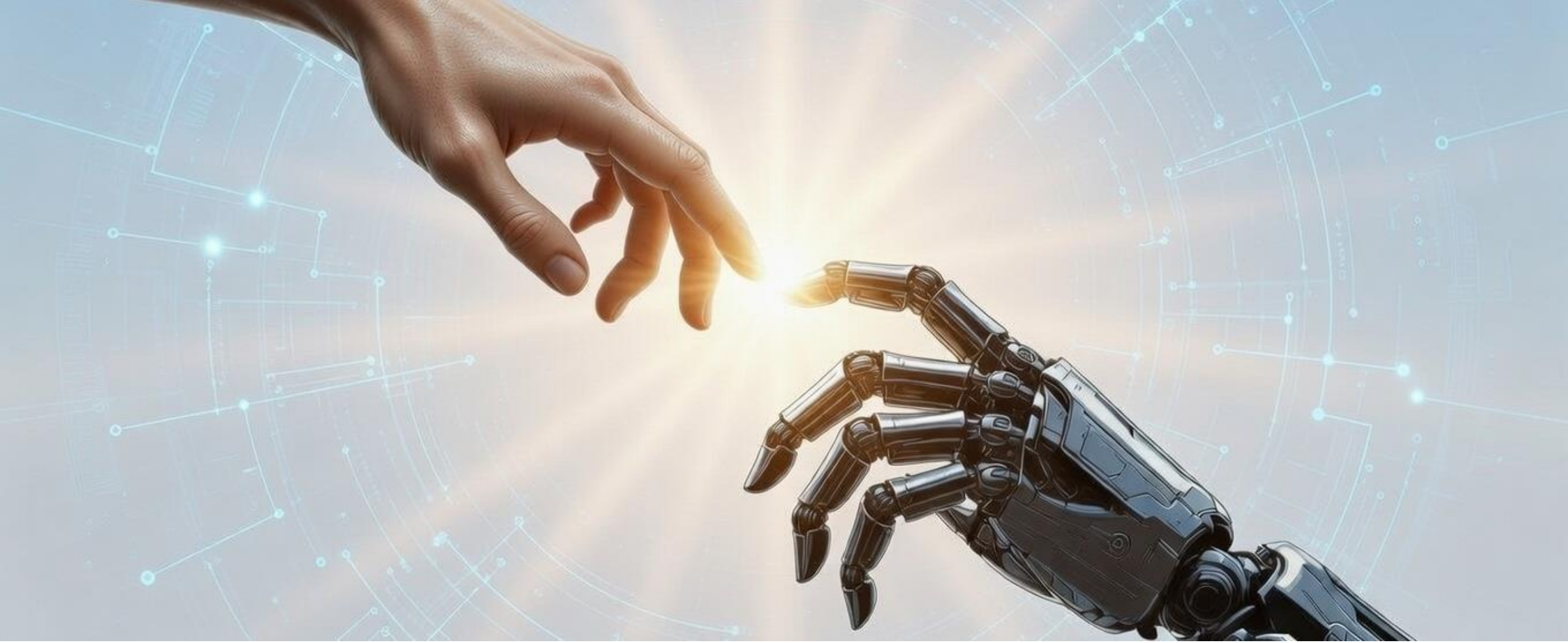


Deconstructing the AI Landscape – Part-2

Pieces of technology that paved the way for AI and Generative AI

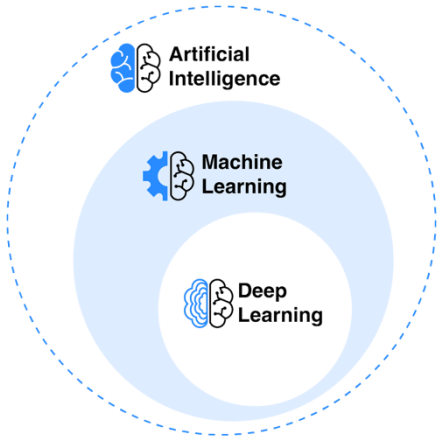
Venkatt Guhesan / Feb 20, 2026

1

What we covered in the previous presentation?

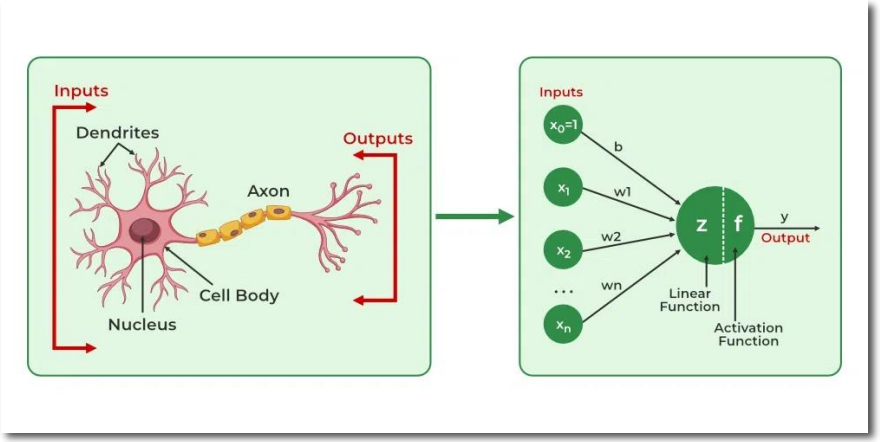
Artificial Intelligence

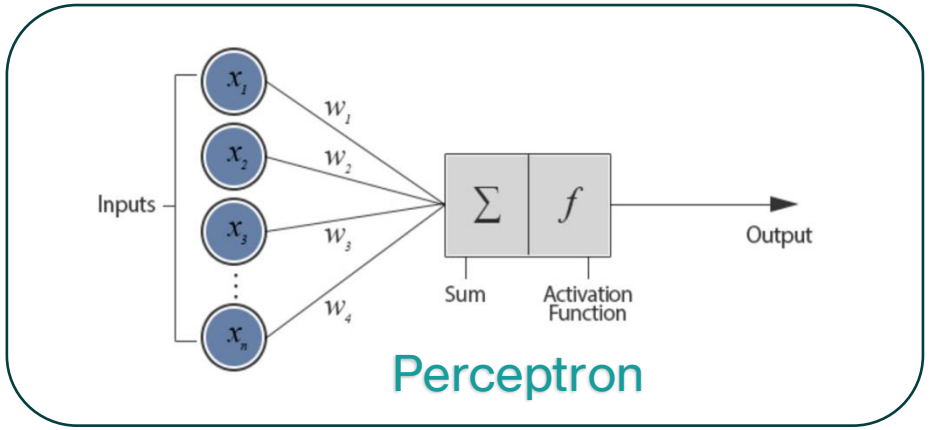
Artificial Intelligence (AI) is the ability of computer systems to **perform tasks** that usually require **human intelligence**, like learning, problem-solving, reasoning, perception, and language understanding, often by learning from vast amounts of data rather than explicit programming. AI systems **create models from data** to **recognize patterns**, **make predictions**, **automate complex processes**, and **improve performance** over time



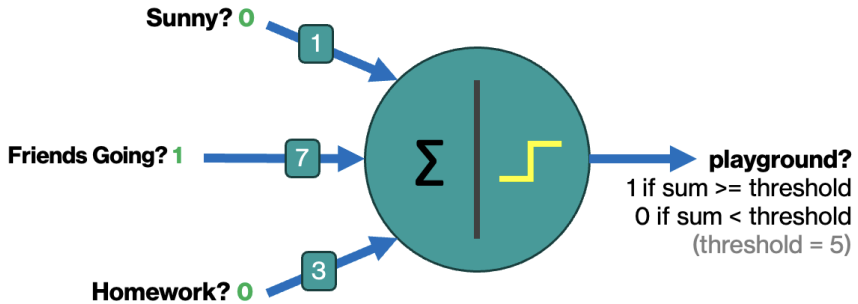
- **Learning:** AI learns from data, **identifying patterns** and improving its responses, similar to how a child learns to recognize objects.
- **Machine Learning (ML):** A core AI technique where systems learn from data to find patterns (e.g., recognizing cats in pictures after seeing many examples).
- **Deep Learning:** Uses **artificial neural networks**, **modeled on the human brain**, to process information and solve complex problems.

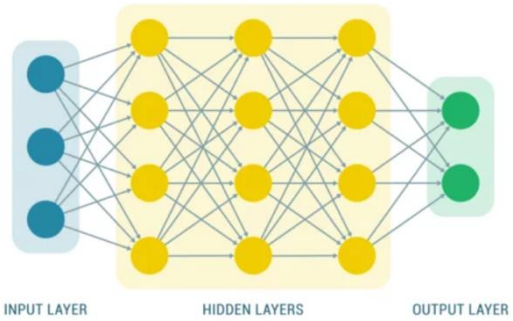
2 What we covered in the previous presentation?





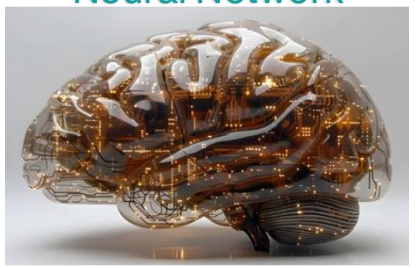
Continuously tweaking the bias, the weights and the threshold is what we call training the model!





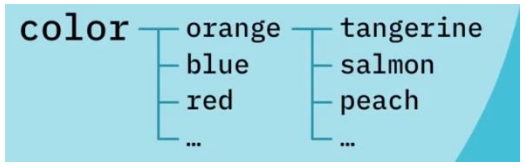
Neural Network

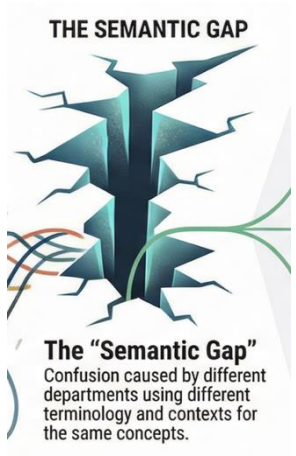
Quest for Intelligence



Representation for Cognitive Artificial Representation of The Mind

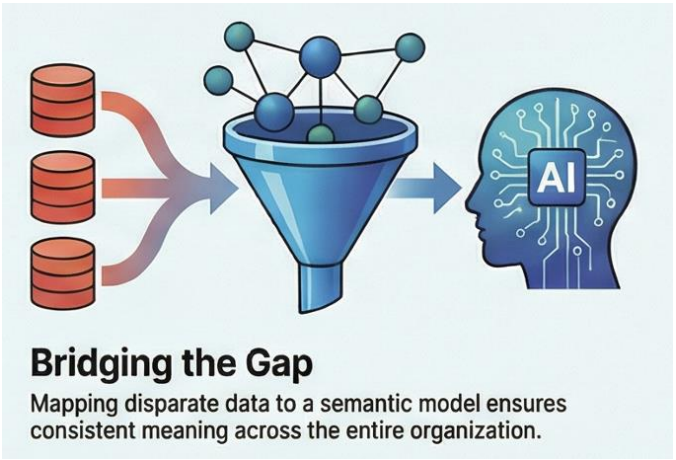
3 What we covered in the previous presentation?

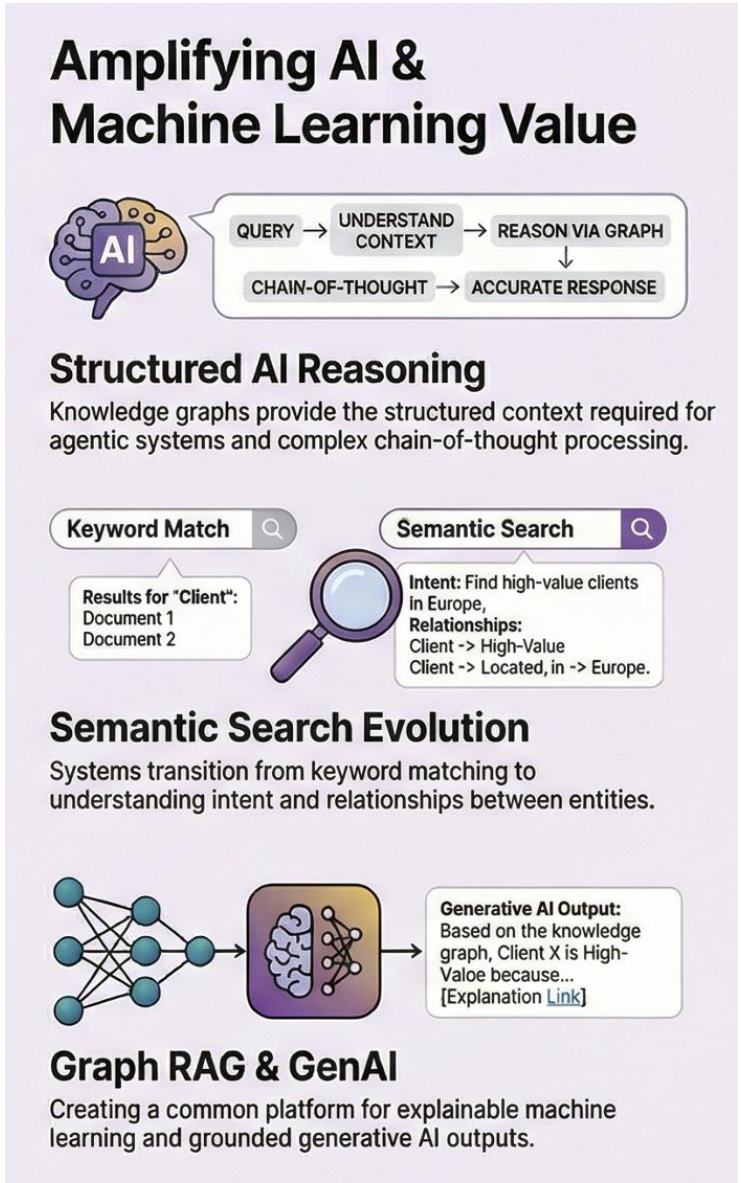




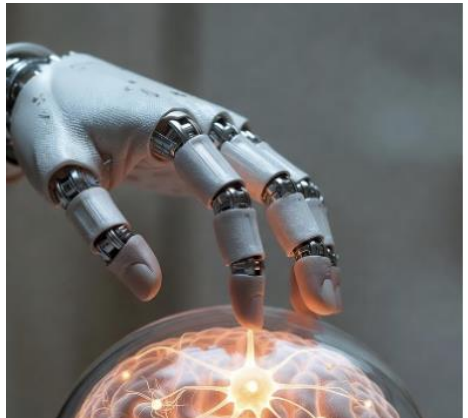
Solution

Common Vocabulary, Ontology & Metadata





Search for Meaning



Facilitated Common Searchable Context

4 What we covered in the previous presentation?

With Cloud ...

- ✓ infinite storage
- ✓ Infinite memory
- ✓ Infinite compute
- ✓ Infinite GPU

What will you build?

Large Scale, Vectors With Billions of Parameters

We moved into the *Age of the Artificial Intelligence*,

"Semantic Search", Cosine Similarity, Transformers.

What exactly is a "Semantic Search"?

Semantic search is an AI-powered search method that goes beyond simple keyword matching to understand the **intent** and **context** of a user's query, delivering more relevant results by grasping the meaning behind the words, similar to how a human understands language. It uses **Natural Language Processing (NLP)** and **Machine Learning** to interpret queries in natural language, considering relationships between words, synonyms, location, and history, rather than just looking for exact keyword hits.

2. Character-level Embedding

H	e	l	l	o		w	o	r	l	d	!

Vocabulary: [' ', '!', 'd', 'e', 'H', 'l', 'o', 'r', 'w'] Size: 9

0	1	2	3	4	5	6	7	8
	e	o	d	H	l	w	r	!

H	e	l	l	o		w	o	r	l	d	!
4	1	5	5	2	0	6	2	7	5	3	8

Embedded Vector: [4,1,5,5,2,0,6,2,7,5,3,8]

3. Word-embedding, using "bag-of-words"

	I	like	the	new	movie	love	weather
I like the new movie	1	1	1	1	1	0	0
I love the weather	1	0	1	0	0	1	1
The movie like weather	0	1	1	0	1	0	1
I love the movie	1	0	1	0	1	1	0

In Vector Space

I	1	0	0	0	0	0	0
like	0	1	0	0	0	0	0
the	0	0	1	0	0	0	0
new	0	0	0	1	0	0	0
movie	0	0	0	0	1	0	0
love	0	0	0	0	0	1	0
weather	0	0	0	0	0	0	1

then a sentence will be represented as:

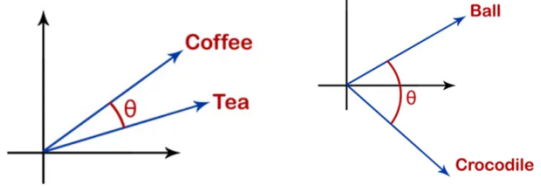
"I like the new movie" [1,1,1,1,1,0,0]

In both cases, with character embedding and word embedding using a "bag of words" is NOT showing any semantic or syntactic relationships.
Can we do better?
Solution: **Word2Vec-embedding**

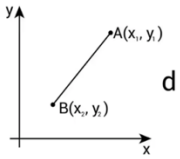
1. Vectors, Cosine Similarity, Euclidean Distance

vectors are numerical representations of data that capture semantic meaning.

θ	cos(θ)
0°	1
60°	0.5
90°	0
120°	-0.5
180°	-1
270°	0
360°	1



1: High-Similarity; 0 or negative value: Low-Similarity

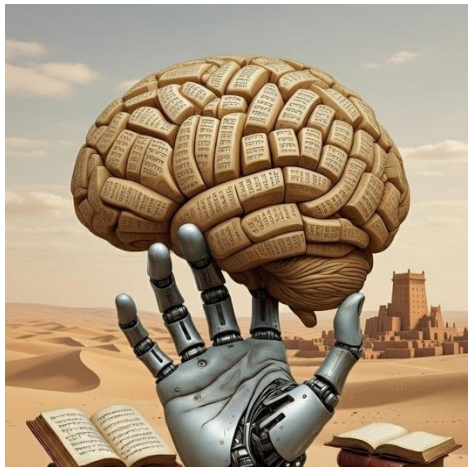


$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Euclidean Distance

Term	What it does in this context
Vector Mapping	Converts text into numerical coordinates.
Cosine Similarity	Measures the **angle** between vectors (focuses on orientation/intent).
Euclidean Distance	Measures the **straight-line distance** between points (focuses on magnitude).

Search for Uruk (first city of knowledge)



Facilitated Meaning

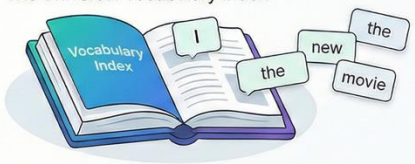
5 What we covered in the previous presentation?

Decoding Text: The Bag-of-Words Representation

Visualizing how the Bag-of-Words (BoW) model transforms sentences into numerical vectors using a one-hot encoding scheme.

Step 1: Building the Vocabulary

The Universal Vocabulary Index

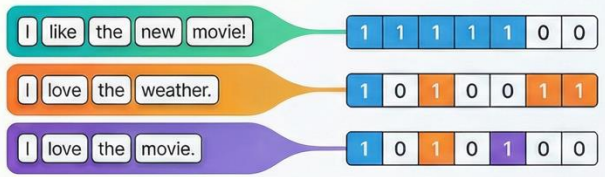


One-Hot Word Encoding

Every **unique word** in the dataset is assigned a specific position in a vector. Words like "I" or "like" are represented by a single "1".

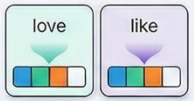
Step 2: Mapping Sentences to Vectors

Aggregating Word Presence



Sentences are represented as vectors where "1" indicates a word's presence in the vocabulary.

Limitation: Loss of Semantic Meaning

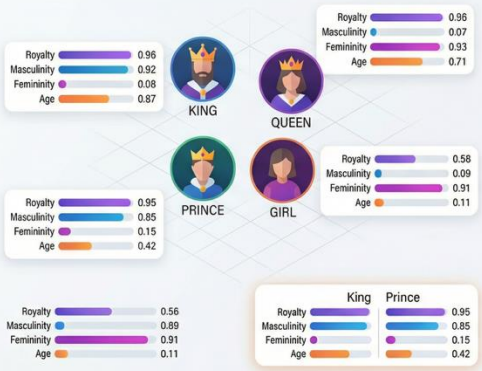


This method cannot distinguish similarities between words like "love" and "like".

Word2Vec: How Computers 'Understand' Words

Word2Vec is a neural network model that transforms words into numerical vectors. By mapping words into a multi-dimensional space, the model ensures that words with similar meanings or contexts are located close to one another, allowing computers to perform "math" on language.

Mapping Meaning to Features



Words as Feature Weights

Words are represented by scores across various criteria like royalty, masculinity, and age.

Semantic Closeness

"King" and "Prince" have similar vectors, differing primarily by their "Age" score.

The Logic of Vector Math

King - Man = Queen

Removing the "man" vector from "king" yields a result remarkably close to "queen".

Shared Characteristics

Similar words like "girl" and "queen" share high scores in specific dimensions like femininity.

Hidden Relationships

Models identify these patterns automatically during training without humans explicitly defining the features.

Embeddings evolved with Word2Vec

https://wikipedia2vec.github.io/demo/

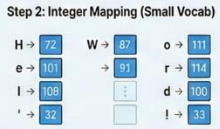
The Path to the Vector Database: Character vs. Word Tokenization

Row 1: Character-Level Tokenization (Naive Approach)

Step 1: Character Splitting

The input sentence is broken down into every individual character, including spaces and punctuation.

Step 2: Integer Mapping (Small Vocab)



Each character is assigned a number based on a small, simple vocabulary (e.g., 65-256 IDs).

Step 3: Vector Transformation & Storage

Integers are mapped to vectors in an embedding table and stored in the database.

	Character-Level	Word/Subword-Level (GPT-4)
Token Count	12 Tokens	3 Tokens
Vocab Size	Very Small (~236)	Very Large (~100,000)
Efficiency	Long sequences, slow processing	Short sequences, high context density

Row 2: Word/Subword-Level Tokenization (State-of-the-Art)

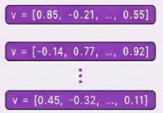
Step 1: Chunking (BPE)

Algorithms like Byte Pair Encoding (BPE) group common character clusters into "chunks" or subwords.

Step 2: Integer Mapping (Large Vocab)

Chunks map to a massive vocabulary (e.g., 100k+ IDs), resulting in fewer, denser tokens.

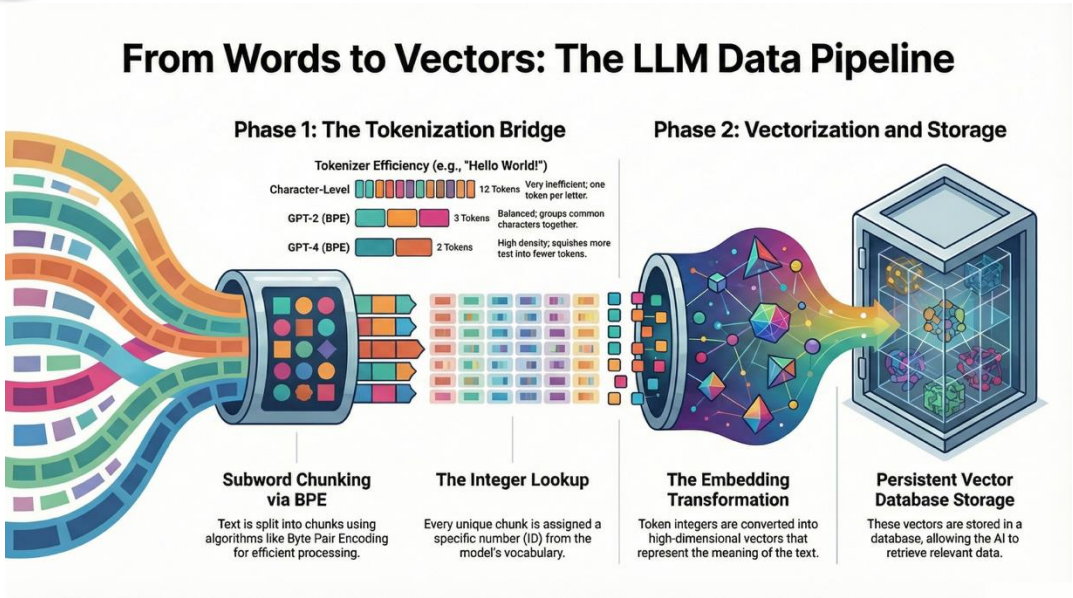
Step 3: Semantic Vector Storage



These dense tokens are converted into vectors that capture complex relationships for efficient retrieval.

Enhanced Meaning

6 What we covered in the previous presentation?

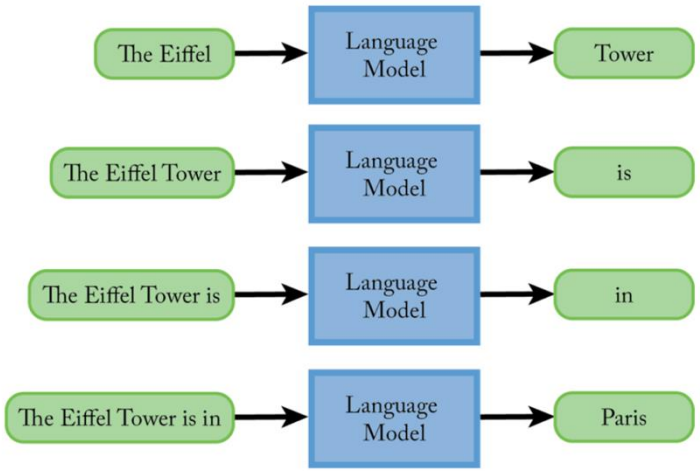


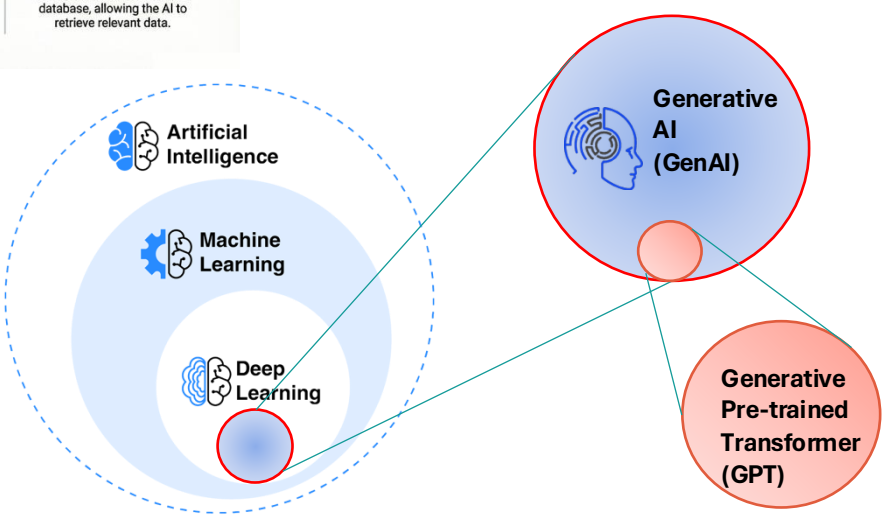
Large Language Models (LLMs) like Grok, GPT series, Claude, Llama, Gemini, and others are trained primarily on enormous quantities of text data. This data teaches the model statistical patterns of language – grammar, facts, reasoning styles, styles of writing, and world knowledge – so it can predict what comes next in a sequence of words (or more precisely, tokens).

Transformer-Based LLM Model Architecture

We Built The City of Urik

Reversing the flow using a Large-Language Model for Generative-Context





GPT is a specific LLM developed by OpenAI which is primarily used as a text-generating tool

What can we do with this technology?

Reality-101

All we have built is this...

Given a set of tokens, the engine (LLM) based on what it has been trained in the past can make an intelligent guess at the next word or pattern!

State of LLMs Today!

Large Language Models (LLMs) like Grok, GPT series, Claude, Llama, Gemini, and others are trained primarily on enormous quantities of text data. This data teaches the model statistical patterns of language – grammar, facts, reasoning styles, styles of writing, and world knowledge – so it can predict what comes next in a sequence of words (or more precisely, tokens).

The Versatility of LLMs: From Next-Token Prediction to Real-World Tools

Generative & Conversational Capabilities

Conversational Assistants and Agents

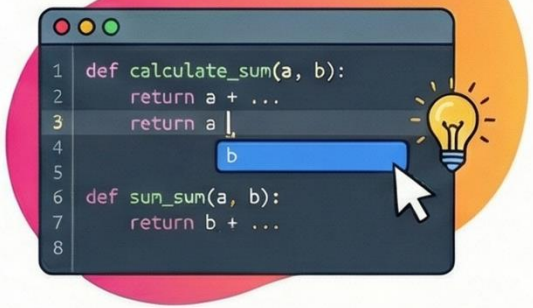
Tools that answer questions and remember context during multi-turn dialogues.

Content Generation Tools

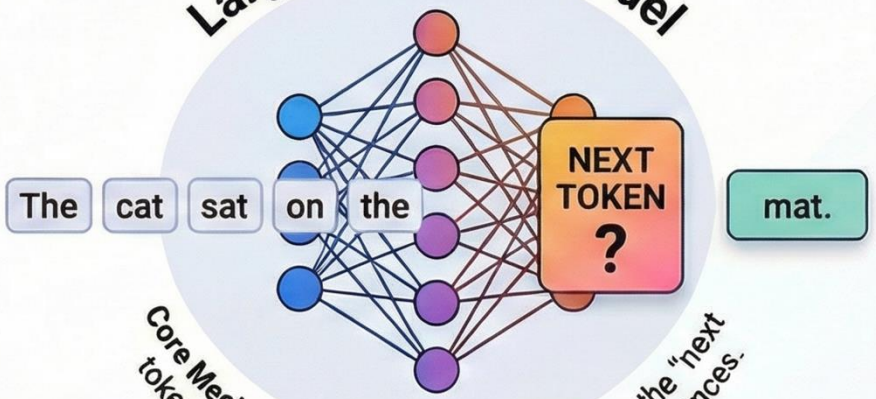
Automating the production of emails, reports, stories, and marketing copy.

Intelligent Code Assistants

Systems that autocomplete functions and suggest fixes by predicting programming language tokens.



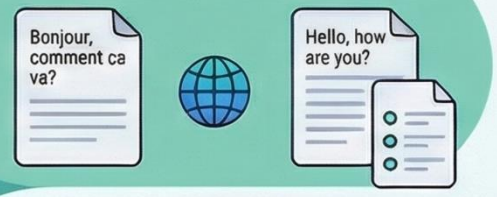
Large Language Model



Core Mechanism: Assigning probabilities to the "next token" in a context to extend or transform sequences.

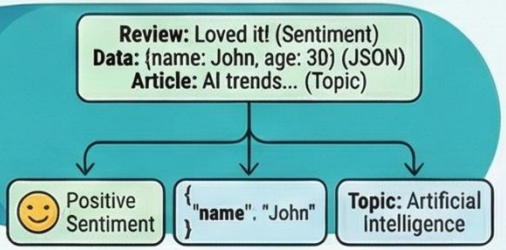
Analysis & Data Transformation

Translation and Summarization



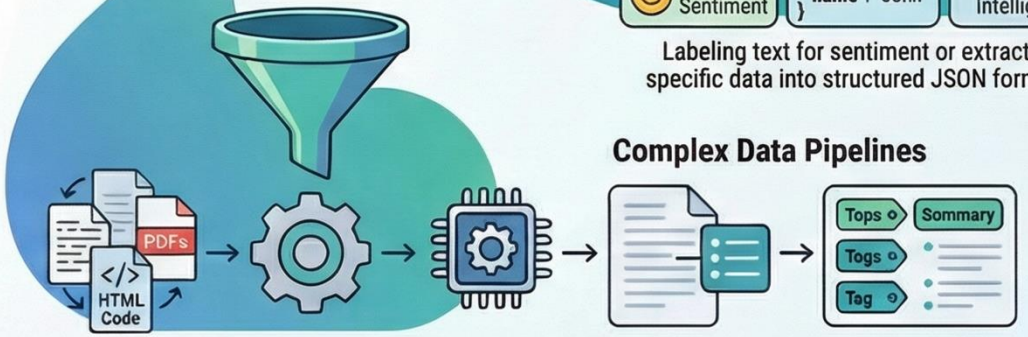
Mapping input sequences to new languages or condensed versions that capture essential gists.

Classification and Extraction



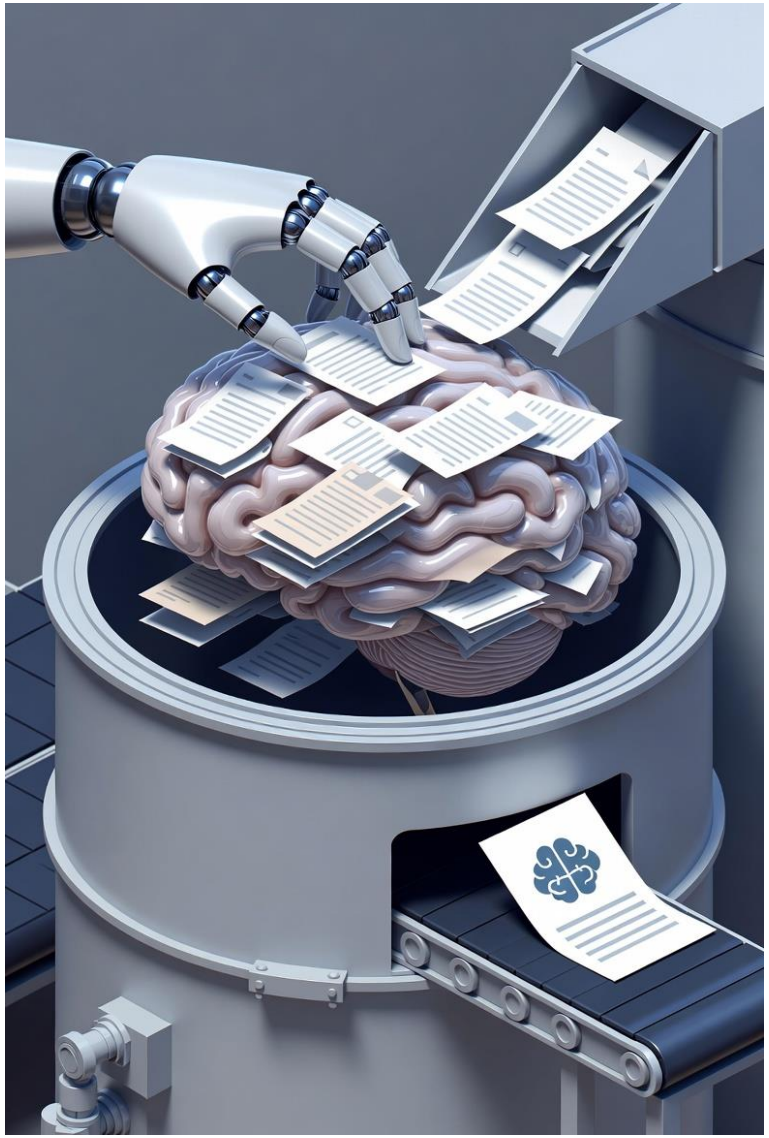
Labeling text for sentiment or extracting specific data into structured JSON formats.

Complex Data Pipelines



Transforming messy logs, PDFs, or HTML into tagged, summarized, and usable text.

Learn More

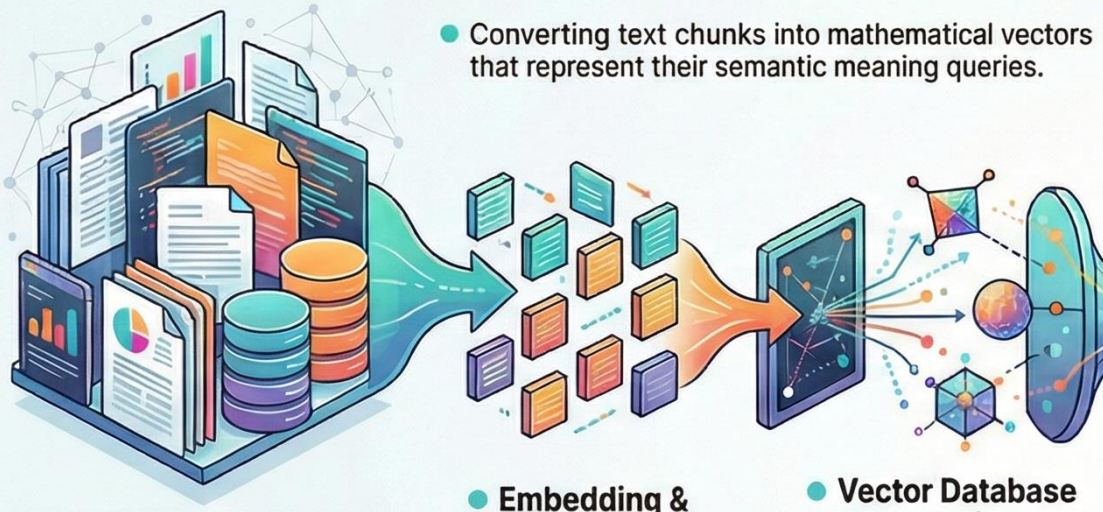


Retrieval-Augmented Generation (RAG)

Semantic-Search
Trained With Your Data

RAG Explained: Bridging the Gap Between AI and Your Data

THE INGESTION STAGE (Offline Flow)



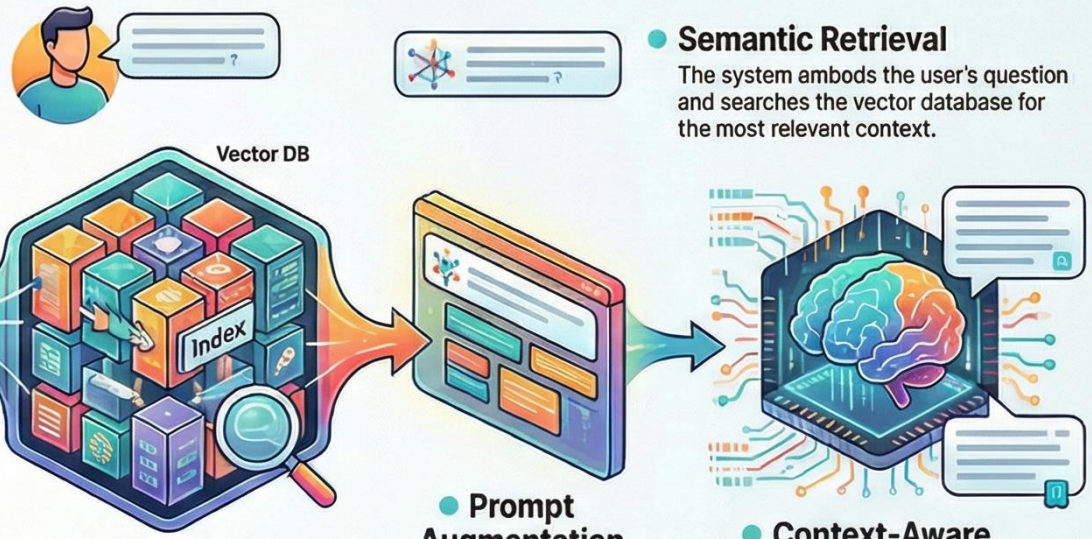
- Converting text chunks into mathematical vectors that represent their semantic meaning queries.

- **Document Chunking**
  Breaking large files into bite-sized, digestible pieces to preserve context without overwhelming the system.

- **Embedding & Vectorization**
  Converting text chunks into mathematical vectors that represent their semantic meaning.

- **Vector Database Indexing**
  Organizing vectors in a specialized database for lightning-fast similarity searches during queries.

THE INFERENCE STAGE (Online Flow)



- **Semantic Retrieval**
  The system embods the user's question and searches the vector database for the most relevant context.

- **Semantic Retrieval**
  The system embads the user's question and searches the vector database for the most relevant context.

- **Prompt Augmentation**
  Combining the original query with the retrieved facts into a single "grounded" prompt.

- **Context-Aware Generation**
  The LLM uses the provided facts to craft an accurate, human-like response with citations.

COMPARISON: RAG vs. STANDARD LLM FINE-TUNING

	Feature	Standard RAG	Model Fine-Tuning
1	Knowledge Updates	Real-time / Dynamic	Static (requires retraining)
2	Hallucination Risk	Low (grounded in facts)	Moderate to High
3	Cost	Lower (no heavy training)	Higher (compute intensive)

Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a framework that enhances large language models (LLMs) by combining retrieval of relevant external information with generative capabilities.

1. LangChain: The Framework for Linear Workflows

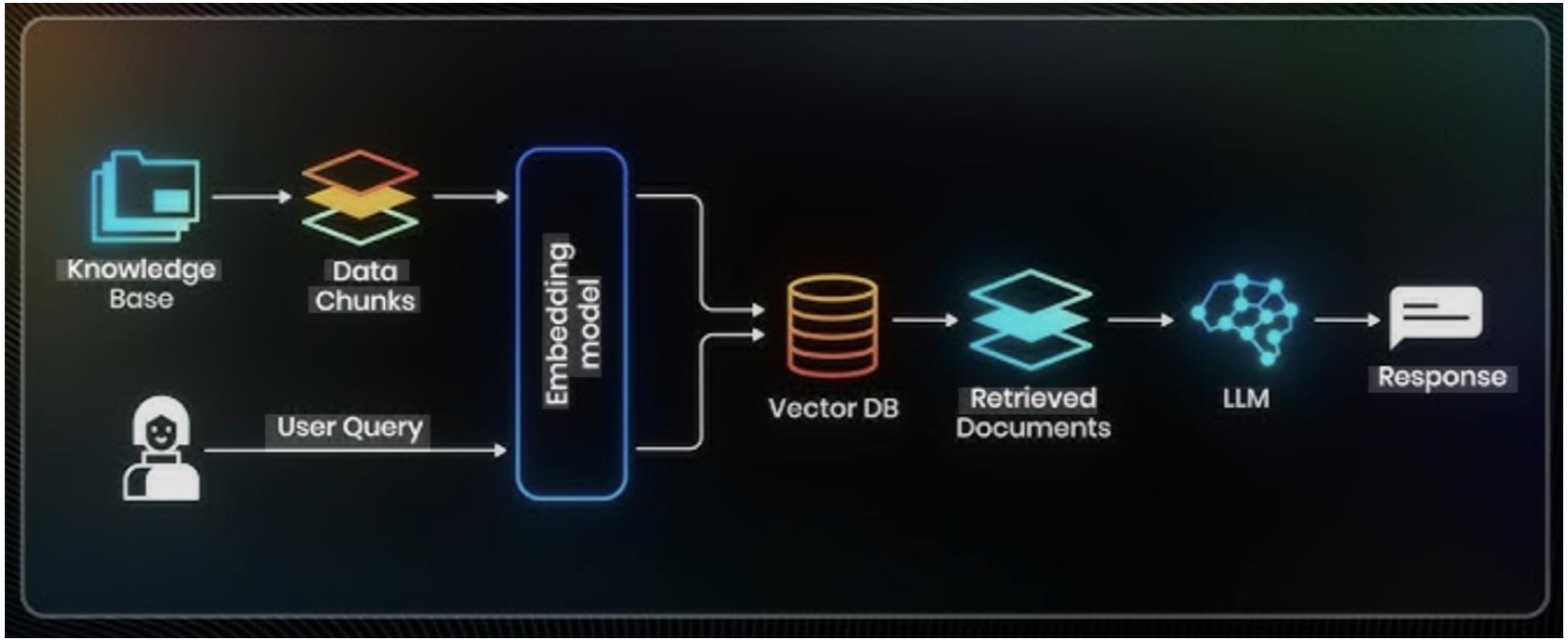
What it is: A foundational software development framework that connects LLMs with external data sources, tools, and memory. It provides modular, reusable components (chains) to build applications rapidly.

Best Use Cases:

- **Retrieval-Augmented Generation (RAG):** Connecting LLMs to private data for answering questions.
- **Simple Chatbots:** Linear question-and-answering systems.
- **Data Transformation:** Summarizing text, extracting data, or simple API interactions.

Key Strength: Prototyping and building simple, structured applications, like a sequential "chain".

{ Data Ingestion & Indexing } | { Retrieval } | { Generation }



Other RAG Tools

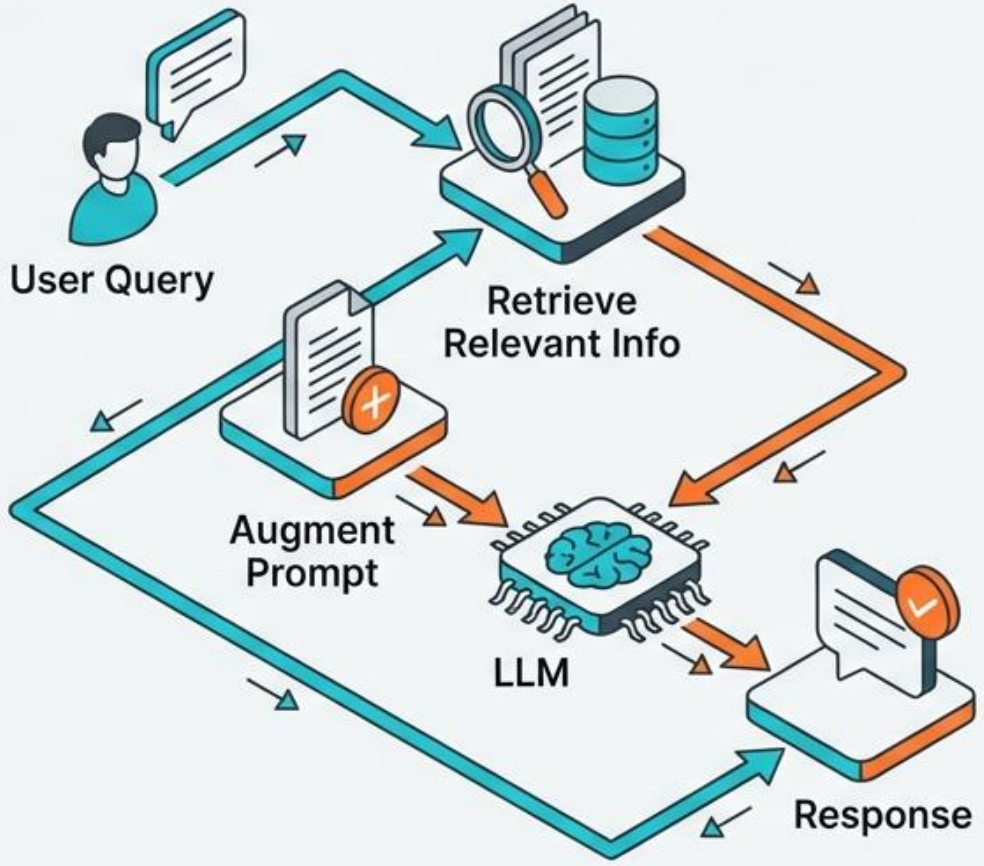
LlamaIndex: Specifically designed for RAG, it excels at ingestion, structuring, and accessing private data sources (PDFs, SQL, APIs). It is highly efficient for Retrieval-Augmented Generation-heavy applications.	Haystack (by deepset): An open-source framework ideal for building production-ready RAG pipelines and search systems, featuring strong connectors for document stores like Elasticsearch and FAISS.	RAGFlow: A RAG engine that offers advanced document parsing, focusing on accurate, deeply-researched, and structured answers with visual citations.	Flowise: A UI-based, low-code tool that allows users to build RAG applications, chatbots, and agents visually.
CrewAI: Focused on orchestrating role-based, autonomous agent teams, ideal for complex multi-agent workflows.	PydanticAI: A framework focused on type-safe, enterprise-level application development.	Mastra: A TypeScript-based framework for building RAG systems that integrates directly into application code.	**Cloud-Native Tools:** Azure OpenAI/AI Search, Vertex AI Agent Builder, and AWS Bedrock AgentCore are strong options for established cloud ecosystems.

The RAG Imperative: Accuracy & Context

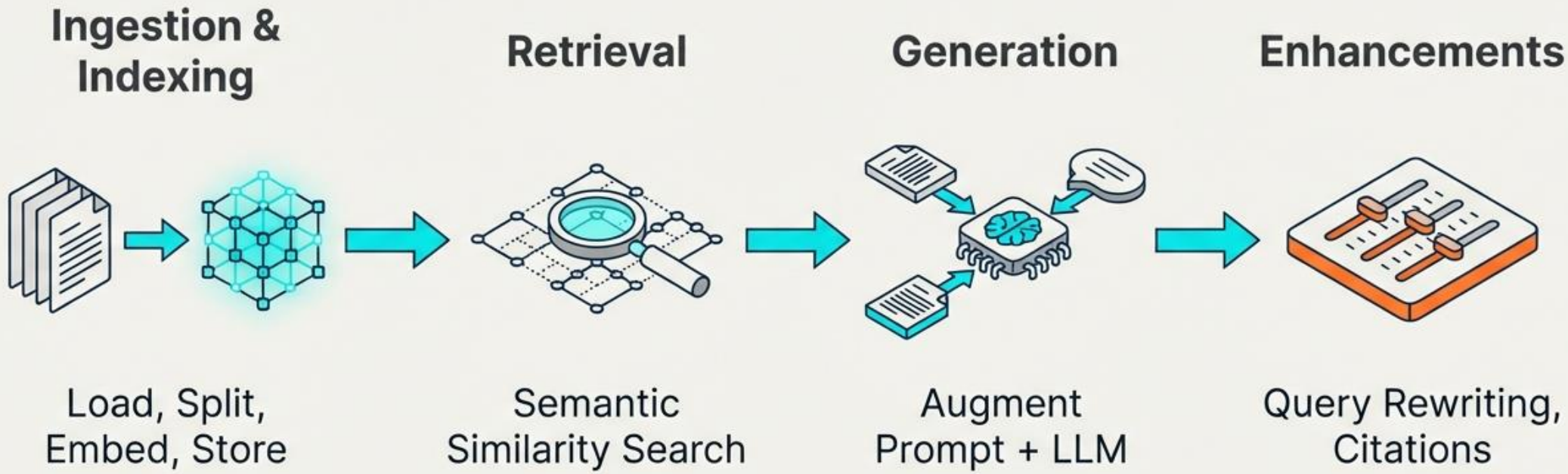
Retrieval-Augmented Generation (RAG)

Introduced by Meta AI (2020), RAG solves key LLM limitations:

1. Outdated Knowledge (Training cut-offs)
2. Hallucinations (Confidently wrong answers)
3. Lack of Domain Context (Proprietary data)



The Architecture of a RAG System



Prerequisites & Environment Setup

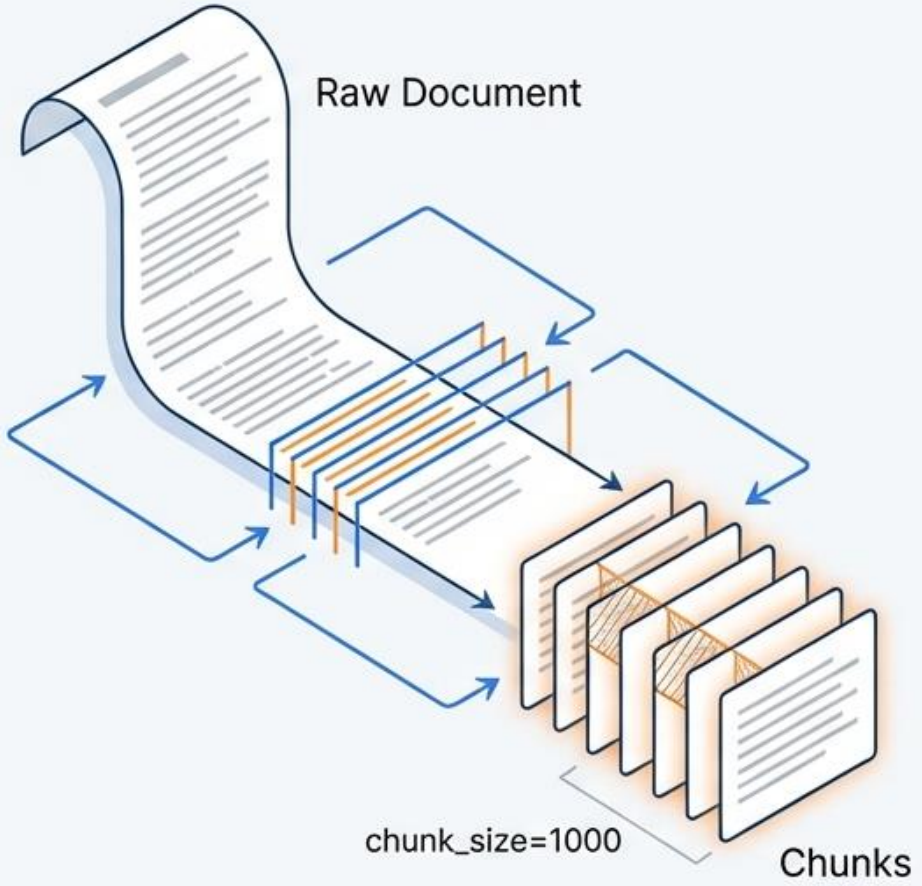
```
> pip install langchain langchain-openai
langchain-chroma openai chromadb

> import os
> os.environ['OPENAI_API_KEY'] = 'your-api-key'
```

The Toolkit

- → **LangChain**: Orchestration framework
- → **Chroma**: Vector Database
- → **OpenAI**: Embeddings & LLM logic

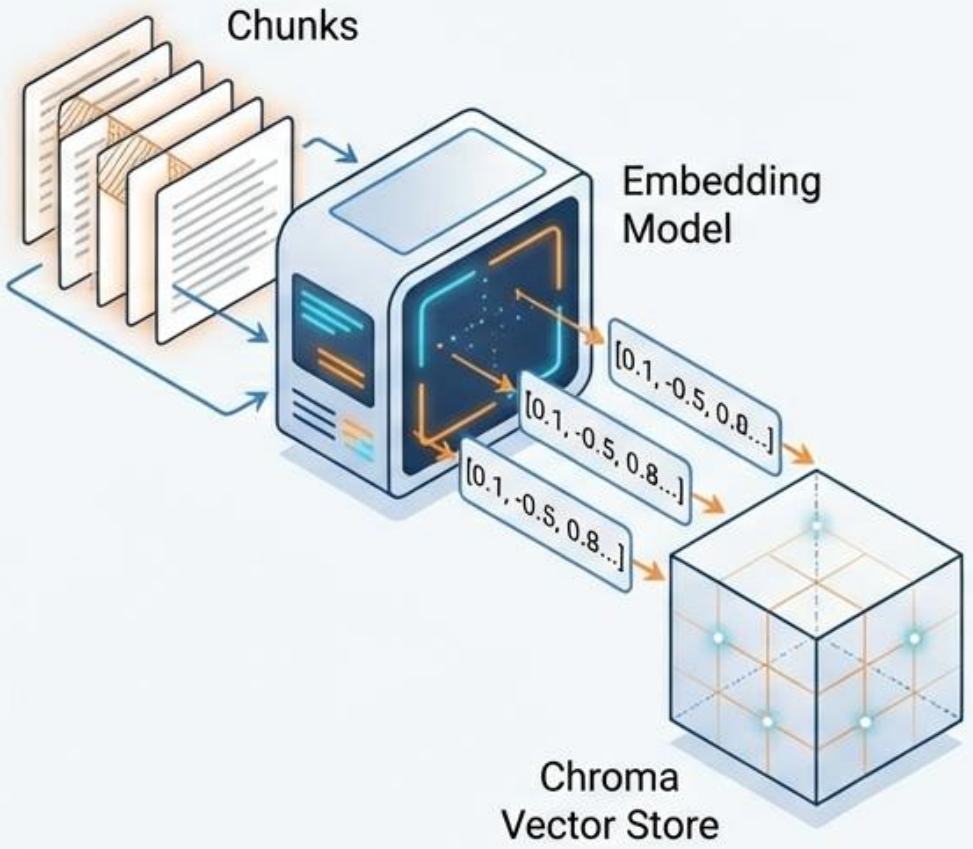
Step 1: Load and Split Documents

```
from langchain_community.document_loaders import
WebBaseLoader
from langchain_text_splitters import
RecursiveCharacterTextSplitter

# Load
loader = WebBaseLoader('https://lilianweng.githu
b.io/posts/2023-06-23-agent/')
docs = loader.load()

# Split
text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=1000, chunk_overlap=200)
splits = text_splitter.split_documents(docs)
```



Step 2: Embed and Index

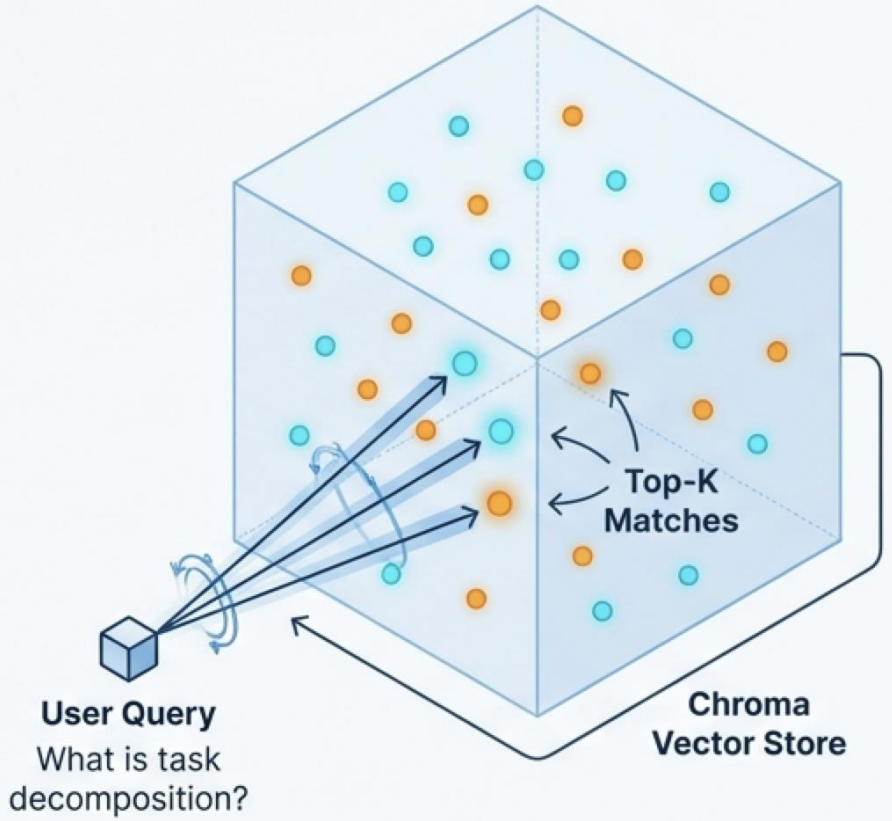
```
from langchain_openai import
OpenAIEmbeddings
from langchain_chroma import Chroma

embeddings = OpenAIEmbeddings(model='text-
embedding-3-small')
vector_store = Chroma.from_documents(
  documents=splits, embedding=embeddings)
```



For production: Consider scalable DBs like SingleStore.

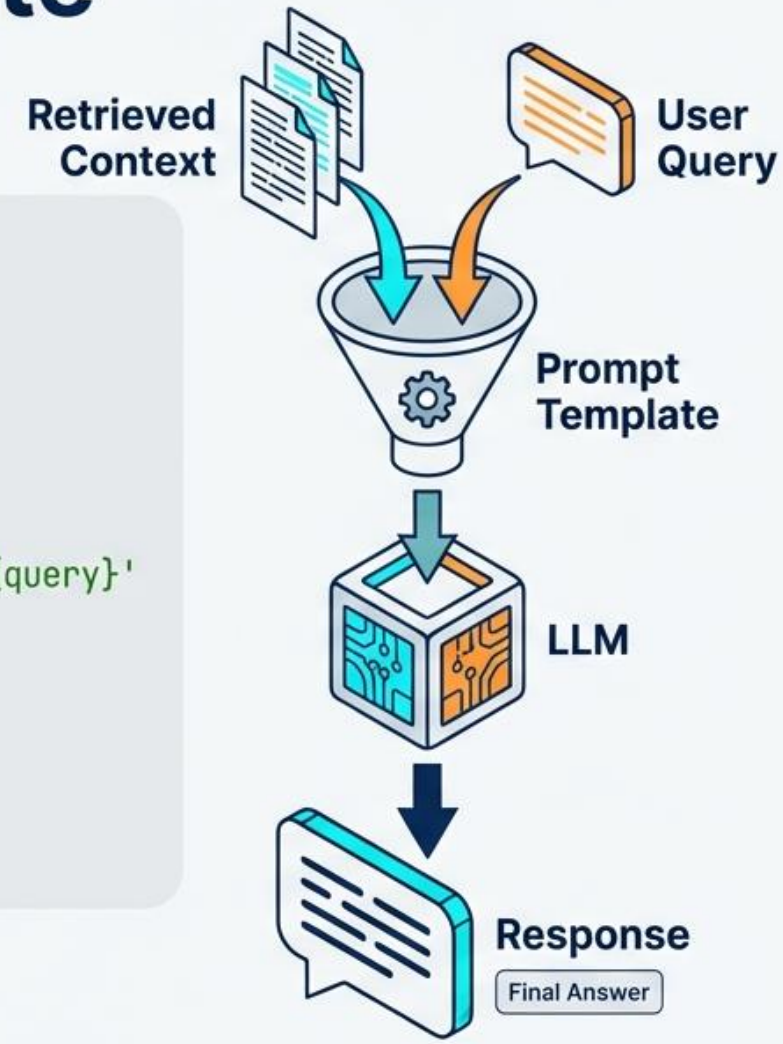
Step 3: Retrieval

```
query = 'What is task decomposition?'
retriever = vector_store.as_retriever(
    search_type='similarity',
    search_kwargs={'k': 3})
retrieved_docs = retriever.invoke(query)


# Combine content
context = '\n\n'.join(doc.page_content
for doc in retrieved_docs)
```



Step 4: Augment and Generate

```python
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

llm = ChatOpenAI(model='gpt-3.5-turbo')

prompt_template = ChatPromptTemplate.from_template(
    'Answer the question based on this context: {context}\n\nQuestion: {query}'
)
chain = prompt_template | llm

response = chain.invoke({'context': context, 'query': query})
print(response.content)
```



Code Samples

RAG

We'll implement a simple RAG system over a text document using LangChain, OpenAI for embeddings/LLM, and Chroma as the vector DB. This draws from official LangChain tutorials and can be adapted for local LLMs (e.g., via Ollama).

Prerequisites

Install packages:

```bash
pip install langchain langchain-openai langchain-chroma openai chromadb
```

Set your OpenAI API key:

```python
import os
os.environ["OPENAI_API_KEY"] = "your-api-key"
```

Step 1: Load and Split Documents

Load data and chunk it to ~500-1000 tokens per piece with overlap for context.

```python
from langchain_community.document_loaders import WebBaseLoader  # Or TextLoader for local files
from langchain_text_splitters import RecursiveCharacterTextSplitter

# Load example data (e.g., a blog post)
loader = WebBaseLoader("https://lilianweng.github.io/posts/2023-06-23-agent/")
docs = loader.load()

# Split into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
splits = text_splitter.split_documents(docs)
```

For PDFs or custom files, use `PyPDFLoader` or similar.

Step 2: Embed and Index in Vector DB

Use embeddings to vectorize chunks and store them.

```python
from langchain_openai import OpenAIEmbeddings
from langchain_chroma import Chroma

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")
vector_store = Chroma.from_documents(documents=splits, embedding=embeddings)
```

Alternatively, with SingleStore (a scalable vector DB):

```python
from langchain_community.vectorstores.singlestoredb import SingleStoreDB

vector_database = SingleStoreDB.from_documents(splits, embeddings, table_name="my_rag_table")
```

SingleStore integrates well for production-scale apps with SQL querying alongside vectors.

Step 3: Retrieval

Embed the query and fetch relevant chunks.

```python
query = "What is task decomposition?"
retriever = vector_store.as_retriever(search_type="similarity", search_kwargs={"k": 3})
retrieved_docs = retriever.invoke(query)

# Combine retrieved content
context = "\n\n".join(doc.page_content for doc in retrieved_docs)
```

Step 4: Augment and Generate

Craft a prompt with context and query, then generate.

```python
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

llm = ChatOpenAI(model="gpt-3.5-turbo")

prompt_template = ChatPromptTemplate.from_template(
    "Answer the question based on this context: {context}\n\nQuestion: {query}"
)
chain = prompt_template | llm

response = chain.invoke({"context": context, "query": query})
print(response.content)
```

This creates a basic RAG chain: retrieve → augment → generate.

For local LLMs (e.g., Llama 3 via Ollama):

- Install Ollama and pull a model: `ollama pull llama3`.
- Replace `ChatOpenAI` with `ChatOllama(model="llama3")`.

RAG Implementation using LangChain, OpenAI, [ChromaDB or SingleStore]

LEARN MORE

Infrastructure Variations: Local & Scalable

Local / Private (Ollama)

```
llm = ChatOllama(model='llama3')
# Run locally: ollama pull llama3
```

Run inference on your own hardware for zero cost and total privacy.

Scalable Production (SingleStore)

```
vector_database = SingleStoreDB.from_documents(
    splits, embeddings, table_name='my_rag_table'
)
```

Combine SQL + Vector search for enterprise-grade applications.

Orchestrating the Agent

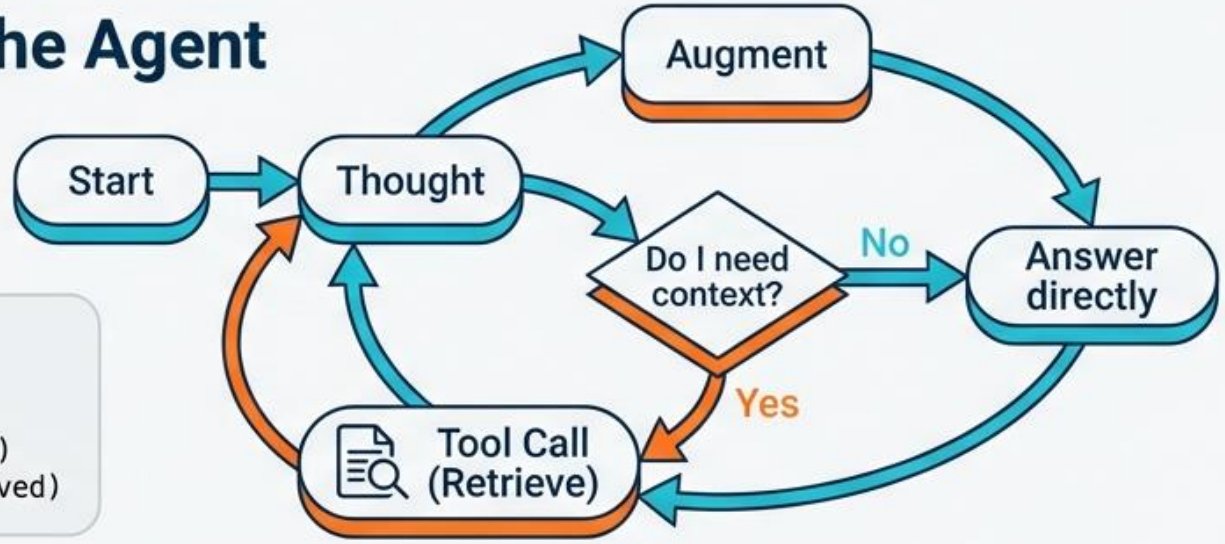
```python
from langchain.agents import create_tool_calling_agent, AgentExecutor

# System Prompt with Instructions
prompt = ChatPromptTemplate.from_messages([
    ('system', 'You are a helpful assistant. Use the retrieve_context tool when needed.'),
    ('human', '{input}'),
    ('placeholder', '{agent_scratchpad}'),
])

# Initialize Agent
agent = create_tool_calling_agent(llm, [retrieve_context], prompt)
agent_executor = AgentExecutor(agent=agent, tools=[retrieve_context])

# Run
result = agent_executor.invoke({'input': query})
```

Evolution: From Chains to Agents

The Chain

Rigid / Predictable

The Agent




```python
@tool
def retrieve_context(query: str) -> str:
    '''Retrieve relevant context for a query.'''
    retrieved = vector_store.similarity_search(query, k=2)
    return '\n\n'.join(doc.page_content for doc in retrieved)
```

RAG

Enhancing it while implementing RAG as an Agent. **Benefit**: Operation more fluid and iterative!

Implementing RAG as an Agent

For more dynamic setups (e.g., deciding when to retrieve), use LangChain agents. docs.langchain.com

Create a Retrieval Tool

Python

```python
from langchain.tools import tool

@tool
def retrieve_context(query: str) -> str:
    """Retrieve relevant context for a query."""
    retrieved = vector_store.similarity_search(query, k=2)
    return "\n\n".join(doc.page_content for doc in retrieved)
```

Build the Agent

Python

```python
from langchain.agents import create_tool_calling_agent
from langchain_core.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant. Use the retrieve_context tool when needed."),
    ("human", "{input}"),
    ("placeholder", "{agent_scratchpad}"),
])

agent = create_tool_calling_agent(llm, [retrieve_context], prompt)
```

Run the Agent

Python

```python
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent, tools=[retrieve_context])
result = agent_executor.invoke({"input": query})
print(result["output"])
```

Agents allow conditional retrieval, making them suitable for complex queries.

LEARN MORE

RAG as an Agent

RAG

Same example but implemented using locally available tools such as NumPy and a locally downloaded Ollama LLM Model. **Very limited in capability.**

```python
import os
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np
import ollama

# Step 1: Load and Split
def load_documents(directory):
    documents = []
    for filename in os.listdir(directory):
        if filename.endswith(".txt"):
            with open(os.path.join(directory, filename), 'r', encoding='utf-8') as f:
                documents.append(f.read())
    return documents

def split_documents(documents, chunk_size=500, chunk_overlap=50):
    chunks = []
    for doc in documents:
        for i in range(0, len(doc), chunk_size - chunk_overlap):
            chunks.append(doc[i:i + chunk_size])
    return chunks

# Step 2: Embed and Index
embedder = SentenceTransformer('all-MiniLM-L6-v2')

def embed_chunks(chunks):
    return embedder.encode(chunks)

def index_in_faiss(embeddings):
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatL2(dimension)
    index.add(embeddings)
    return index
```

```python
# Step 3: Retrieval
def retrieve(query, index, embedder, chunks, top_k=3):
    query_embedding = embedder.encode([query])
    distances, indices = index.search(query_embedding, top_k)
    retrieved_chunks = [chunks[i] for i in indices[0]]
    return retrieved_chunks

# Step 4: Augment and Generate
def augment_and_generate(query, retrieved_chunks, model='llama2'):
    context = "\n".join(retrieved_chunks)
    prompt = f"Context: {context}\n\nQuestion: {query}\nAnswer:"
    response = ollama.generate(model=model, prompt=prompt)
    return response['response']

# Run the pipeline
docs = load_documents("path/to/your/documents")  # Replace
chunks = split_documents(docs)
embeddings = embed_chunks(chunks)
index = index_in_faiss(embeddings)
faiss.write_index(index, "faiss_index.index")

query = "Your query here"  # Replace
retrieved = retrieve(query, index, embedder, chunks)
response = augment_and_generate(query, retrieved)
print("Final Response:", response)
```

Source Code Not Shared For This Example.
Only for demonstrating simplistic use-case

Using Local Tools

RAG

Another variation where everything is local but slightly better

```python
#!/usr/bin/env python3
"""
Single-file RAG pipeline using:
- LlamaIndex (open source)
- Local Ollama (llama3 for both embeddings + generation)
- SingleStore DB (local vector store)
No internet, no paid APIs required.
"""

import os
from llama_index.core import (
    Settings,
    SimpleDirectoryReader,
    VectorStoreIndex,
    StorageContext,
)
from llama_index.core.node_parsers import SentenceSplitter
from llama_index.embeddings.ollama import OllamaEmbedding
from llama_index.vector_stores.singlestore import SingleStoreVectorStore
from llama_index.llms.ollama import Ollama

# ========================= CONFIGURATION =========================
# Change these to match your environment
DOCS_DIR = "/path/to/your/documents"          # ← your PDF/TXT/DOCX folder
OLLAMA_BASE_URL = "http://localhost:11434"    # default Ollama port
SINGLESTORE_CONFIG = {
    "host": "localhost",
    "port": 3306,
    "user": "root",
    "password": "your_password",              # ← change this
    "database": "rag_db",                     # ← change this
    "table": "vector_table",
}
EMBEDDING_MODEL = "llama3"                    # must be pulled with `ollama pull llama3`
LLM_MODEL = "llama3"
CHUNK_SIZE = 1024
CHUNK_OVERLAP = 20
TOP_K = 3

# Sample query (change as needed)
QUERY = "What is the main topic discussed in these documents?"

# ========================= STEP 1: LOAD & SPLIT =========================
print("🔄 Step 1: Loading and splitting documents...")
documents = SimpleDirectoryReader(
    input_dir=DOCS_DIR,
    required_exts=[".pdf", ".txt", ".docx", ".doc"],  # supported formats
).load_data()

parser = SentenceSplitter(chunk_size=CHUNK_SIZE, chunk_overlap=CHUNK_OVERLAP)
nodes = parser.get_nodes_from_documents(documents)

print(f"✅ Loaded {len(documents)} documents → {len(nodes)} nodes")

# ========================= STEP 2: EMBED & INDEX IN SINGLESTORE =========================
print("🔄 Step 2: Setting up embeddings + SingleStore vector store...")

Settings.embed_model = OllamaEmbedding(
    model_name=EMBEDDING_MODEL,
    base_url=OLLAMA_BASE_URL,
)

vector_store = SingleStoreVectorStore(
    host=SINGLESTORE_CONFIG["host"],
    port=SINGLESTORE_CONFIG["port"],
    user=SINGLESTORE_CONFIG["user"],
    password=SINGLESTORE_CONFIG["password"],
    database=SINGLESTORE_CONFIG["database"],
    table=SINGLESTORE_CONFIG["table"],
    embedding_dimension=4096,  # llama3 dimension
)

storage_context = StorageContext.from_defaults(vector_store=vector_store)

index = VectorStoreIndex(nodes, storage_context=storage_context)

print("✅ Documents embedded and indexed in SingleStore")
```

RAG

One more way forward

```python
# ======================== STEP 3: RETRIEVAL ========================
print("\n🔄 Step 3: Retrieval demo...")
retriever = index.as_retriever(similarity_top_k=TOP_K)
retrieved_nodes = retriever.retrieve(QUERY)

print(f"Top {TOP_K} retrieved nodes:")
for i, node in enumerate(retrieved_nodes, 1):
    print(f"  {i}. {node.text[:200]}...")

# ======================== STEP 4: AUGMENT + GENERATE ========================
print("\n🔄 Step 4: Augment & Generate with local Ollama...")
Settings.llm = Ollama(
    model=LLM_MODEL,
    base_url=OLLAMA_BASE_URL,
    temperature=0.7,
    request_timeout=120.0,
)

query_engine = index.as_query_engine(similarity_top_k=TOP_K)

response = query_engine.query(QUERY)

print("\n" + "="*60)
print("FINAL RAG RESPONSE")
print("="*60)
print(response)
print("="*60)

# Optional: show source nodes used for the final answer
print("\nSources used:")
for i, node in enumerate(response.source_nodes, 1):
    print(f"  {i}. {node.node.metadata.get('file_name', 'unknown')} "
          f"(score: {node.score:.3f})")
```

How to run

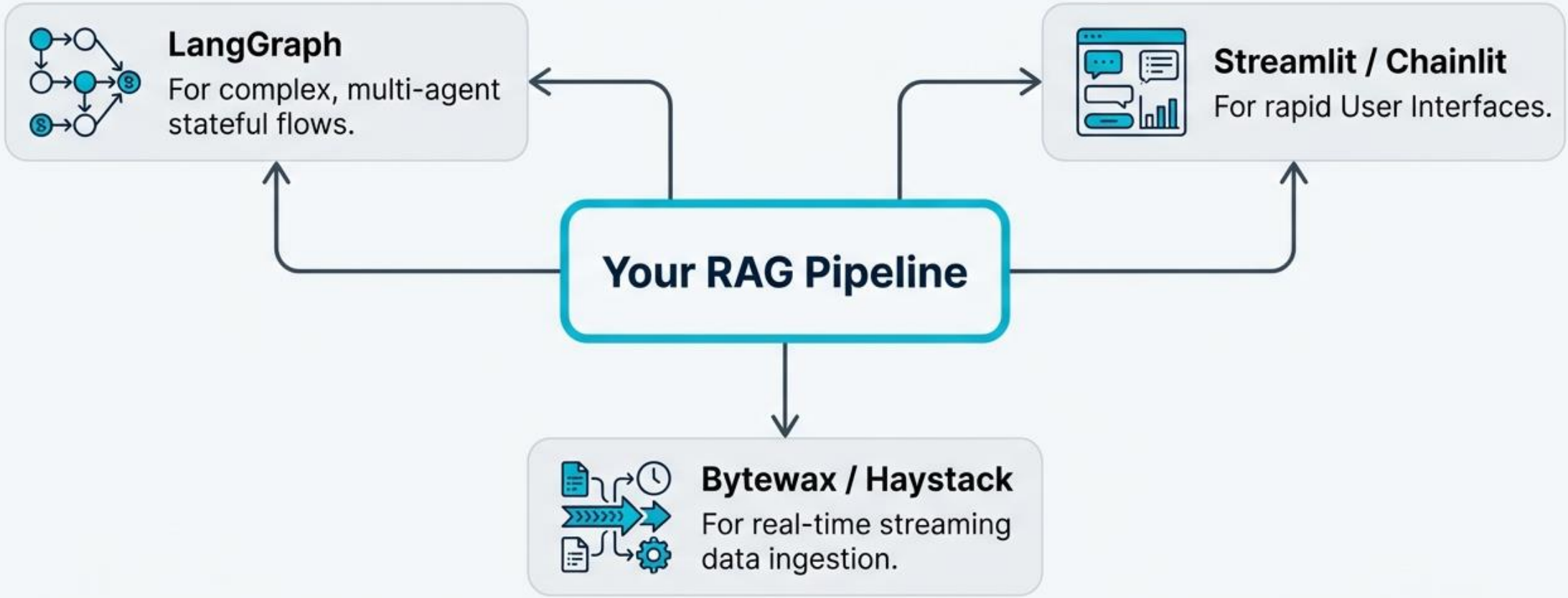
1. Make sure Ollama is running with llama3:
   `ollama pull llama3 && ollama run llama3`
2. Have SingleStore running locally (Docker example):
   `docker run -d -p 3306:3306 --name singlestore singlestore/cluster-in-a-box`
   (create the database `rag_db` manually or let the script handle it)
3. Install dependencies (once):

```bash
pip install llama-index llama-index-embeddings-ollama llama-index-llms-ollama \
            llama-index-vector-stores-singlestore singlestoredb
```

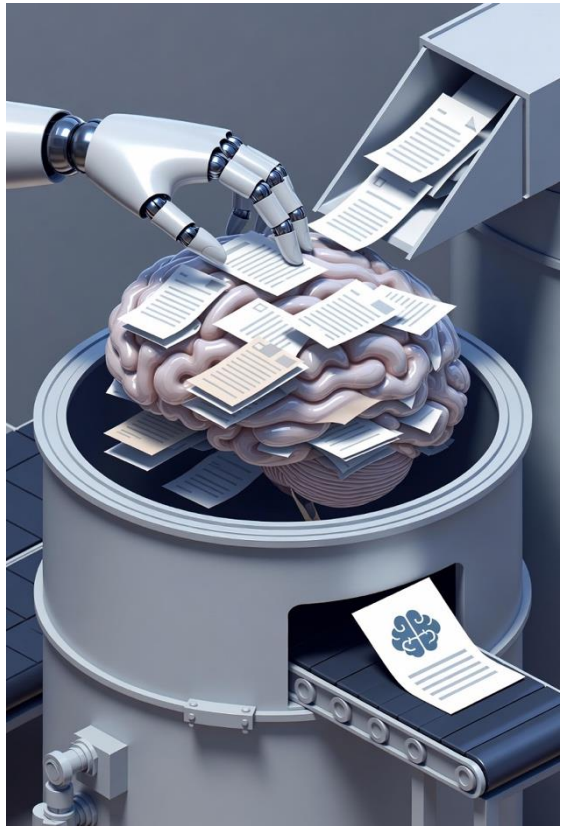
4. Update the `CONFIGURATION` section with your paths/credentials.
5. `python rag_pipeline.py`

The script now runs the complete pipeline end-to-end in one file. Enjoy! 🚀

Extending to Production



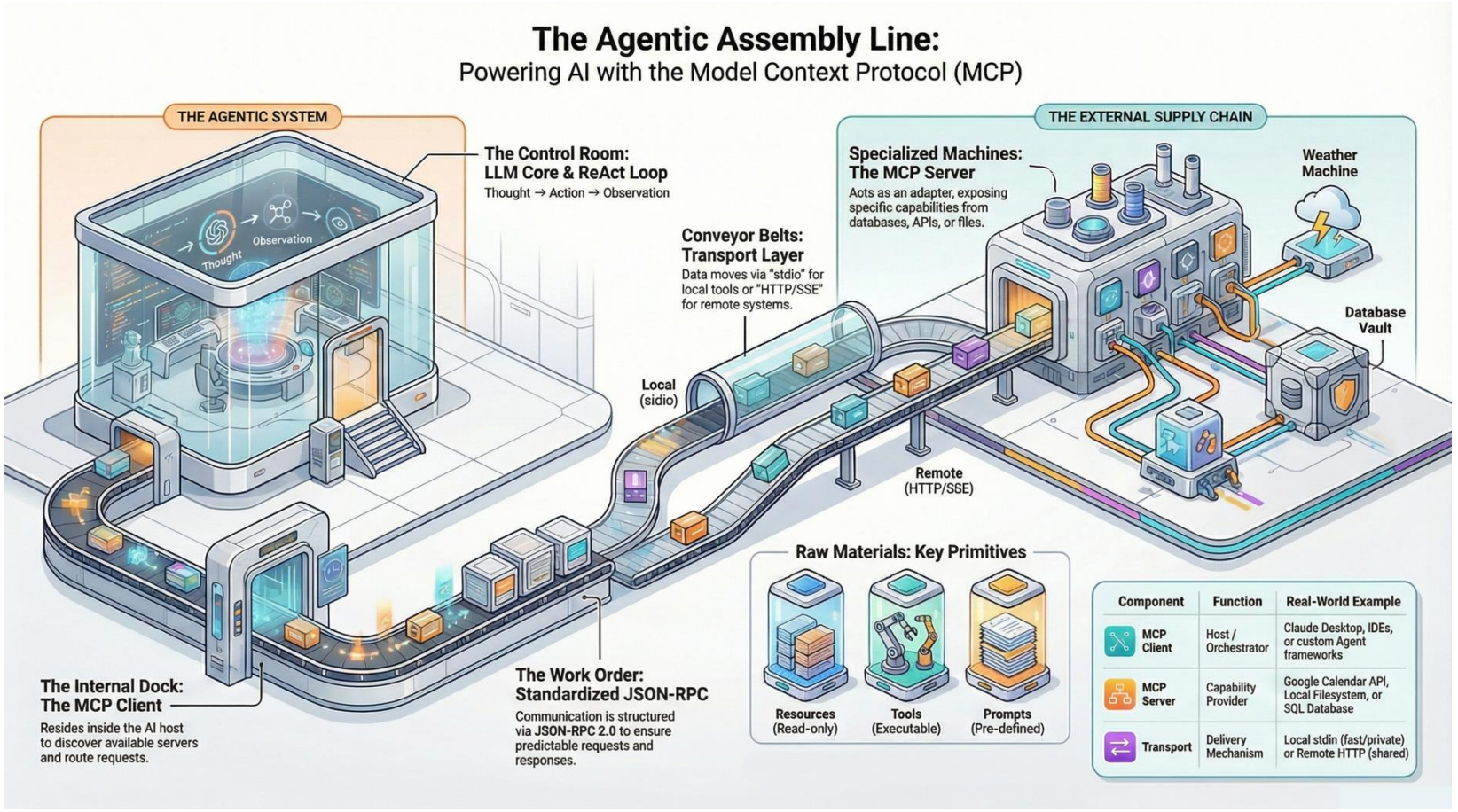
This code is the foundation. Production requires constant iteration on data quality and evaluation.



Key Benefits of RAG:

With RAG, you can enhance semantic-searches using a combination of local, private data combined with semantics. This overcomes two major limitations of LLMs:

1. Time-Bound Data
2. Mixing Corporate Knowledge to make searches more relevant.

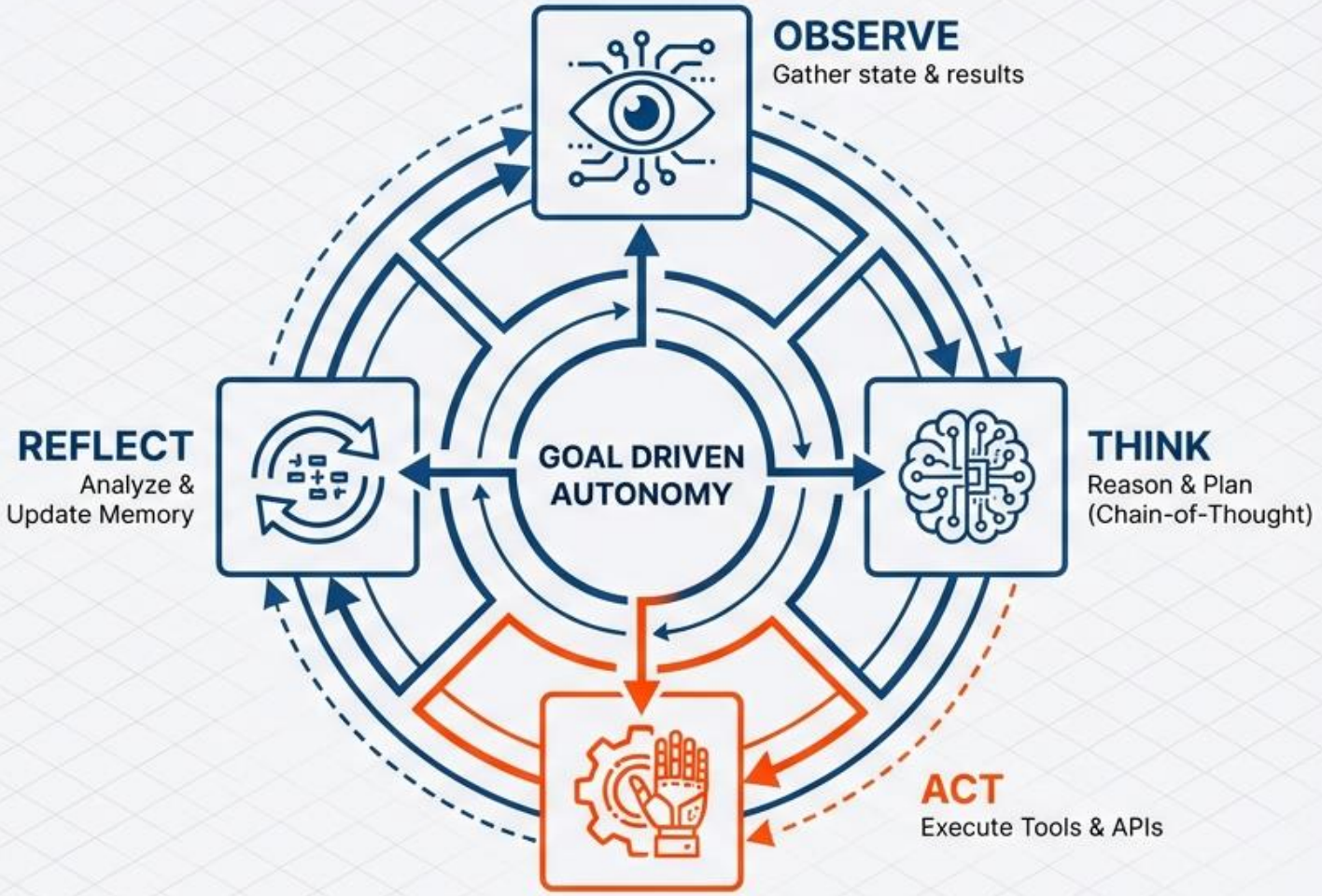


Agentic Systems (MCP/A2A)

Autonomous Planning, Reasoning, and using Tools to complete tasks without human supervision

Beyond Chatbots: The Rise of the Agentic Loop

THE CHATBOT PARADIGM

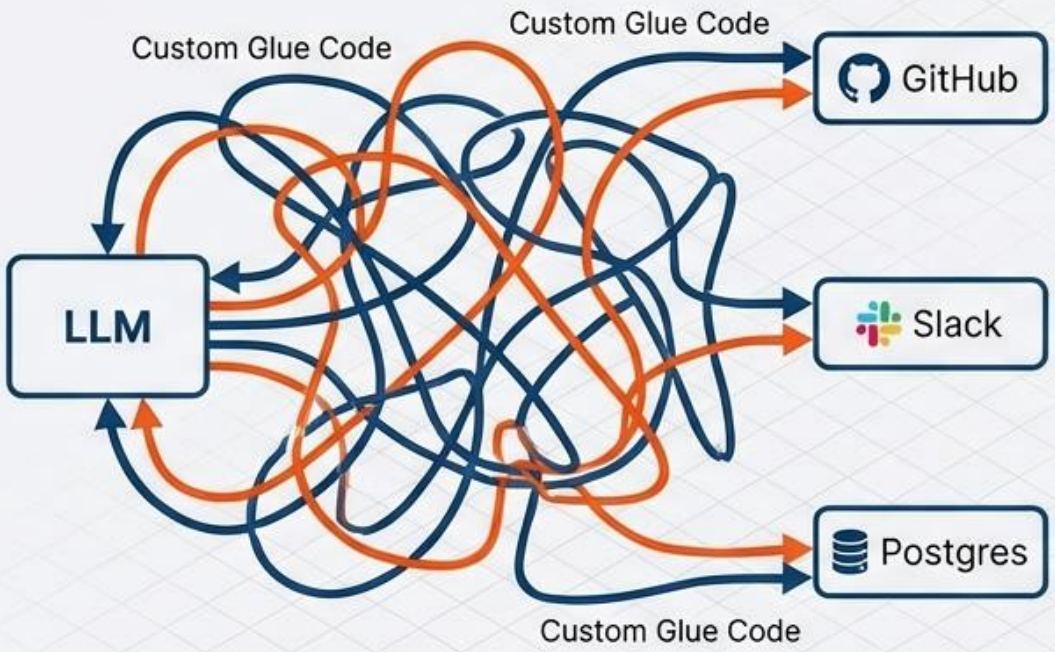


THE AGENTIC LOOP (O-T-A-R)

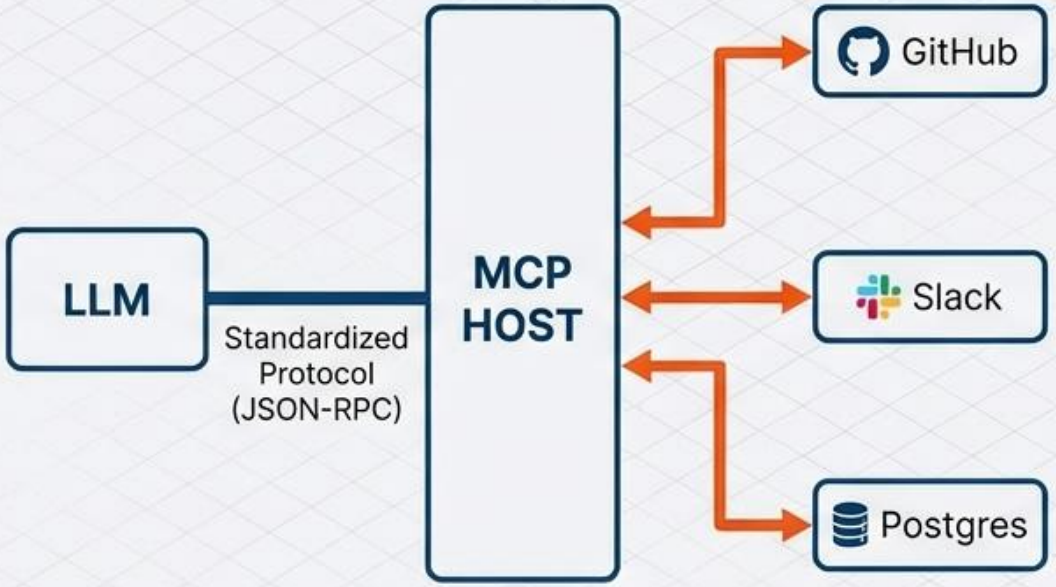
The Connectivity Crisis & The Solution

Solving the N x M Integration Problem

BEFORE: THE GLUE CODE TRAP

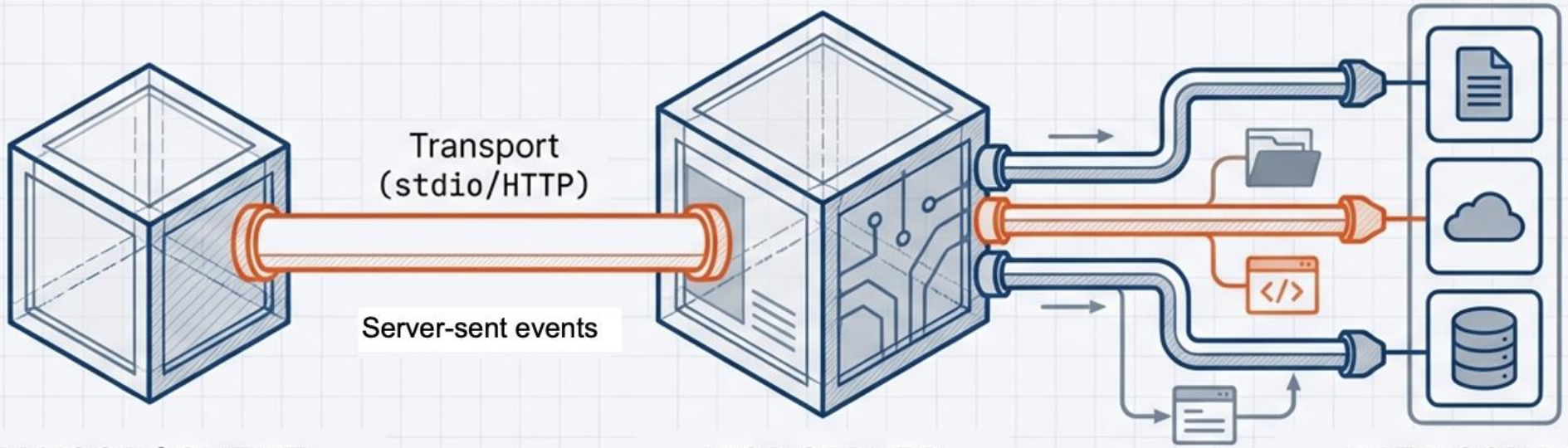


AFTER: MCP (THE USB-C FOR AI)



The USB-C for AI: A universal connector for secure, consistent tool access.

MCP Architecture: Client, Host, & Primitives



AI HOST / CLIENT
(e.g., Claude Desktop, IDE)

MCP SERVER
(The Adapter)

DATA SOURCE
(Local Files, API, DB)

RESOURCES
Passive Data.
Read-only context
(logs, files).

TOOLS
Executable Actions.
Functions the
model can call.

PROMPTS
Templates.
Pre-defined
workflows.

Building the Server: Setup & Initialization

Implementation Step 1: Python FastMCP Wrapper

```
# 1. Install the SDK
# pip install mcp

from mcp.server.fastmcp import FastMCP

# 2. Initialize the Server
mcp = FastMCP(
    name="Weather & File Server",
    description="Exposes a fake weather tool and local file reading"
)

# This handles session management and JSON-RPC automatically.
```

The Standard Wrapper for 2026

Defining Tools: The Executable Layer

The Code

```python
@mcp.tool
def get_current_weather(city: str) -> str:
    """Get the current weather for a city."""
    return f"Weather in {city}: Cloudy, 42F."
```

AUTO-GENERATED

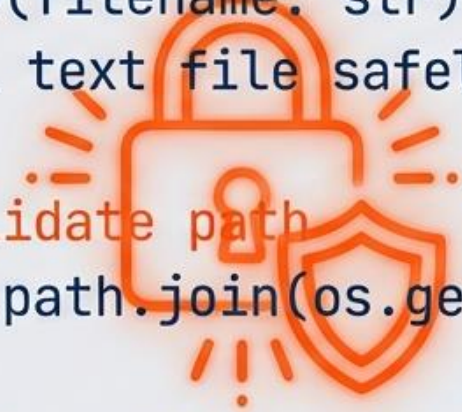
The LLM View

```
{
  "name": "get_current_weather",
  "description": "Get the current weather
for a city.",
  "parameters": {
    "type": "object",
    "properties": {
      "city": { "type": "string" }
    }
  }
}
```

Defining Resources: Context & Security

URI Scheme for Passive Access

```python
@mcp.resource("read_local_file://{filename}")
def read_local_file(filename: str) -> str:
    """Read a local text file safely."""

    # SECURITY: Validate path
    safe_path = os.path.join(os.getcwd(), filename)

    return open(safe_path).read()
```

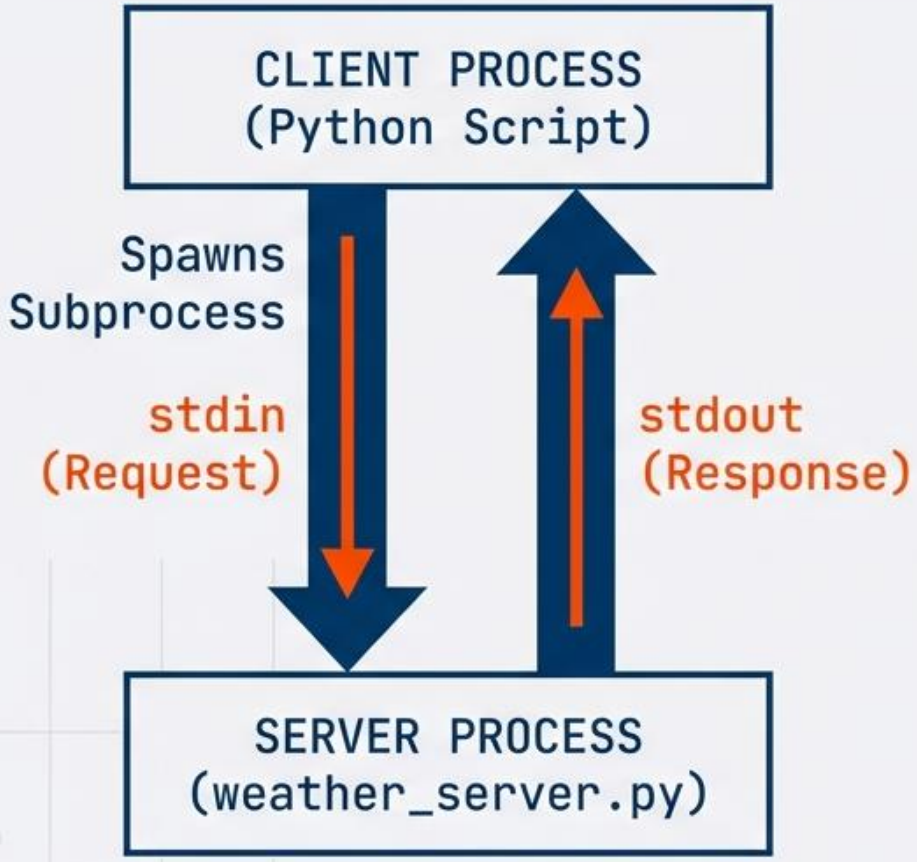


Sandboxing Required

Reference the resource definition code.

Building the Client: The `stdio` Connection

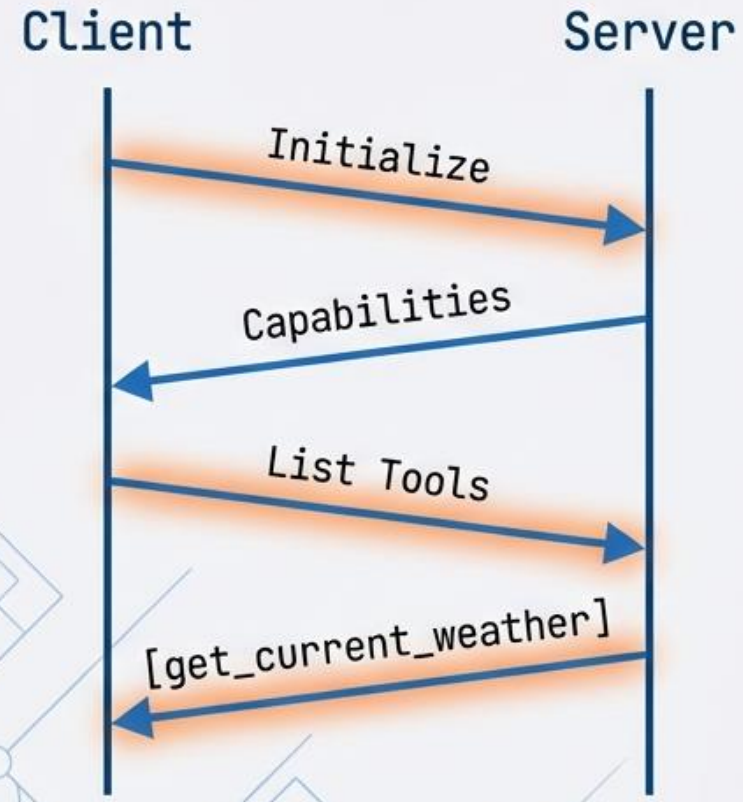
Connecting the AI to the Script




```python
from mcp import ClientSession, StdioServerParameters

server_params = StdioServerParameters(
    command="python",
    args=["weather_server.py"]
)
```

Reference the client connection code.

Client Initialization & Discovery




```python
async with stdio_client(server_params) as (read, write):
    async with ClientSession(read, write) as session:
        await session.initialize()

        # Dynamic Discovery
        tools = await session.list_tools()
        resources = await session.list_resources()

        print(f"Found {len(tools)} tools.")
```

Reference the discovery code.

Execution: Calling Tools & Fetching Resources

The Trigger Code

```
result = await session.call_tool(
    "get_current_weather",
    arguments={"city": "Glen Burnie"}
)
```

Reference the execution code.

The Output Terminal

Console

```
> Tool Output: Current weather in Glen Burnie: Cloudy, 42 degrees.
> Resource Output: [Content of notes.md loaded]
```

Code Samples

MCP Server in Python Using fastMCP

This example creates a simple MCP server with tools for basic math operations and a resource for version info.

1. **Install fastMCP**: Run `pip install fastmcp` (or `uv pip install fastmcp` for better dependency management).
2. **Create the Server File**: Save as `mcp_server.py`.

Python

```python
# Filename: mcp_server.py
from fastmcp import FastMCP

# Initialize the MCP server
mcp = FastMCP(name="Math MCP Server", description="A server for basic math operations")

# Define a tool for addition
@mcp.tool
def add(a: int, b: int) -> int:
    """Add two integers and return the result."""
    return a + b

# Define a tool for multiplication
@mcp.tool
def multiply(a: float, b: float) -> float:
    """Multiply two floats and return the result."""
    return a * b

# Define a static resource for server version
@mcp.resource("config://version")
def get_version():
    """Return the server version."""
    return "1.0.0"

if __name__ == "__main__":
    # Run the server over HTTP for remote access
    mcp.run(transport="http", host="127.0.0.1", port=8000, path="/mcp")
```

3. **Run the Server**: Execute `python mcp_server.py`. The server will be available at `http://127.0.0.1:8000/mcp`.
4. **Test**: Clients can now discover tools (e.g., `add`, `multiply`) and resources (e.g., `config://version`).

MCP Client in Python Using fastMCP

This example creates a client that connects to the above server, lists tools, and calls one.

1. **Create the Client File**: Save as `mcp_client.py`.

Python

```python
# Filename: mcp_client.py
import asyncio
from fastmcp import Client

async def main():
    # Connect to the server over HTTP
    async with Client("http://127.0.0.1:8000/mcp") as client:
        # List available tools
        tools = await client.list_tools()
        print("Available Tools:", tools)

        # Call the 'add' tool
        add_result = await client.call_tool("add", {"a": 5, "b": 3})
        print("Add Result:", add_result.content[0].text)

        # Read a resource
        version = await client.read_resource("config://version")
        print("Server Version:", version.content)

# Run the async main function
asyncio.run(main())
```

2. **Run the Client**: Ensure the server is running, then execute `python mcp_client.py`. Output will show tools, the addition result (8), and the version.

This approach keeps original code unchanged while exposing it via MCP.

Taking an Existing REST API and Its OpenAPI/Swagger Docs and Exposing Them as an MCP Server

To expose an existing REST API as an MCP server in Java, use the MCP Java SDK or Spring AI (preferred for Spring apps). The process involves:

- Adding MCP dependencies.
- Annotating service methods that call the REST API with `@Tool` to expose them as MCP tools.
- Using OpenAPI/Swagger docs to generate tool descriptions (manually copy or use tools like OpenRewrite for automation).
- Registering tools in the MCP server.

Assume an existing REST API for user management with endpoints like `/users/{id}` (GET) and OpenAPI spec describing it. We'll create tools that wrap API calls.

Example: Full code for a standalone Java MCP server wrapping a fictional REST API (use `RestTemplate` for calls).

```java
// Filename: McpServerApp.java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
import java.util.List;
import org.springframework.ai.mcp.Tool;
import org.springframework.ai.mcp.ToolCallbacks;

@SpringBootApplication
public class McpServerApp {

    public static void main(String[] args) {
        SpringApplication.run(McpServerApp.class, args);
    }

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @Bean
    public UserService userService(RestTemplate restTemplate) {
        return new UserService(restTemplate);
    }

    @Bean
    public List<ToolCallbacks> toolCallbacks(UserService userService) {
        return List.of(ToolCallbacks.from(userService));
    }
}
```

```java
// Filename: UserService.java (Wraps REST API calls)
import org.springframework.web.client.RestTemplate;

public class UserService {
    private final RestTemplate restTemplate;
    private final String baseUrl = "https://api.example.com";  // Existing REST API base URL

    public UserService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }
    // From Swagger docs
    @Tool(name = "get_user", description = "Get user by ID from existing REST API (from OpenAPI: Retrieves user details)")
    public User getUser(String id) {
        return restTemplate.getForObject(baseUrl + "/users/" + id, User.class);
    }
    // From Swagger docs
    @Tool(name = "create_user", description = "Create a new user via existing REST API (from OpenAPI: Posts new user data)")
    public User createUser(User user) {
        return restTemplate.postForObject(baseUrl + "/users", user, User.class);
    }
}

// Filename: User.java (Data model from OpenAPI)
public record User(String id, String name, String email) {}
```

- Build and run: Use Maven/Gradle with `spring-ai-starter-mcp-server` dependency.
- The `@Tool` descriptions are derived from OpenAPI/Swagger docs.
- For automation, use OpenRewrite recipes (from GitHub link) to scan controllers and generate `@Tool` methods.

LEARN MORE

MCP Server/Client as a proxy to existing Java Spring REST API

In this example, we will build a MCP Server/Client as a proxy to existing Java Spring REST API.

Assume an existing Spring Boot REST API app with a `ProductController` exposing `/products` endpoints. We'll add MCP support using Spring AI.

1. **Add Dependencies** (pom.xml):

```
<dependencies>
    <!-- Existing Spring Boot deps -->
    <dependency>
        <groupId>org.springframework.ai</groupId>
        <artifactId>spring-ai-starter-mcp-server-webmvc</artifactId>
    </dependency>
</dependencies>
```

2. **Configure Properties** (application.properties):

```
spring.ai.mcp.server.enabled=true
spring.ai.mcp.server.name=product-mcp-server
spring.ai.mcp.server.transport=http  # Expose over HTTP
```

3. **Annotate Existing Service**:

Assume existing `ProductService` with methods calling repo or external API.

```
// Filename: ProductService.java
import org.springframework.ai.mcp.Tool;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class ProductService {
    // Existing logic, e.g., repo or REST calls

    @Tool(name = "get_products", description = "Retrieve all products from the existing REST API")
    public List<Product> getProducts() {
        // Existing code: e.g., call repo.findAll() or RestTemplate.getForList("/products")
        return List.of(new Product("1", "Laptop", 999.99));
    }

    @Tool(name = "get_product_by_id", description = "Get a product by ID from the existing REST API")
    public Product getProductById(String id) {
        // Existing code: e.g., repo.findById(id) or RestTemplate.getForObject("/products/" + id)
        return new Product(id, "Product " + id, 100.0);
    }
}

// Filename: Product.java
public record Product(String id, String name, double price) {}
```

4. **Register Tools in Main App**:

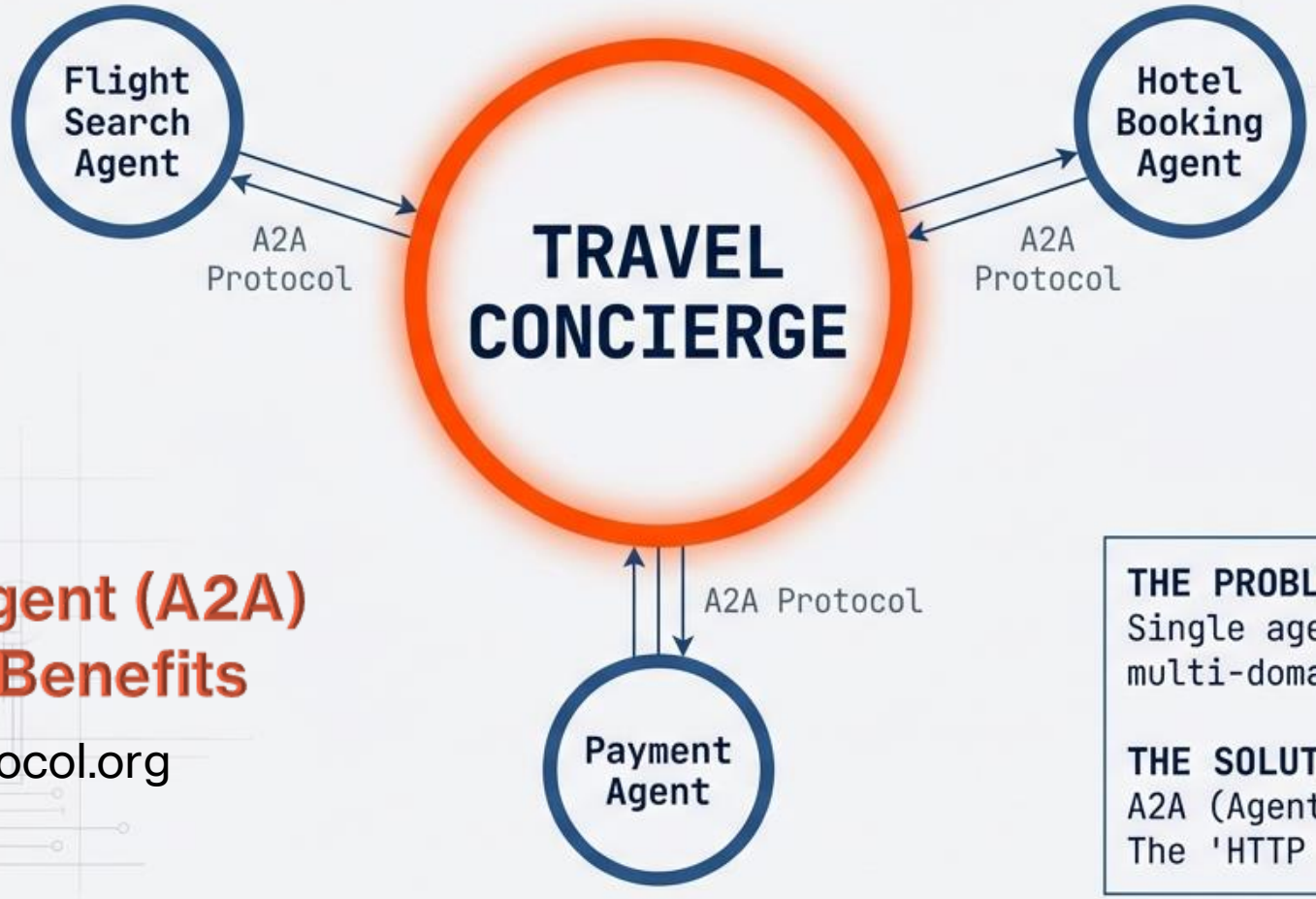
```
// Filename: ExistingRestApp.java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.ai.mcp.ToolCallbacks;
import java.util.List;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class ExistingRestApp {

    public static void main(String[] args) {
        SpringApplication.run(ExistingRestApp.class, args);
    }

    @Bean
    public List<ToolCallbacks> productTools(ProductService productService) {
        return List.of(ToolCallbacks.from(productService));
    }
}
```

5. **Run**: Start the app. The MCP server exposes tools at the configured endpoint (e.g., HTTP).
6. AI clients can now use get_products etc., leveraging the existing REST logic.

Future of MCP → A2A

The Next Limit: Collaboration & Complexity

When One Agent Isn't Enough



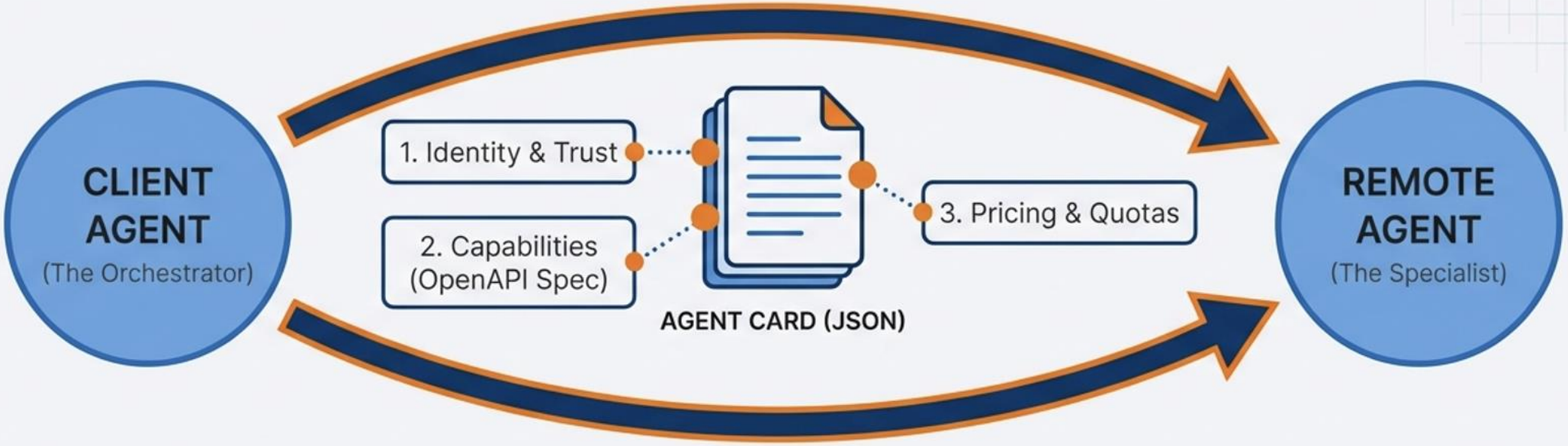
Agent to Agent (A2A) Protocol & Benefits

a2a-protocol.org

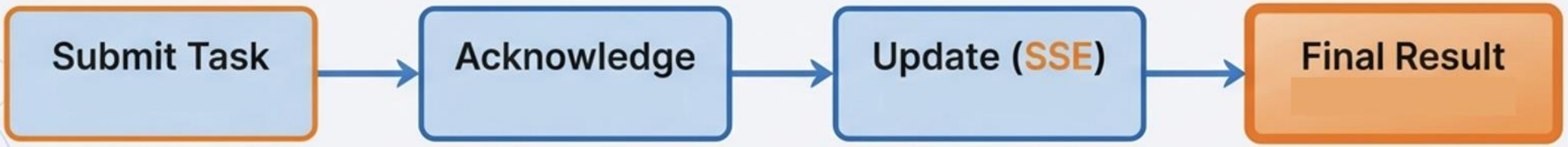
THE PROBLEM:
Single agents struggle with multi-domain complexity.

THE SOLUTION:
A2A (Agent-to-Agent Protocol). The 'HTTP for AI Agents'.

A2A Architecture: The Social Network of Agents



Task Lifecycle

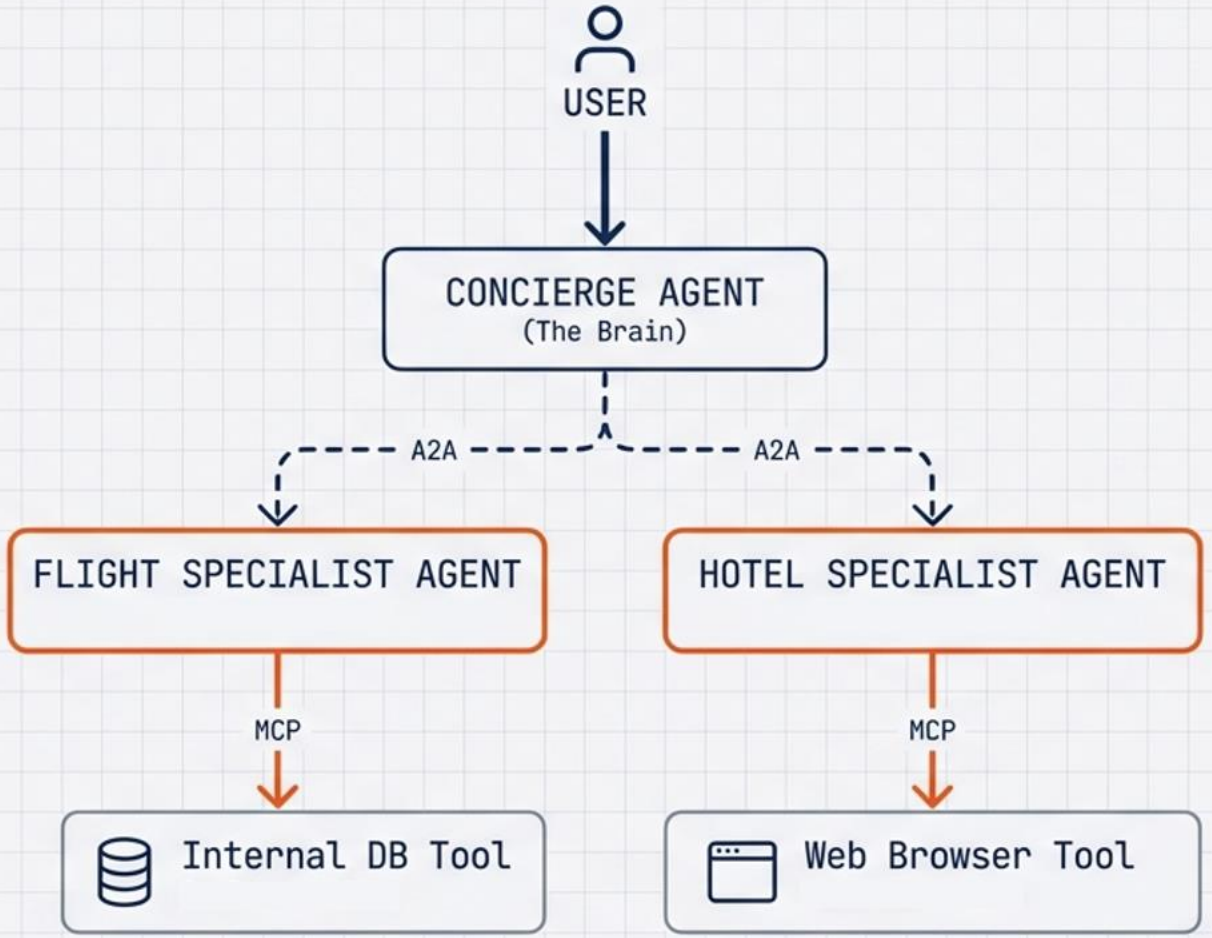


Protocol Showdown: MCP vs. A2A

Aspect	MCP (Model Context Protocol)	A2A (Agent-to-Agent Protocol)
Primary Focus	Agent <-> Tool (Vertical)	Agent <-> Agent (Horizontal)
Interaction	Function Calling	Task Delegation & Negotiation
Discovery	List Tools (Dynamic)	Agent Cards (Metadata)
Transport	Local (stdio)	Remote (HTTP/Webhooks)

RULE OF THUMB: Use MCP to give an agent HANDS. Use A2A to give an agent a TEAM.

The Unified Ecosystem



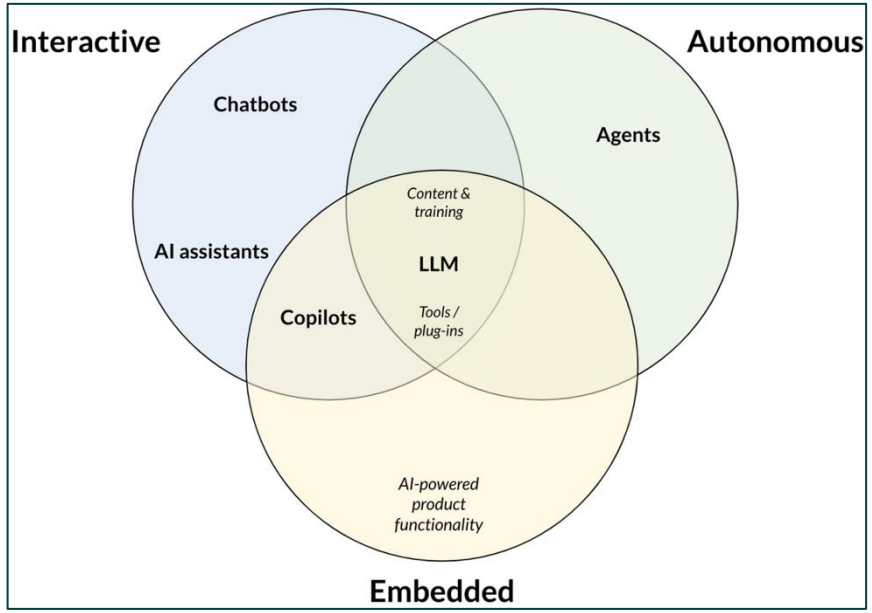
LEGEND:

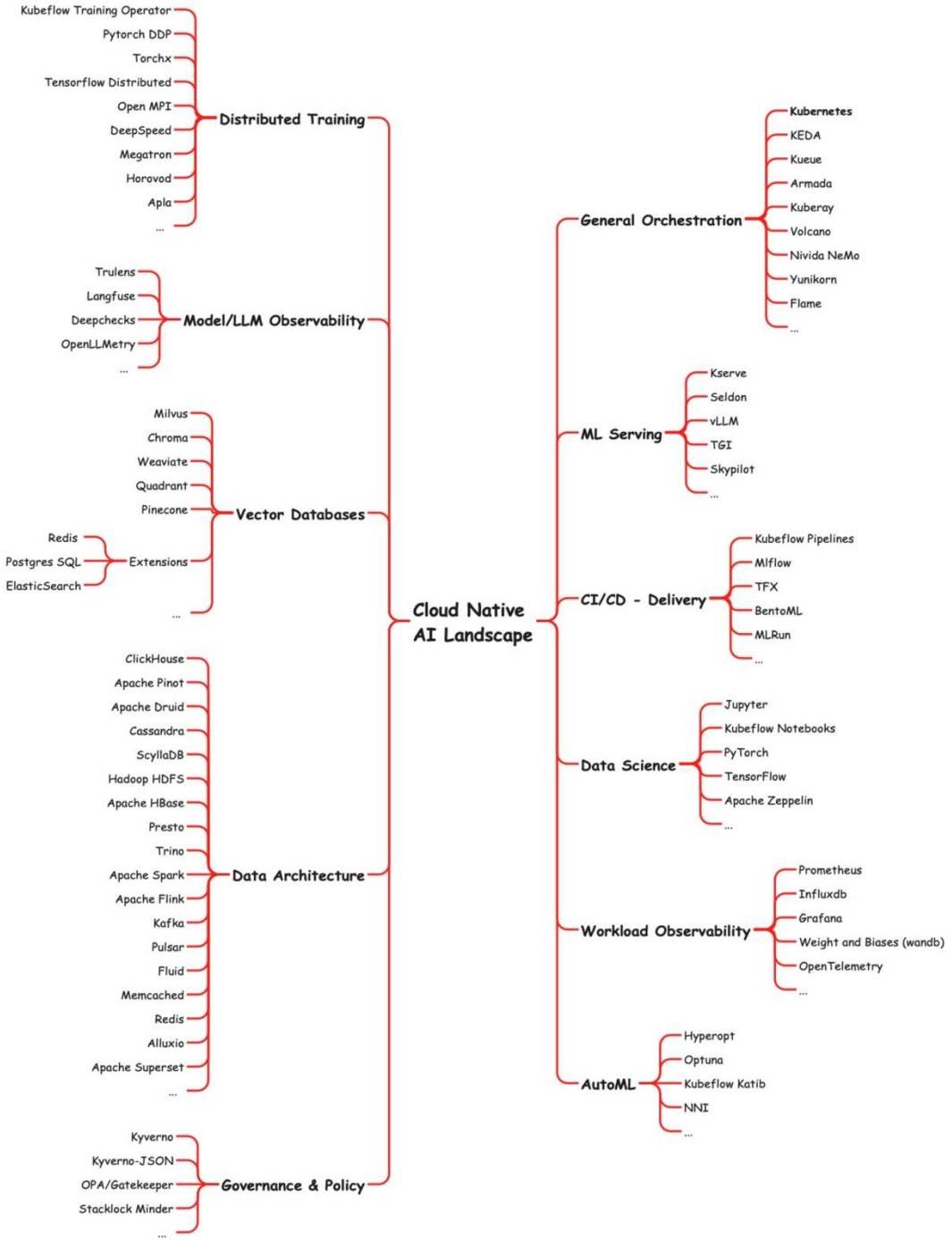
--- Dashed Line = A2A (Collaboration);

— Solid Line = MCP (Capabilities)

Your Builder's Roadmap

THE FUTURE IS COLLABORATIVE.


Interactive
Autonomous
Chatbots
Agents
Content & training
AI assistants
LLM
Copilots
Tools / plug-ins
AI-powered product functionality
Embedded


Cloud Native AI Landscape
Distributed Training
Kubeflow Training Operator
Pytorch DDP
Torchx
Tensorflow Distributed
Open MPI
DeepSpeed
Megatron
Horovod
Apla
...
Model/LLM Observability
Trulens
Langfuse
Deepchecks
OpenLLMetry
...
Vector Databases
Milvus
Chroma
Weaviate
Quadrant
Pinecone
Extensions
Redis
Postgres SQL
ElasticSearch
...
Data Architecture
ClickHouse
Apache Pinot
Apache Druid
Cassandra
ScyllaDB
Hadoop HDFS
Apache HBase
Presto
Trino
Apache Spark
Apache Flink
Kafka
Pulsar
Fluid
Memcached
Redis
Alluxio
Apache Superset
...
Governance & Policy
Kyverno
Kyverno-JSON
OPA/Gatekeeper
Stacklock Minder
...
General Orchestration
Kubernetes
KEDA
Kueue
Armada
Kuberay
Volcano
Nivida NeMo
Yunikorn
Flame
...
ML Serving
Kserve
Seldon
vLLM
TGI
Skypilot
...
CI/CD - Delivery
Kubeflow Pipelines
Mlflow
TFX
BentoML
MLRun
...
Data Science
Jupyter
Kubeflow Notebooks
PyTorch
TensorFlow
Apache Zeppelin
...
Workload Observability
Prometheus
Influxdb
Grafana
Weight and Biases (wandb)
OpenTelemetry
...
AutoML
Hyperopt
Optuna
Kubeflow Katib
NNI
...

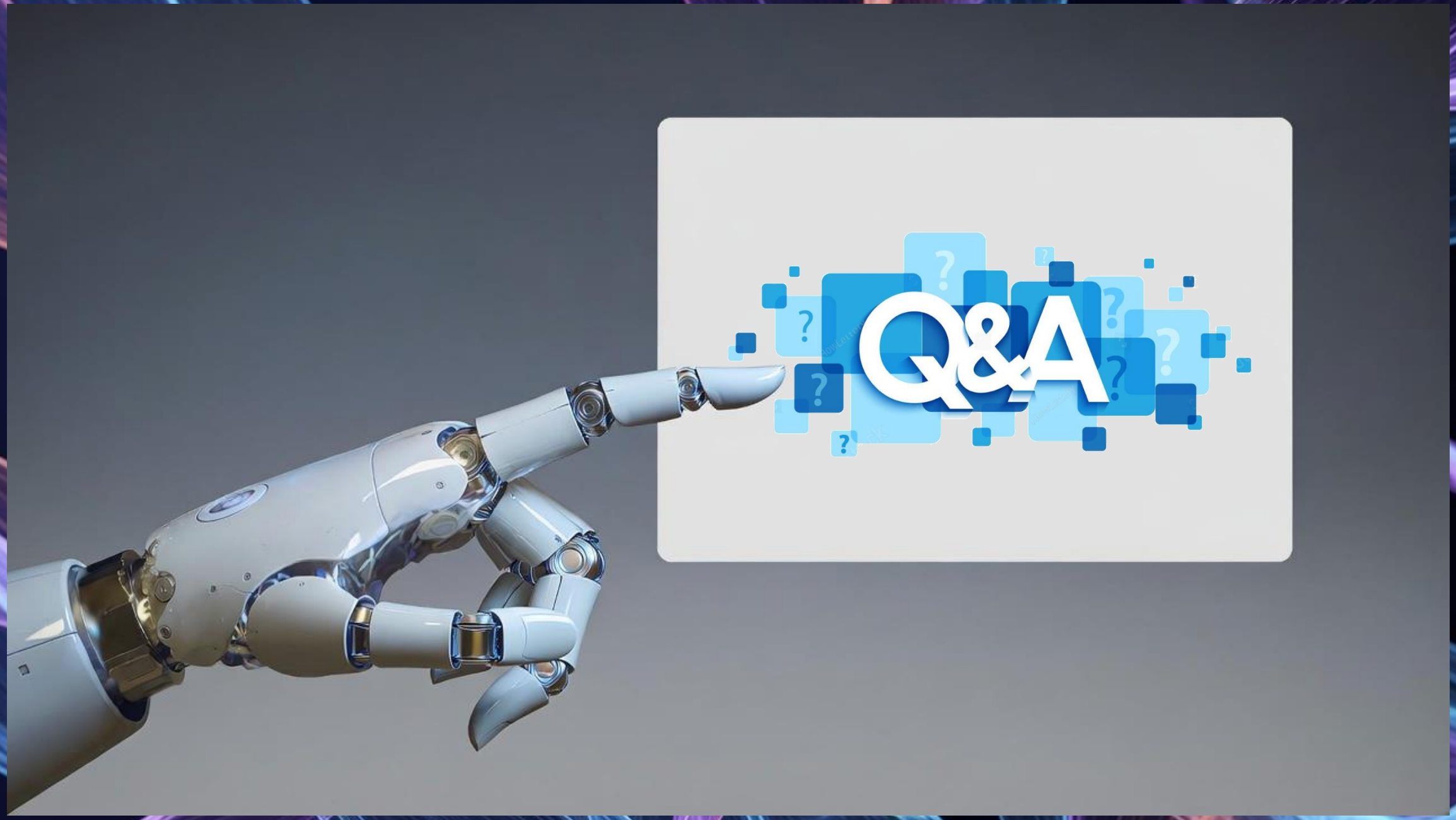
There is no magic here.

What looks like sophisticated orchestration of tools (Copilot, Claude Code, and many more) is exactly that: *a carefully choreographed, deliberately designed orchestration — nothing more, nothing less.*

Once You Learn The Fundamentals...

The Future Is Bright!

What will YOU build?


Q&A