Introduction to RAG and Agentic RAG with LlamaIndex and LangChain

AI Engineering: Agents, Vibe Coding and Full-Stack AI Course

John Alexander

Slides and code repo: https://github.com/JRAlexander/Intro-to-RAG-Agentic-RAG-2602

Why RAG?

Large Language Model challenges

- Trained with point-in-time data to ensure effectiveness
- Concerns about costs and security
- Hallucinations – confidently wrong answers

Integrating domain knowledge

Fine tuning

Learn new skills (permanently)

💵 High cost, time 🕰️

Retrieval Augmented Generation

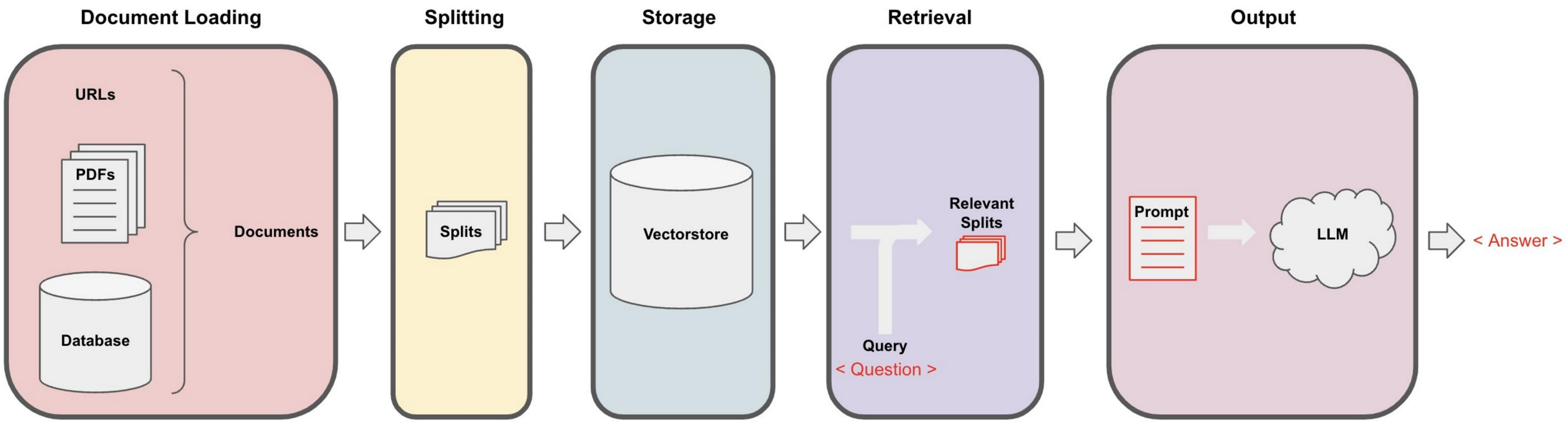
Learn new facts (temporarily)

Source: Pamela Fox, https://aka.ms/pythonai/slides/rag

RAG 101

What is Retrieval Augmented Generation (RAG)

- a pattern that works with pretrained Large Language Models (LLM) and **<u>your own data</u>** to generate responses.
- Use tools and components to augment the prompting!



DEMO

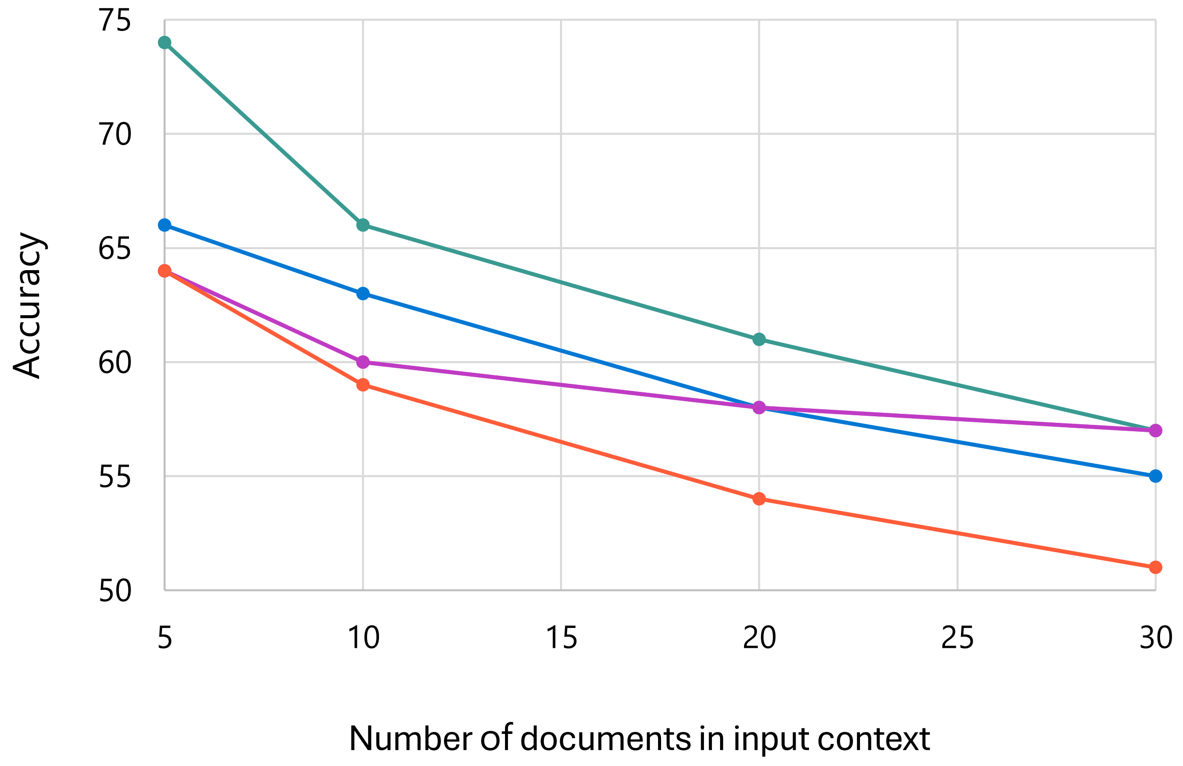
Simple RAG

Why do we need to split documents?

1. LLMs have limited context windows (4K – 128K)

2. When an LLM receives too much information, it can get easily distracted by irrelevant details.

3. The more tokens you send, the higher the cost, the slower the response.



Source: Lost in the Middle: How Language Models Use Long Contexts, Liu et al. arXiv:2307.03172

Source: Pamela Fox, https://aka.ms/pythonai/slides/rag

Optimal size of document chunk

How big should chunks be?

# of tokens per chunk	Recall@50
512	**42.4**
1024	37.5
4096	36.4
8191	34.9

Source: aka.ms/ragrelevance

A **token** is the unit of measurement for an LLM's input/output. ~1 token/word for English, higher ratios for other languages.
More on token ratios: aka.ms/genai-cjk

Where to split chunks?

Chunk boundary strategy	Recall@50
Break at token boundary	40.9
Preserve sentence boundaries	42.4
10% overlapping chunks	43.1
25% overlapping chunks	**43.9**

Source: aka.ms/ragrelevance

A chunking algorithm should also consider tables, and avoid splitting tables when possible.

Source: Pamela Fox, https://aka.ms/pythonai/slides/rag

Tokens – the coin of the realm

- Models process text using **tokens**

```
Many words map to one token, but some don't: indivisible.

Unicode characters like emojis may be split into many tokens containing
 the underlying bytes: ������

Sequences of characters commonly found next to each other may be grouped
 together: 1234567890
```

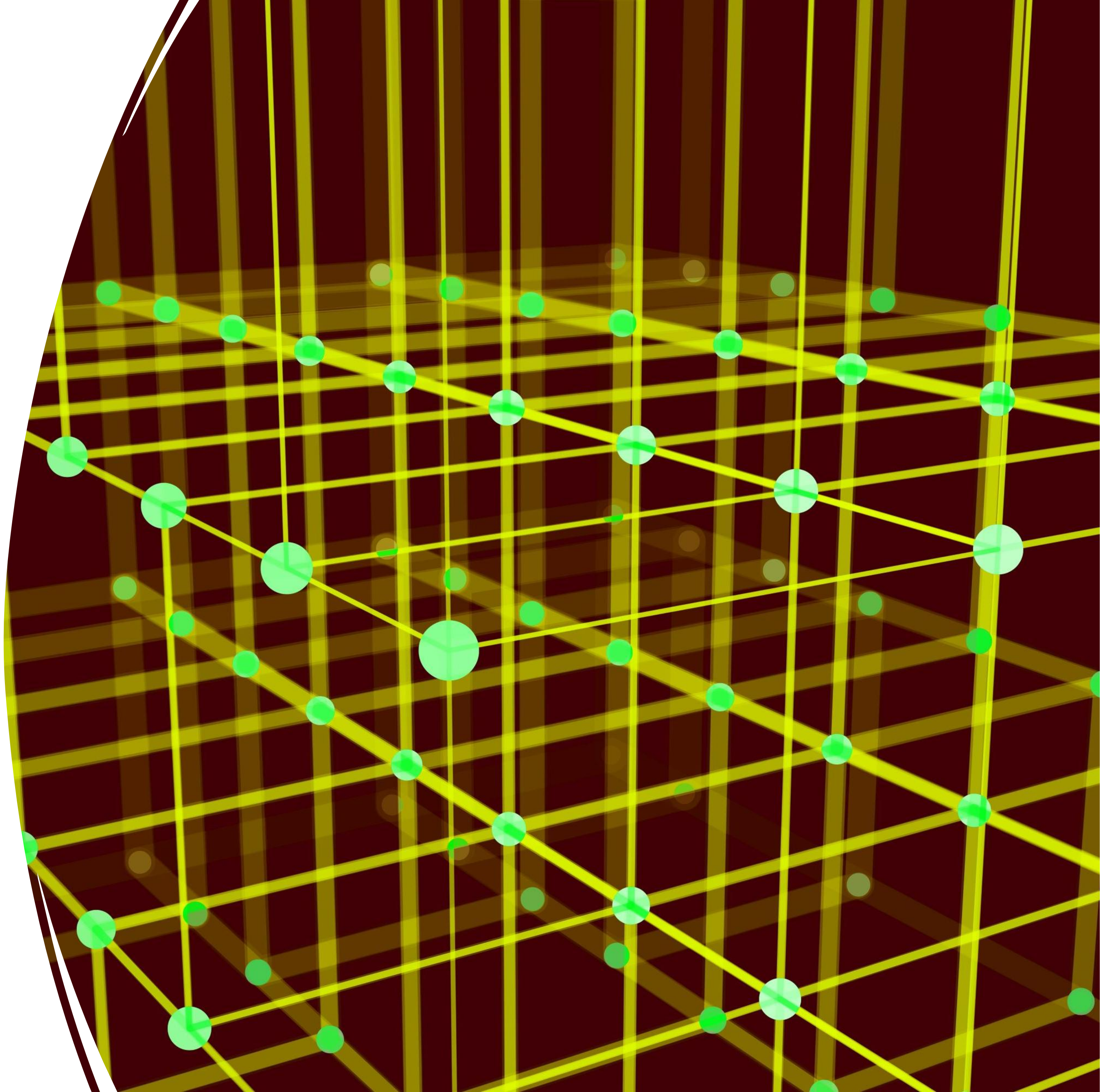
```
[7085, 2456, 3975, 284, 530, 11241, 11, 475, 617, 836, 470, 25, 773, 452,
12843, 13, 198, 198, 3118, 291, 1098, 3435, 588, 795, 13210, 271, 743,
307, 6626, 656, 867, 16326, 7268, 262, 10238, 9881, 25, 12520, 97, 248,
8582, 237, 122, 198, 198, 44015, 3007, 286, 3435, 8811, 1043, 1306, 284,
1123, 584, 743, 307, 32824, 1978, 25, 17031, 2231, 30924, 3829]
```

Each token counts towards the model's maximum limit, so it's important to consider the token count when designing your prompt (and tool calls and context).

DEMO

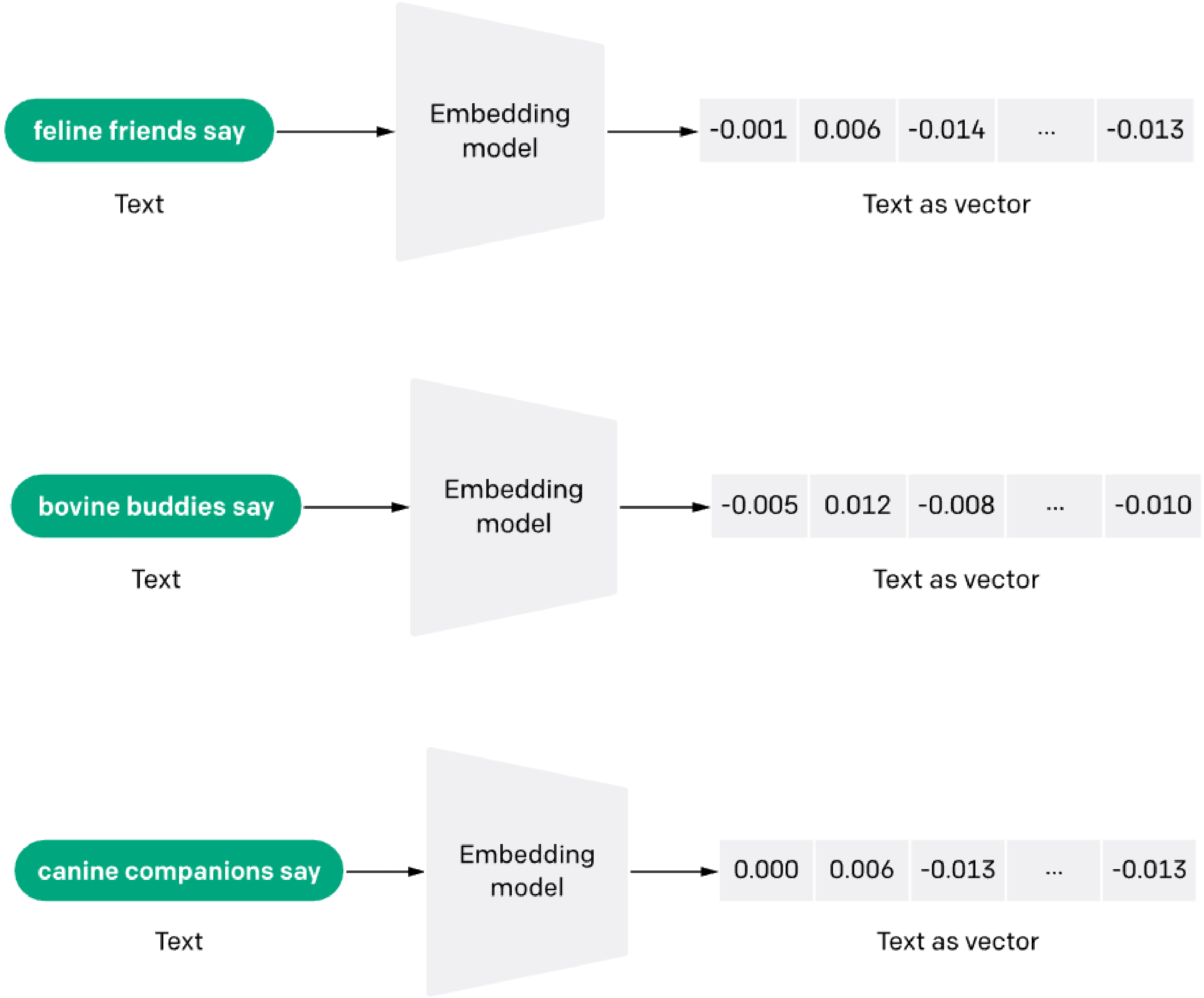
Chunking

Introduction to Vector Databases



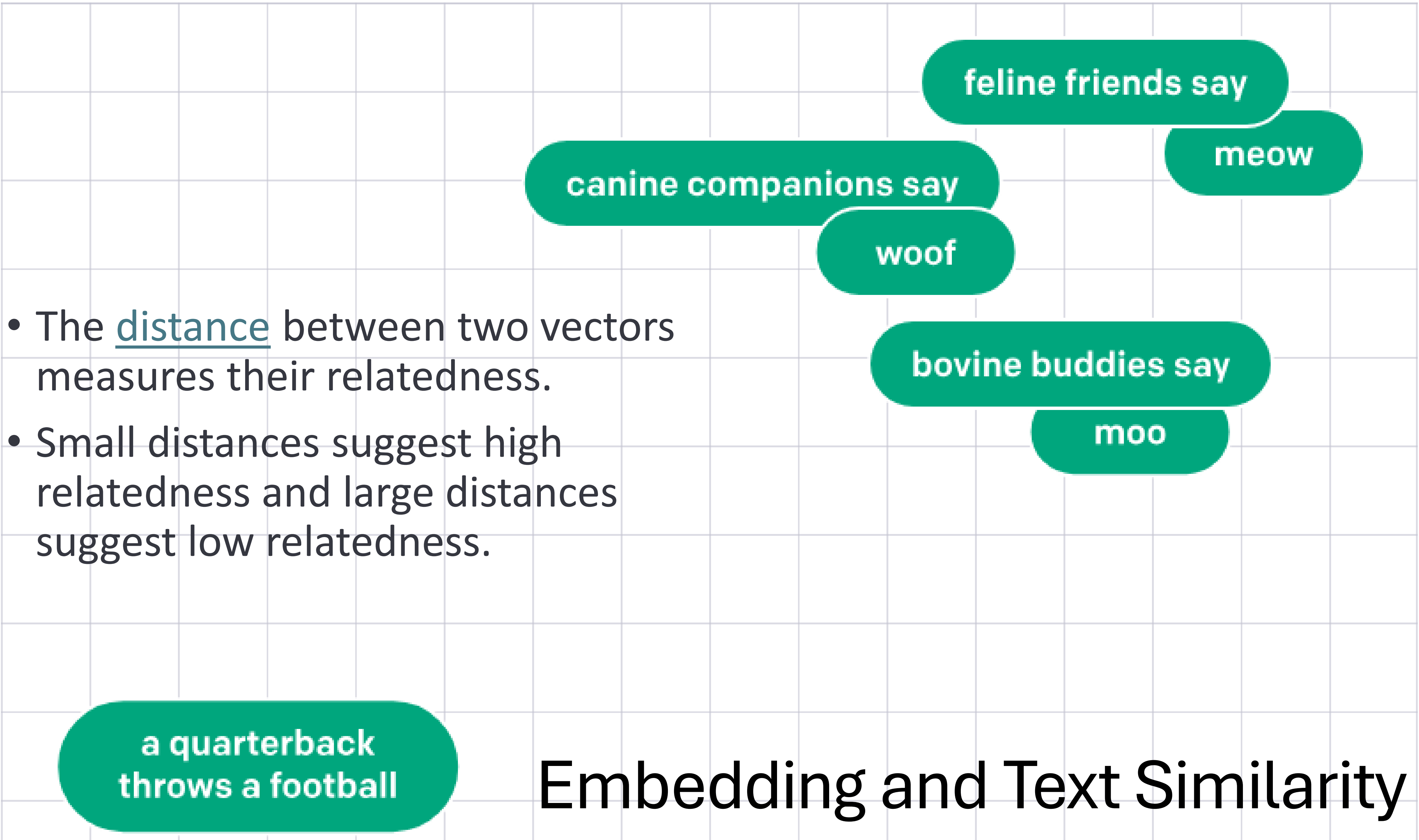
What are Vector Embeddings?

- a vector (list) of floating-point numbers
- measure the relatedness of text strings



Sources: https://openai.com/blog/introducing-text-and-code-embeddings, https://platform.openai.com/docs/guides/embeddings/what-are-embeddings

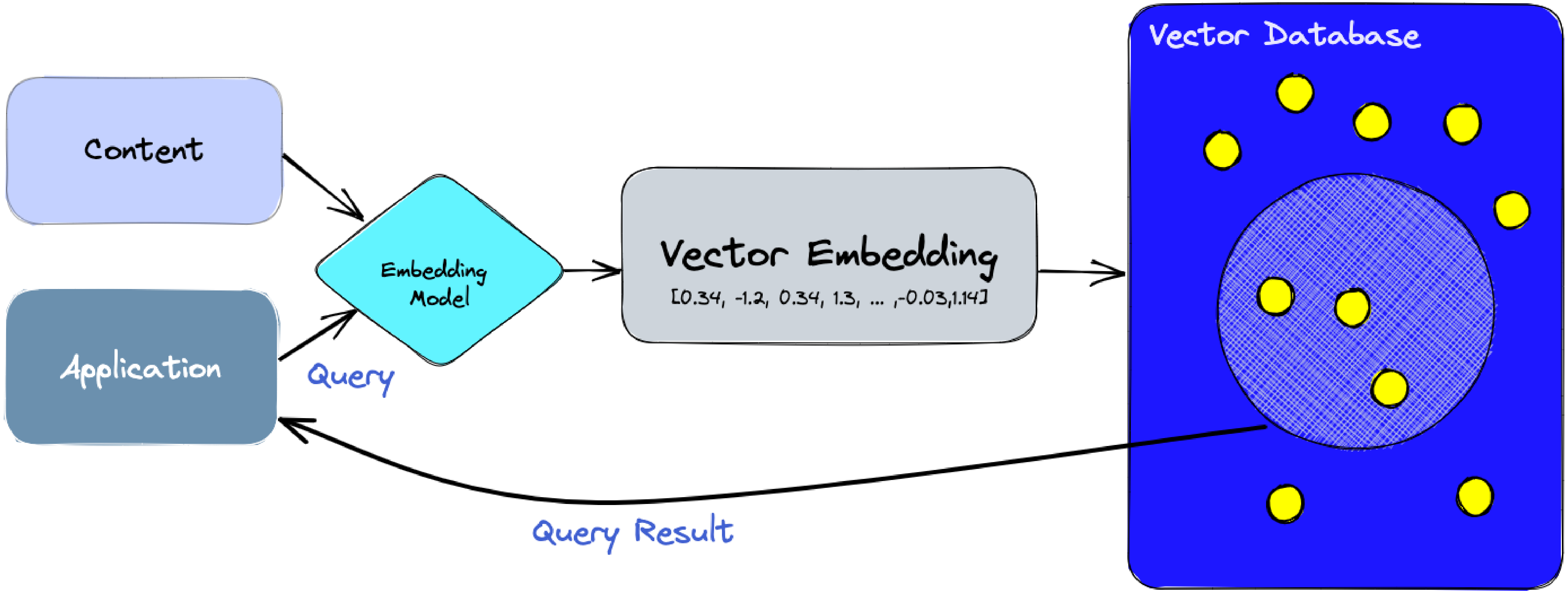
Embedding and Text Similarity



- The distance between two vectors measures their relatedness.
- Small distances suggest high relatedness and large distances suggest low relatedness.

Sources: https://openai.com/blog/introducing-text-and-code-embeddings, https://platform.openai.com/docs/guides/embeddings/what-are-embeddings

What is a Vector Database?



Source: https://www.pinecone.io/learn/vector-database/

DEMO

Embedding and Text Similarity

RAG data source types

Database rows
(Structured data)

You need a way to **vectorize** target columns with an **embedding model**.

You need a way to **search** the vectorized rows.

Documents
(Unstructured data)

PDFs, docx, pptx, md, html, images

You need an ingestion process for **extracting**, **splitting**, **vectorizing**, and **storing** document chunks.

You need a way to **search** the vectorized **chunks**.

Source: Pamela Fox, https://aka.ms/pythonai/slides/rag

RAG document ingestion

For long/unstructured documents, we need an ingestion flow such as this one:

PDF → pymupdf → LangChain → OpenAI → JSON

```
[{"id": "chunk1",
  "doc": "bee.pdf",
  "text": "the bee...",
  "vec": [0.1234..]}]
```

Extract text from PDF

Other options for this step:
Azure Document Intelligence,
LlamaParse,
LangChain document loaders,
OCR services, Unstructured, etc.

Split data into chunks

Split text based on sentence boundaries and token lengths.

You could also use "semantic" splitters and your own custom splitters.

Vectorize chunks

Compute embeddings using embedding model of your choosing.

Store chunks

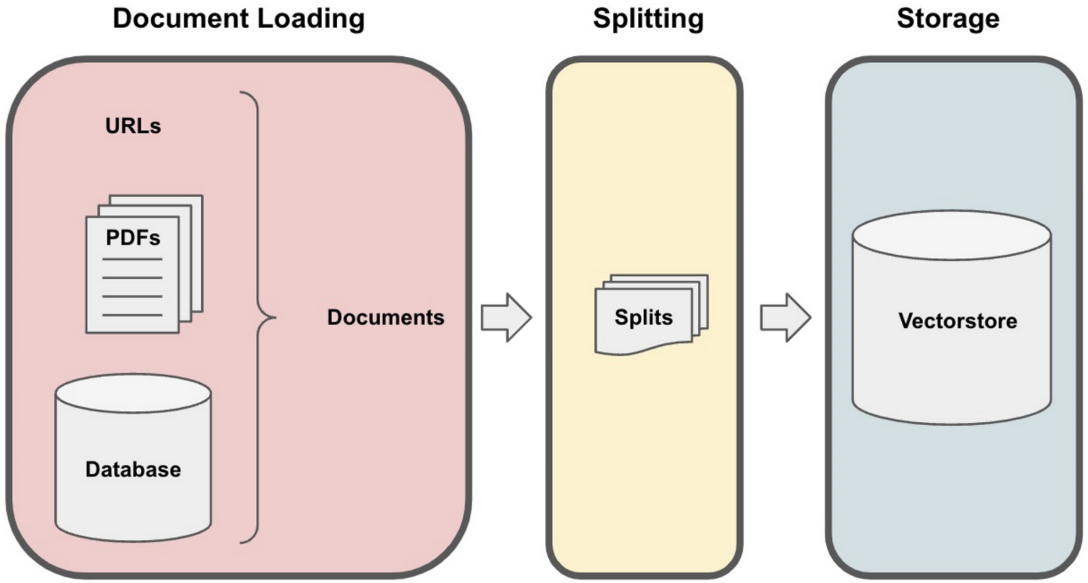
This is where you'd typically use a search service like
Azure AI Search or a database like PostgreSQL.

Source: Pamela Fox, https://aka.ms/pythonai/slides/rag

Orchestration Tools

Introduction to LlamaIndex

- Designed for efficient information retrieval and data analysis
- Optimized to handle large datasets and provide fast, accurate data access
- Built to scale with increasing data sizes and complexity, making it suitable for various applications
- Supports many different data types and retrieval methods



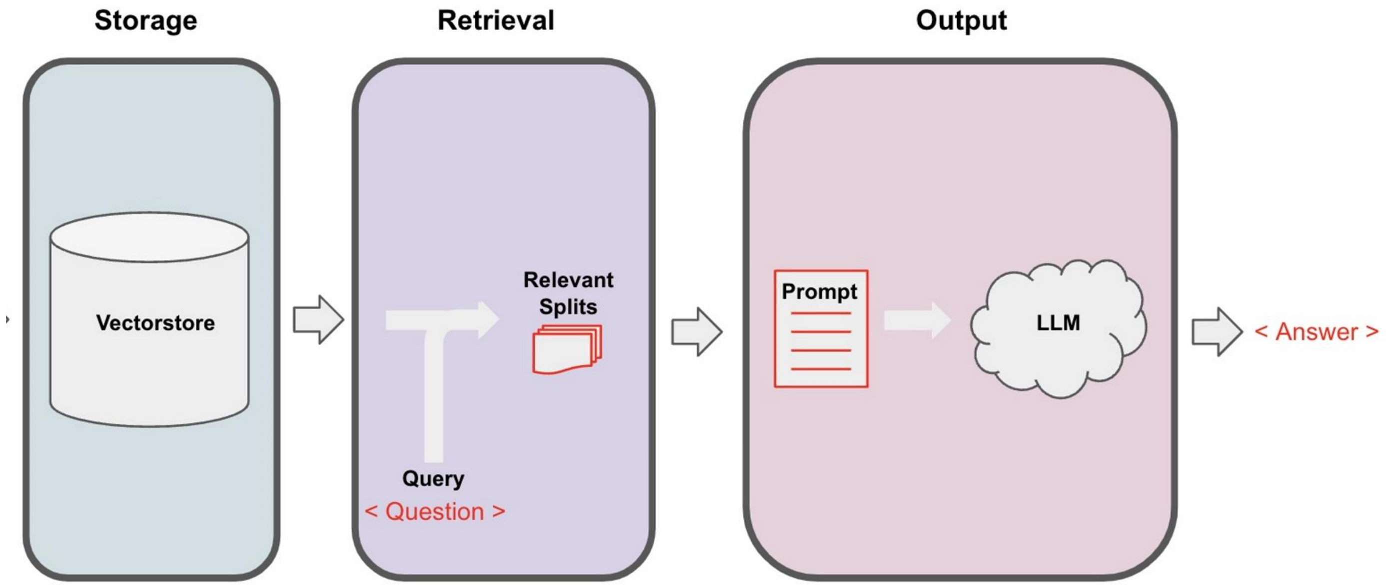
LlamaIndex: Real-World Applications

Used in applications such as:

- semantic search
- contextual data retrieval, and real-time data analysis.

Introduction to LangChain

- Powerful framework designed for creating intelligent pipelines for text generation and processing.
- Leverages the capabilities of large language models
- Chain = pipeline
- Large number of tools and frameworks integrations
- Modular design allows easy customization and integration of different components.



LangChain: Real-World Applications

Used in various applications such as:

- automated report generation
- intelligent document summarization
- real-time sentiment analysis

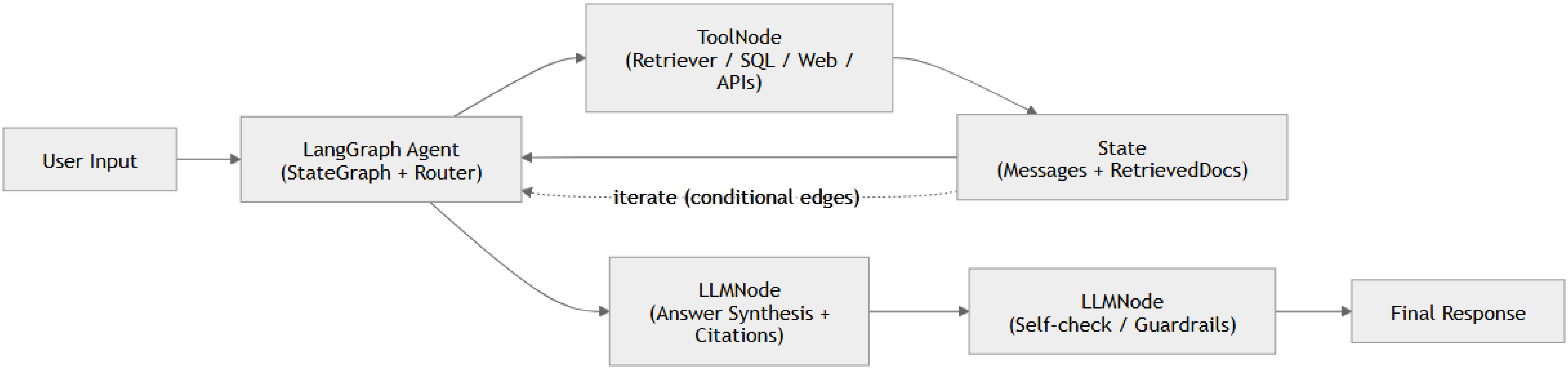
Introduction to LangGraph

- Library for building stateful, multi-actor applications with LLMs, used to create agent and multi-agent workflows
- Provides fine-grained control over both the flow and state of your agent applications
- Agentic Features
  - Human-in-the-loop
  - Parallelization
  - Subgraphs
  - Reflection

When to use LangGraph

- You need fine-grained, low-level control over agent orchestration.
- You need durable execution for long-running, stateful agents.
- You're building complex workflows that combine deterministic and agentic steps.
- You need production-ready infrastructure for agent deployment.

Agentic RAG – LangChain / LangGraph Architecture



Why plain RAG breaks

- Retrieves irrelevant chunks (keyword bias, poor query)
- Needs multiple hops (A → B → C), but retrieval is one-shot
- Conflicts across sources are not reconciled
- Answer quality depends on luck: one query, one retrieval, one pass

Combining LlamaIndex and LangChain

Agentic RAG Demo

Workflow

- Step 1: Define the data retrieval task with LlamaIndex.
- Step 2: Process and analyze the retrieved data.
- Step 3: Generate answer using LangChain.

Questions?