EPAM CONTINUUM

Unlocking the Power of LLMs
Creating RAG Applications

David Camacho

Associate Chief Software Engineer

Oct 2025

David Camacho

Associate Chief Software Engineer

David Camacho is a data & AI architect focused on **RAG and agentic architectures**, helping teams move from demos to **production-grade GenAI**. He designs **Azure-centric lakehouse** stacks with strong **DevSecOps and IaC** practices (Terraform/Bicep), builds **ETL/ELT pipelines**, **feature stores**, and **LLM-powered applications**, and leads hands-on workshops on **Graph RAG, Reflective RAG, and other advanced patterns**. With a track record across Latin America, he aligns engineering, governance, and cost controls to deliver **reliable, observable, and scalable** AI services.

As **Associate Chief Software Engineer**, David leads architecture and delivery for AI platforms, mentors teams, and sets coding/quality standards—turning **prototype RAG ideas** into **secure, maintainable, and scalable** products.

Outline

01 Intro and Project Setup

02 RAG, Vectorial Data Bases

03 Advanced RAG + Coding Time

04 Graph RAG + Coding Time

05 Reflective RAG + Coding Time

06 Metrics, Quality and Comparison

07 Takeaways

Intro and Project Set up

Setting up the project:

1. Go to github: https://github.com/LuisDavidCamacho/rag_architecture_workshop
2. Go to branch advanced RAG
3. Clone the repo
4. Download the enron dataset from: https://www.kaggle.com/datasets/wcukierski/enron-email-dataset
5. Install docker if you don’t have it yet
6. Open your IDE pointing to the cloned repo
7. To start the project, you will need to run the compose file

Repo content:

- Enron email data set csv file
- Poetry toml file to manage dependencies
- Chat front end
- Vectorial database
- Ollama local model (we will use local llama model to avoid public LLM service auth)
- Fast API backend (you will work here)
- Docker files and docker compose files

RAG adds context. The way you add it makes all the difference.

What is RAG?

Retrieval-Augmented Generation is a technique that enriches an LLM's answers by injecting external, relevant context at inference time.

Naïve RAG

- Injects raw document text directly into the prompt.
- Fast to implement but lacks control.
- Risks including irrelevant or excessive context.
- No scoring or filtering — everything goes in.

Advanced RAG

- Embeds and stores the document corpus in a vector database.
- At runtime, the user's query is embedded and matched against the corpus.
- Only the **most relevant snippets** are retrieved and passed to the LLM.
- Scalable, efficient, and measurable.

Embeddings unlock meaning. Vector DBs make it searchable.

What are Embeddings?

- Numerical representation of text that captures **semantic meaning**.
- Allows similarity search using **distance metrics** (cosine, Euclidean, etc.).
- Critical in RAG for finding **relevant context** across large corpora.

Why Use a Vector Database?

- Optimized for storing and querying **millions of embeddings** efficiently.
- Fast **approximate nearest neighbor (ANN)** search.
- Supports **filters, metadata, hybrid search**, and scalability.
- Built for GenAI, not traditional SQL search.

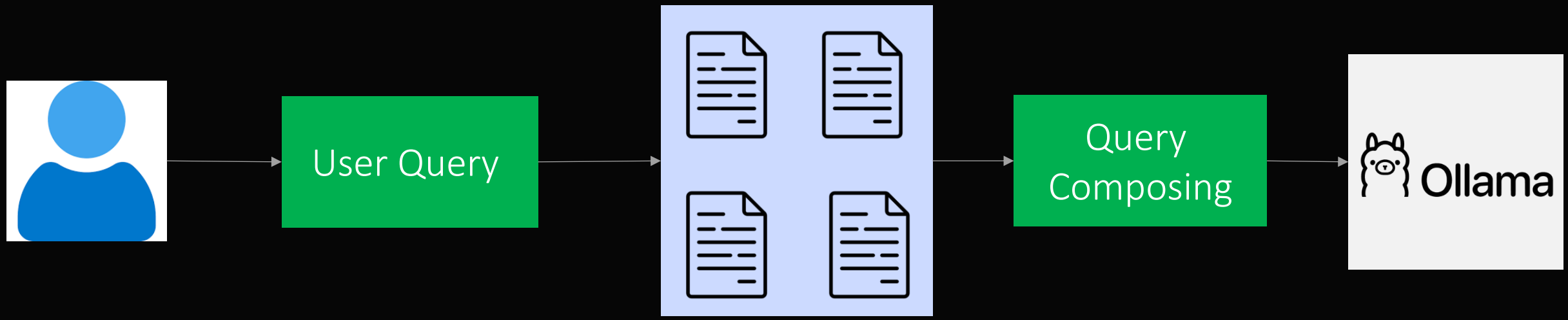
Popular Vector DB Providers & Features

Vendor	Approach	Highlights
FAISS	Local, open-source	High-performance, no metadata
Qdrant	Open-source / hosted	Metadata filters, REST/gRPC
Pinecone	Cloud-native SaaS	Scalable, hybrid search, API-first
Weaviate	Open-source / hosted	Schema, hybrid search, GraphQL
Milvus	Distributed open-source	High-scale, community-supported

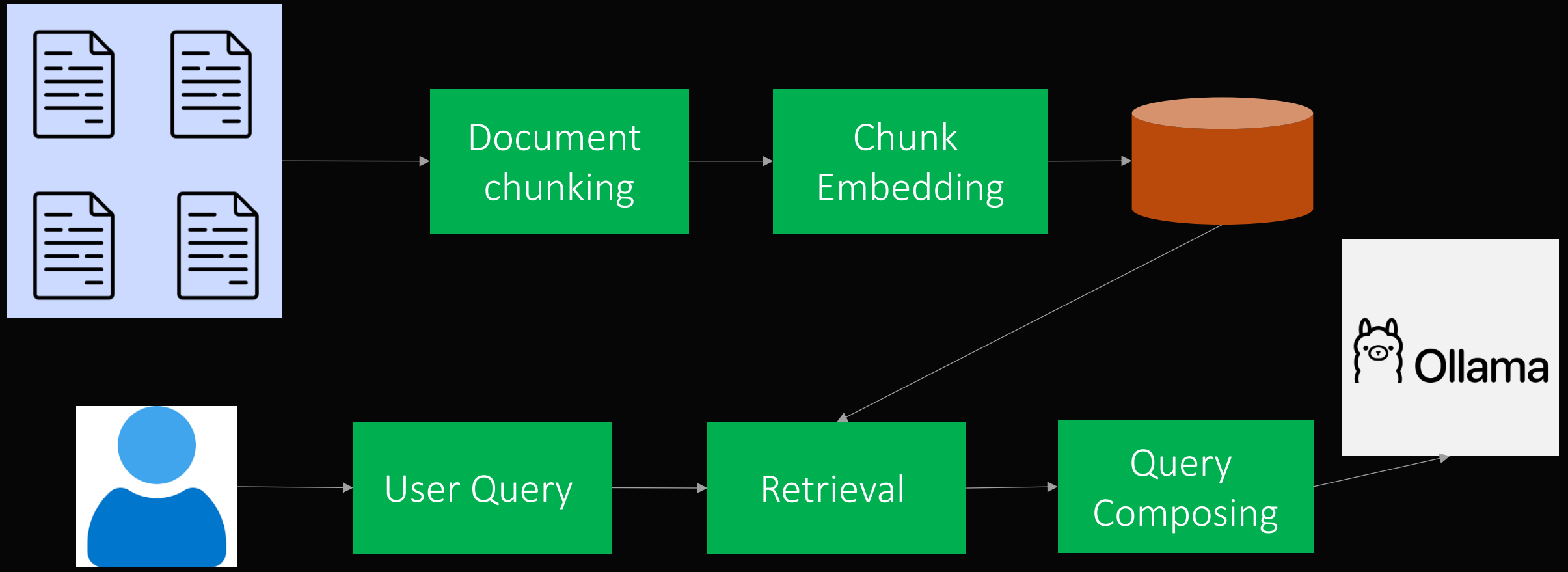
Common Embedding Models

Model	Source	Dim	Notes
text-embedding-3-small	OpenAI	1536	Fast, accurate, ideal for RAG
all-MiniLM-L6-v2	HuggingFace	384	Lightweight, good general-purpose
bge-base-en-v1.5	BAAI (OSS)	768	Optimized for semantic search
e5-base	HuggingFace	768	Fine-tunable, versatile

RAG and Advanced RAG Architectures



RAG and Advanced RAG Architectures



Graphs brings structure to your retrieval.

What is Graph RAG?

- Graph RAG enhances retrieval by using a **knowledge graph** built from the document corpus. It captures structured relationships between entities (e.g., people, places, concepts, components) and uses that structure to improve relevance and reasoning.

Key Components:

- **Graph Constructor**: Extracts entities/relations from docs using NLP (NER, OpenIE).
- **Knowledge Graph DB**: Stores entities and their relationships (Neo4j, RDF, NetworkX).
- **Retriever + Graph Traversal**: Uses graph paths or filters to narrow down context candidates.
- **LLM Prompt Composer**: Uses graph-selected context, often more focused and interpretable.

Structure improves reasoning and reduces noise.

Precision Improvement

- Graph constraints help **narrow down** the set of possible context nodes.
- You retrieve **fewer, more relevant chunks**—especially in large, dense corpora.

Better Multi-Hop Reasoning

- Graph paths enable connecting **indirect relationships** (e.g., X → Y → Z).
- LLM can generate answers that require **reasoning across multiple entities**.

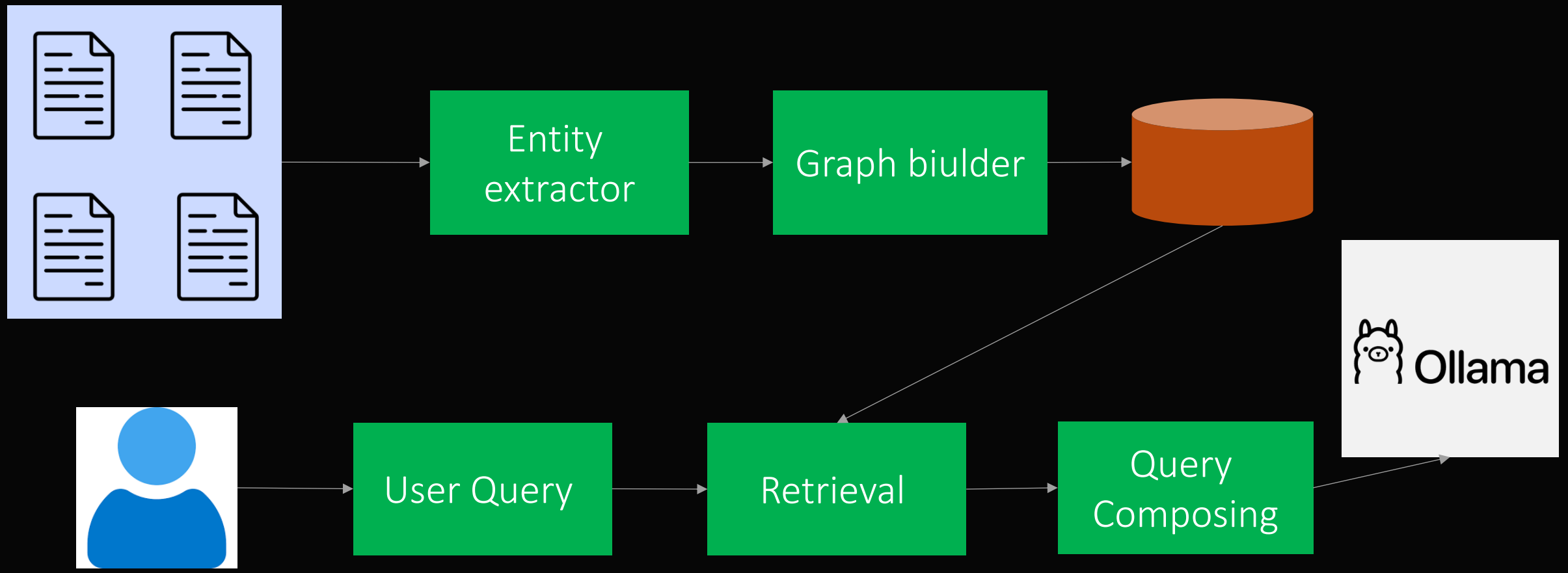
Interpretability

- You can **trace the path** that led to the retrieved nodes (e.g., in Neo4j).
- Helps for debugging, auditability, and trust in output.

Domain Fit

- Works great for **technical documentation**, **product catalogs**, **legal logic trees**, or **biomedical ontologies**.

Graph RAG Architecture



Reflective RAG: When one LLM isn’t enough.

What is Reflective RAG?

- Reflective RAG introduces a **second reasoning step**: once an answer is generated by the initial LLM, a **reflection agent** (usually another LLM call or prompt chain) evaluates, critiques, or rewrites the answer.

It may:

- Detect hallucinations
- Refine vague or incomplete answers
- Offer citations or confidence scores
- Compare multiple candidate responses

Where Reflection Adds Value

- Regulated environments: legal, medical, finance
- High-stakes domains: safety, contracts
- Customer-facing systems needing **factual reliability**

Better answers, at a cost.

Improvements

- **Reduces hallucinations** by checking source alignment.
- Adds **self-consistency**: answers are reranked or rewritten with better coherence.
- **Confidence-aware**: can generate citations, uncertainty indicators, or retry fallback logic.
- Promotes **multi-agent behavior** — a step toward agentic architectures.

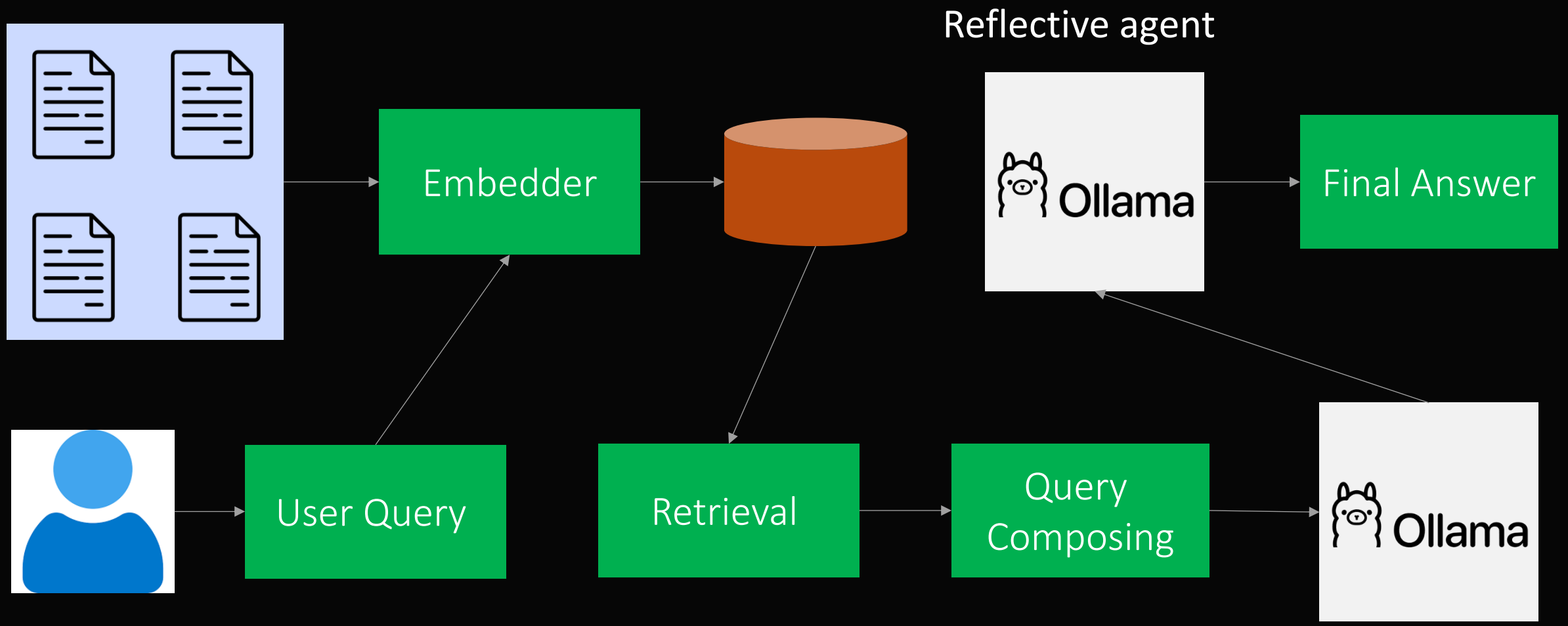
Pros

- Boosts **accuracy and trustworthiness**
- Great for **auditable or critical environments**
- Easier to detect weak responses and **recover gracefully**
- Modular: can use the same LLM or a cheaper model as the critic

Cons

- **Higher cost** (2x+ LLM calls or more)
- **Increased latency**: double the inference time
- Needs **careful prompt design** to avoid biasing the reflection
- Complexity in chaining, scoring, or retry logic

Graph RAG Architecture



Comparing The Models

Feature / Metric	Advanced RAG	Graph RAG	Reflective RAG
Context Selection	Embedding similarity	Graph traversal + filter	Embedding + reflection scoring
Multi-hop Reasoning	Limited	Supported	Partial (via critique)
Factuality Check	None	Limited via structure	Strong via reflection agent
Interpretability	Opaque	Graph traceable	Depends on reflection prompt
Latency	Low	Medium	High (multi LLM calls)
Cost	Efficient	Medium	High
Use Case Fit	General Q&A	Technical, Structured Data	High-stakes, Regulated Content

How do you measure a good RAG response?

1. Context Precision at K

Measures how many of the top-K retrieved chunks are actually relevant to the user's query.

How to calculate:

- For each query, store expected reference answer or tags.
- Compare top-K retrieved chunks against gold annotations or keyword overlap.

```
precision_at_k = num_relevant_chunks / k
```

How do you measure a good RAG response?

2. Answer Grounding Score

Checks how well the answer is grounded in the retrieved context (vs hallucinated).

How to calculate:

- Use fuzzy string matching (e.g., fuzz.partial_ratio) between the answer and the context.
- Optionally: prompt another LLM to judge if the answer is supported by the provided text.

```
from fuzzywuzzy import fuzz
grounding_score = fuzz.partial_ratio(answer,
                                     context_text)
```


How do you measure a good RAG response?

3. Latency / Cost Per Query

Tracks inference performance for each RAG variant.

How to calculate:

- Log timestamps at each pipeline stage.
- Count number of LLM calls + token usage.

```
import time
start = time.time()
# run RAG pipeline
elapsed = time.time() - start
```

Thank you!

RAG Architectures Workshop

Python Brazil Oct 2025

David Camacho

david_camacho@epam.com
https://www.linkedin.com/in/luisdcamachog/

Join us at EPAM!
Contact me if you're interested in joining.
https://invite.epam.com