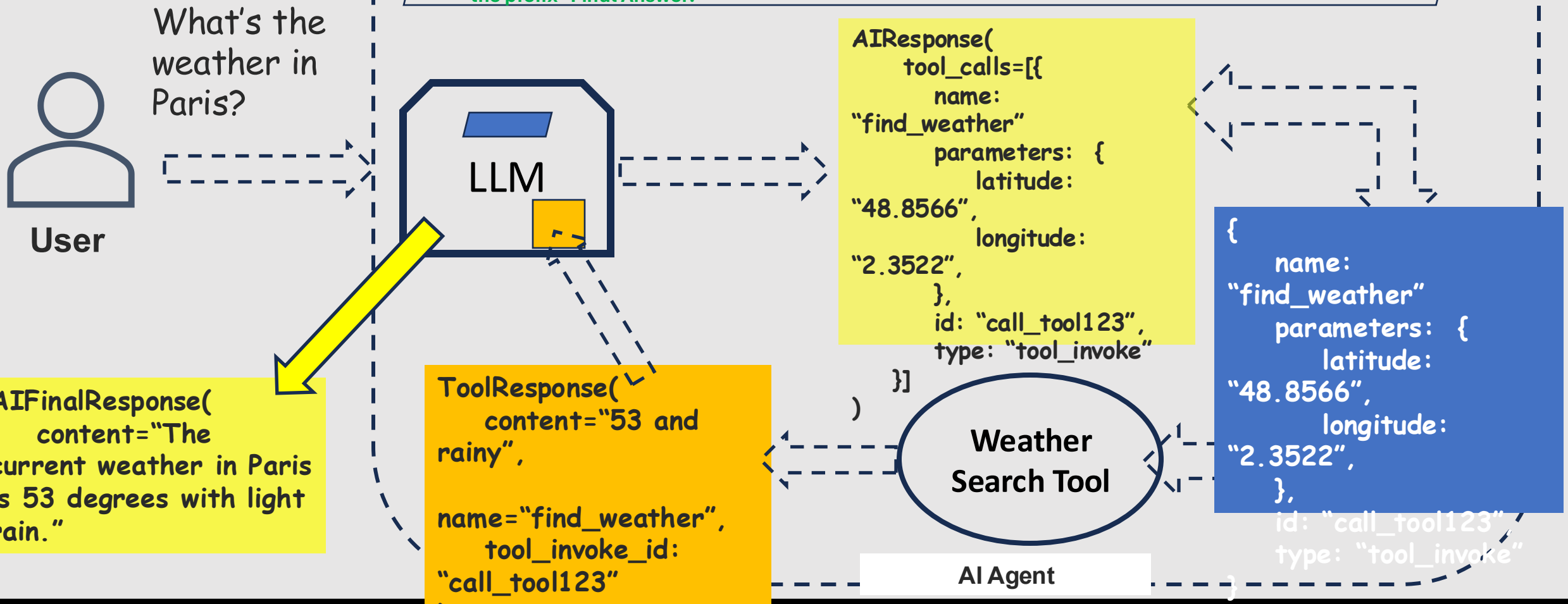
Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""



Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""



Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:"""

What's the weather in Paris?

User

LLM

```
AIResponse(
    tool_calls=[{
        name:
"find_weather"
        parameters:  {
            location: "Paris",
        },
        id: "call_tool123",
        type: "tool_invoke"
    }]
)
```

```
{
    name:
"find_weather"
    parameters:  {
        latitude:
"48.8566",
        longitude:
"2.3522",
    },
    id: "call_tool123",
    type: "tool_invoke"
}
```

Weather Search Tool

```
ToolResponse(
    content="53 and
rainy",

name="find_weather",
    tool_invoke_id:
"call_tool123"
)
```

```
AIFinalResponse(
    content="The
current weather in Paris
is 53 degrees with light
rain."
)
```

AI Agent

Architectural Features of AI Agents

Planning

- AI autonomously outlines and executes a logical series of steps for accomplishing a given objective.
- Provides the AI with a way to dynamically adapt its approach based on real-time data and feedback..
- Might employ reflection to evaluate and improve responses
- Example: A research agent plans search → summarize → generate report.

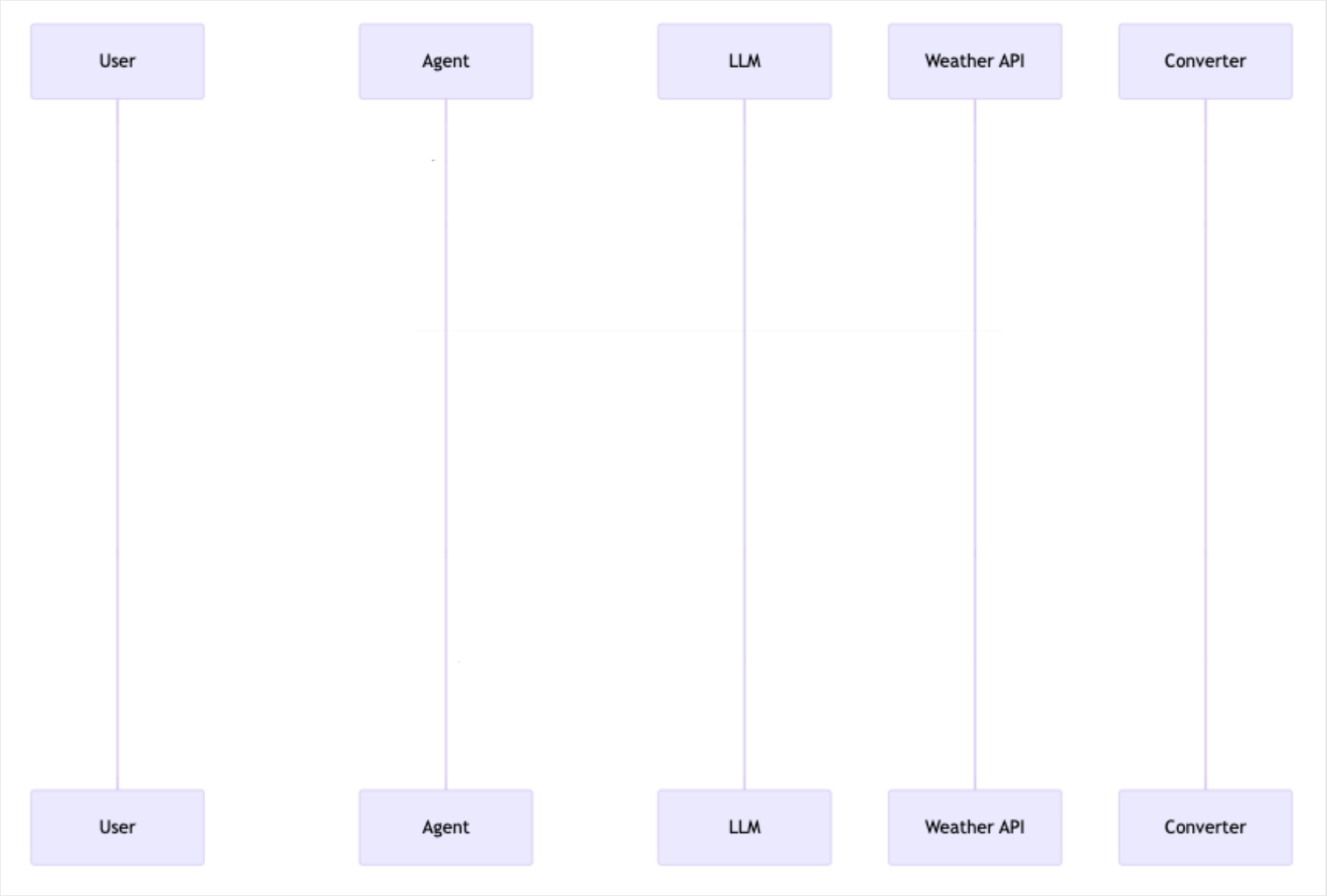
Tool Use

- AI agents interact with external APIs, databases, and functions.
- Enhances LLMs by providing access to real-world knowledge.
- Reduces hallucinations by using retrieval-augmented generation (RAG).
- Example: Calling a Python function to perform complex calculations.

Memory

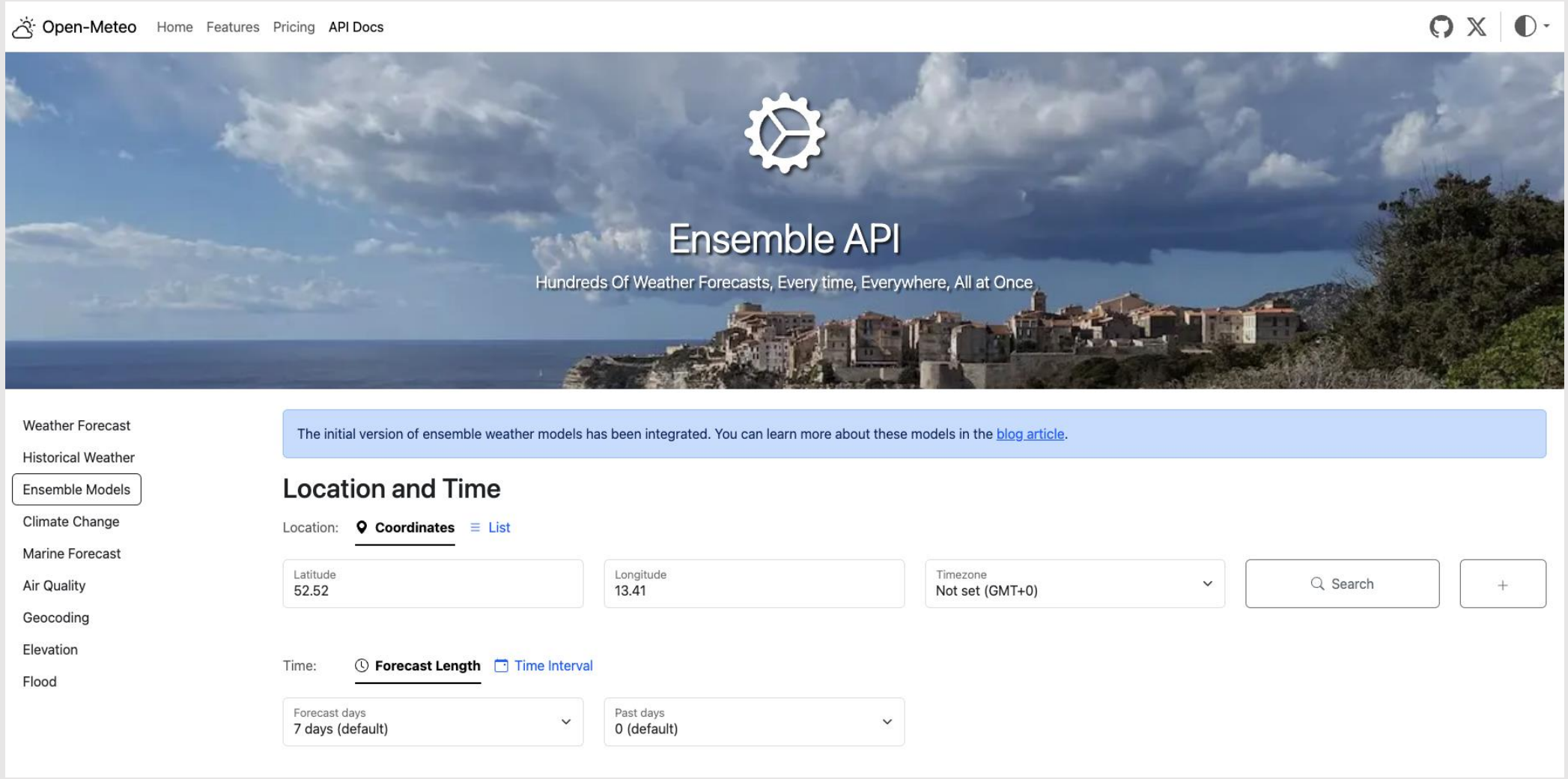
- Short-term handles tasks; long term stores knowledge and experience
- Memory ensures consistency and efficiency in multi-step decisions
- Memory recalls preferences to enhance personalization and user experience
- Example: Storing user preferences for future reference or personalized responses

Workflow for Lab 2 Agent



Free weather API

- https://open-meteo.com/en/docs/ensemble-api



Lab 2 – Creating a Simple Agent

Purpose: In this lab, we'll learn about the basics of agents and take our first pass at writing one. We'll also see how Chain of Thought occurs with LLMs

Model Context Protocol (MCP)

Problem: Different models handle function calls differently

OpenAI

```
{
 "index": 0,
 "message": {
  "role": "assistant",
  "content": null,
  "tool_calls": [
   {
    "name": "get_current_stock_price",
    "arguments": "{\n \"company\": \"AAPL\",\n \"format\": \"USD\"\n}"
   }
  ]
 },
 "finish_reason": "tool_calls"
}
```

LLaMA

```
{
 "role": "assistant",
 "content": null,
 "function_call": {
  "name": "get_current_stock_price",
  "arguments": {
   "company": "AAPL",
   "format": "USD"
  }
 }
}
```

Claude

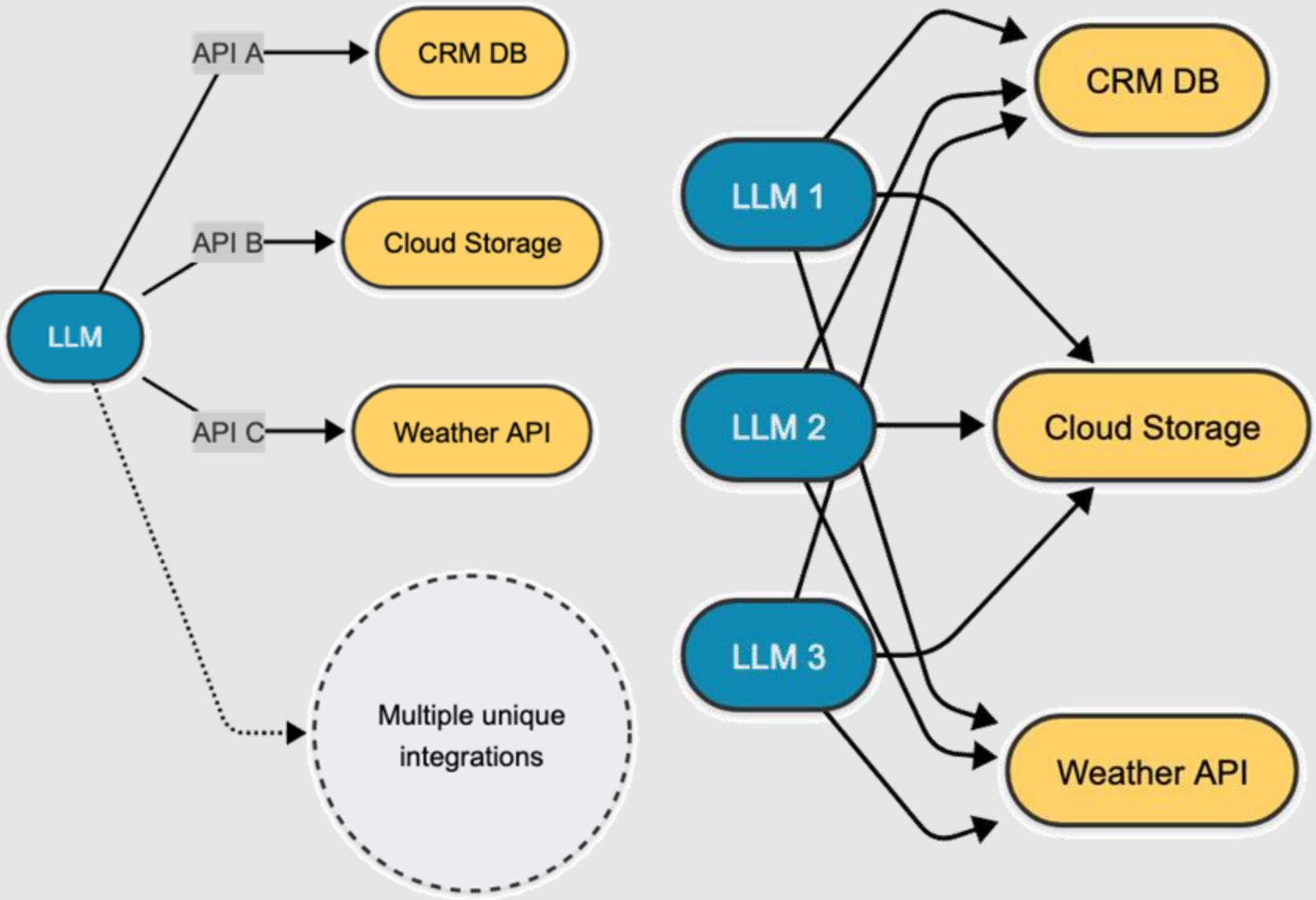
```
{
 "role": "assistant",
 "content": [
  {
   "type": "text",
   "text": "<thinking>To answer this question, I will: ...</thinking>"
  },
  {
   "type": "tool_use",
   "id": "1xqaf90qw9g0",
   "name": "get_current_stock_price",
   "input": {"company": "AAPL", "format": "USD"}
  }
 ]
}
```

Gemini

```
{
 "functionCall": {
  "name": "get_current_stock_price",
  "args": {
   "company": "AAPL",
   "format": "USD"
  }
 }
}
```

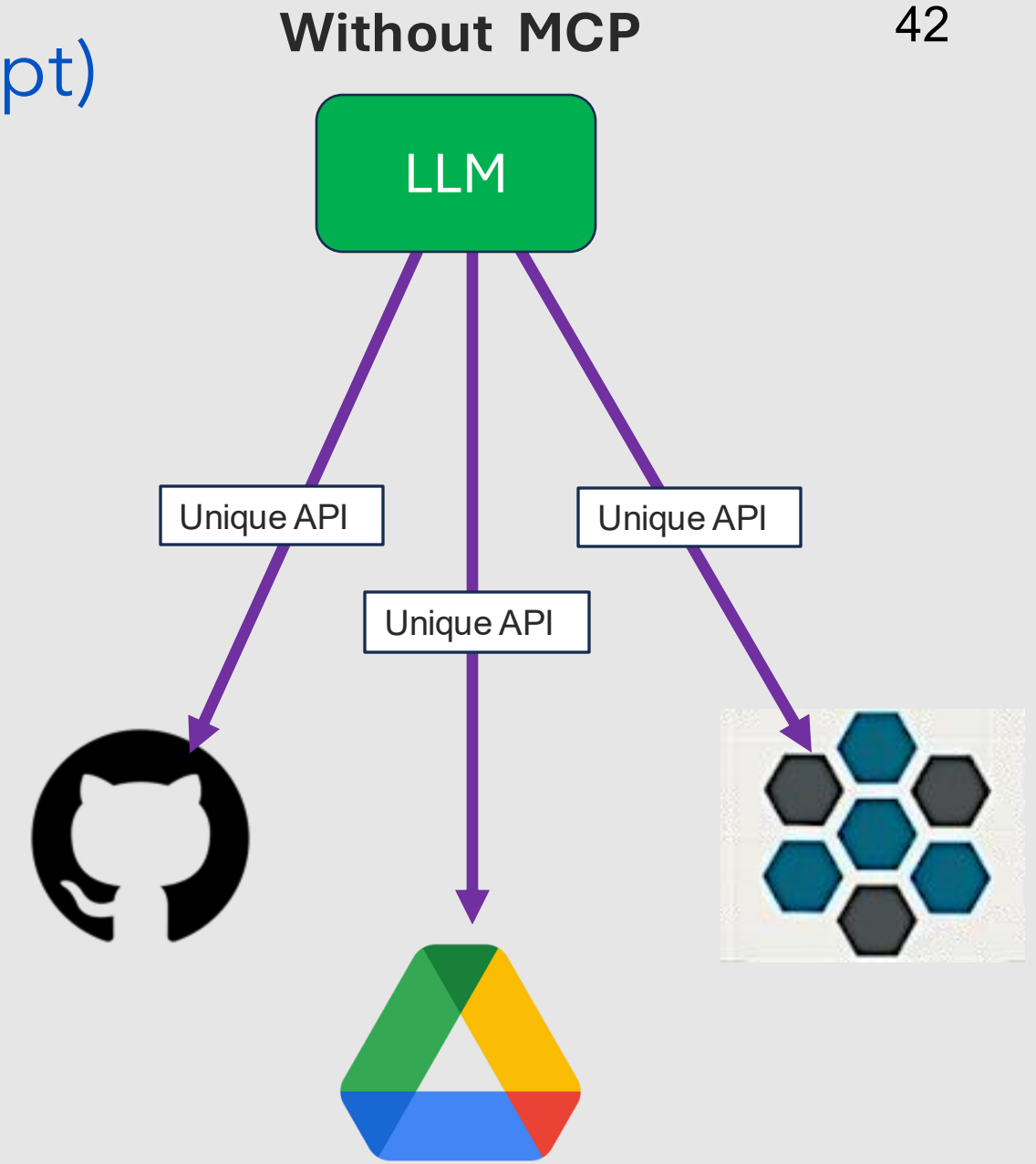
M x N Integration problem

- Many AIs × many tools → exponential integration burden
- Every AI-tool pair needs its own custom link
- Leads to high complexity, cost, and maintenance pain



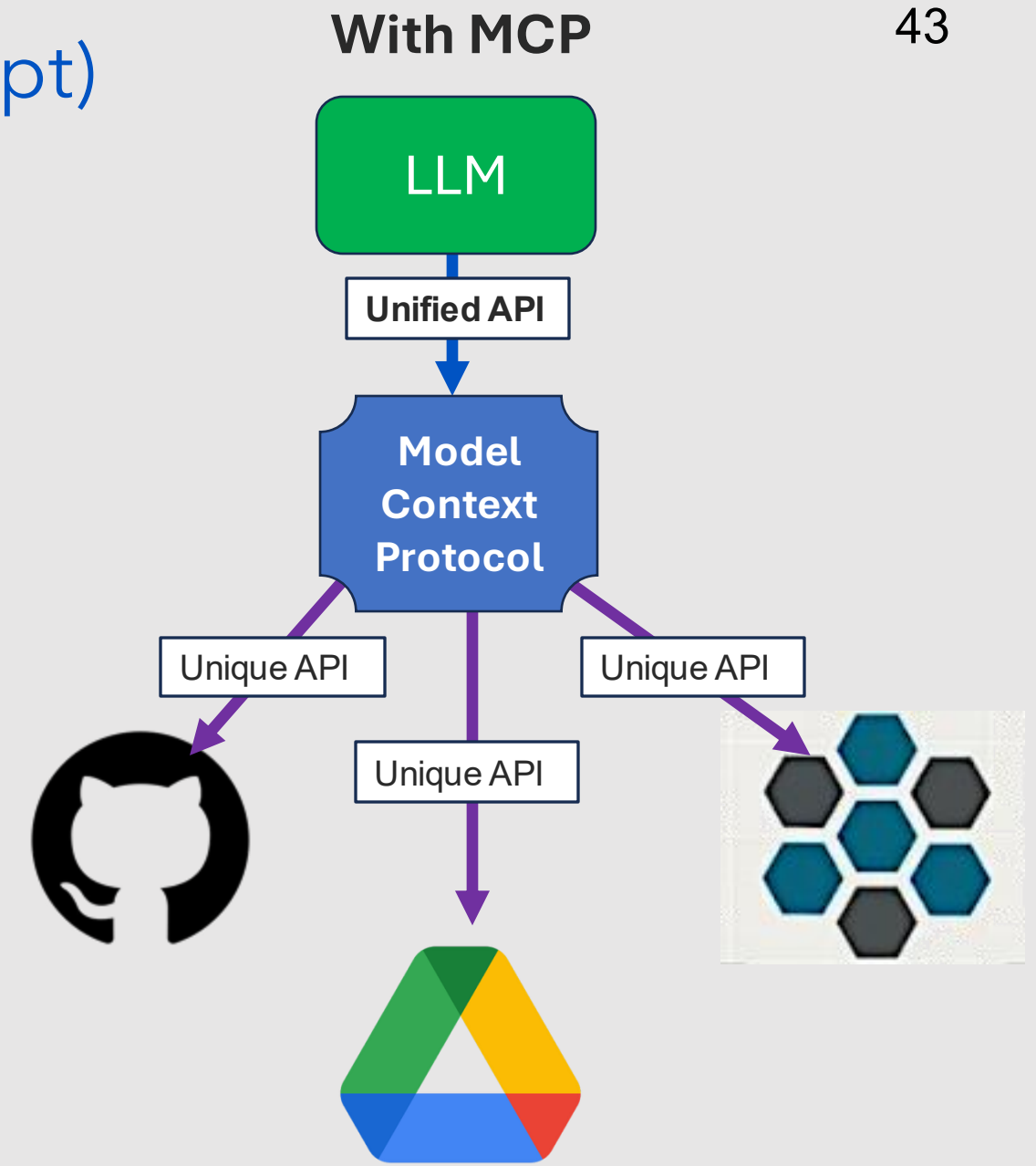
Model Context Protocol (Concept)

- MCP is an open protocol for standardizing how applications provide tools and resources to AI applicaations
- Like a universal connector for AI (sometimes called "usb-c" of AI)
- Manages
  - Tool discovery: Identifies right tool for request
  - Invocation: Executes the function call
  - Response handling : Returning results in a structured format



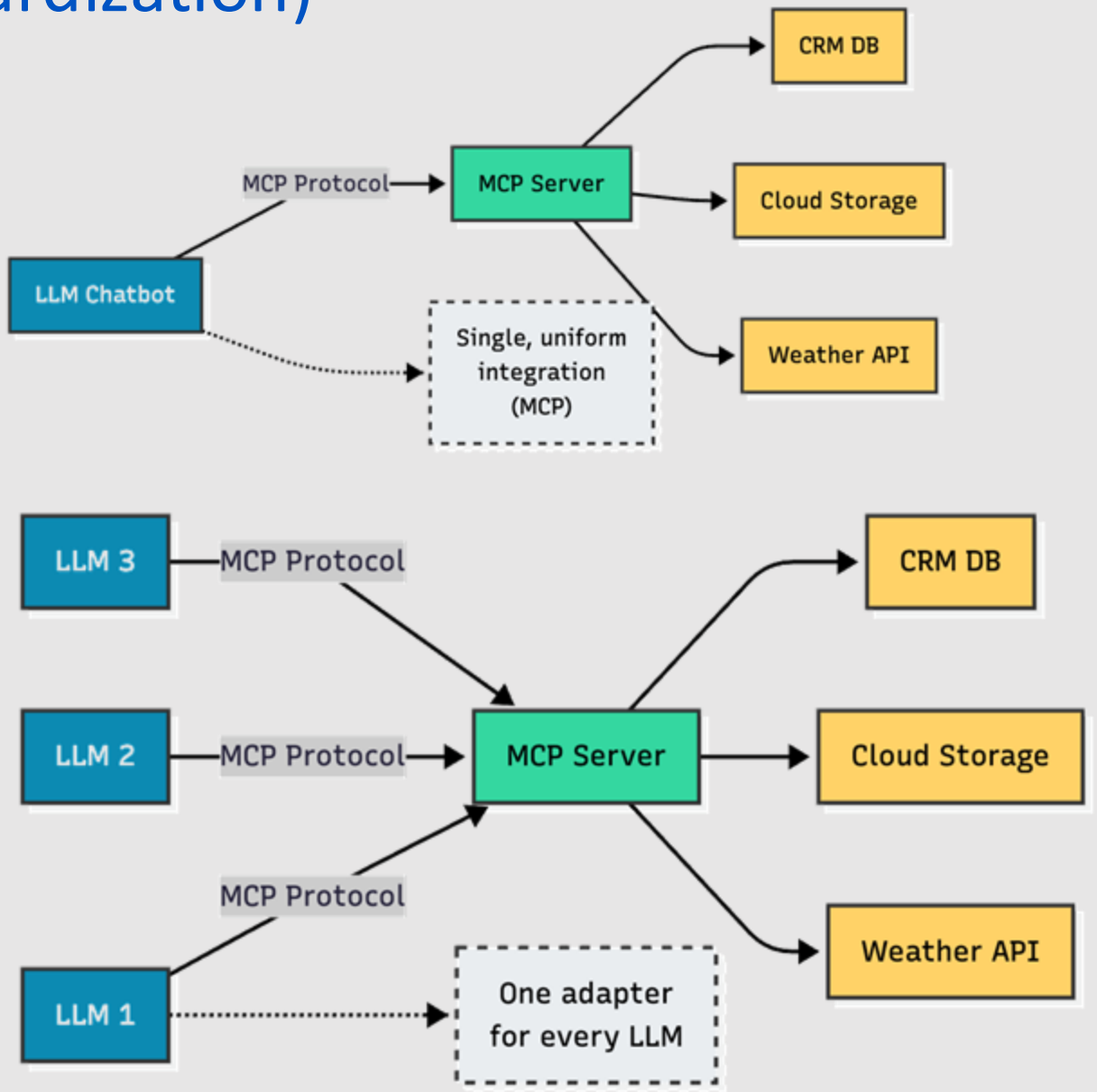
Model Context Protocol (Concept)

- MCP is an open protocol for standardizing how applications provide tools and resources to AI applicaations
- Like a universal connector for AI (sometimes called "usb-c" of AI)
- Manages
  - Tool discovery: Identifies right tool for request
  - Invocation: Executes the function call
  - Response handling : Returning results in a structured format



How does MCP Help? (Standardization)

- MCP introduces one shared interface for everyone
- Each AI app implements the MCP client side once
- Each tool or data source implements the server side once
- AI apps can easily "plug in" new capabilities through MCP servers
- Integration count falls from **M × N** to **M + N**
- Outcome: faster lower cost, easier scaling

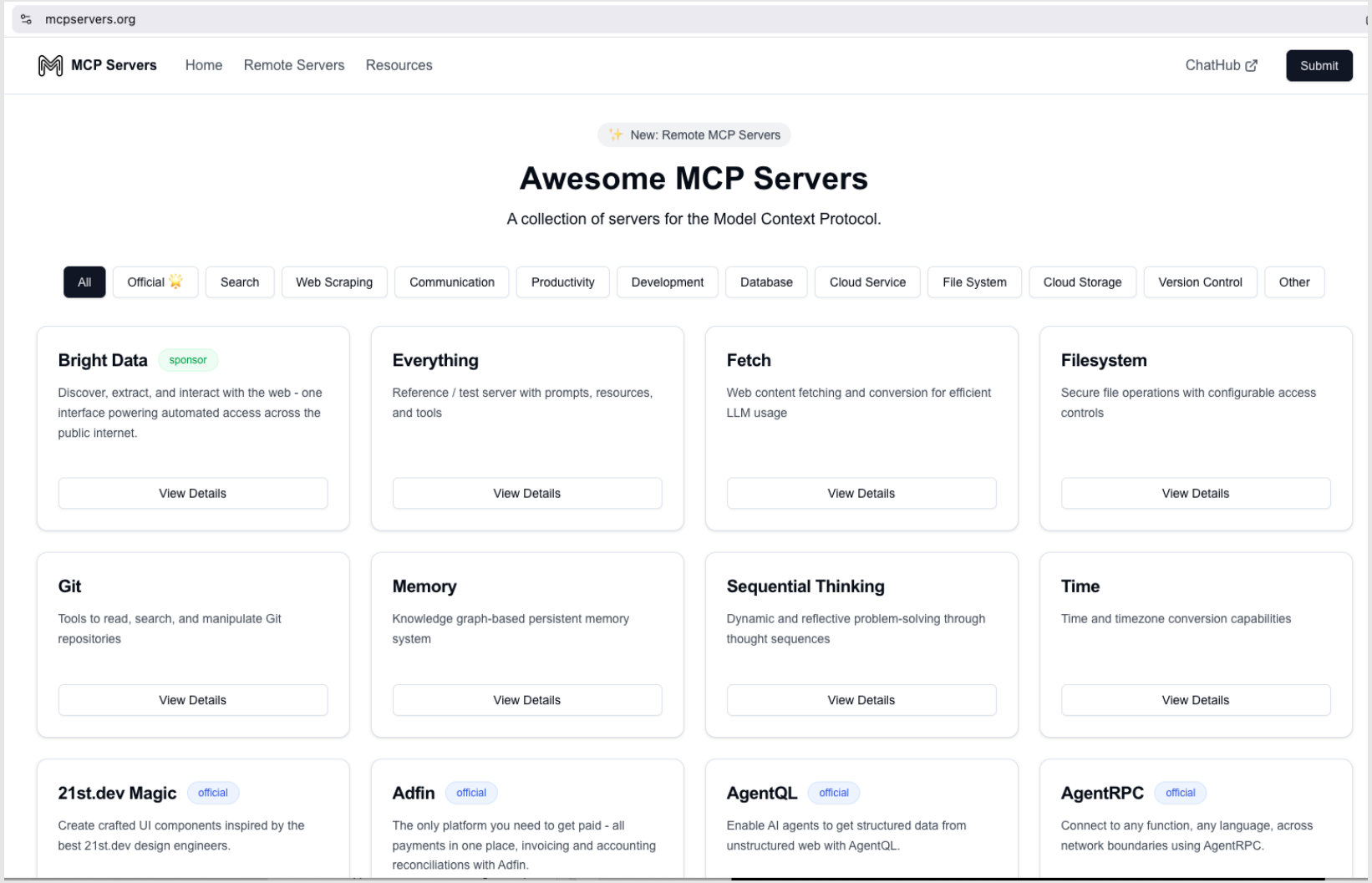


Before MCP: each agent must know each service's custom API.

With MCP: agents speak a single protocol; the MCP server handles all downstream specifics.

How does MCP Help? (Discoverability)

- Makes tools for AI apps discoverable
- Standardization means any app using an MCP client can use the tool
- Discovery features of MCP means its easy to know how to use the tool
  - MCP providers can "advertise"
- Opens up much larger ecosystem for AI apps to leverage external tools



How does it work?

Overview of MCP Architecture

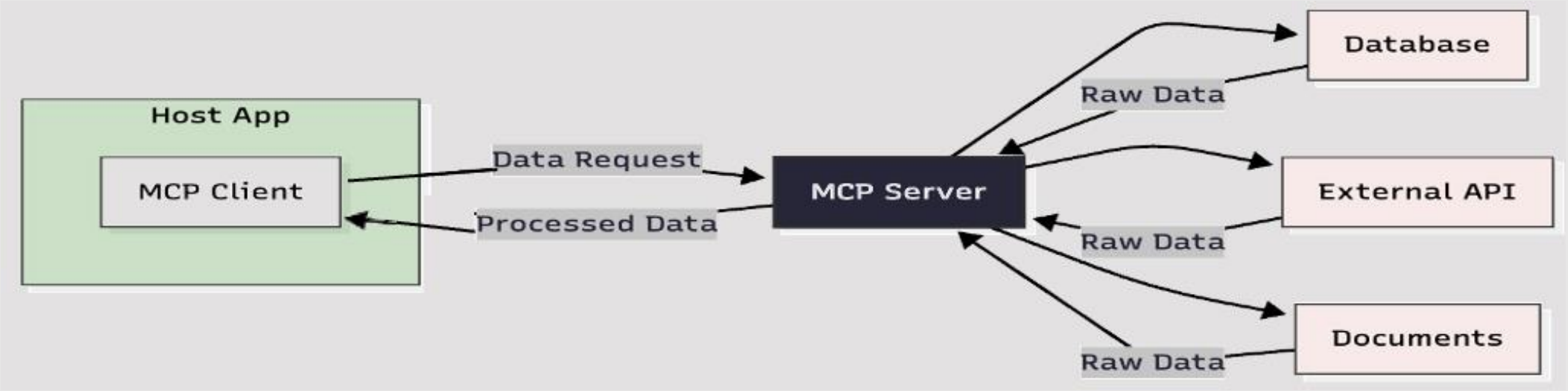
- MCP uses a **client-server architecture** for structured AI-tool communication.
- Enables systems to access external tools through a common protocol.

Core Components

MCP hosts - apps (Claude Desktop, Windsurf, Cursor, VS Code, custom apps) that want to access data via MCP

MCP clients – implement client protocol and maintain 1:1 connections with MCP servers
act as communications bridge

MCP servers – lightweight programs to expose specific capabilities (calling an APi, reading data, etc.) via the server protocol



MCP Server Spec Endpoints for Server and Client

Server

- **GET /discover**
  - Returns JSON catalog of:
    - » **tools**: name, description, I/O schema, version
    - » **resources**: name, schema, version
    - » **prompts**: name, template, version
    - » **samplings**: name, settings
- **POST /invoke**
  - Body: { tool: string, args: object }
  - → Executes the tool, returns { result: … }
- **GET or POST /resources/{name}**
  - Fetches data for that resource given any parameters
- **POST /prompts/{name}**
  - Renders the named prompt template with provided inputs

- All endpoints must expose consistent schema metadata so clients can validate inputs/outputs and generate UI or code

Client

- **discover()**
  - Fetch full catalog of Tools, Resources, Prompts & Samplings
  - Under the hood:
    - » GET /discover → JSON with lists of each capability
- **invoke(tool_name, args)**
  - Call a named Tool with its input arguments
  - Under the hood:
    - » POST /invoke
    - » { "tool": "tool_name", "args": { … } } ->JSON result
- **fetch(resource_name, params)**
  - Retrieve a Resource by name (loads data into context)
  - Under the hood:
    - » GET /resources/{resource_name}?… or POST /resources/{resource_name} with params
- **prompt(prompt_name, inputs)**
  - Expand a Prompt template server-side with given inputs
  - Under the hood:
    - » POST /prompts/{prompt_name} → filled-in text

MCP Framework Comparisons

- **MCP SDK:** The foundation for all MCP development; use for production, standardized projects.
- **FastMCP 1.0:** Historical, now integrated into the MCP Python SDK; not used as a standalone today.
- **FastMCP 2.0:** The modern, feature-rich toolkit for advanced MCP workflows, server composition, and client integration.
- **Other Frameworks:** The Java SDK is official; other frameworks may exist but are less common.

FrameworkCli	Role & Features	Integrations	Status	Client/Server	Notes
MCP SDK	• Official, language-specific SDKs (Python, Java, etc.)• Full MCP spec: tools, resources, prompts, transports	• Any MCP-compliant tool or transport (STDIO, SSE, Streamable)	Current, maintained	Both	Standard for production; includes FastMCP 1.0 in Python SDK
FastMCP 1.0	• Early Pythonic decorators for tools & resources• Minimal boilerplate	• Now part of the MCP Python SDK	Integrated, legacy support only	Server-only	Legacy API—avoid as standalone
FastMCP 2.0	• Standalone, feature-rich toolkit• Server composition, proxying, auth, testing• OpenAPI/FastAPI support• Client-side LLM sampling	• Compatible with MCP SDK, Prefect, FastAPI, REST, other SDKs	Current, actively maintained	Server-only (compatible with client written with SDK)	Recommended for advanced & production MCP workflows; rewritten in a language-agnostic style for TypeScript and future Java/C# ports
Other Frameworks	• Java SDK (official)• Third-party libs in other languages	• Interoperate via the MCP protocol	Varies (Java SDK maintained)	Both	Use official Java SDK for JVM; others for niche needs

Overview of MCP Architecture

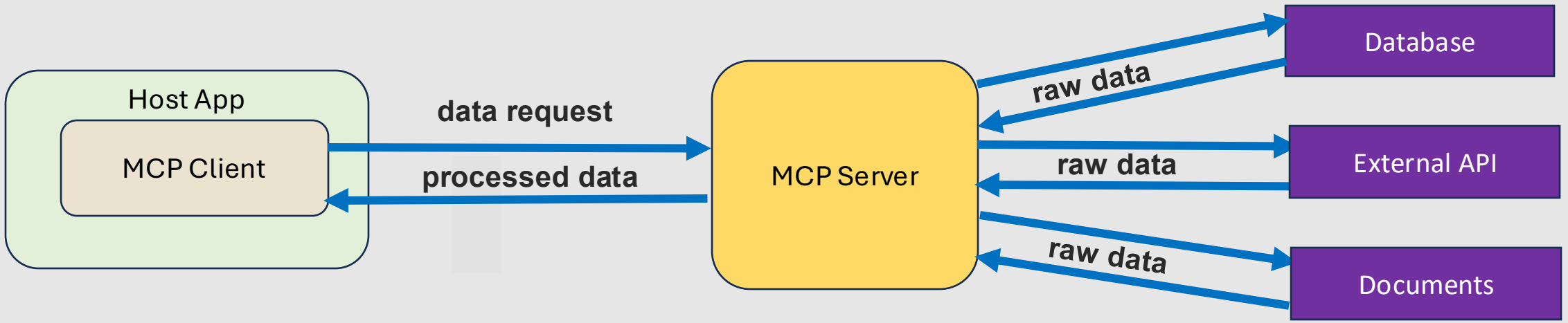
- MCP uses a **client-server architecture** for structured AI-tool communication.
- Enables systems to access external tools through a common protocol.

Core Components

MCP hosts - apps (Claude Desktop, Windsurf, Cursor, VS Code, custom apps) that want to access data via MCP

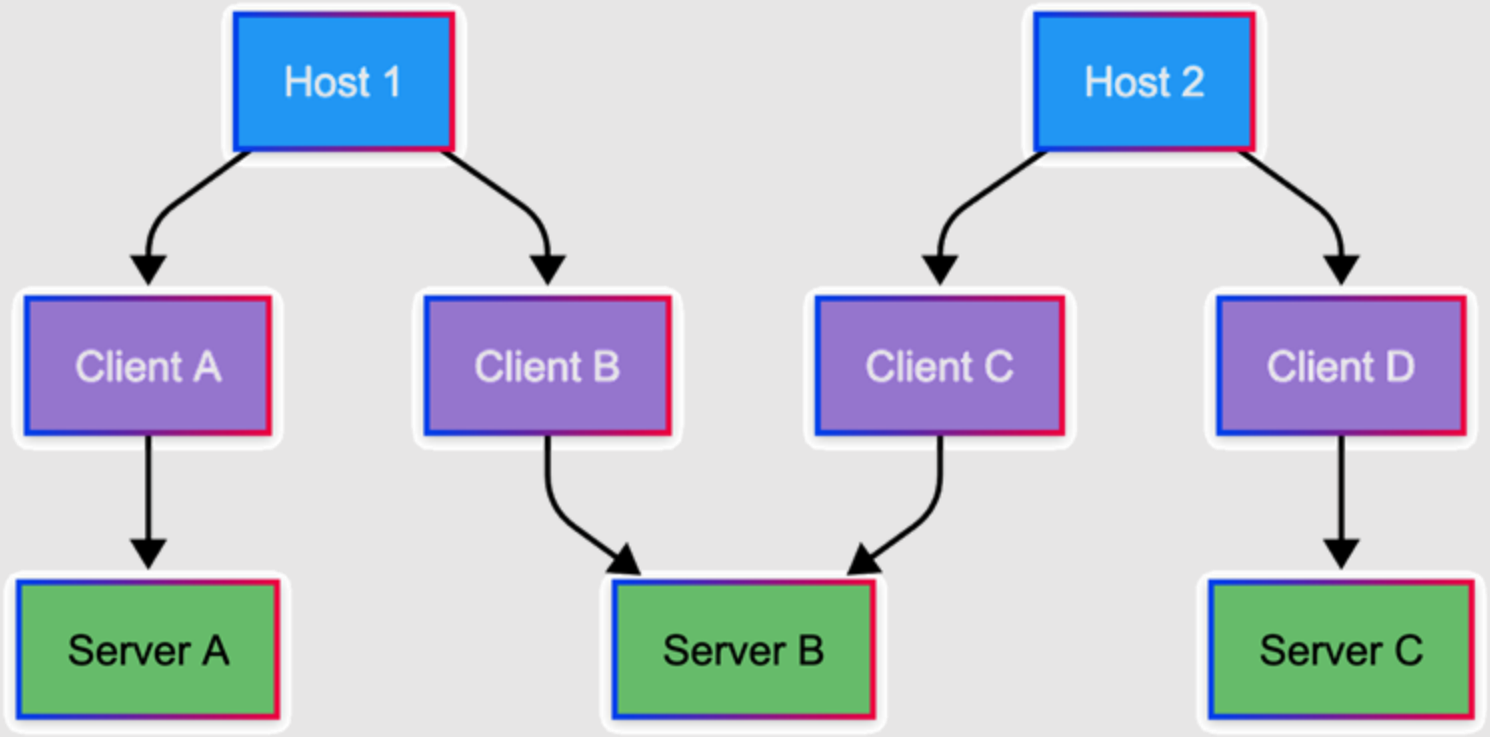
MCP clients – implement client protocol and maintain 1:1 connections with MCP servers
act as communications bridge

MCP servers – lightweight programs to expose specific capabilities (calling an APi, reading data, etc.) via the server protocol



Modularity and Scalability

- Host can connect to **multiple Servers** via multiple Clients.
- Servers can be added **without changing Hosts**.
- Supports:
- Scalable and reusable tooling
- Reduced integration complexity (M+N vs M×N)



Lab 3 – Exploring MCP

Purpose: In this lab, we'll see how MCP can be used to standardize an agent's interaction with tools.

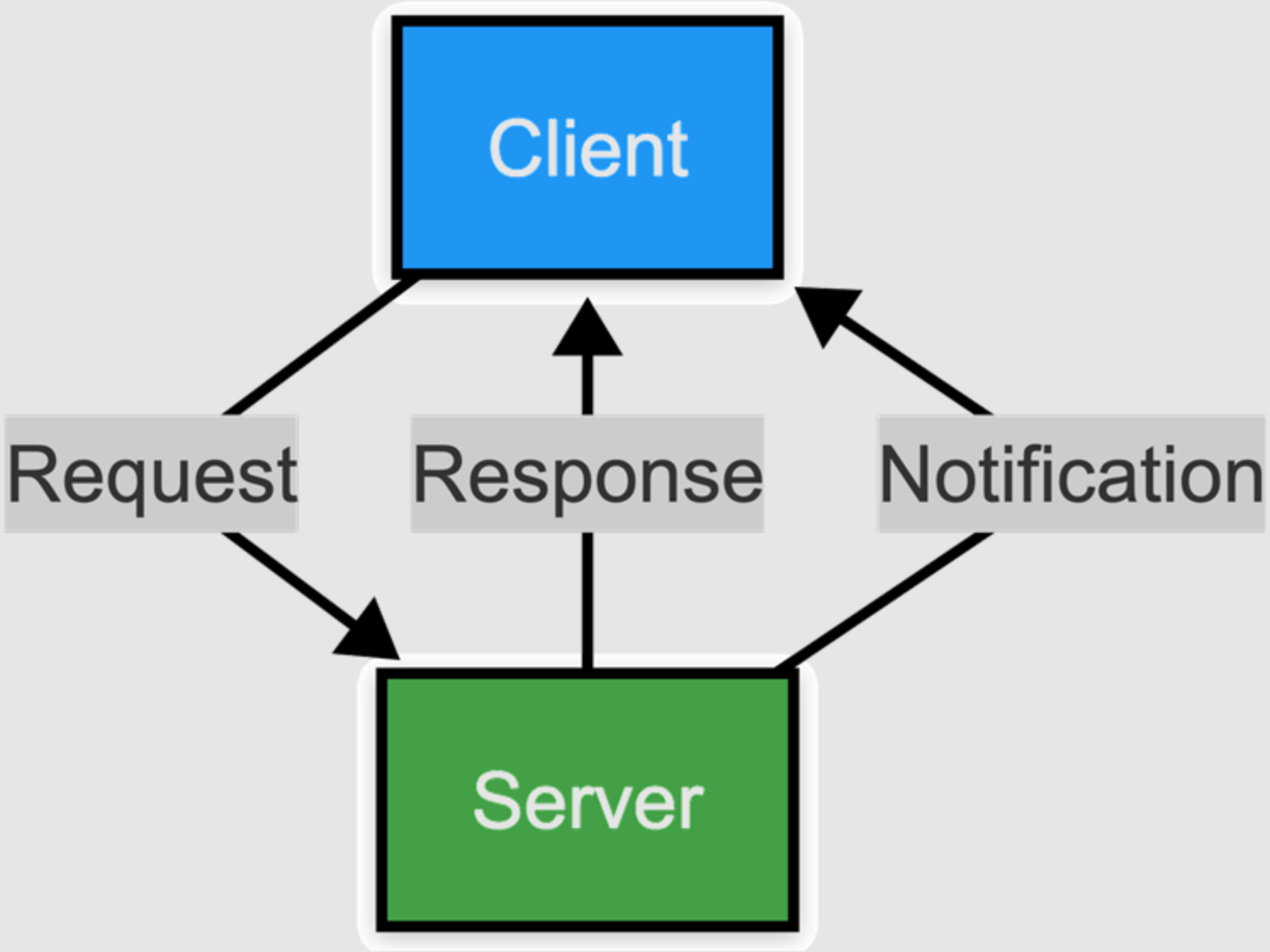
MCP Tool

- Defined by annotating a Python (or other-SDK) function with @mcp.tool
- Registers a **name**, **description**, typed **inputs** and **outputs**
- Exposed over the MCP transport for clients to invoke via JSON-RPC
- Server dispatches calls to the underlying function (synchronously or streamed)
- Versioning through the MCP protocol ensures clients & servers stay in sync

```
# 1) TOOL: simple "add" operation
@mcp.tool(name="add", description="Add two numbers")
def add(a: int, b: int) -> int:
    return a + b
```

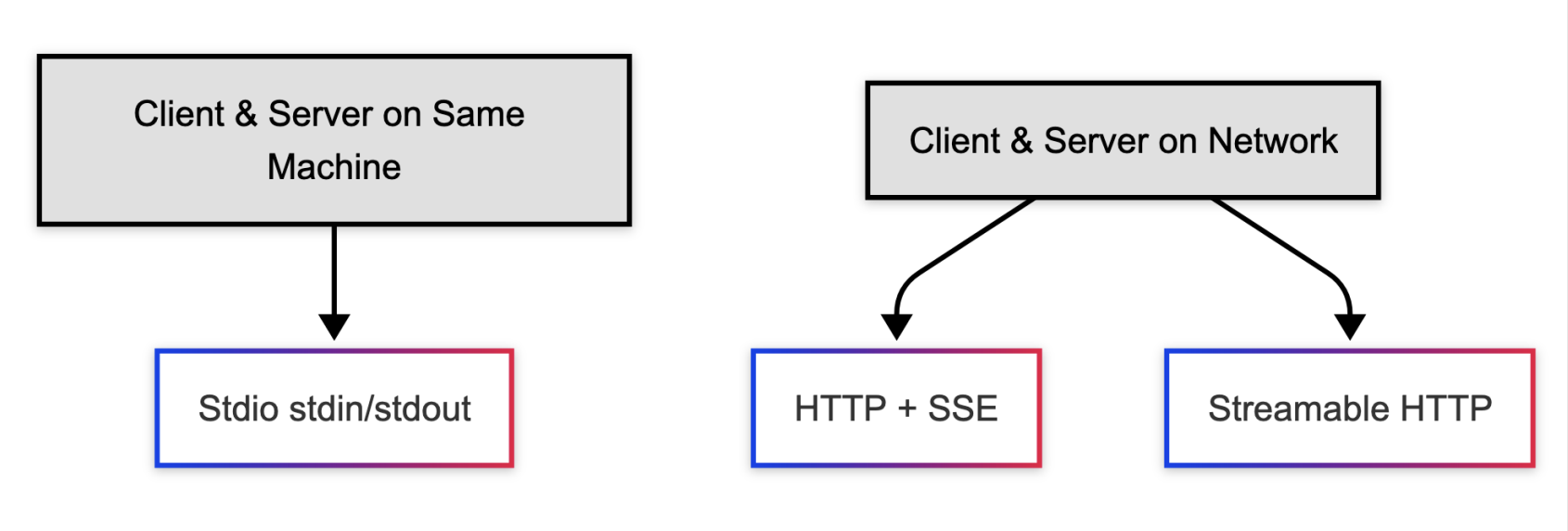
MCP Communication Protocol Overview

- Defines standard message exchange between Clients and Servers
- Based on JSON-RPC 2.0
- Supports Requests, Responses, and Notifications
- Ensures consistency and interoperability



Transport Mechanisms in MCP

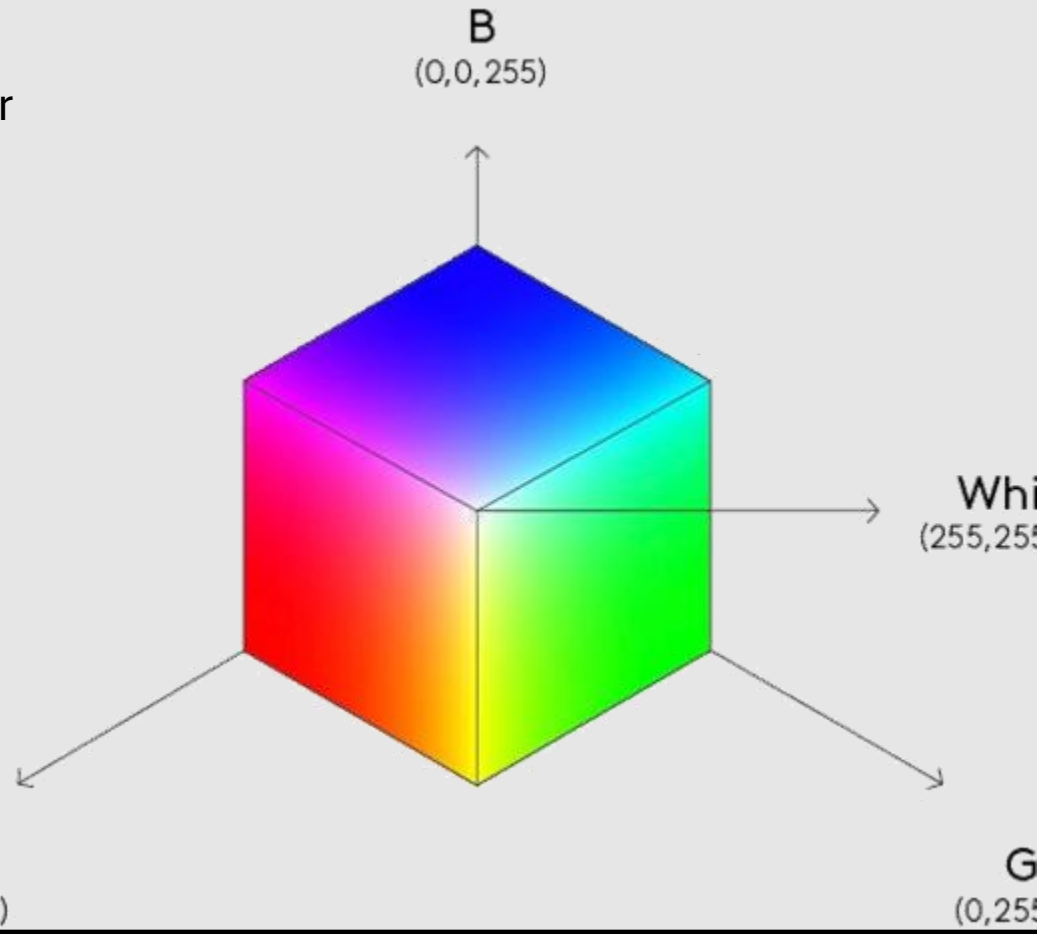
- Stdio: Local process communication via stdin/stdout
- HTTP + SSE: Remote communication with Server-Sent Events
- Streamable HTTP: Dynamic streaming for serverless support



Background for RAG: Embeddings and Vectors

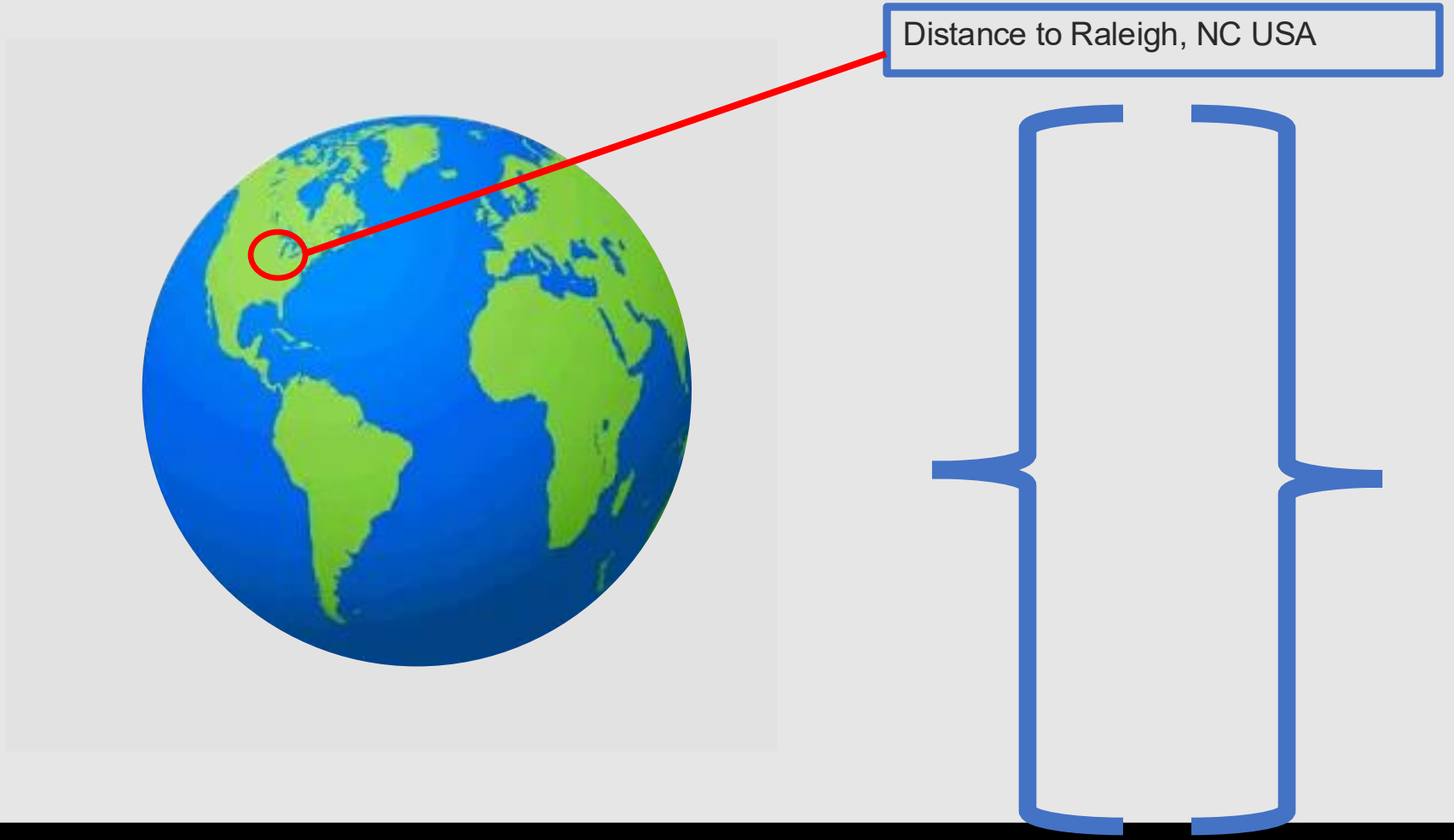
Embeddings

- Embeddings represent text as sets of numeric data - tensors (lots of dimensions)
- Each dimension stores some info about the text's meaning, context, or syntactical aspects
- Words or sentences with similar meanings are stored closer together in the vector space
  - If two pieces of text are similar syntactically, they will have similar embeddings (smaller distance between their vectors)
- During training, models learn to place text with similar meanings closer together in the embedding space
- Common pre-trained models used for generating embeddings include BERT and variants (RoBERTa, DistilBERT)
- Once you have embeddings, you can use them for NLP tasks like semantic search, text classification, sentiment analysis



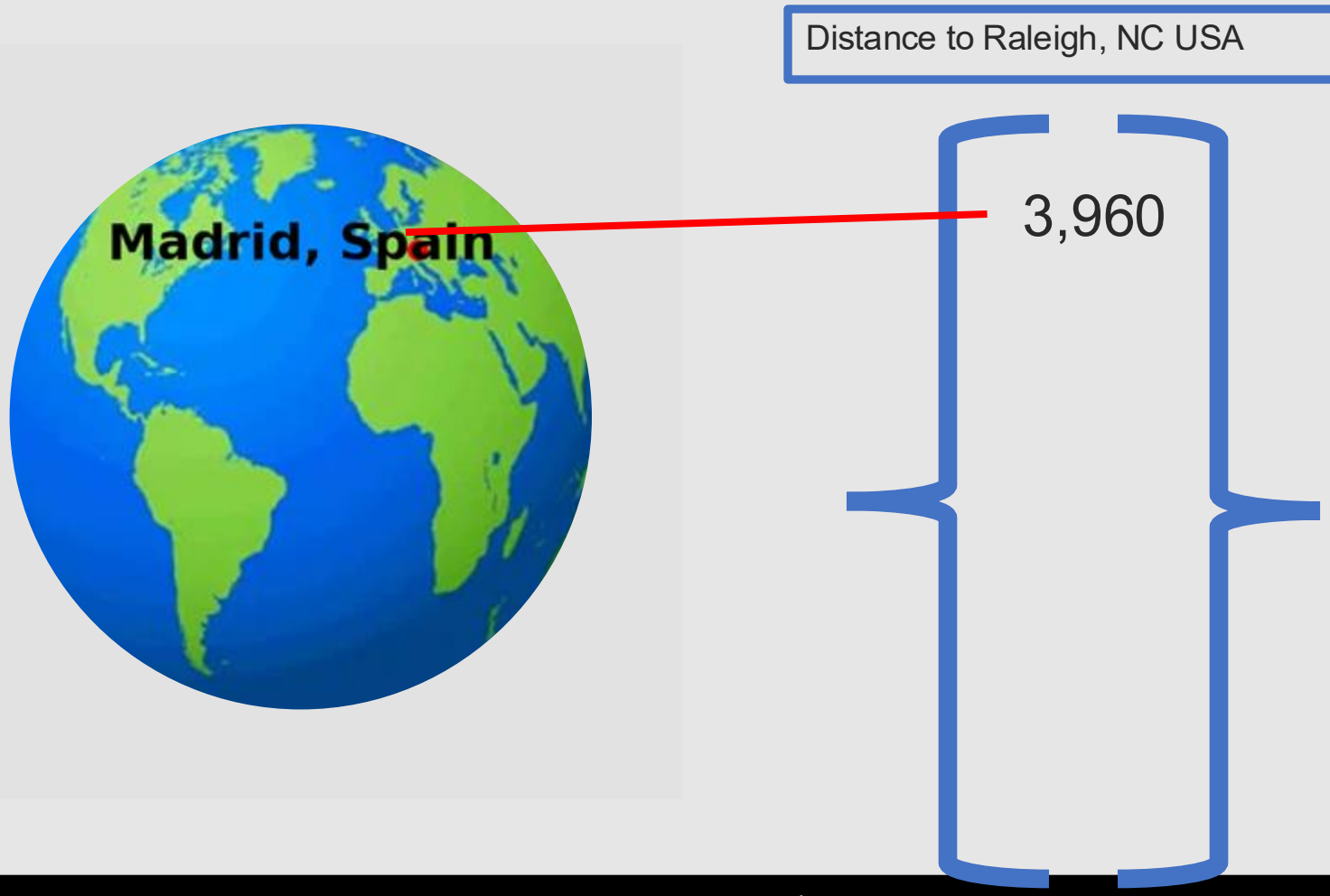
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



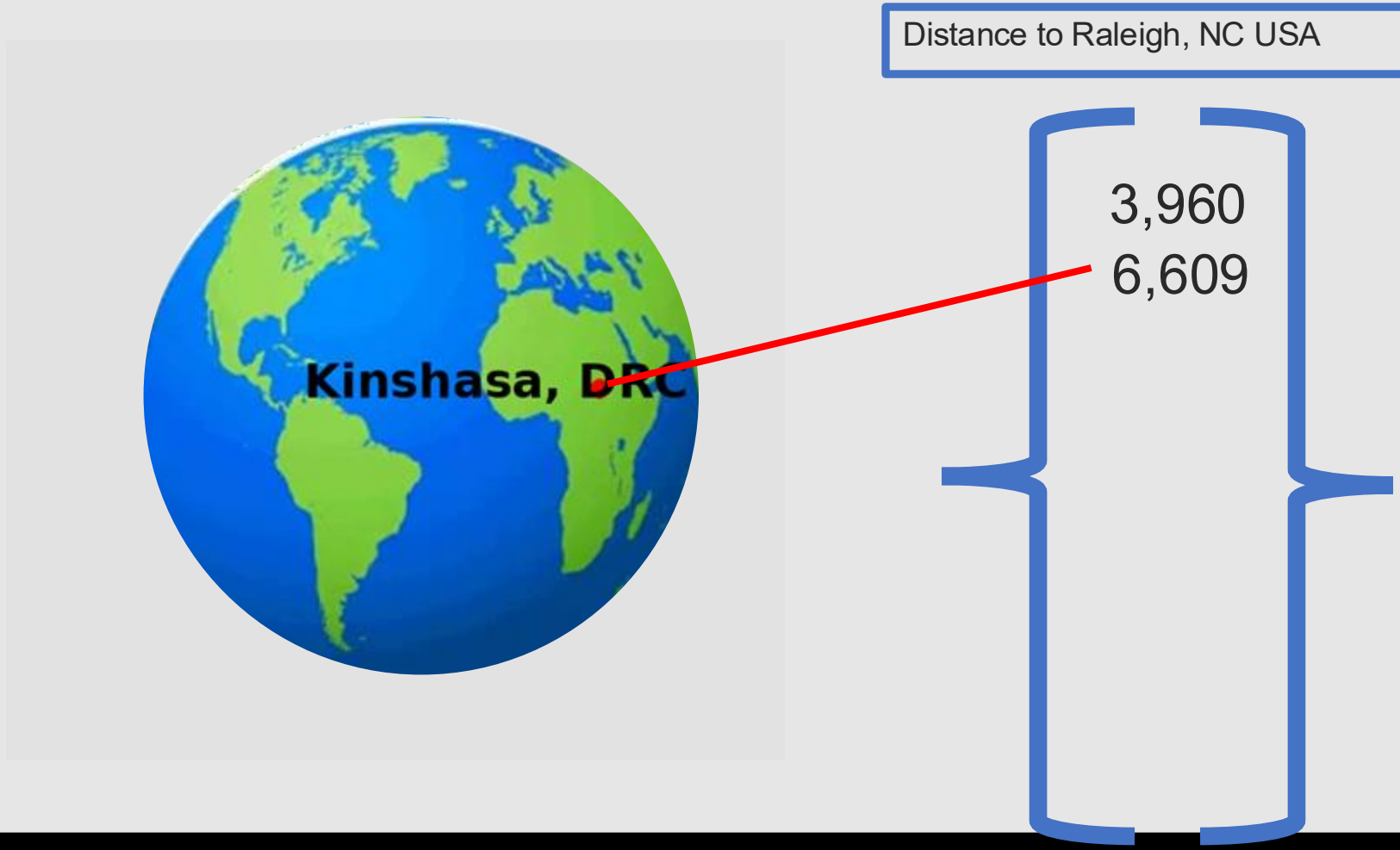
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



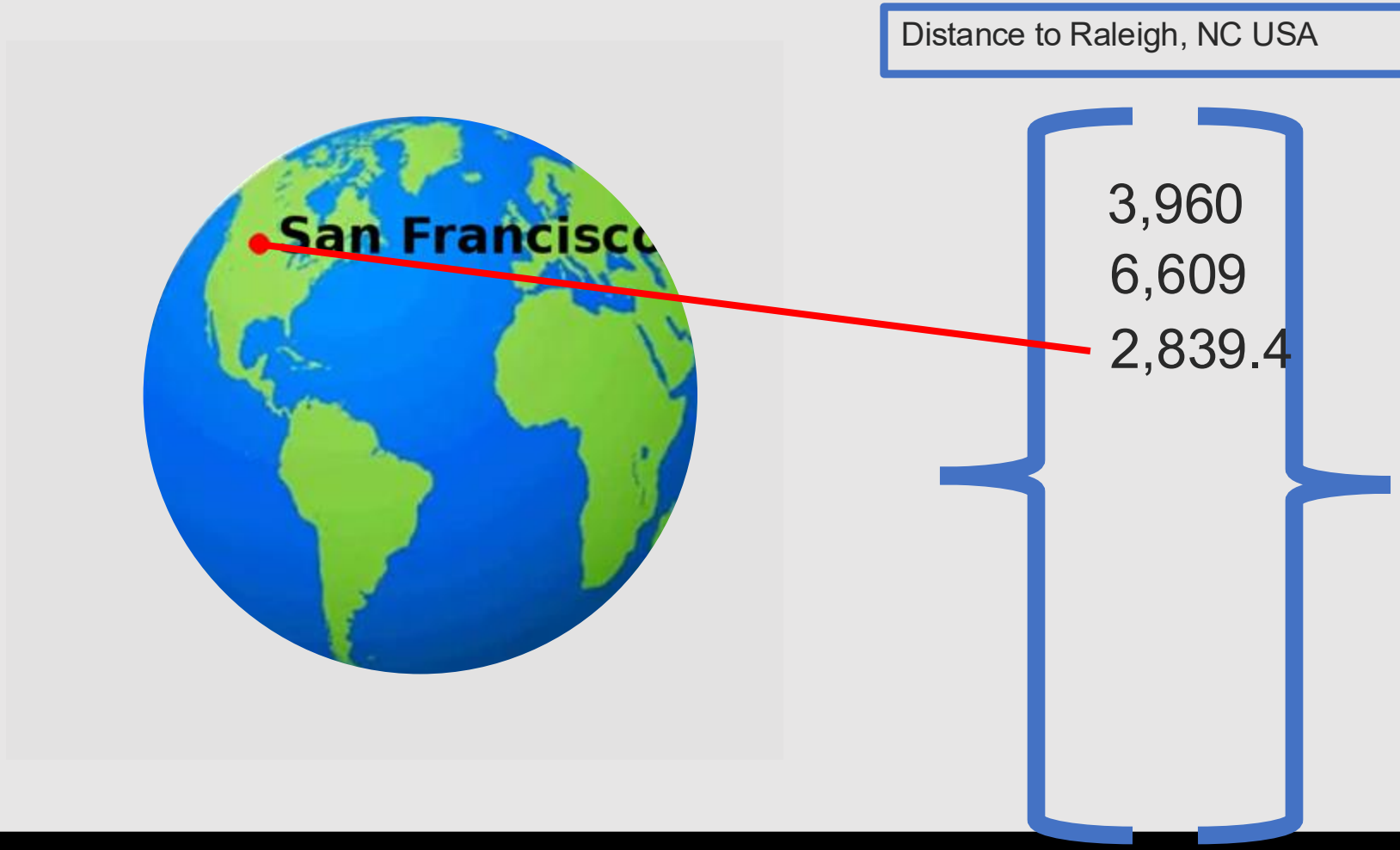
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



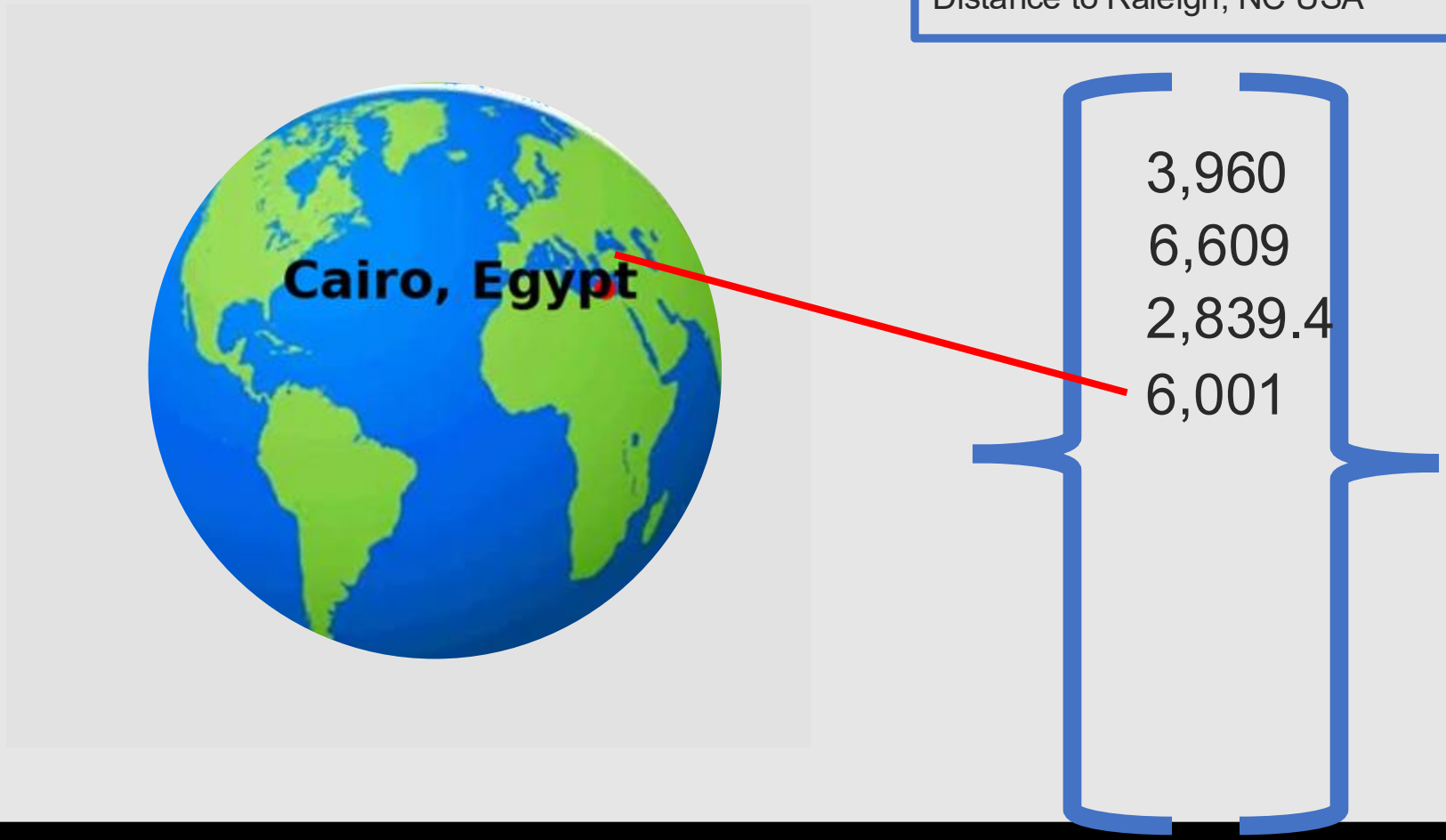
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



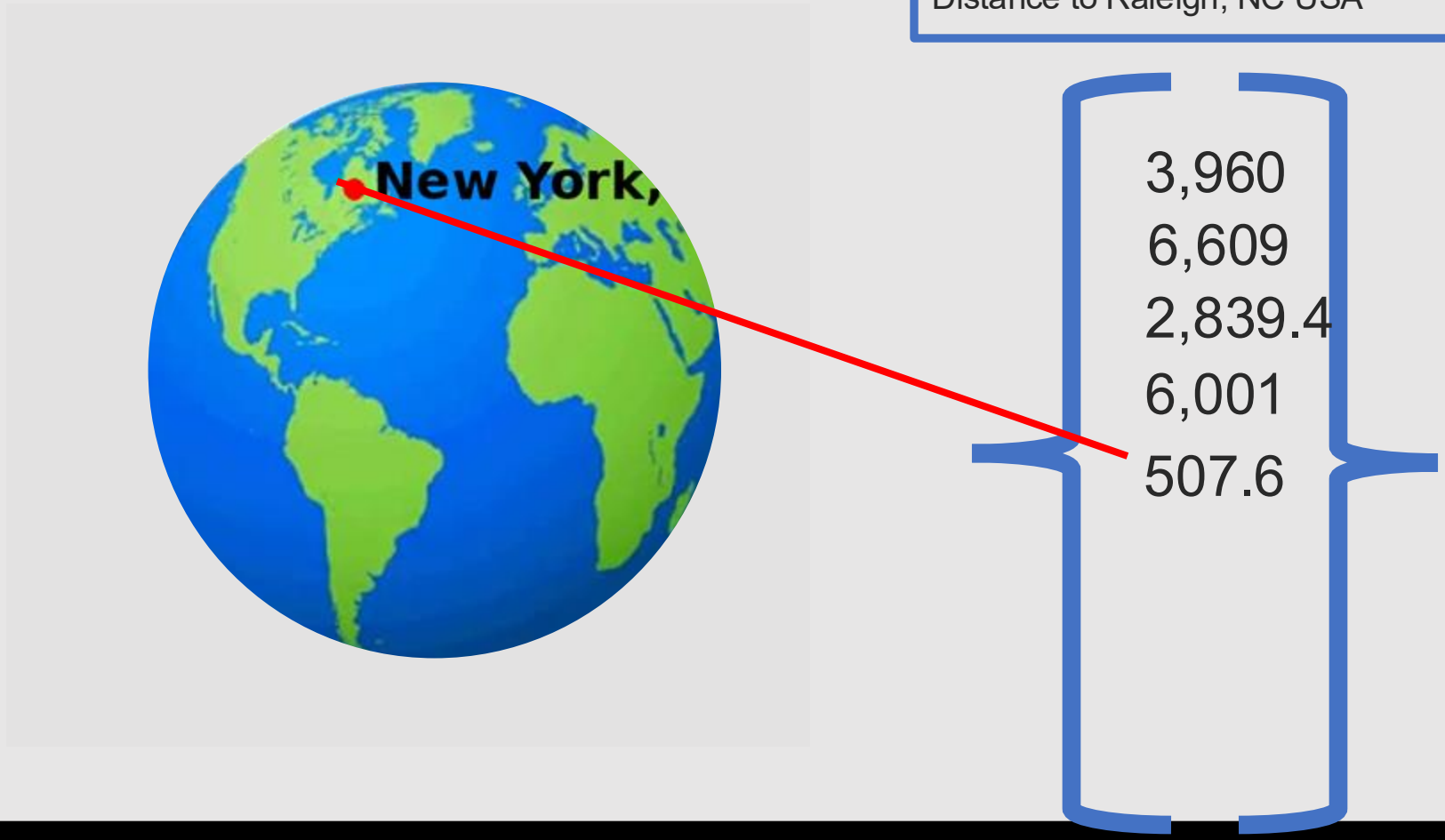
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



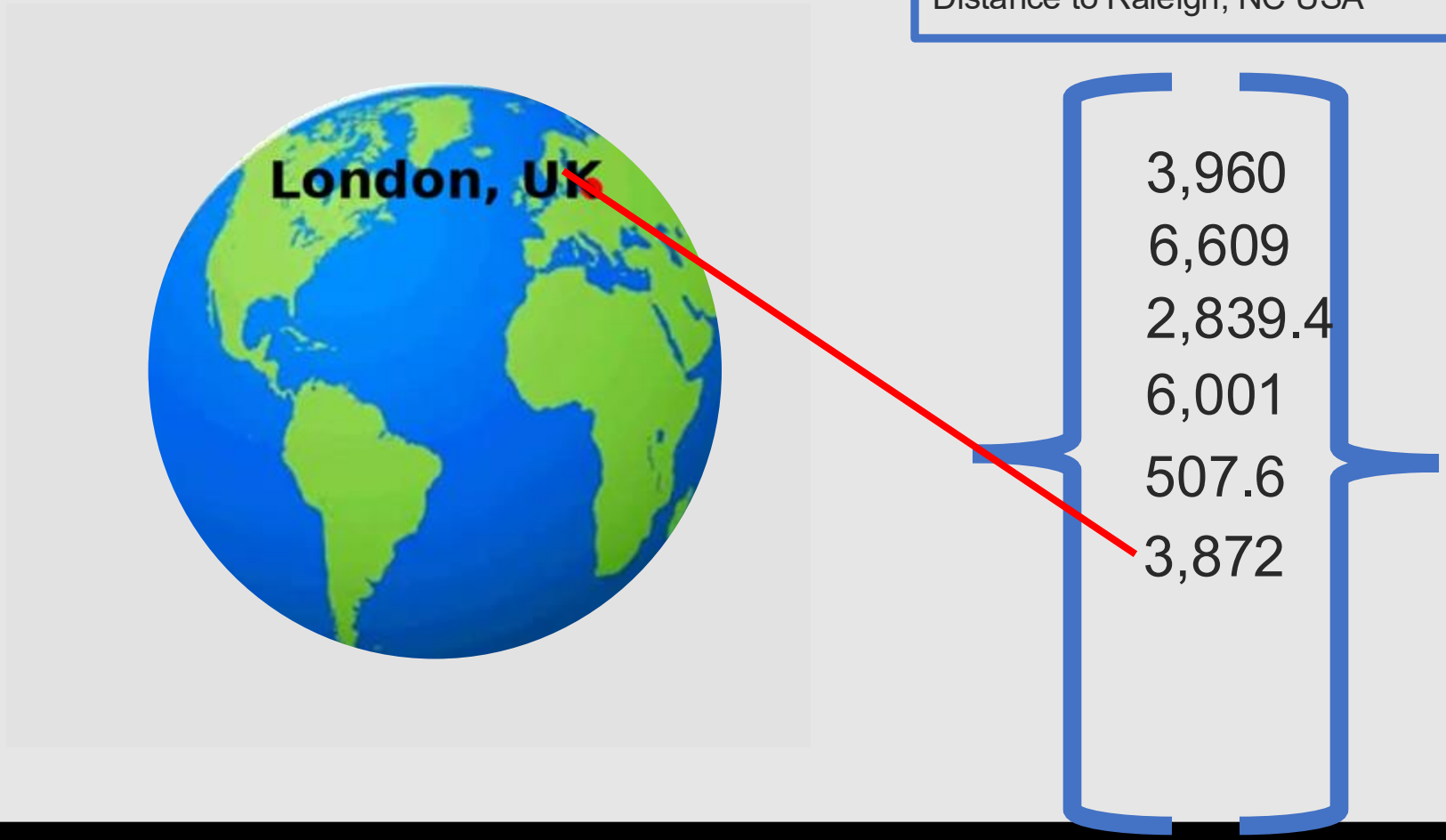
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



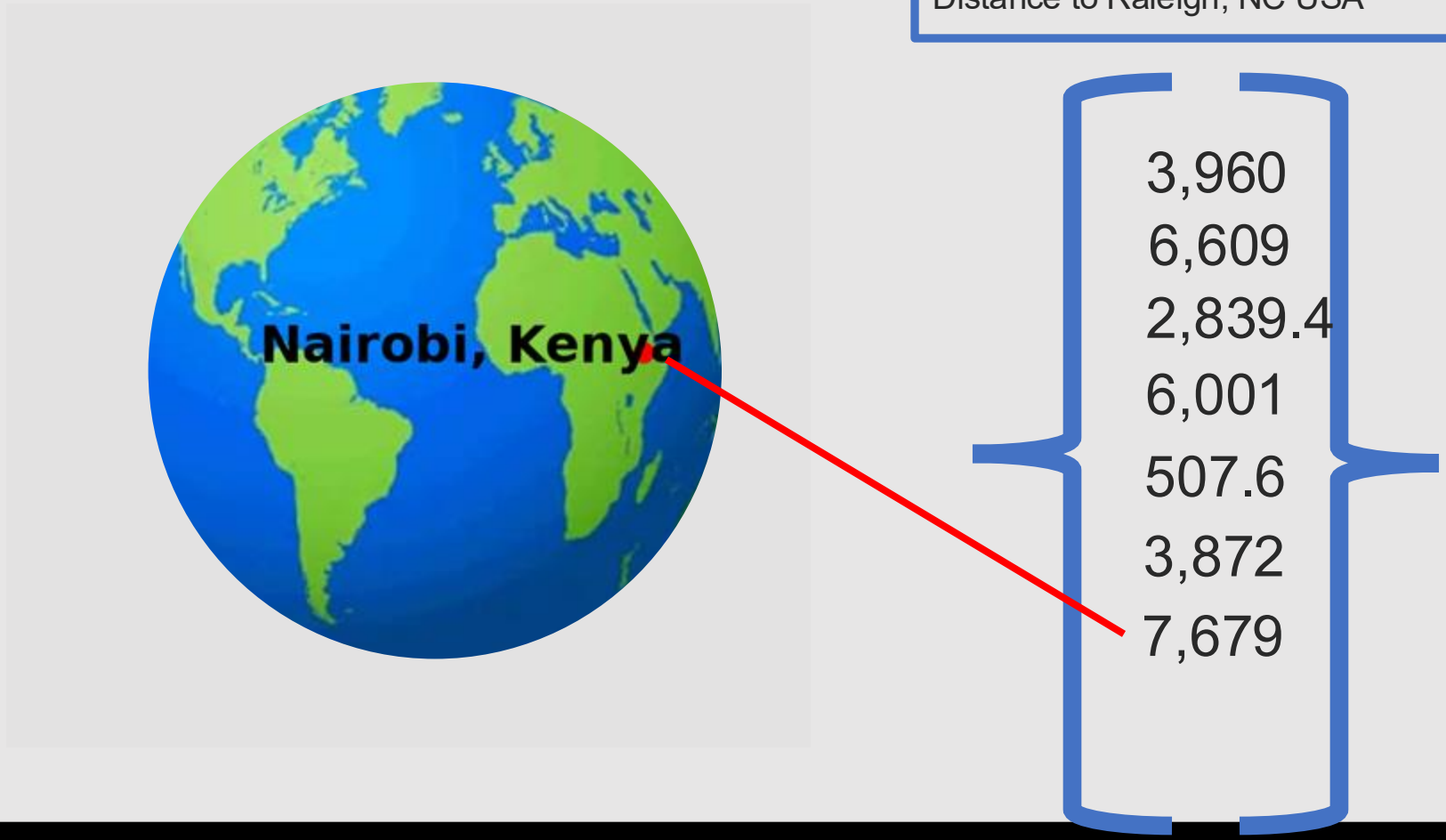
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



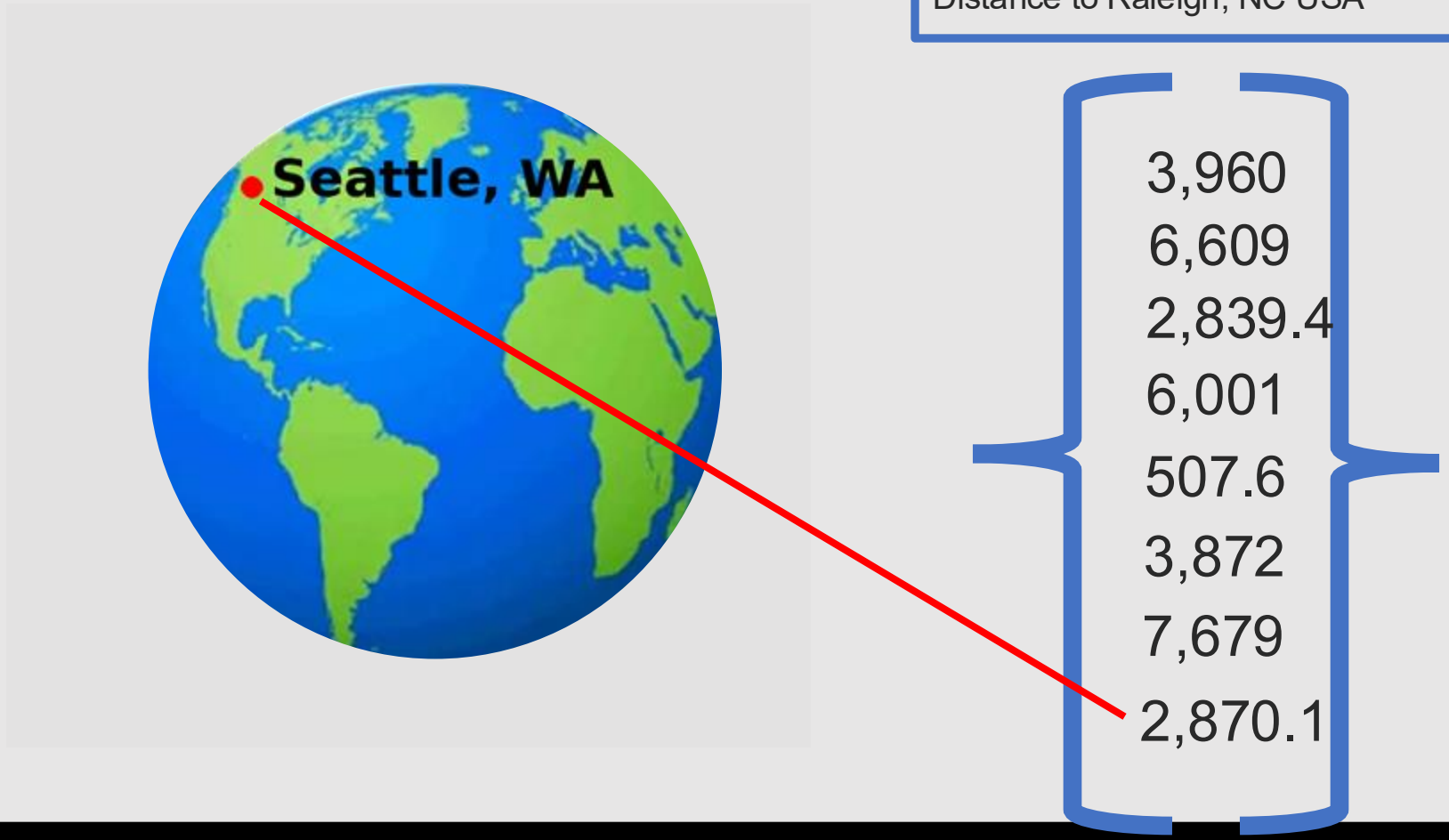
Understanding vectors in AI

- Collection of data points that encapsulate an item's relationship to other items



Understanding vectors in AI

- Collection of data points that encapsulate information about an item including it's relationship to other items



Semantic meaning / relationships

- Suppose we have 3 words
- King and Queen are more similar to each other than they are to lunch
- In order for neural net to understand the relationships, each word needs to be represented as a vector
- Suppose each word is represented by a 2-dimensional vector

King $\begin{bmatrix} -130.16 \\ 89.5 \end{bmatrix}$

Queen $\begin{bmatrix} -115.43 \\ 95.2 \end{bmatrix}$

Lunch $\begin{bmatrix} -89.5 \\ 34.3 \end{bmatrix}$

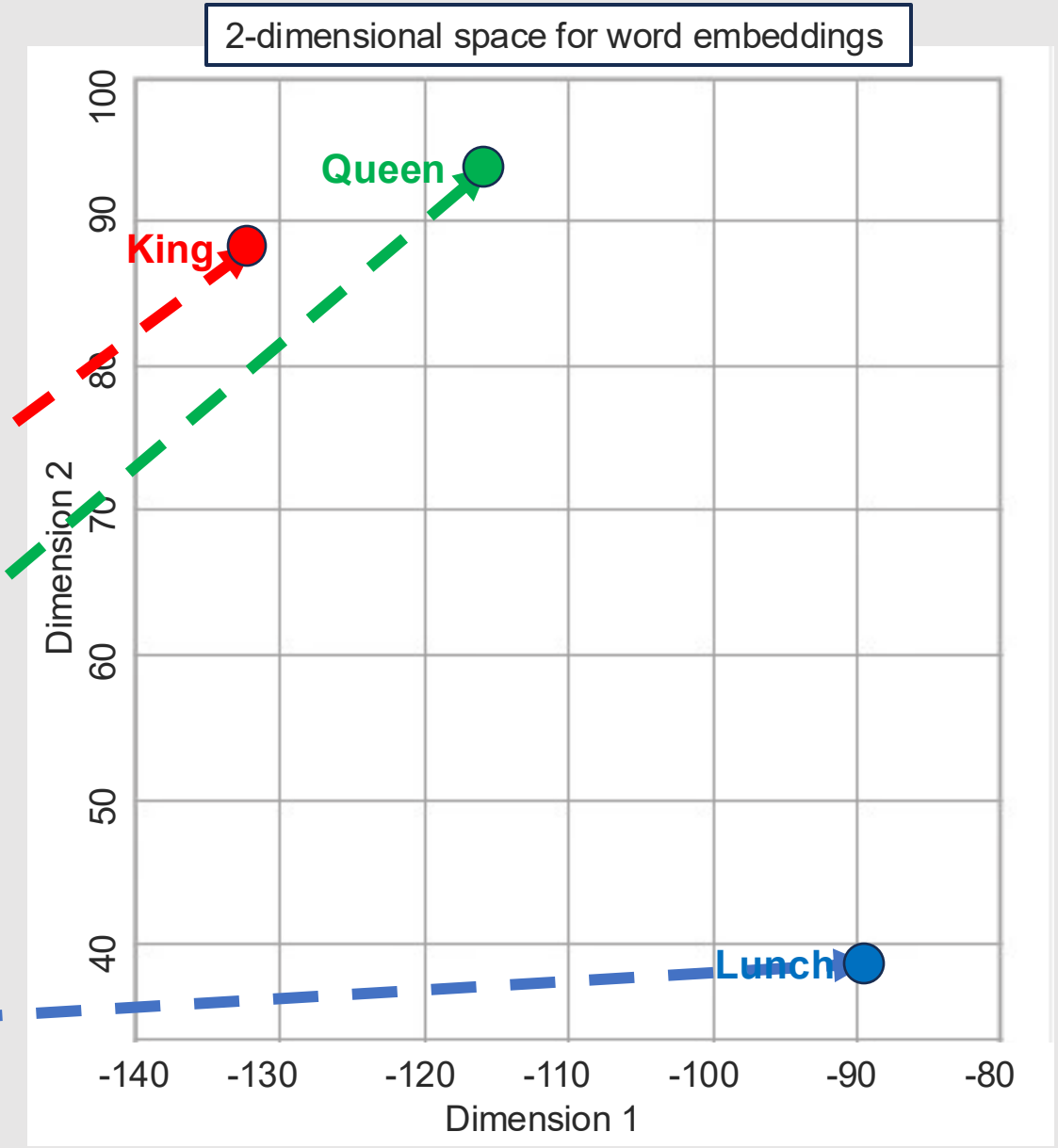
Embedding space

- Plotting in 2-dimensional embedding space shows relationships
- Way to let NN understand relationships between words
- We want the NN to learn that King and Queen are more similar to each other than they are to lunch

King [-130.16, 89.5]

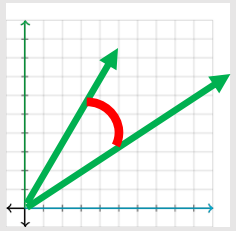
Queen [-115.43, 95.2]

Lunch [-89.5, 34.3]

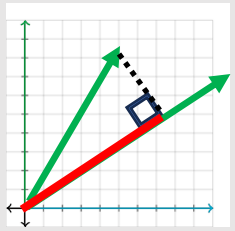


Searching for Vectors - similarity metrics

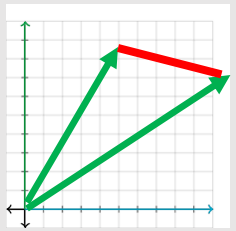
- 3 metrics commonly used to determine similarity of two vectors (2-dimensional representation)



Cosine similarity - measure the angle between two vectors; values from -1 to 1; 1 = both point in same direction; -1 point in opposite directions; 0 = orthogonal (perpendicular)



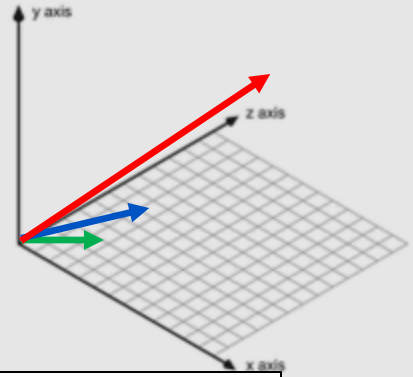
Dot product / inner product - measures how well 2 vectors align with each other; values from - ∞ to ∞; positive values indicate vectors are in same direction; negative values indicate opposite directions; 0 = orthogonal



Euclidean distance - measures the distance between two vectors; values from 0 to ∞; 0 = identical; larger numbers farther apart

imagine 3 vectors - a,b,c

$$a=\begin{bmatrix}.01\\.07\\.1\end{bmatrix} \quad b=\begin{bmatrix}.01\\.08\\.11\end{bmatrix} \quad c=\begin{bmatrix}.91\\.57\\.6\end{bmatrix}$$



Cosine similarity

$$sim(u,v)=\frac{u\cdot v}{\|u\|\|v\|}=\frac{\sum_{i=1}^{n}a_n b_n}{\sqrt{\sum_{i=1}^{n}u_n^2}\sqrt{\sum_{i=1}^{n}v_n^2}}$$

$$sim(a,b)=\frac{(a_1*b_1)+(a_2*b_2)+(a_3*b_3)}{\sqrt{a_1^2+a_2^2+a_3^2}\sqrt{b_1^2+b_2^2+b_3^2}}$$

$$=\frac{(0.01*0.01)+(0.07*0.08)+(0.1*0.11)}{\sqrt{0.01^2+0.07^2+0.1^2}\sqrt{0.01^2+0.08^2+0.11^2}}$$

0.0141

Dot product / inner product

$$u\cdot v=|u||v|cos\theta=\sum_{i=1}^{n}a_n b_n$$

Dot product formula

$$a\cdot b=(a_1b_1)+(a_2b_2)+(a_3b_3)$$

$$=(0.01*0.01)+(0.07*0.08)+(0.1*0.11)$$

0.0167

Euclidean distance

$$d(u,v)=\sqrt{\sum_{i=1}^{n}(u_i-v_i)^2}$$

Euclidean distance formula

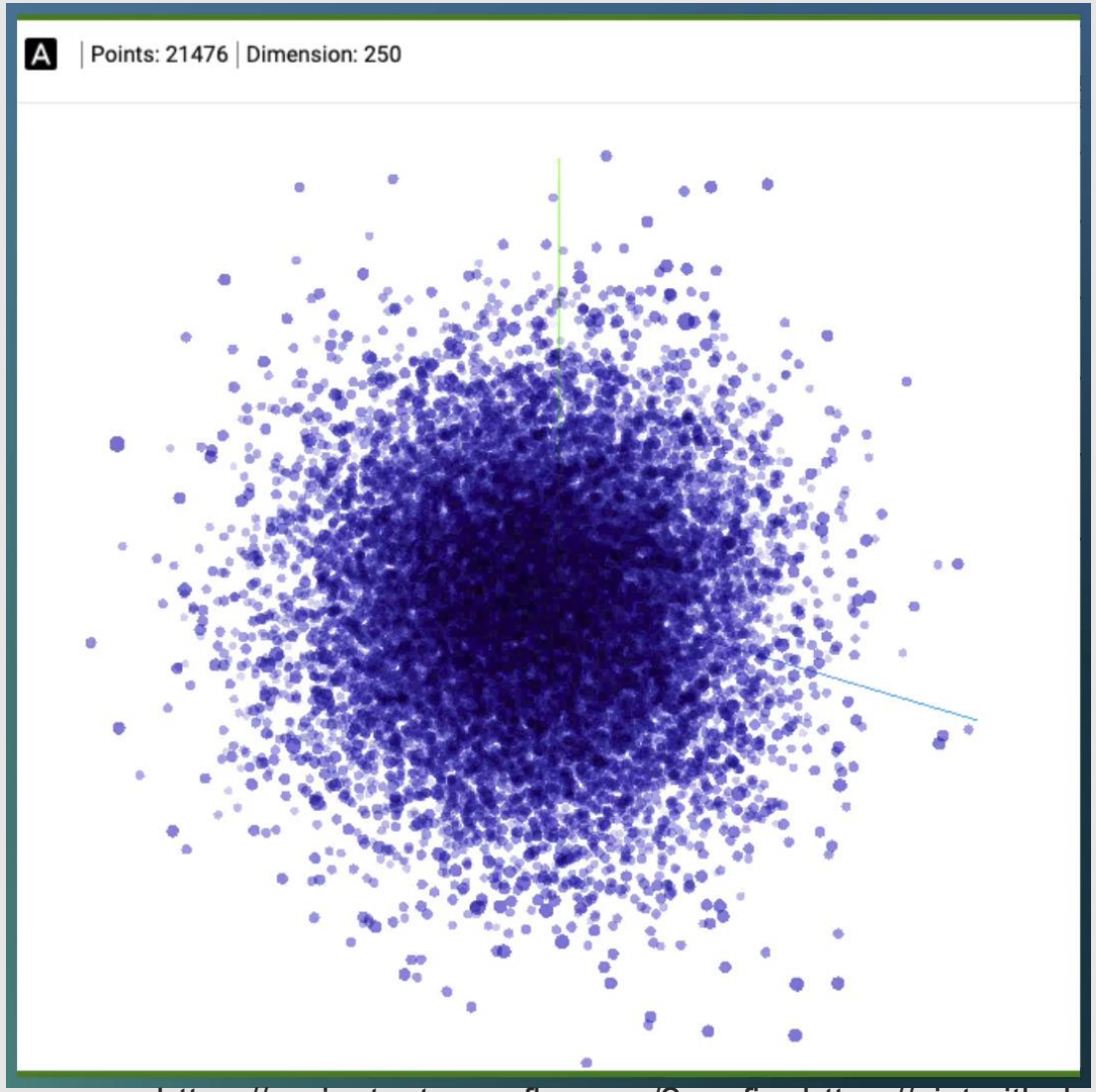
$$d(a,b)=\sqrt{(b_1-a_1)^2+(b_2-a_2)^2+(b_3-a_3)^2}$$

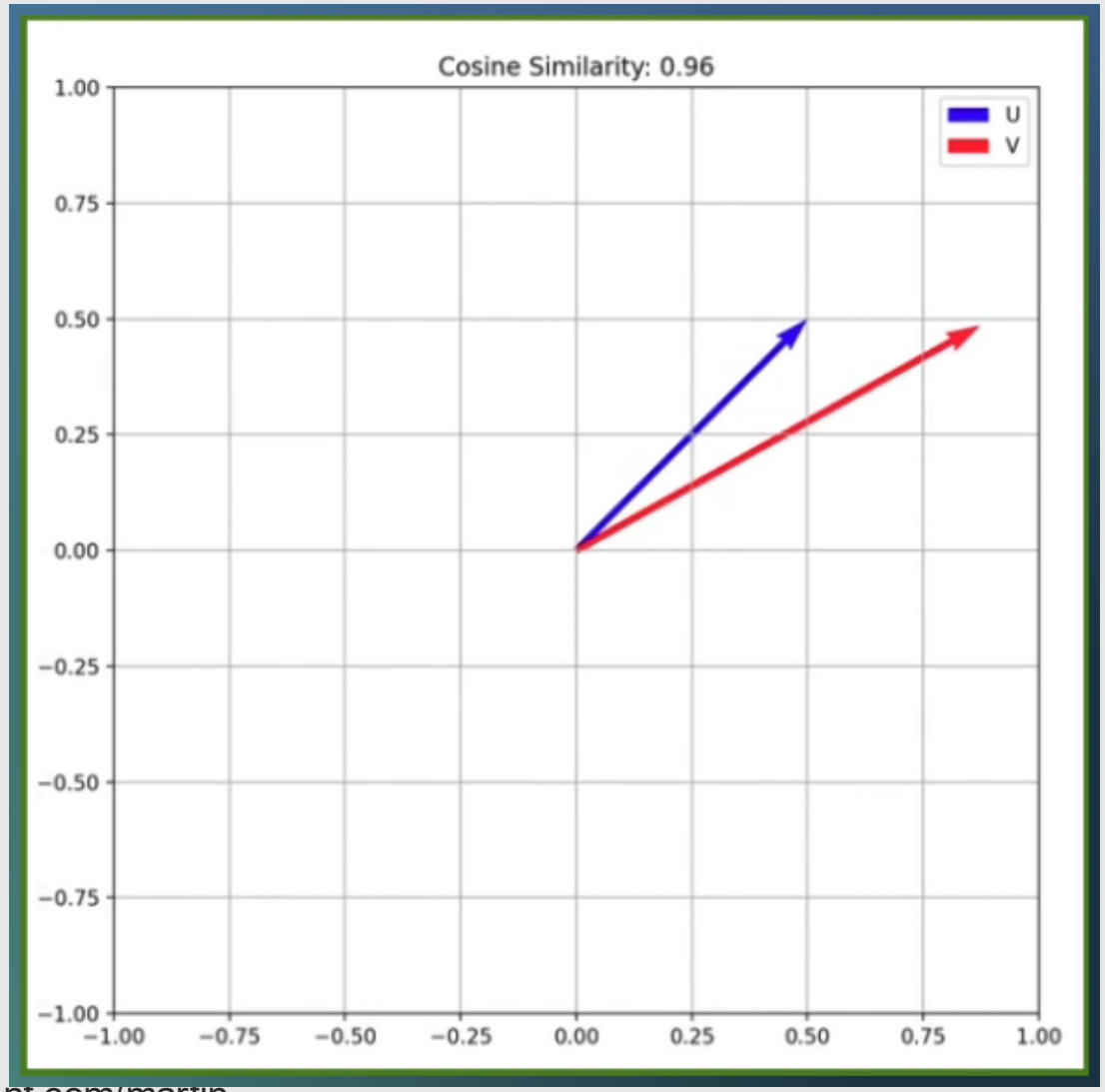
$$=\sqrt{(0.01-0.01)^2+(0.08-0.07)^2+(0.11-0.1)^2}$$

0.9998

Credit: https://towardsdatascience.com/similarity-metrics-in-nlp-acc0777e234c

Visualizing Embeddings and Vector Similarity

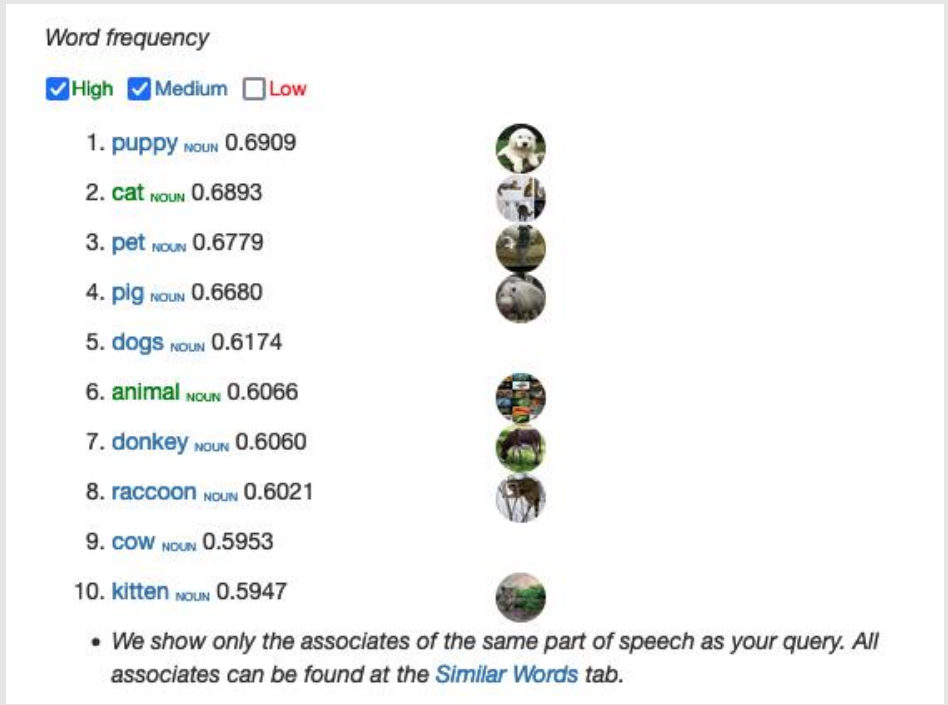


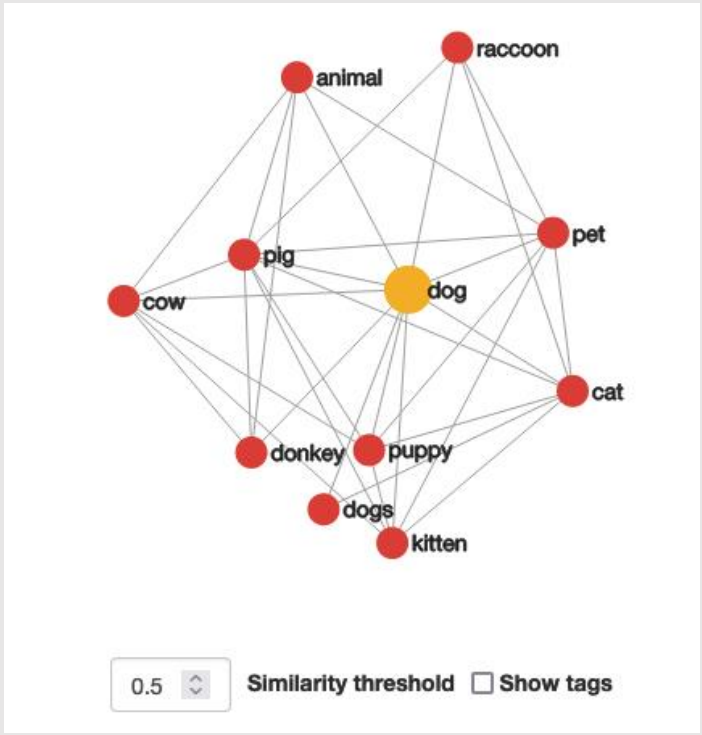


source: https://projector.tensorflow.org/?config=https://gist.githubusercontent.com/martin-labrecque/4483ff5a104f0b56417585c3bc9a12f1/raw/57348e12a70c8d70c2c573d3dbc0122ac077556b/journaux_config.json

Vectors and relationships example

- Query - what words are related to "dog" in model "English Wikipedia"?





Show the raw vector of «dog» in model MOD_enwiki_upos_skipgram_300_2_2021:

[-0.03301828354597092, 0.05134638026356697, 0.0036009703762829304, -0.04066073149442673, 0.10361430048942566, 0.013021323829889297, 0.028161464259028435, -0.0027567853685468435, 0.03388035297393799, -0.044882044196128845, 0.005169689189642668, -0.05818631127476692, 0.0533536821603775, 0.016616210341453552, 0.02030780538916588, -0.008570297621190548, -0.10925538837909698, -0.0708925873041153, 0.04675082117319107, -0.03091960959136486, -0.05172094330191612, 0.04471702128648758, 0.008674593642354012, -0.01816382259130478, 0.05909318849444389, 0.10409023612737656, 0.05633684620261192, -0.024881813675165176, 0.01872968301177025, 0.007228093687444925, -0.023127363994717598, 0.01528552919626236, -0.0643191784620285, -0.010359424166381359, -0.06104437634348869, -0.13868044316768646, -0.023004498332738876, 0.0038427673280239105, -0.021551262587308884, -0.03467748314142227, 0.010687021538615227, -0.017304275184869766, 0.026886526495218277, -0.0030398862436413765, -0.03685504570603371, -0.06017328053712845, 0.047442398965358734, -0.10714898258447647, 0.14808930456638336, -0.06579480320215225, -0.004342162515968084, 0.06226382404565811, 0.08031187951564789, -0.055930640548467636, -0.07030591368675232, 0.015474628657102585, 0.05367768555879593, 0.0917837843298912, 0.031899698078632355, 0.055091146379709244, -0.025078952312469482, -0.048126623034477234, -0.09730836749076843, -0.07128141075372696, 0.019415033981204033, -0.025872433558106422, -0.01761292852461338, 0.015608762390911579, -0.029876720160245895, -0.008602319285273552, 0.049825914204120636, 0.06784739345312119, 0.005586292129009962, -0.07148509472608566, -0.03097137063741684, -0.020296750590205193, 0.05099814385175705, 0.14920306205749512,

Source: http://vectors.nlpl.eu/explore/embeddings/en/MOD_enwiki_upos_skipgram_300_2_2021/dog_NOUN/

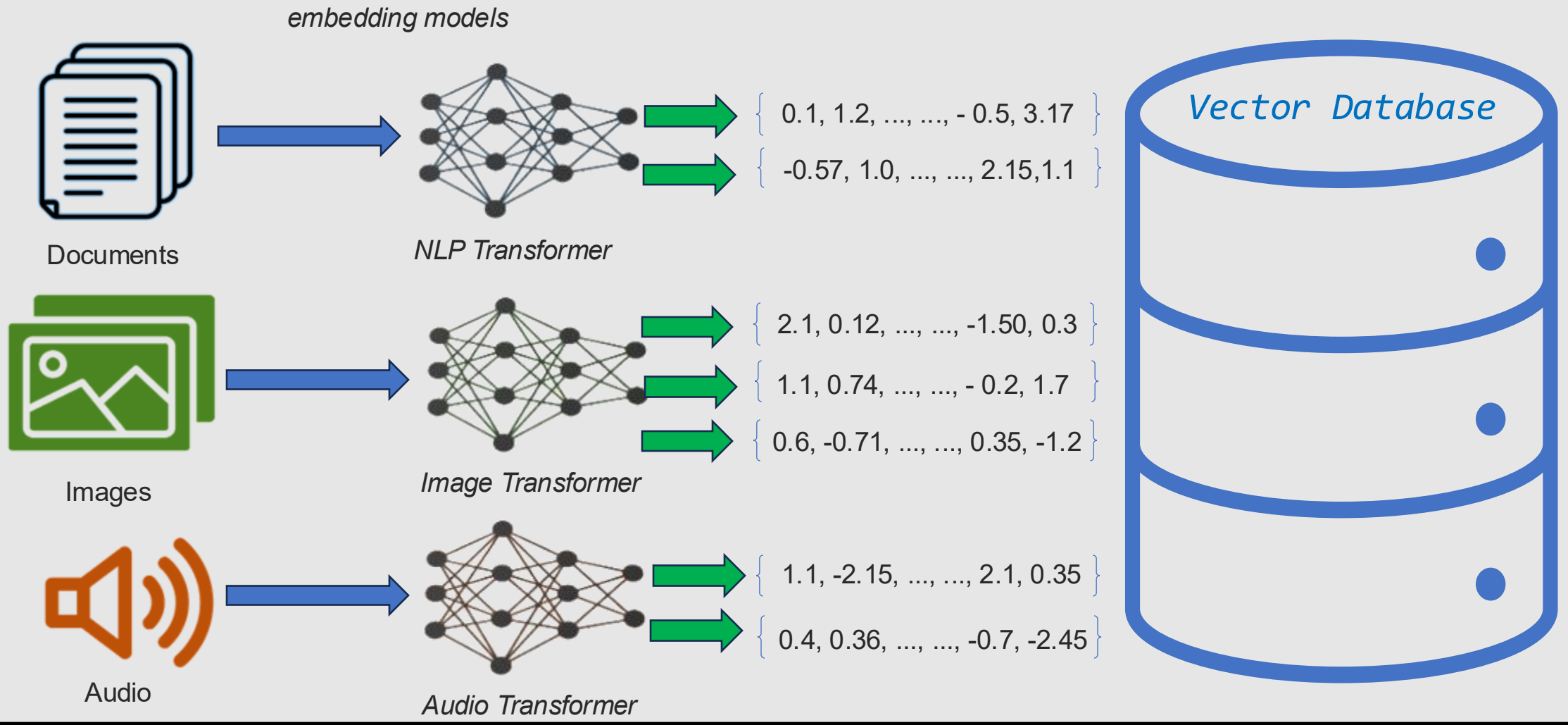
Vector Databases

Vector Databases

- Specialized database that index and stores *vector embeddings*
- Useful for
  - fast retrieval
  - similarity search
- Offer comprehensive data management capabilities
  - metadata storage
  - filtering
  - dynamic querying based on associate metadata
- Scalable and can handle large volumes of vector data
- Support real-time updates
- Play key role in AI and ML applications

How data gets into Vector Databases

- Data is input, converted to embeddings (vectors) and stored
- Queries are input, converted to embeddings (vectors) and then **similarity metrics** are used to find results ("nearest neighbors")



Lab 4 – Working with Vector Databases

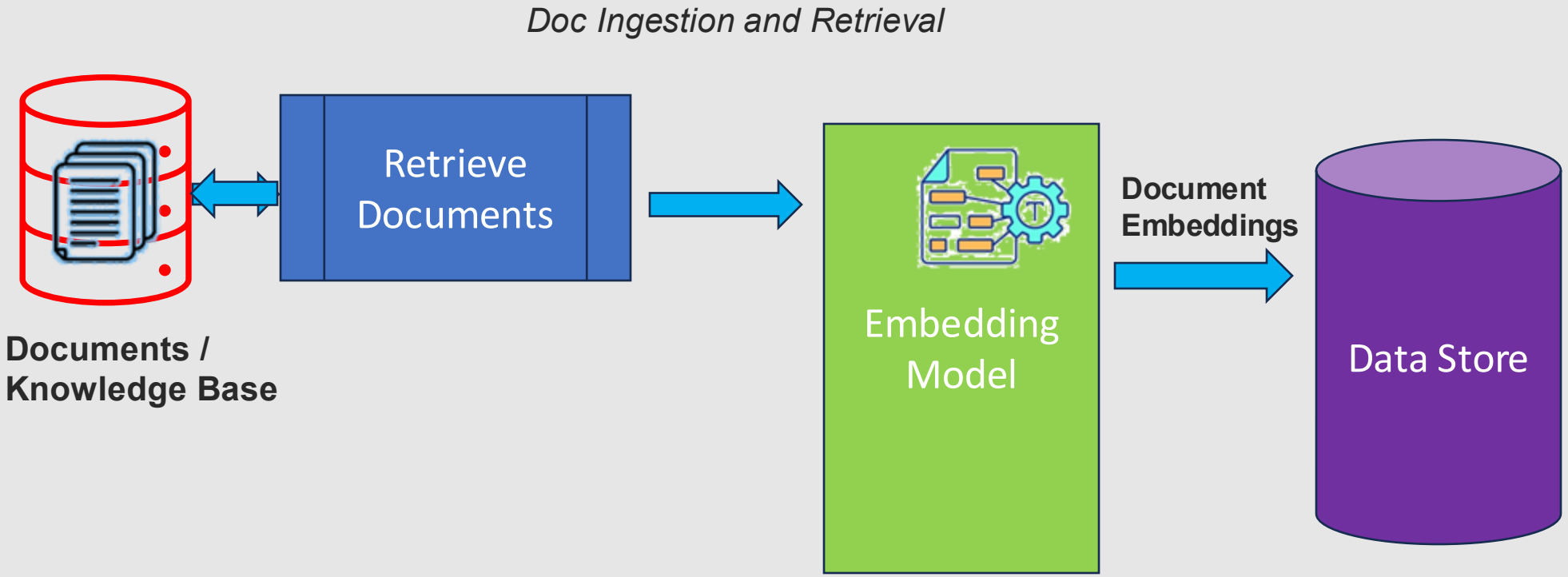
Purpose: In this lab, we'll learn about how to use vector databases for storing supporting data and doing similarity searches.

RAG

What is Retrieval Augmented Generation (RAG)?

- Search local data for related "hits" and add them to prompt for context
- Unlike keyword search, RAG understands *meaning*
- Your data is turned into numeric representations (embeddings) where each piece of data has information about how it relates to others
- **Retrieval:** When you ask a question/do a search, RAG turns your question/search into its own numeric representation (embedding) and uses calculations to find data that is numerically related (has similar meanings)
- **Augmentation:** The top "hits" (search results) are then added to the prompt we send to the LLM
- **Generation:** Those search results give the LLM some local context (passed in through the prompt) for it to consider in responding

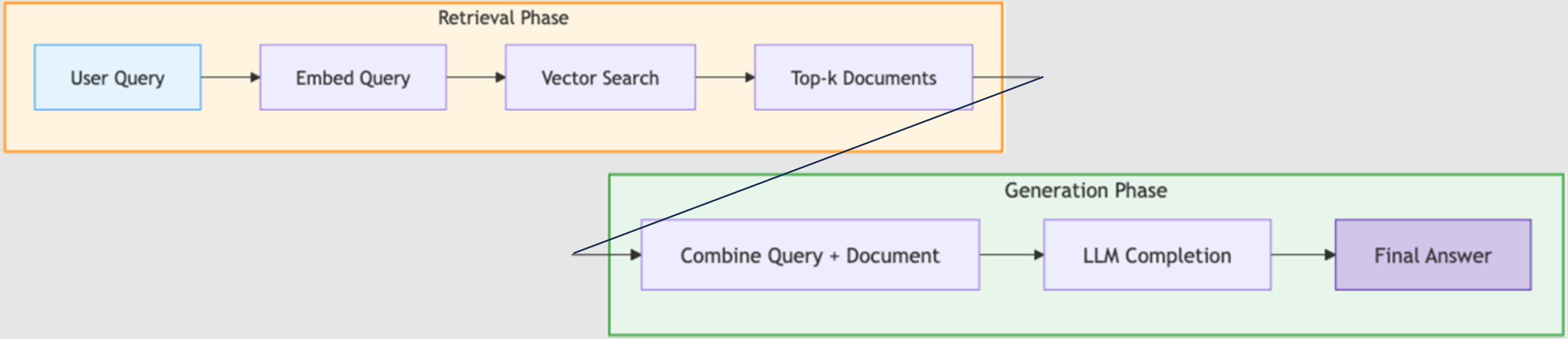
What is RAG and how does it work? (the setup)



- Your data is parsed and stored with information about other data it's related to in a data store
  - parsing = tokens, storing as numeric data = embeddings, information about other data = vectors (set of numbers that represent how much a term is related to others in the data store)

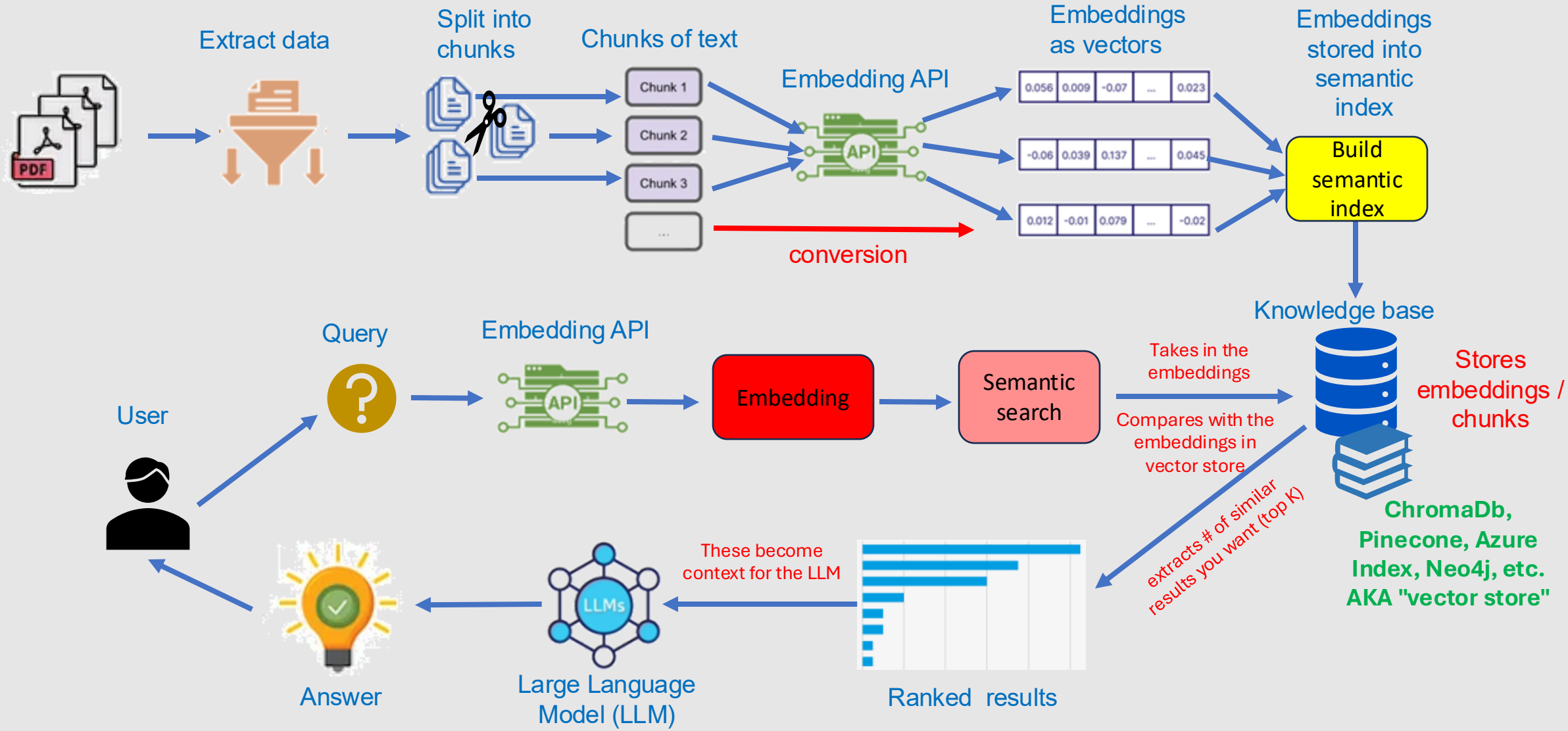
What is RAG and how does it work? (the application)

- When ready to prompt LLM, a separate "search" is done first to find related information in the data store
  - search strings are parsed and turned into embeddings
  - search is done using calculations on values in vectors to figure out which things are most related
  - top results are returned
- Top results are then added to LLM prompt/query to give it more context to consider
- LLM considers top hits passed to it from your data in addition to its own training data when generating results



Source: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/

Typical RAG Pipeline



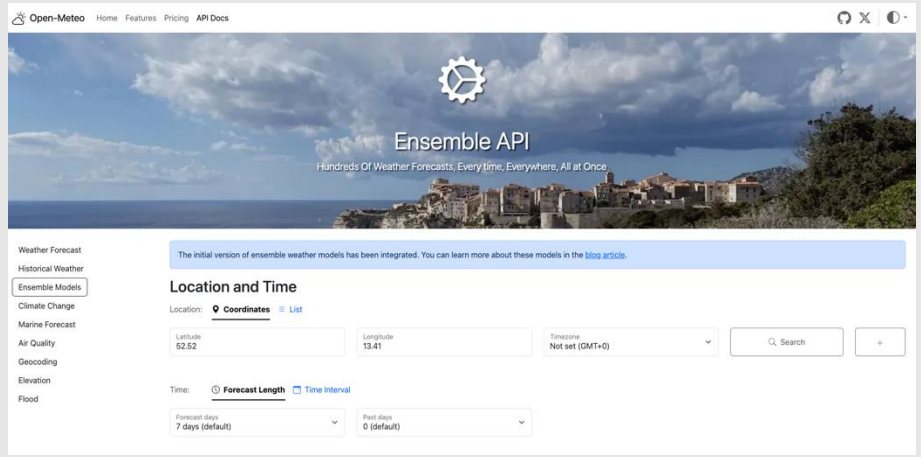
Lab 5 – Using RAG with Agents

Purpose: In this lab, we'll explore how agents can leverage external data stores via RAG and tie in our previous tool use.

Dealing with multiple data types

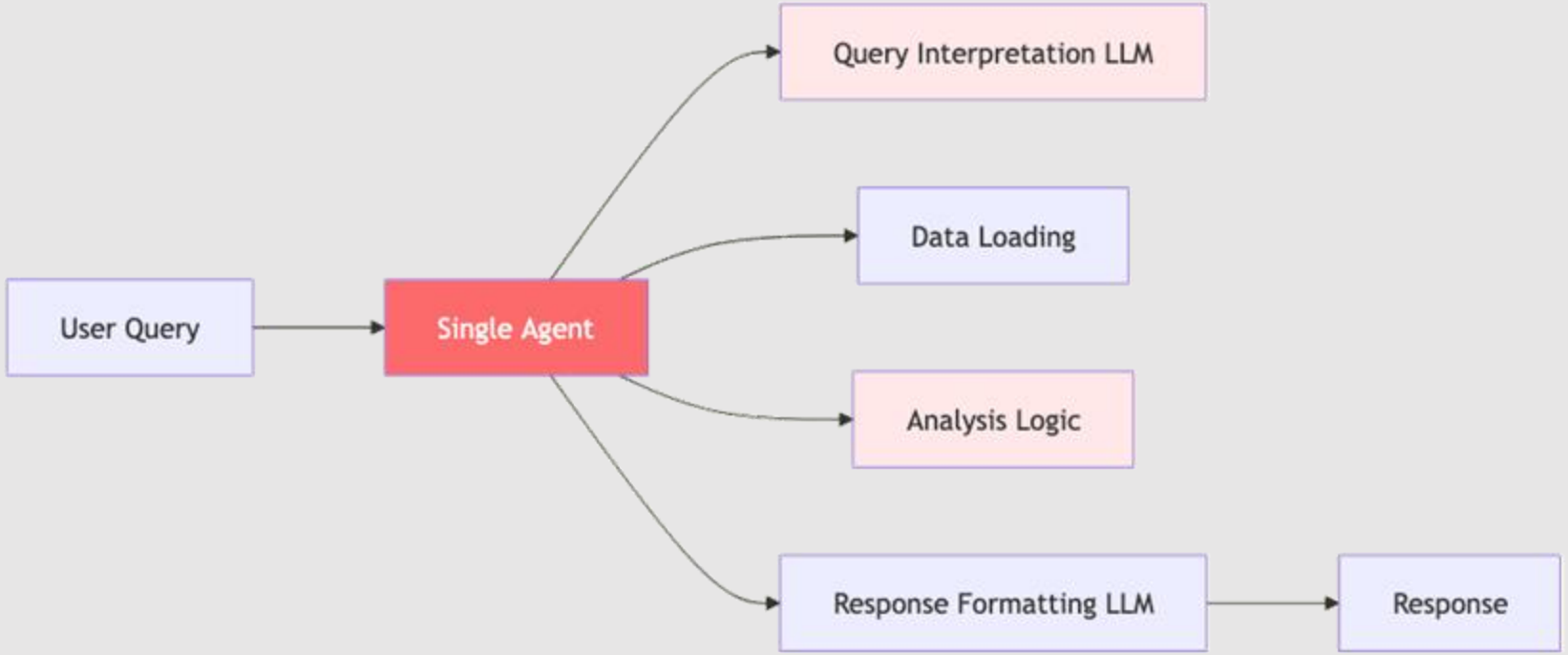
- We can have multiple types of data
- Structured – well-formed, static, can be easily processed
  - Useful for queries like "Which office has highest revenue?"
  - Best for:
    - Structured business data (CSV/SQL)
    - Predictable questions
    - Numerical analysis
    - Fast, accurate responses
    - Good fit for vector storage and search
- Unstructured – dynamic, variable format, harder to process
  - Useful for queries like "What is the weather in Paris?"
  - Best for:
    - Unstructured documents
    - Semantic search needs
    - External API calls
    - Open-ended queries
- Both are prevalent
- Multiple approaches to dealing with both in AI apps

1	city	employees	revenue_million	opened_year
2	San Francisco	250	45.3	2015
3	New York	380	62.1	2012
4	London	175	38.7	2017
5	Tokyo	290	51.2	2014
6	Seattle	210	42.5	2016
7	Austin	145	28.9	2018
8	Berlin	165	33.4	2019
9	Singapore	195	41.8	2016
10	Toronto	220	39.6	2017
11	Sydney	185	35.2	2018

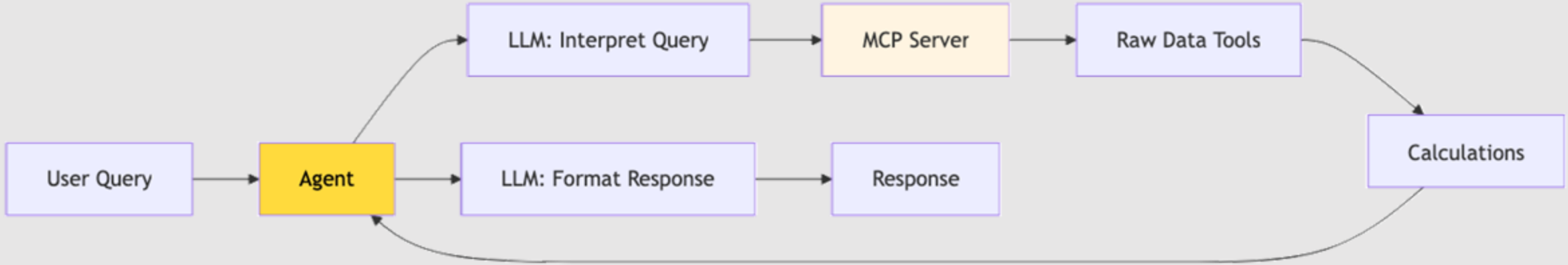


Architectural Approach #1: Agent-Only Architecture

- Agent handles everything
  - Data-loading/analysis
  - Uses LLM for query interpretation/response formatting
- Problems:
  - All logic embedded in one agent
  - Code duplication across queries
  - Tight coupling - hard to modify
  - Difficult to scale

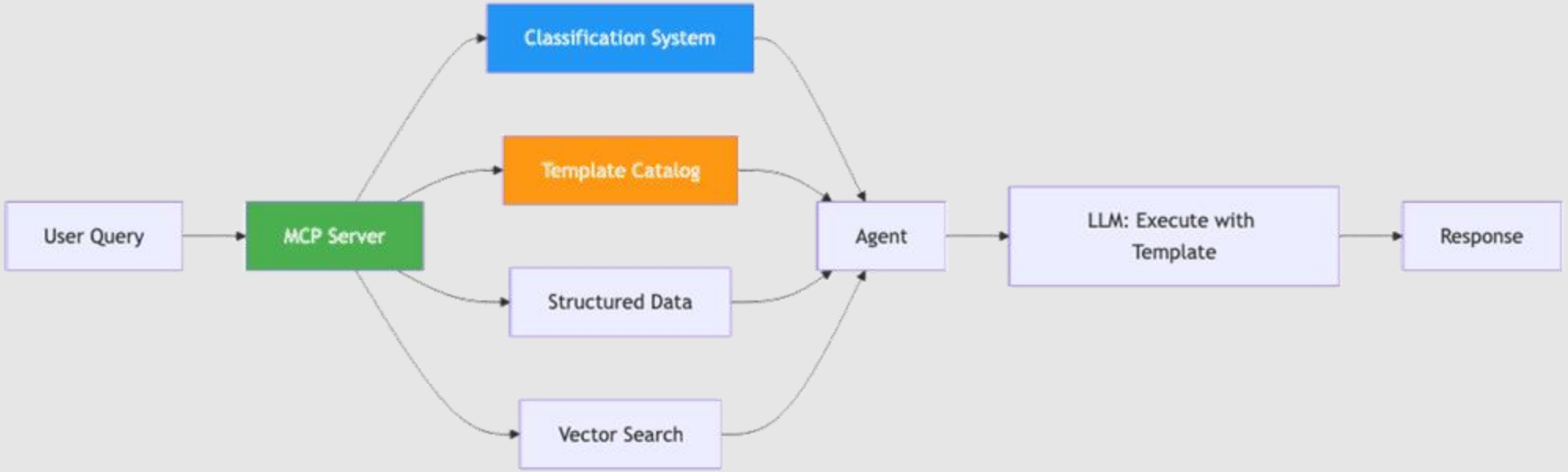


Architectural Approach #2: Pure Data Tools



- Uses MCP Server to handle all data
- LLM interprets/formats queries
- Problems:
  - Query interpretation still ad-hoc (subject to LLM's understanding/interpretation)
  - Each agent must figure out which tool to use
  - No standardized query patterns

Architectural Approach #3: Classification & Templates



- Best approach
  - Structured query interpretation (uses programming logic rather than LLM only)
  - Reusable templates (for prompts)
  - Centralized query logic
  - Server manages classification and data
  - Useful approach to employ "Canonical Queries"

Canonical Queries

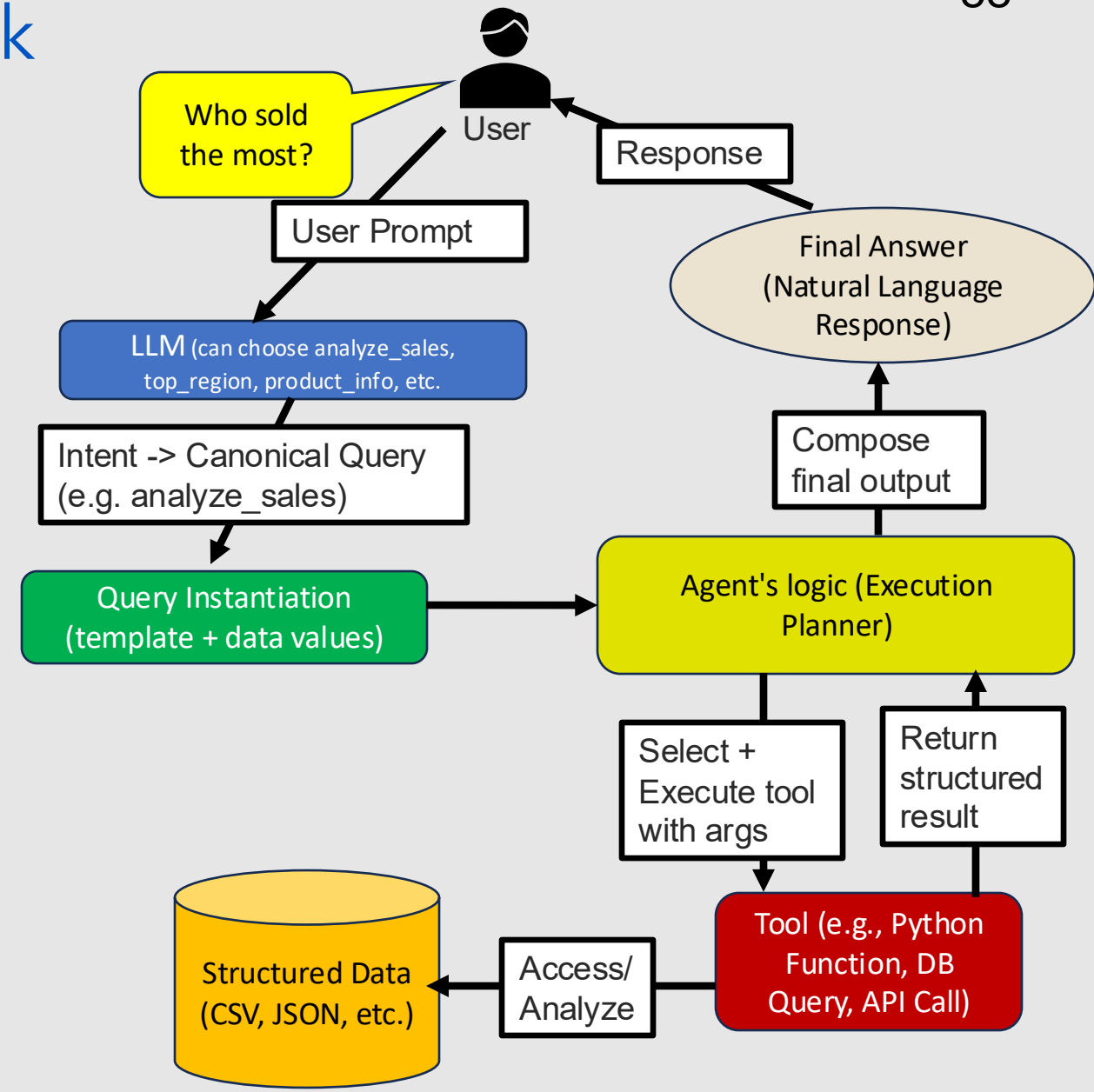
What are Canonical Queries?

- Definition
  - A *canonical query* is a **standard version** of a question
  - It helps AI systems understand different ways people ask for the same thing
  - Think of it as a common template indicated from different phrasings/wordings
- How to think about it
  - A function that can be called to do deterministic processing
  - LLM/Routing logic know available functions
  - Based on keywords, similarity metrics, etc. prompt can be mapped to one of these available functions
- Why It Matters
  - Makes AI answers **consistent** even if users phrase things differently.
  - Reduces confusion and improves **accuracy and search results**
  - Useful in **customer support, analytics, and business dashboards**

User Question	Canonical Query
"Show me last month's sales in Europe."	`get_sales(region="Europe", period="last_month")`
"How did we perform in Europe in September?"	`get_sales(region="Europe", period="last_month")`

How Canonical Queries Work

- User asks a natural question
- AI or routing logic converts it to a standard query form
- System runs query logic
- Results come back in a consistent, clear format



Canonical Queries Implementation

- A canonical query is a simple, standardized string that represents a specific analysis task
- Example: "average_revenue", "most_employees", "opened_after_2015"
- These strings are:
  - Chosen by the developer
  - Used by tools as reliable, exact-match identifiers
- Who decides the canonical query?
  - LLM-Decided (Dynamic)
    - » LLM uses prompt instructions or examples to decide which canonical query to use
    - » Infers right one from context and wording
  - Predefined or Middleware-Decided (Static)
    - » Router, orchestrator, or parser maps the user query to the canonical form
    - » Improves reliability and reduces "hallucinated" mappings

```python
# --- Tool: Analyze sales records using canonical query strings ---
def analyze_sales(data: dict, query: str, memory: list) -> str:
    df = pd.DataFrame(data)
    q = query.lower().strip()

    if q == "total_sales":
        return f"Total sales: ${df['amount'].sum():,.2f}"

    if q == "top_region":
        top = df.groupby("region")["amount"].sum().idxmax
        return f"Top sales region: {top}"

    if q.startswith("product_"):
        prod = q.split("_", 1)[1]
        filtered = df[df["product"].str.lower() == prod]
        return f"Sales for {prod}: ${filtered['amount'].sum():,.2f}"

    return "No matching query."
```

function that implements action & context can have more detailed/bespoke logic

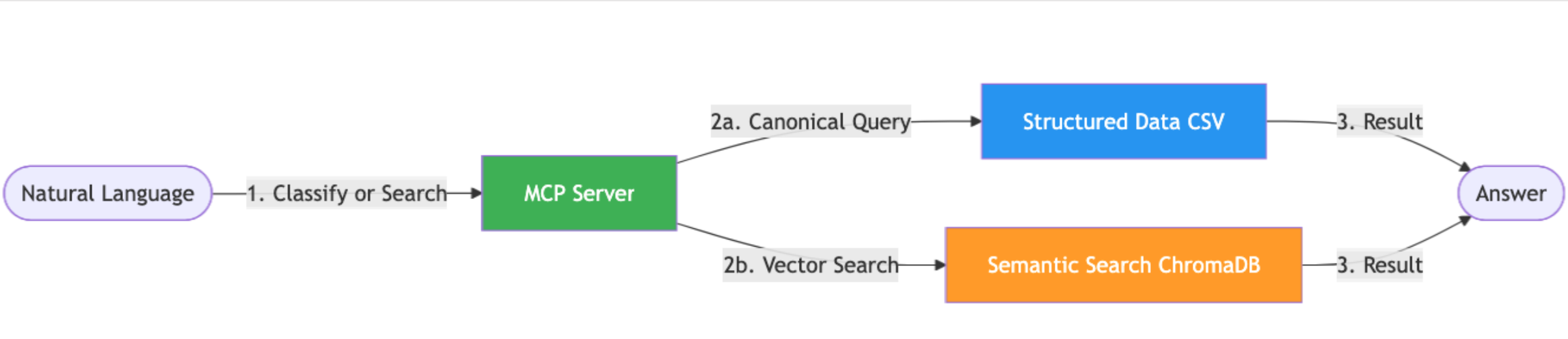
```
# --- System Prompt ---
"""
You are a helpful assistant analyzing sales data.

Use the following canonical queries:
  - total_sales
  - top_region
  - product_<name>  (e.g. product_laptop)

Respond in this JSON format:
{
  "thought": "why you chose the tool",
  "action": "analyze_sales",
  "args": {
    "query": "<canonical query string>"
  }
}
"""
```

LLM processes user prompt and selects "best fit" from shortcut phrases

LLM calls appropriate action (function) with context (shortcut phrases)

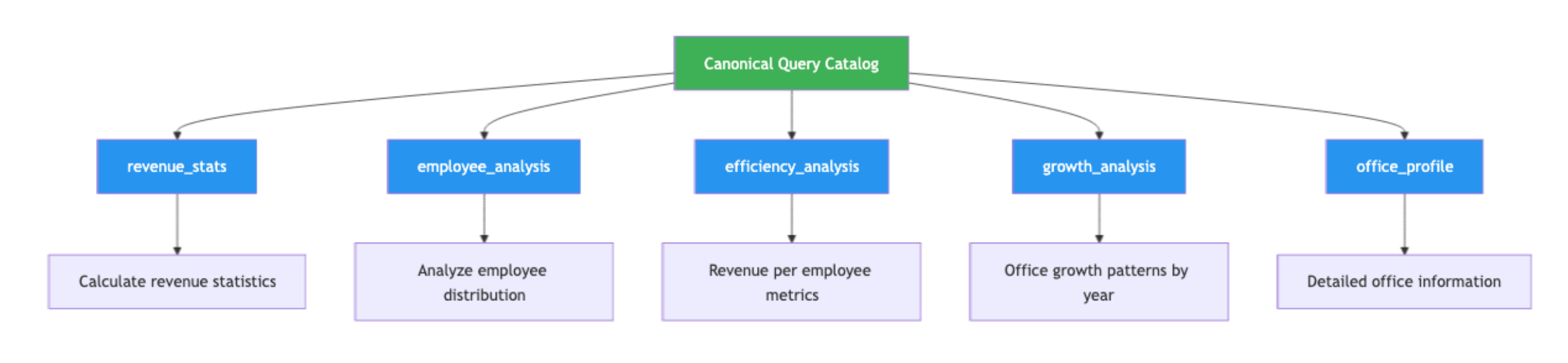
Classification MCP Server : High-level Architecture

- Classification System (Keyword-Based)
  - Maps natural language to canonical queries
  - Uses keyword scoring and matching
  - Returns confidence levels



Classification MCP Server : High-level Architecture

- Canonical Query Catalog



Classification MCP Server : High-level Architecture

- Vector Database Layer

- **ChromaDB**: Persistent vector database at `./mcp_chroma_db`
- **Embedding Model**: `all-MiniLM-L6-v2` (SentenceTransformer)
- **Two collections**:
  - `office_locations` - PDF embeddings (one per line from offices.pdf)
  - `office_analytics` - CSV embeddings (descriptive text for each office)
- **Semantic search**: Finds relevant data beyond keywords
- **Fuzzy matching**: Handles variations like "HQ" vs "headquarters"
- **Auto-population**: Server populates collections on startup if empty

`data/offices.pdf`

Office Name	Address			ices Offered
HQ	123 Main St, New York, NY	200	15M	Corporate Operations, Finance
West Coast Hub	456 Market St, San Francisco, CA	150	12M	Tech Development, Customer Support
Midwest Office	789 Elm St, Chicago, IL	100	8M	Sales, Marketing
Southern Office	321 Pine St, Austin, TX	80	5M	Customer Support, Sales
Northeast Office	654 Maple St, Boston, MA	120	10M	Tech Development, HR
London Office	1 High St, London, UK	140	11M	Corporate Strategy, Marketing
Toronto Office	468 Palm St, Toronto, Canada	130	9M	Corporate Operations, Customer Support
Tokyo Office	5-2 Ginza St, Tokyo, Japan	110	10M	Product Development, Tech Support
Sydney Office	77 George St, Sydney, Australia	90	7M	Marketing, Customer Support
Berlin Office	22 Friedrichstrasse, Berlin, Germany	100	8M	Sales, Product Design
Paris Office	88 Champs-Élysées, Paris, France	95	9M	Marketing, Sales

`data/offices.csv`

1	city	employees	revenue_million	opened_year
2	San Francisco	250	45.3	2015
3	New York	380	62.1	2012
4	London	175	38.7	2017
5	Tokyo	290	51.2	2014
6	Seattle	210	42.5	2016
7	Austin	145	28.9	2018
8	Berlin	165	33.4	2019
9	Singapore	195	41.8	2016

Classification MCP Server : High-level Architecture

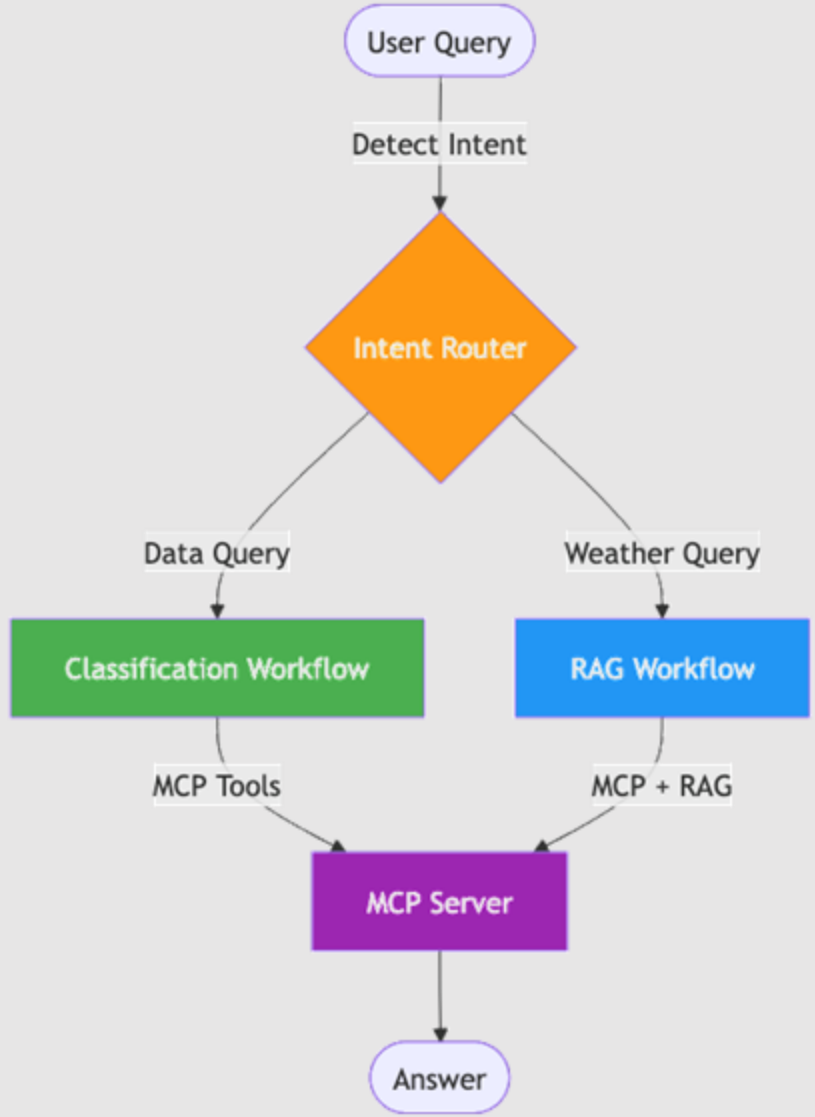
- MCP Tools Provided
  - **Classification & Templates:**
    - list_canonical_queries() - List all available queries
    - classify_canonical_query() - Classify user intent
    - get_query_template() - Get prompt template
    - validate_query_parameters() - Validate query parameters
  - **Structured Data Access:**
    - get_office_dataset() - Get complete office dataset
    - get_filtered_office_data() - Get filtered CSV data with column selection
  - **Vector Search (Semantic):**
    - vector_search_locations() - Semantic search for PDF location data
    - vector_search_analytics() - Semantic search for CSV analytics data
  - **Legacy Location Tools:**
    - search_office_locations() - Keyword search in PDF (legacy)
    - get_all_office_locations() - Get all location data from PDF
  - **Weather & Geocoding:**
    - get_weather() - Weather API via Open-Meteo
    - geocode_location() - Location to coordinates
    - convert_c_to_f() - Temperature conversion

Lab 6 – Building a Classification MCP Server

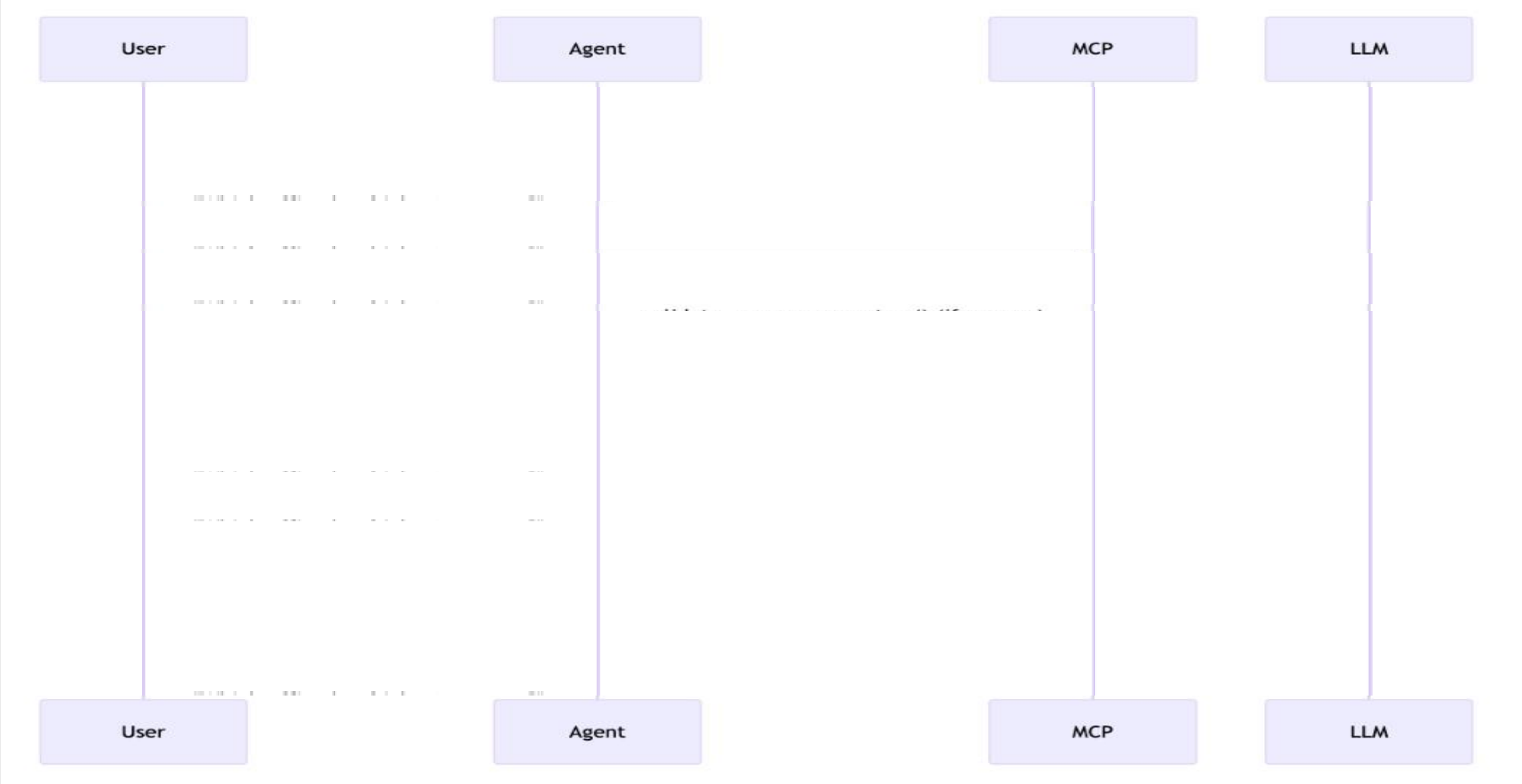
Purpose: In this lab, we'll transform our simple MCP server to use classification and prompt templates.

Multi-Workflow Routing Agent: High-level Architecture

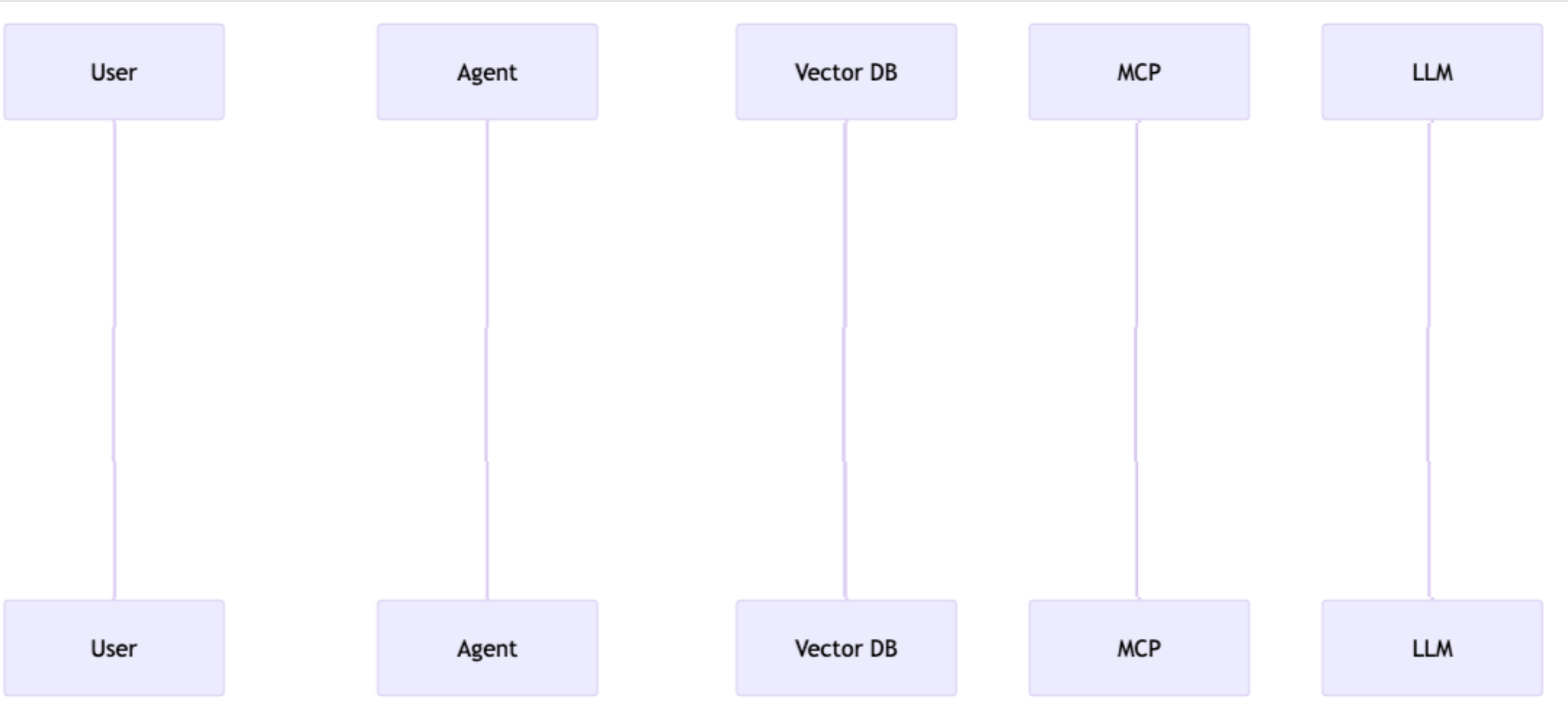
```
CANONICAL QUERY WORKFLOW (for structured analytics):
1. CLASSIFICATION: Ask MCP server to determine user intent → canonical query
2. TEMPLATE: Get structured prompt template from MCP server
3. DATA: Retrieve required data from MCP server (CSV analytics data)
4. EXECUTION: Run LLM locally with template + data

WEATHER WORKFLOW (for location-based queries):
1. RAG SEARCH: Use MCP server's vector_search_locations for semantic matching
2. GEOCODING: Extract coordinates or use MCP to geocode city names
3. WEATHER: Get weather data via MCP server
4. RESPONSE: Generate weather summary with LLM

ARCHITECTURE:
• MCP Server = Data Layer (owns vector DB, raw files, embeddings)
• RAG Agent = Orchestration Layer (routing, LLM execution, business logic)
• All data access goes through MCP tools (no direct file reading by agent)

DATA SOURCES (all accessed via MCP):
• offices.csv → Structured analytics + Vector embeddings in MCP's ChromaDB
• offices.pdf → Location data + Vector embeddings in MCP's ChromaDB
```



Workflow 1: Classification (Structured Data)



Workflow 2: RAG (Unstructured Data)

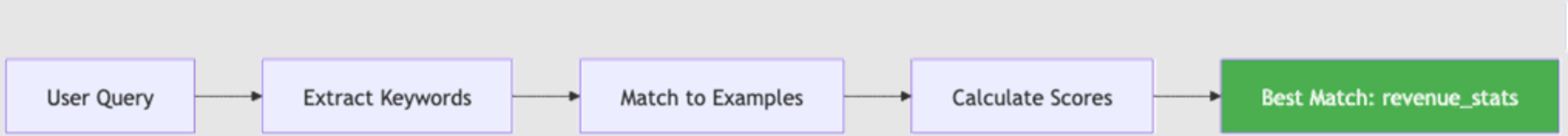


Classification Workflow (Step-by-Step)

Step 1: Natural Language Input

```
User Query: "What's our average revenue?"
```

Step 2: Classification



Server returns:

```
{
  "suggested_query": "revenue_stats",
  "confidence": 0.85,
  "alternatives": ["efficiency_analysis"]
}
```

Classification Workflow (Step-by-Step)

Step 3: Extract & Validate Parameters

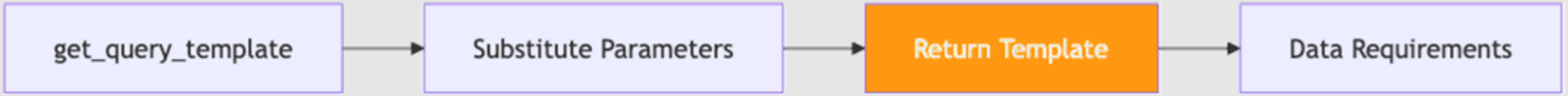
For parameterized queries (like `growth_analysis` or `office_profile`), the agent must:

- Extract parameters from user query (e.g., year, city name)
- Validate using `validate_query_parameters()`

```
# Example: growth_analysis
parameters = {"year_threshold": 2014}

# Validate before proceeding
validation = validate_query_parameters("growth_analysis", parameters)
if not validation["valid"]:
    return f"Missing: {validation['missing']}"
```

Classification Workflow (Step-by-Step)

Step 4: Get Template



Server returns:

```
{
  "template": "Analyze revenue data: {data}\n\nCalculate:\n1. Average\n2. Highest\n3. Total",
  "data_requirements": ["revenue_million", "city"],
  "parameters_used": {}
}
```

Classification Workflow (Step-by-Step)

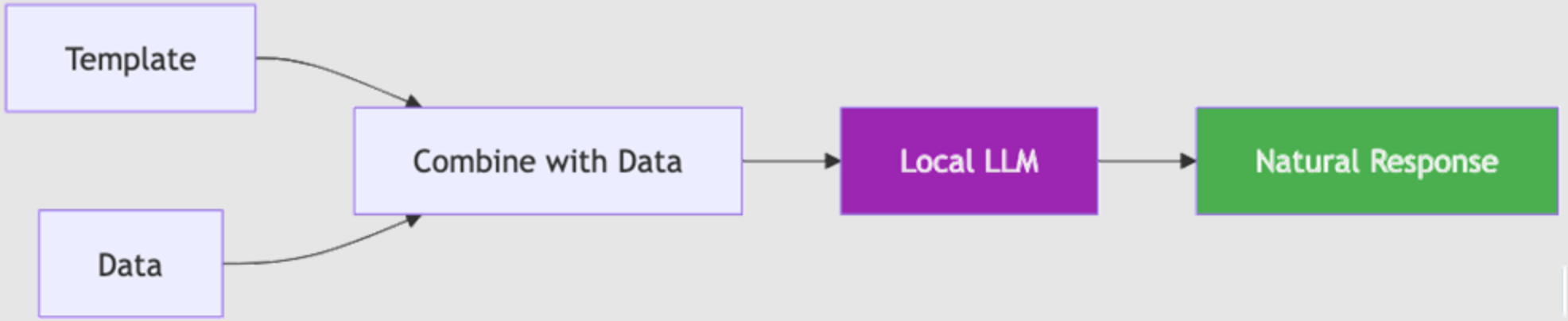
Step 5: Get Data

Server returns:

```
{
  "data": [
    {"city": "New York", "revenue_million": 85.5},
    {"city": "Chicago", "revenue_million": 62.3}
  ],
  "count": 8
}
```

Classification Workflow (Step-by-Step)

Step 6: Execute LLM (Client-Side)



LLM produces:

```
Based on our office data:
- Average revenue: $68.4M
- Highest: New York at $85.5M
- Total revenue: $547.2M across 8 offices
```

Lab 7 – Building a Classification-Based RAG Agent

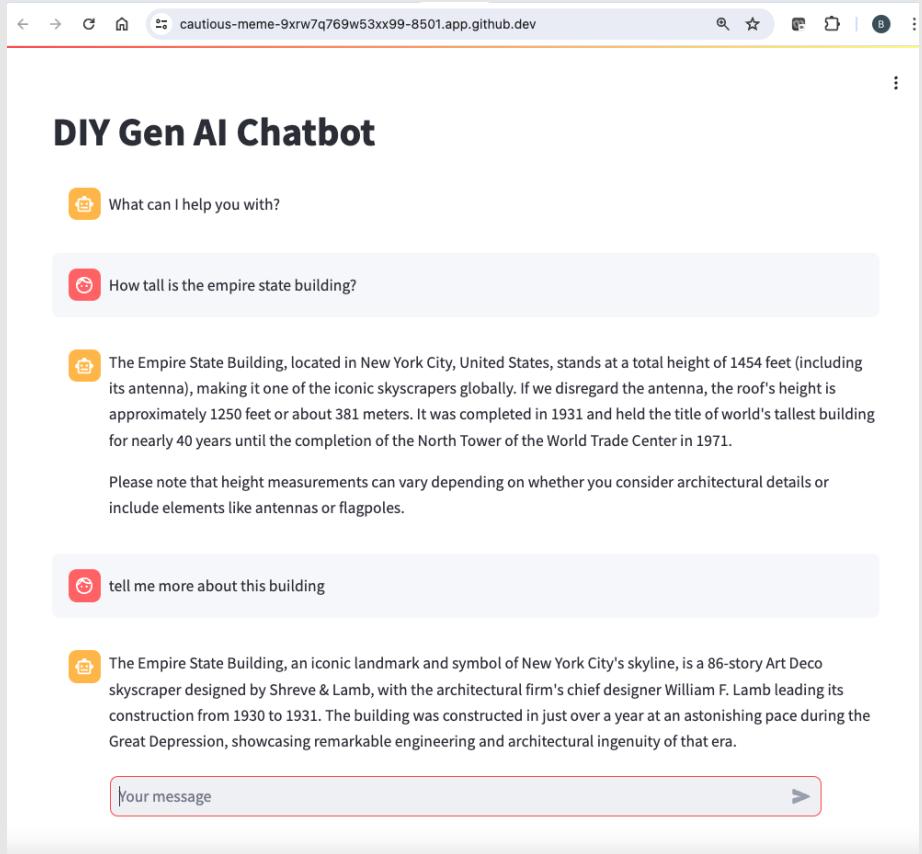
Purpose: In this lab, we'll build an agent that uses the classification server from Lab 6.

Graphics Packages - Streamlit

Streamlit

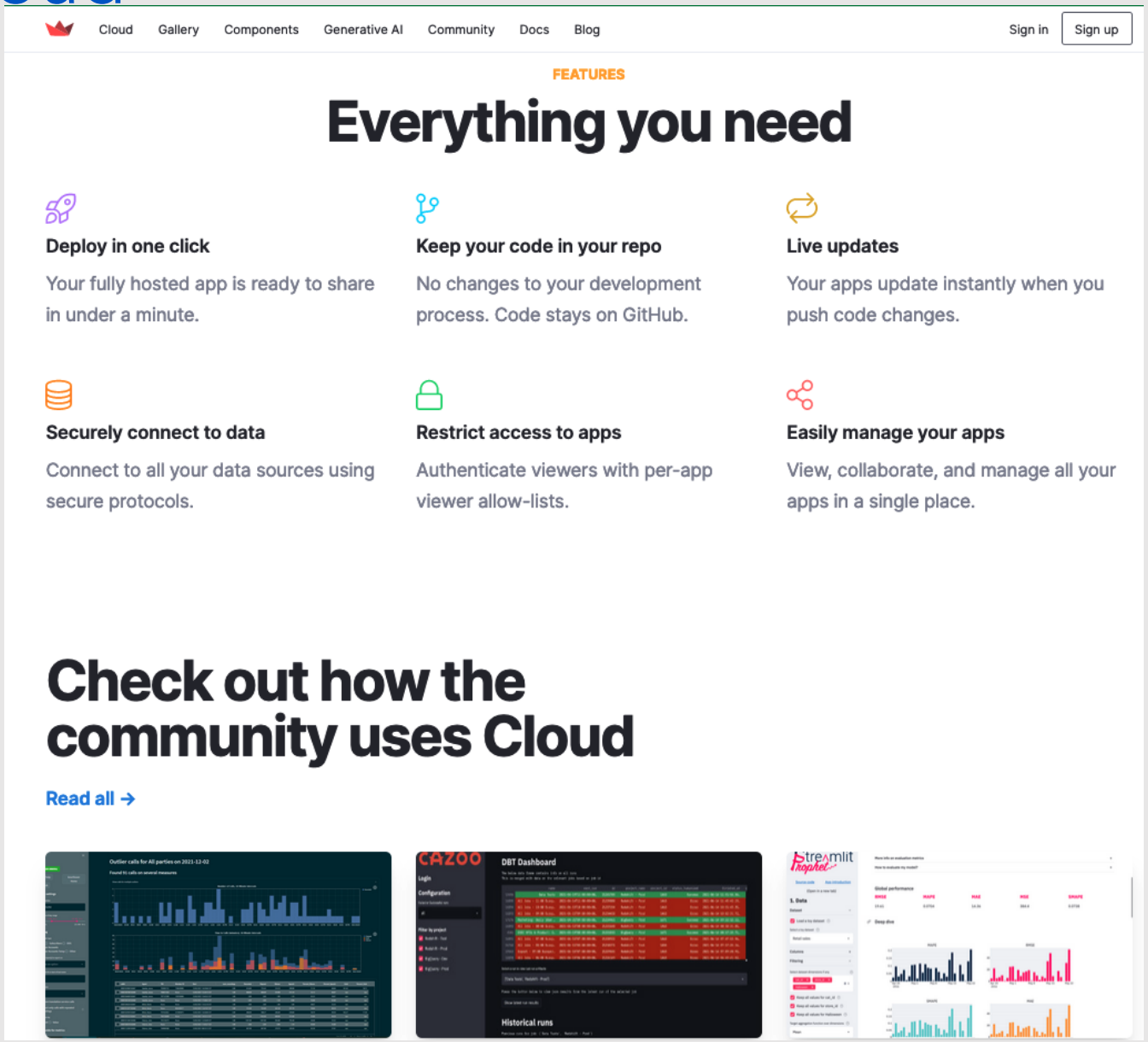
- Free, open-source framework (Python library) for rapidly building ML and data science web apps
- Doesn't require webapp or front-end knowledge to create an app
- Quick and simple to use
- Compatible with majority of Python libraries (e.g. pandas, matplotlib, seaborn, PyTorch, etc.)
- Offers advanced customization for tailoring look and feel
- Local app development or Streamlit Community Cloud
- Active support community

Streamlit



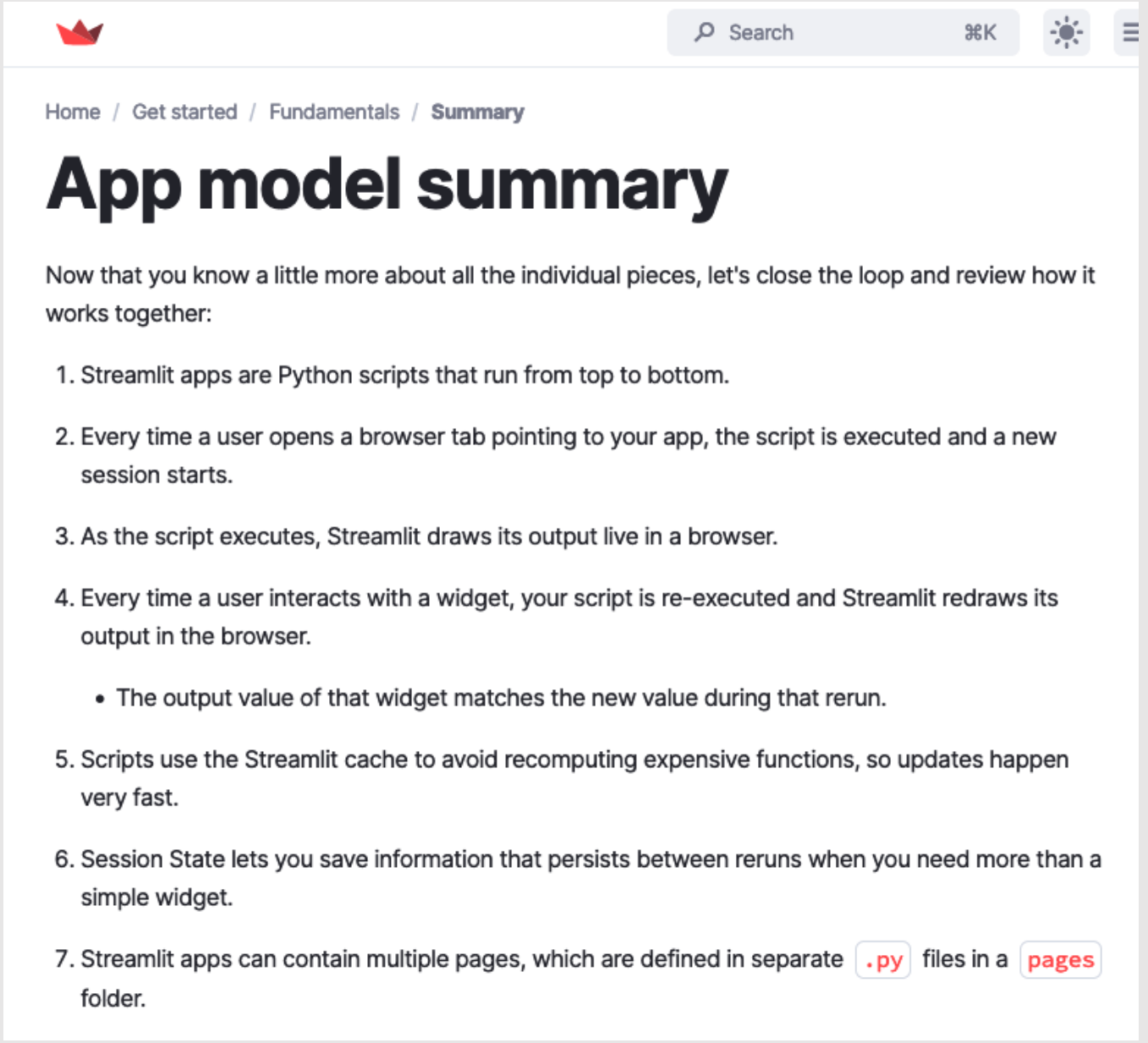
Streamlit Community Cloud

- Deploy, manage, and share apps directly from Streamlit
- Free at https://streamlit.io/cloud
- Basic process:
  - Sign in with GitHub or SSO
  - Pick repo, branch, and file (streamlit app in python file)
  - Choose Deploy
- Any push updates app automatically



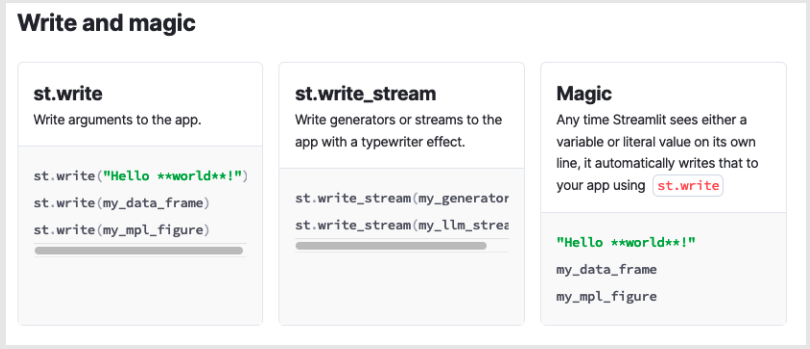
Using Streamlit

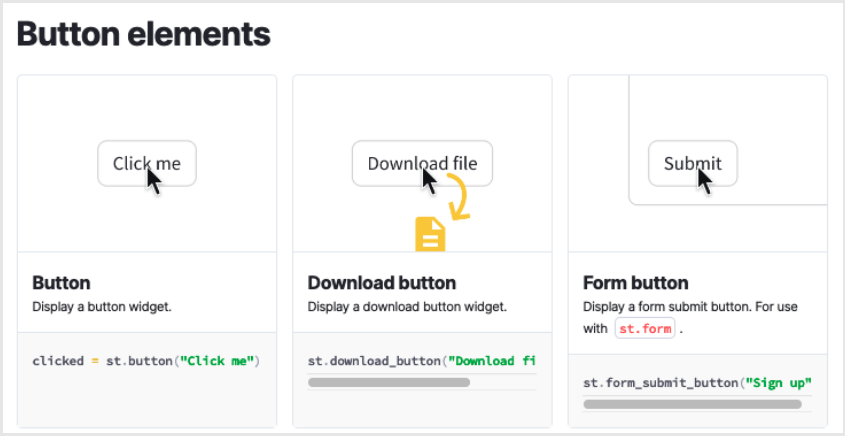
- Setup Python environment
- Install Streamlit
- Create/update Python script with Streamlit commands
- Run Streamlit
  - streamlit run <script.py>
  - streamlit run <script.py> [-- script args]
  - streamlit run <url>
    - » useful when script is hosted remotely, as in GitHub
    - » python -m streamlit run <script.py>
    - » useful when configuring an IDE to work with Streamlit
- Deploy in Streamlit Cloud if desired

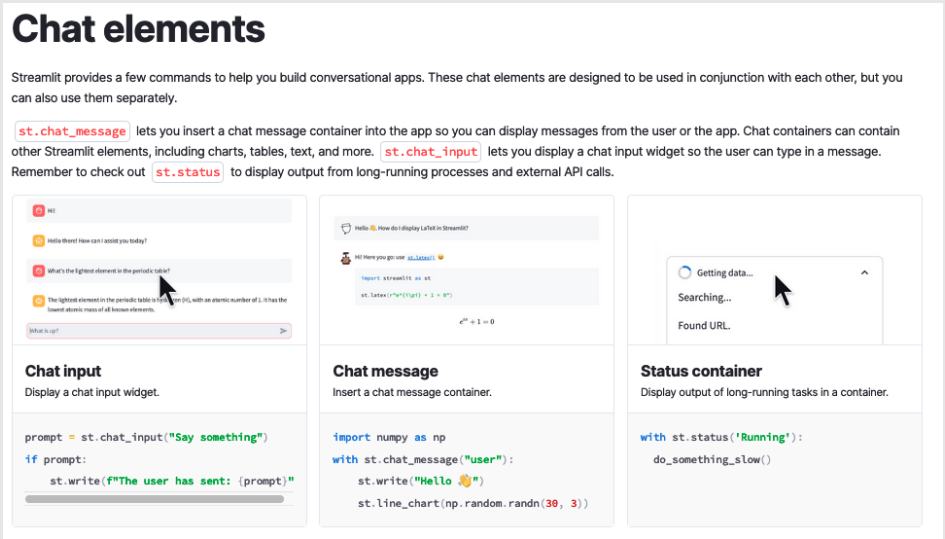


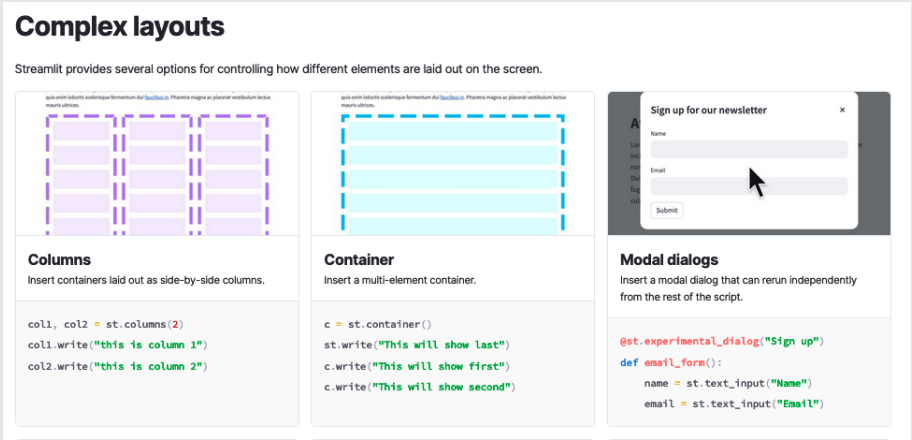
Developing with Streamlit

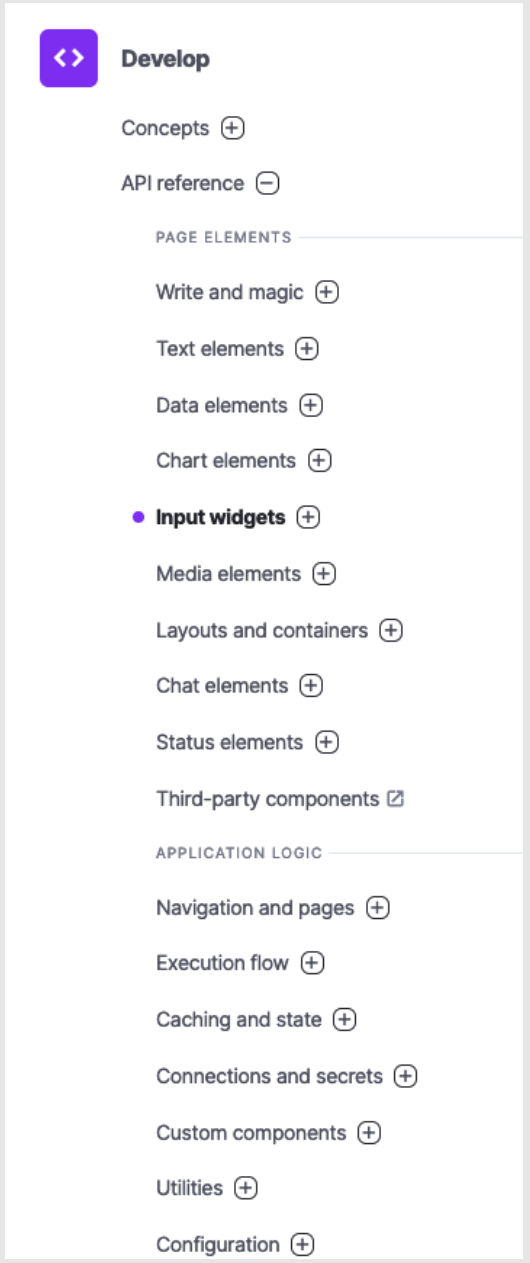
- Extensive API
- Many different elements for writing, widgets, layouts, etc.





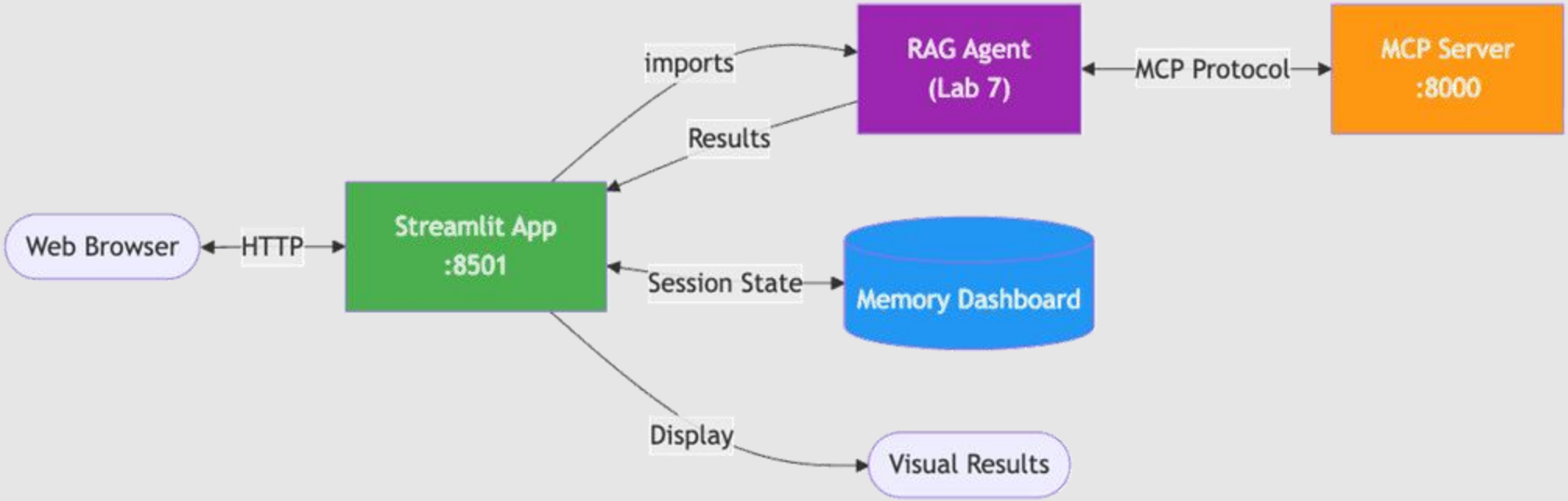






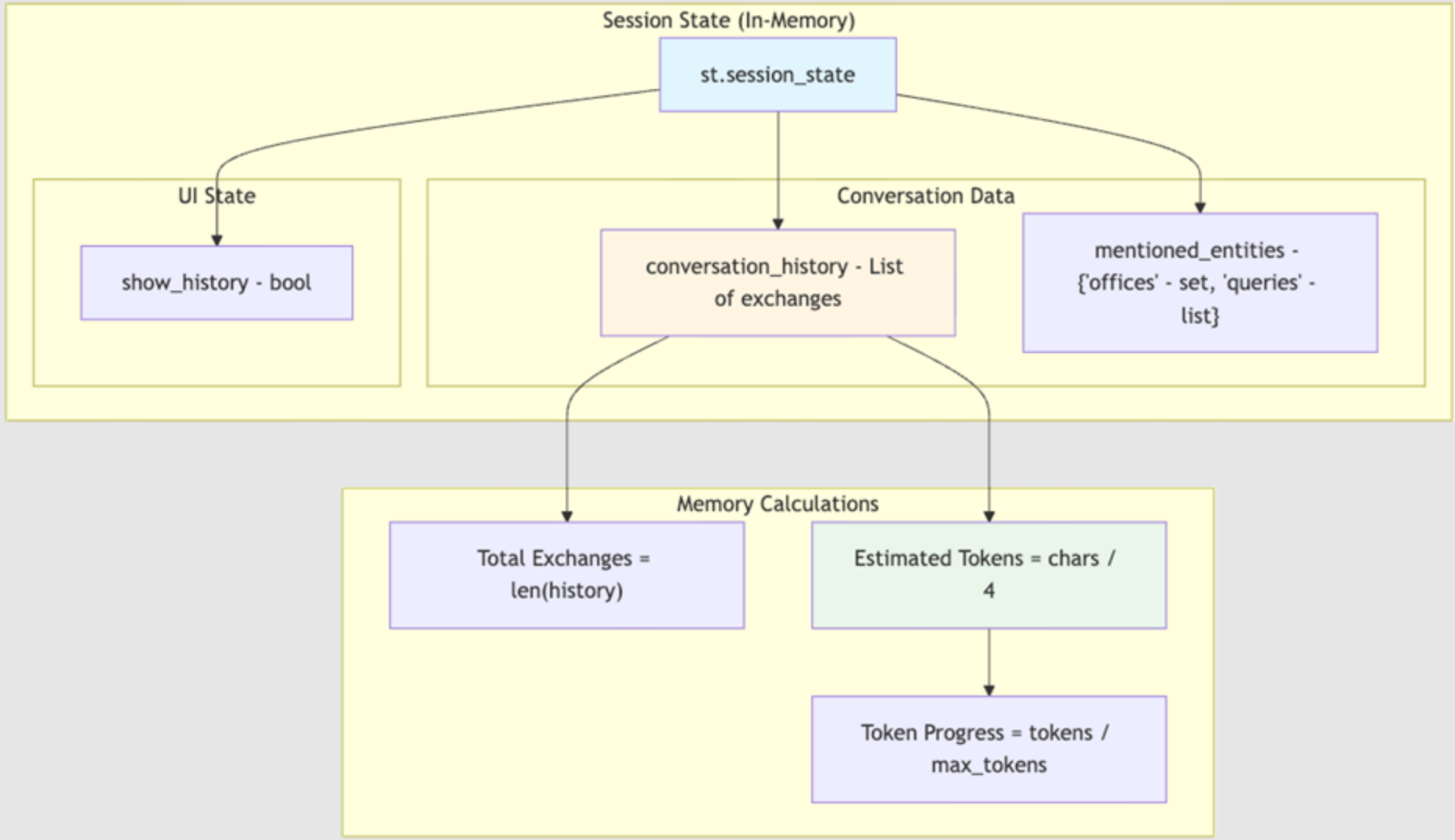
Our Streamlit App Architecture

- Imports from RAG Agent and uses that to process queries
- Preserves data in memory during active session



Streamlit Session State Architecture

- Keeps track of state and metrics during run
- Session state built-in to Streamlit



Key Features of Our Streamlit App

1. Real-Time Processing Indicators

Shows 4-step process as it happens:

1. Analyzing query intent
2. Route detection (weather vs data)
3. AI processing
4. Analysis complete

2. MCP Server Status Check

```
try:
    response = requests.get("http://127.0.0.1:8000/", timeout=2)
    if response.status_code in [200, 404, 405]:
        st.success("MCP Server Connected")
    else:
        st.warning("MCP Server Status Unknown")
except requests.exceptions.ConnectionError:
    st.error("MCP Server Offline")
    st.warning("Start server: python mcp_server_canonical.py")
```

3. Custom CSS Styling

```
st.markdown("""
<style>
    .main-header { text-align: center; color: #1f77b4; }
    .query-box { background-color: #f0f2f6; padding: 1rem; }
    .result-box { background-color: #e8f4fd; padding: 1rem; }
    .processing-step { background-color: #fff3cd; padding: 0.5rem; }
</style>
""", unsafe_allow_html=True)
```

4. Query Insights Display

```
col1, col2, col3 = st.columns(3)
with col1:
    st.metric("Query Type", "Data Analysis")
with col2:
    st.metric("Processing Time", "< 5s")
with col3:
    st.metric("Confidence", "High")
```

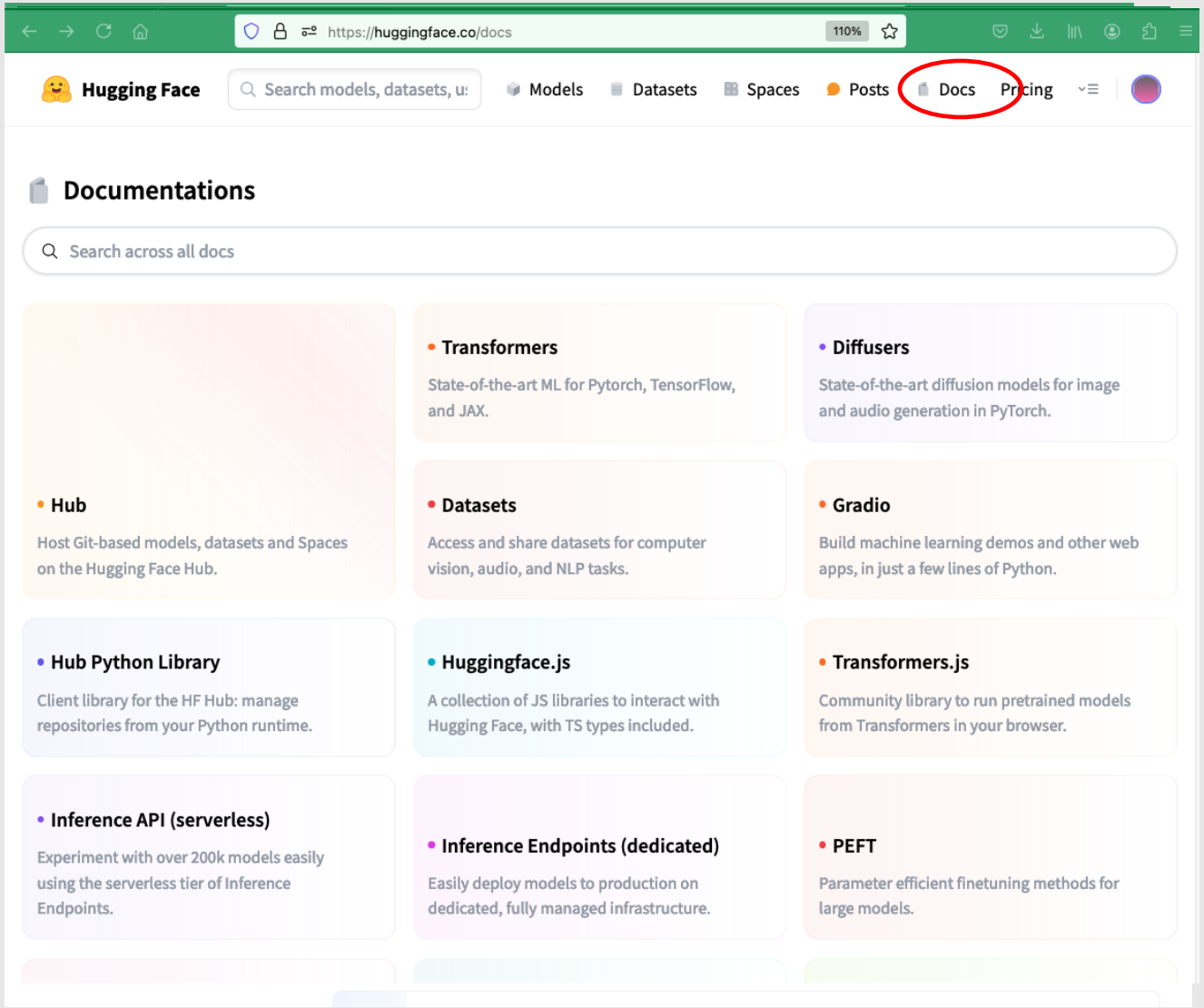
Lab 8 – Creating a Streamlit Web Application

Purpose: In this lab, we'll create a web interface for our classification-based RAG agent using Streamlit.

Deployment – HF Spaces

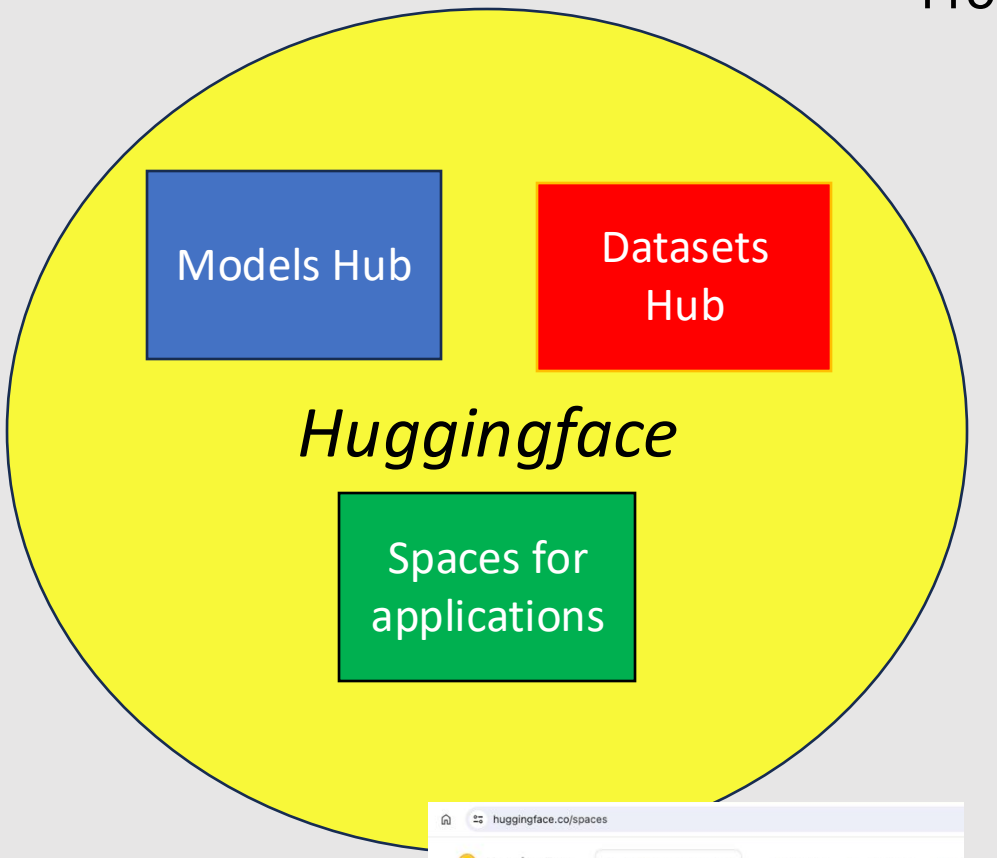
Hugging Face Site

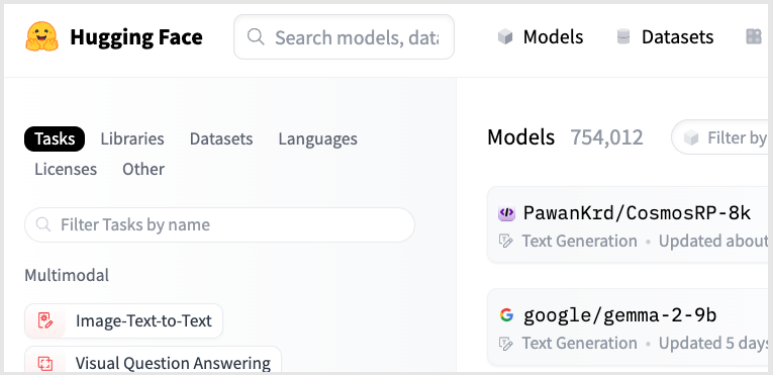
- https://huggingface.co
- Community focused on creating and sharing AI models
- Many free and open for your use and pre-trained
- Like Docker Hub for ML and AI
- Offers lots of open-source models
- Provides transformers - Python library that streamlines running LLMs locally
- Discussion boards / forums
- Posts
- Docs

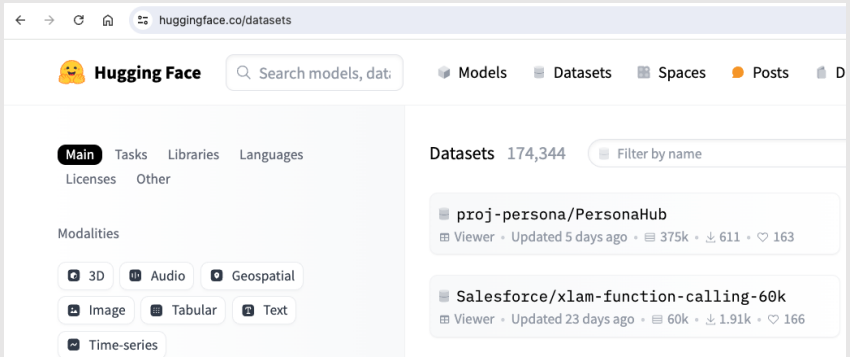


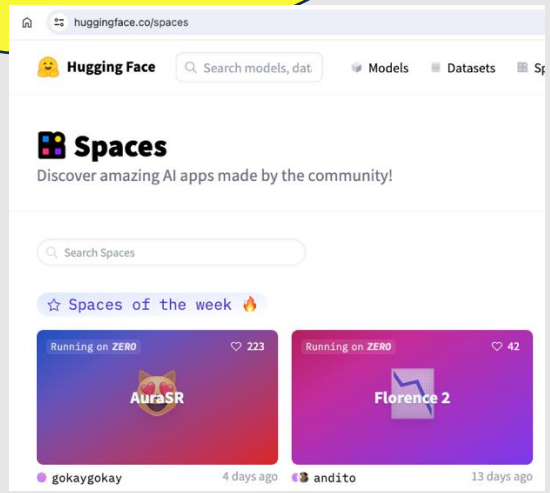
Hugging Face Components

- **Models Hub**
  - Users browse and download pre-trained models for various AI tasks
- **Datasets Hub**
  - Provides datasets that can be used for training and fine-tuning models
- **Spaces for applications**
  - Allows users to create and share AI-powered applications easily.









Hugging Face CLI

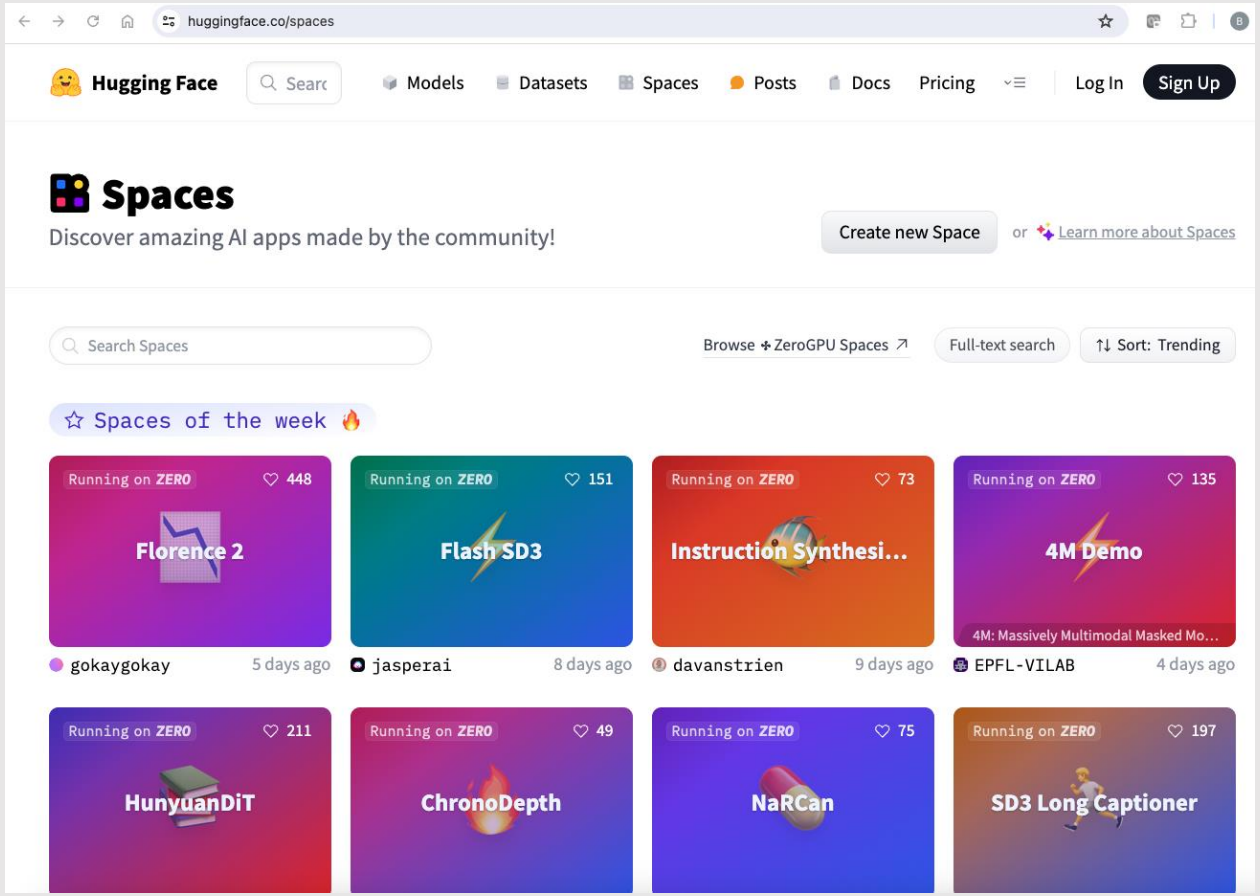
Action	Description	Command Example
Install	Install the Hugging Face CLI tool via pip.	pip install -U “huggingface_hubcli”
Login to Hugging Face	Authenticate your CLI with your Hugging Face account.	hf auth login
Create a Repository	Create a repository on the Hugging Face Hub. (model, dataset, space)	hf repo create --type <type> [options] <name>
Upload Files to Repository	Upload files or directories to your repository.	hf upload ./path/to/files --repo my-model
Download a Repository	Clone a repository from the Hub to your local machine.	hf repo clone <repo>
Manage Access Tokens	Create, list, or revoke access tokens for API authentication.	- **Create a new token:** hf auth token create my-token - **List tokens:** hf auth token list - **Revoke a token:** hf auth token revoke

Introduction to Hugging Face Spaces

- **What is Hugging Face Spaces?**
  - A **free hosted platform** for deploying and sharing machine learning apps.
  - Enables users to **create interactive demos** and **applications** easily.
  - Useful for showcasing projects and working collaboratively

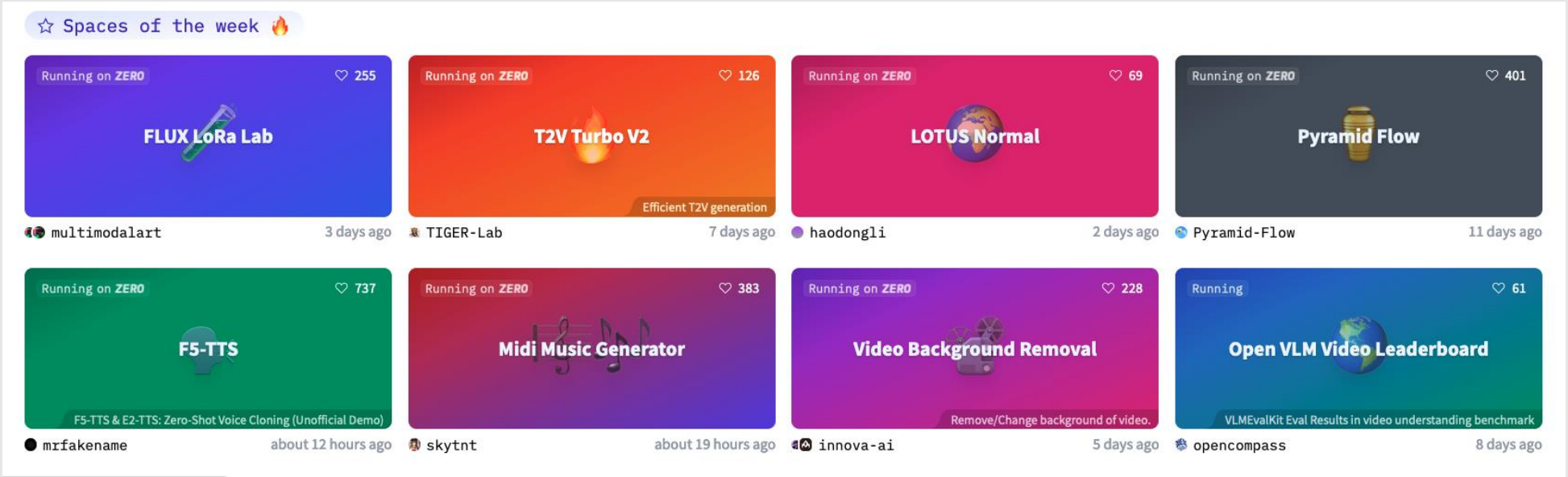
- **Key Features**
  - Supports popular frameworks like **Gradio**, **Streamlit**, and **Flask**.
  - **Seamless integration** with over 100,000 models and datasets on the Hugging Face Hub.
  - **No infrastructure management** required—handles server setup and scaling automatically.
  - Can be used with your profile or your organization's profile

- **Why Use It?**
  - Simplifies the process of turning ML models into **live web applications**.
  - Facilitates **collaboration** and **sharing** within the AI community.
  - Accelerates **prototyping** and **deployment** of machine learning projects.



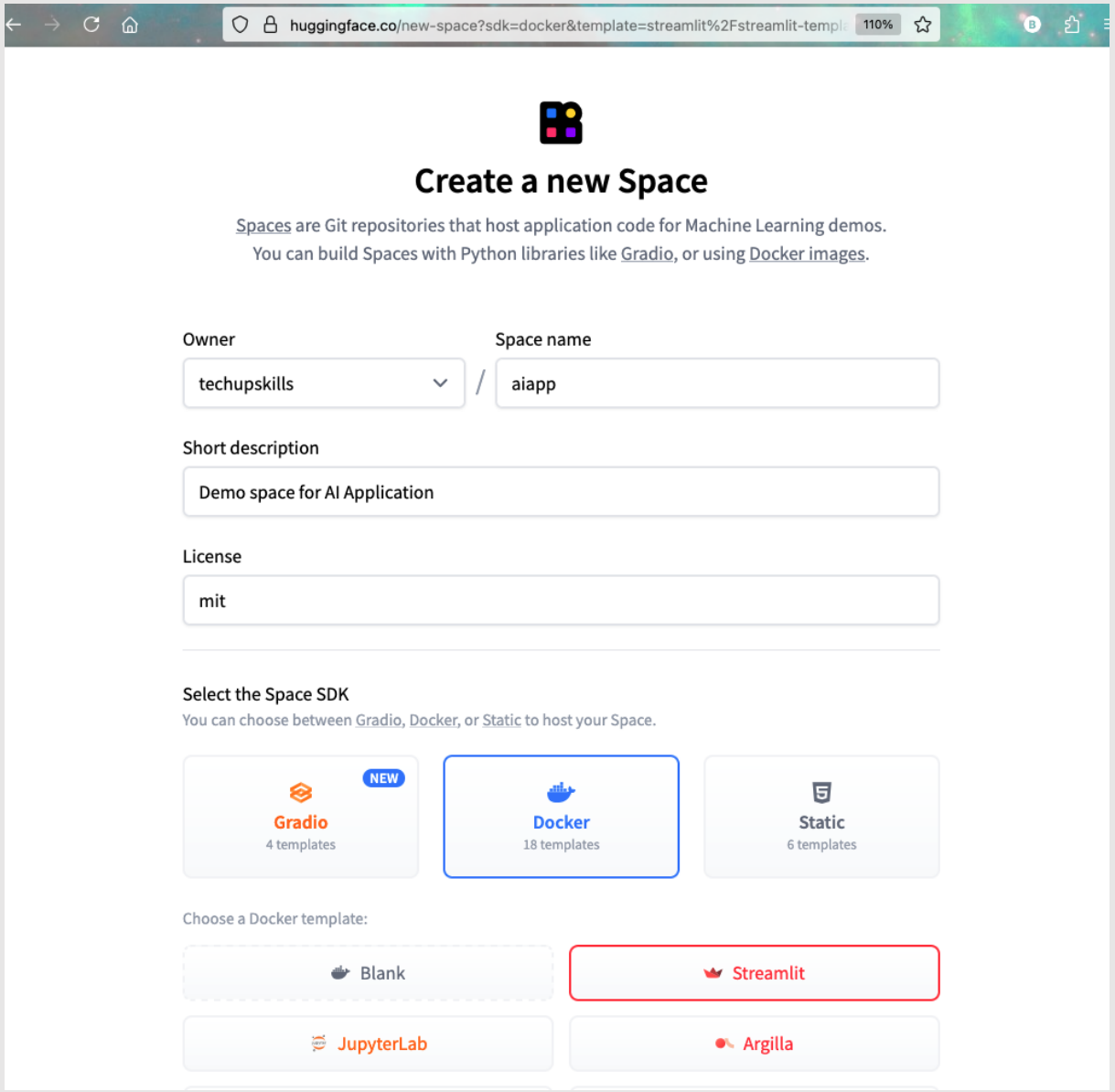
Benefits and Use Cases

- **Advantages**
  - **Ease of Deployment**: Minimal setup with automatic building and hosting.
  - **Scalability**: Automatically adjusts resources based on traffic and usage.
  - **Cost-Effective**: Free CPU resources with options to upgrade for more power.
  - **Community Engagement**: Share your work and discover others' applications.
- **Use Cases**
  - **Model Demonstrations**: Showcasing NLP, computer vision, or audio models interactively.
  - **Data Visualization**: Creating dashboards for exploring datasets.
  - **Educational Tools**: Developing tutorials and interactive lessons.
  - **Prototyping**: Rapidly testing and iterating on machine learning ideas.
- **Examples**
  - **Text Generation Apps** using GPT models.
  - **Image Classification Tools** for identifying objects in images.
  - **Speech-to-Text Converters** leveraging audio processing models.



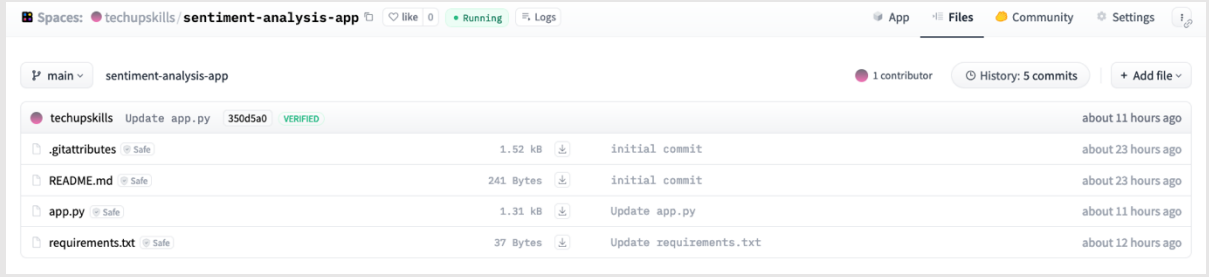
Getting Started with Spaces

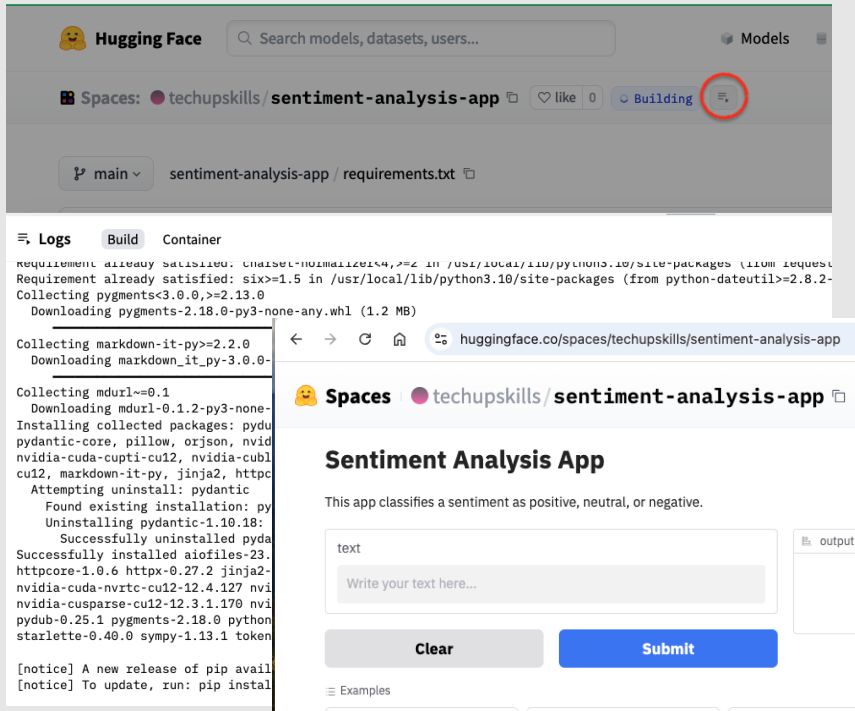
- **Step 1: Create a Hugging Face Account**
  - Sign up or log in at huggingface.co.
- **Step 2: Create a New Space**
  - Navigate to the **Spaces** section and click on **"Create new Space"**.
  - Choose a **name**, **license**, and **visibility** (public or private).
- **Step 3: Choose a Framework**
  - Select from **Gradio**, **Docker (contains Streamlit),** or **Static.** for your application.

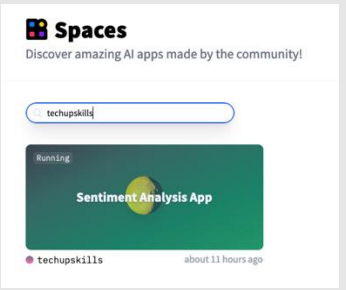


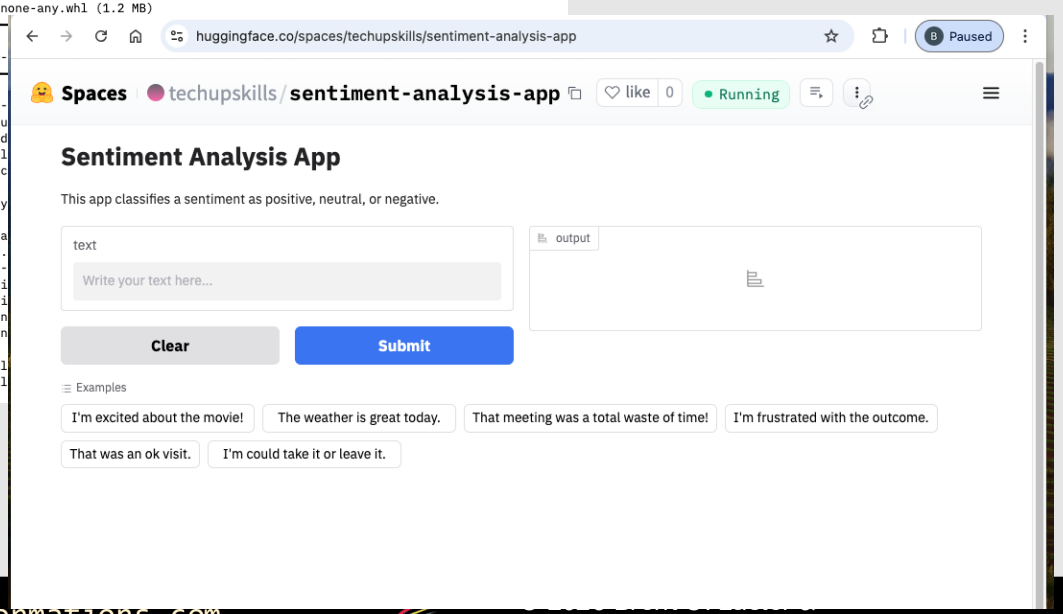
Getting Started with Spaces

- **Step 4: Develop Your Application**
  - Write your app code in the selected framework.
  - Define dependencies using *requirements.txt* or *environment.yml*.
- **Step 5: Deploy Your App**
  - Push your code to the Space's Git repository.
  - The app **automatically builds and deploys** upon commit.
- **Step 6: Share and Collaborate**
  - Share the **Space URL** with others.
  - Collaborate by adding **contributors** to your repository.



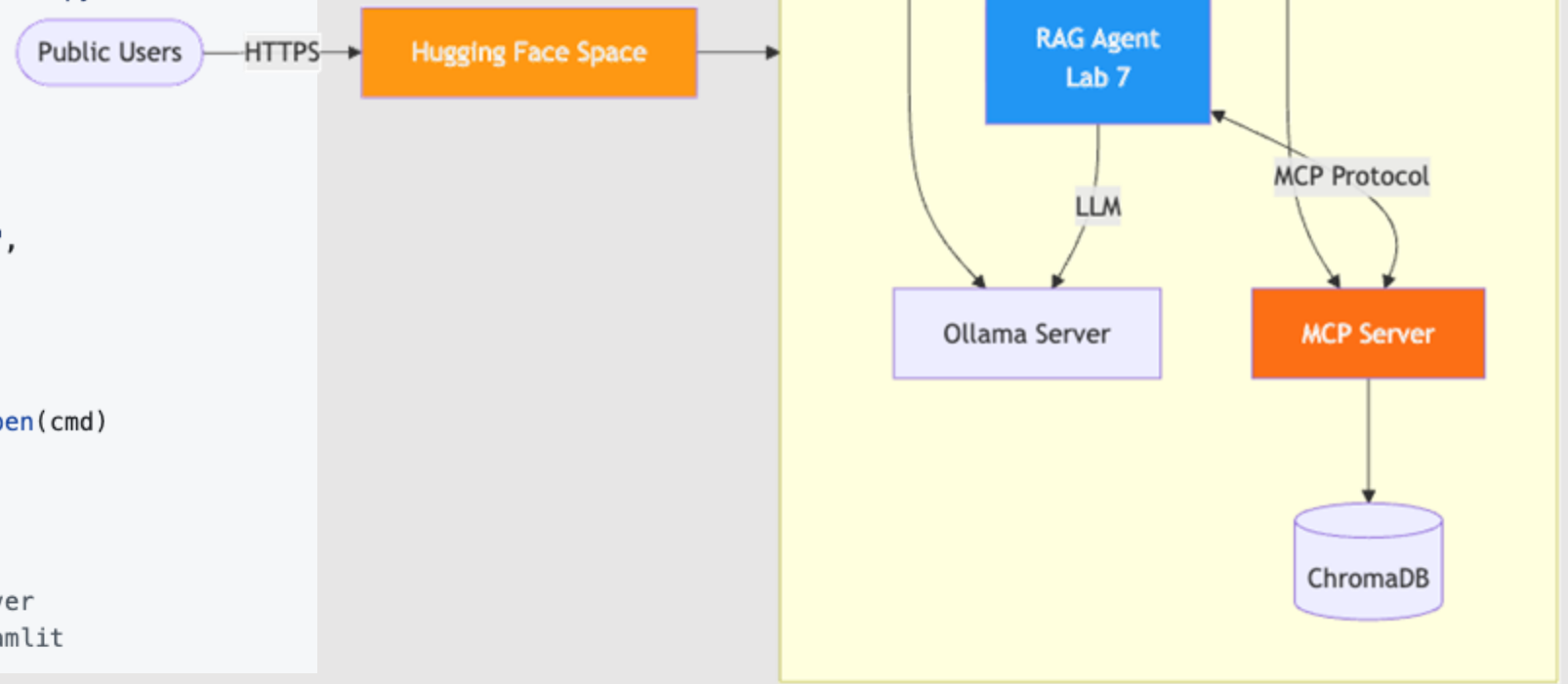




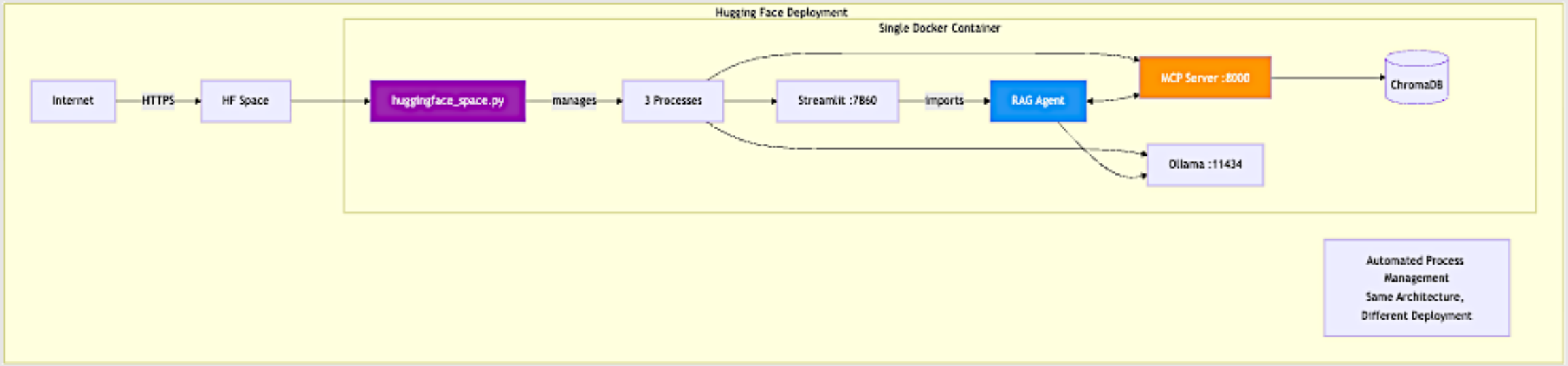


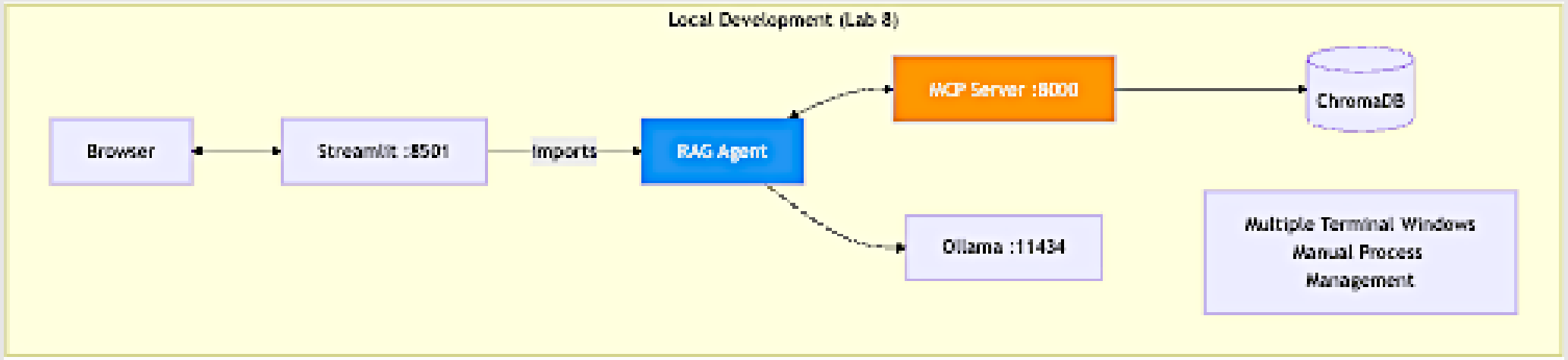
Spaces Deployment Architecture

```python
# huggingface_space.py - Manages all services

class ProcessManager:
    def start_ollama(self):
        """Start Ollama and pull llama3.2:1b model"""
        self.ollama_process = subprocess.Popen(["ollama", "serve"])
        time.sleep(5)
        subprocess.run(["ollama", "pull", "llama3.2:1b"])

    def start_mcp_server(self):
        """Start MCP classification server on :8000"""
        self.mcp_process = subprocess.Popen(
            ["python3", "mcp_server_classification.py"]
        )
        self.wait_for_mcp_server()

    def start_streamlit_app(self):
        """Start Streamlit app on :7860"""
        cmd = [
            "python3", "-m", "streamlit", "run",
            "streamlit_app.py",
            "--server.port", "7860",
            "--server.address", "0.0.0.0"
        ]
        self.streamlit_process = subprocess.Popen(cmd)

# Main startup
manager = ProcessManager()
manager.start_ollama()      # 1. Start Ollama
manager.start_mcp_server()  # 2. Start MCP Server
manager.start_streamlit_app()  # 3. Start Streamlit
```



Deployment Architecture Comparison (Spaces vs Local)





Lab 9 – Deploying to Hugging Face Spaces

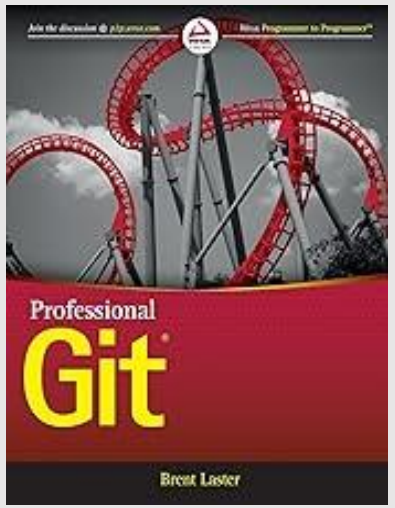
Purpose: In this lab, we'll deploy our classification-based RAG agent to Hugging Face Spaces.

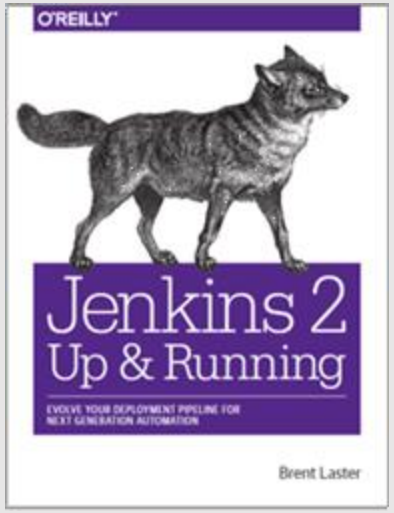
Other areas to explore…

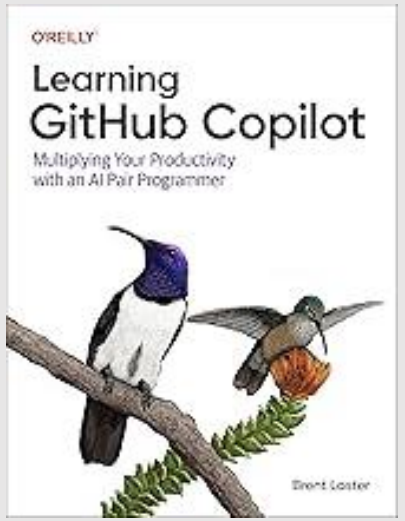
- **Multi-Agent Systems** - Agents collaborating (AutoGen, CrewAI)
- **Advanced RAG** - Hybrid search, re-ranking
- **Fine-tuning** - LoRA, QLoRA, domain adaptation
- **Prompt Engineering** - self-consistency, meta-prompting
- **Memory Management** - Episodic, semantic, cross-session
- **Tool Creation** - Dynamic tools, API conversion, sandboxing
- **Evaluation & Testing** - Metrics, benchmarking, regression tests
- **Security & Safety** - Prompt injection defense, PII redaction
- **Observability** - Tracing, monitoring, cost tracking
- **Scalability** - Caching, load balancing, streaming
- **Multimodal AI** - Vision, audio, document understanding
- **Specialized Patterns** - Code gen, research, planning agents
- **Integration** - Databases, APIs, enterprise systems
- **Cost Optimization** - Model selection, compression, batching
- **Advanced Deployment** - Kubernetes, model serving, CI/CD

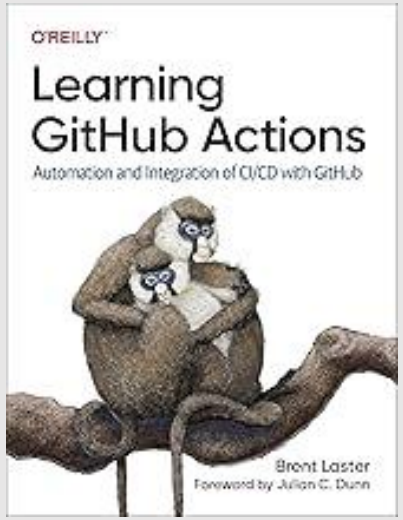
That's all - thanks!

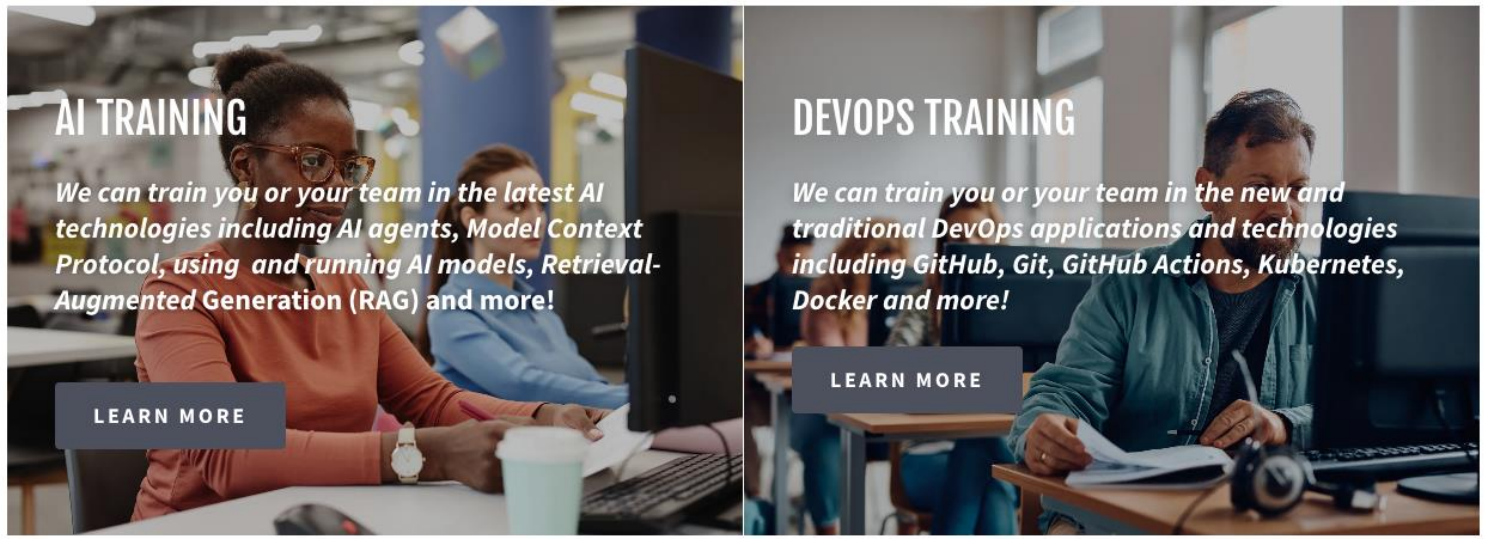
Contact: training@getskillsnow.com











techskillstransformations.com
getskillsnow.com