Databricks Certified Generative AI Engineer Associate Module 2025

Definition:

The Databricks Certified Generative AI Engineer Associate certification exam assesses an individual’s ability to design and implement LLM-enabled solutions using Databricks. This includes problem decomposition to break down complex requirements into manageable tasks as well as **choosing appropriate models, tools, and approaches** from the current generative AI landscape for developing comprehensive solutions.

It also assesses Databricks-specific tools such as

1. **Vector Search** for semantic similarity searches,
2. **Model Serving** for deploying models and solutions,
3. **MLflow** for managing solution lifecycle, and
4. **Unity Catalog** for data governance.

These questions are segmented into six Pillars and 45 sub-segments:

- Design Applications — 14%
- Data Preparation — 14%
- Application Development — 30%
- Assembling and Deploying Apps — 22%
- Governance — 8%
- Evaluation and Monitoring — 12%

Section 1: Design Applications

Section 1: Design Applications

1. Design a prompt that elicits a specifically formatted response

Tips for Effective Prompt Engineering

1. **Model-Specific Prompts**
2. **Iterative Development**
3. **Avoiding Bias and Hallucinations**: Include instructions to avoid generating false information. Example: “Do not make things up if you do not know. Say ‘I do not have that information’.”
4. **Use Delimiters**: Use delimiters to distinguish between instruction and context, such as ###, ` ` ` ` ` `, {}, ` [] ` , or `
5. **Structured Output:** “Return the movie name mentioned in the form of a JSON object. The output should look like {‘Title’: ‘In and Out’}.”

Examples of Advanced Prompting Techniques

- **Zero-shot Prompting** — Without examples
- **Few-shot Prompting** — Fancy word for more than 1 example in a prompt.
- **Prompt Chaining** — Chain of prompts

Key Elements of a Good Prompt:

1. **Clear Instruction**: Provide a clear directive specifying what the model should do.
2. **Contextual Information**: Include background or additional information that helps the model understand the task.
3. **Input / Question**: Specify the query or data the model needs to process.
4. **Output Type / Format**: Define the desired structure or style of the response.

2. Select model tasks to accomplish a given business requirement

Key Steps in Selecting Model Tasks

1. **Identify Business Objectives**
2. **Decompose the Problem**: Break down the overall objective into smaller, manageable tasks. Example: For improving customer service, tasks might include sentiment analysis, FAQ retrieval, and automated response generation.
3. **Task Mapping**: Match the business objectives to specific AI model tasks; you could use multiple LLM for specilized tasks
4. **Select Appropriate Models**: MPT VS Chatgpt VS Lamma VS BERT or RoBERTa Vs other models; Which all models are open source? Text generation model vs other kinds of model.

5. **Consider the Interaction Between Tasks**: Sentiment analysis might be performed first, followed by retrieval of FAQs, and finally, response generation.

6. **Utilize Tools and Frameworks**: Using LangChain for creating multi-stage reasoning chains that handle complex workflows.

7. **Evaluate, Optimize and Iterate**: Continuously evaluate and optimize the models and tasks to ensure they meet the business requirements based on performance, accuracy, and context applicability. Example: Regularly updating the training data and fine-tuning models to improve performance.

3. Select chain components for a desired model input and output

Frameworks and Libraries for Building Chains

1. LangChain, 2. LLamaIndex, 3. OpenAI Agents

Components:

- **Chain:** An ordered sequence of components that processes information step-by-step for a specific AI task.
- **Prompt:** The text input or instruction designed to guide an AI model's response.
- **Retriever**: A component that fetches relevant documents or information from a source to support the AI's output.
- **Tool or ChatGPT Function Calling:** Mechanisms that allow the AI to interact with external functions, APIs, or tools for additional capabilities.
- **LLM:** A large language model that generates human-like text responses based on given input.

Integration and Implementation

1. **Framework Integration**: Use LangChain to build and manage the chain components. Integrate with databases and external APIs for dynamic data retrieval and interaction.

2. **Logging and Monitoring**: Use MLflow to log the performance of each component in the chain. Monitor the entire workflow to ensure high performance and accuracy.

3. **Optimization**: Continuously evaluate the chain's performance and make necessary adjustments. Update models and retrain as needed to maintain accuracy and relevance.

4. Translate business use case goals into a description of the desired inputs and outputs for the AI pipeline

Implementation Considerations

1. **Data Quality**
2. **Model Selection**: Choose models that are well-suited for each task — accuracy VS speed VS cost VS open source VS regulations VS hosting cost VS performance — relevance to the business goals.
3. **Integration** of all components
4. **Testing and Optimization**

5. Define and order tools that gather knowledge or take actions for multi-stage reasoning

Patterns for Agent Reasoning:

1. **ReAct (Reason + Act)**:

- **Thought or Reason**: Reflect on the problem given and previous actions taken.
- **Act**: Choose the correct tool and input format to use.
- **Observe (Continues to Reason)**: Evaluate the result of the action and generate the next thought.

2. **Tool Use / Function Calling**: Agents interact with external tools and APIs to perform specific tasks.

Example Tools:

- **Research / Search Tools**: Web browsing, search engines, Wikipedia.
- **Document Retrieval**: Database retriever, vector DB retriever, document loader.
- **Image Processing**: Image generation, object detection, image classification.
- **Coding**: Code execution, documentation generator, debugging/testing .

3. **Planning**: Agents must dynamically adjust their goals and plans based on changing conditions.

Tasks:

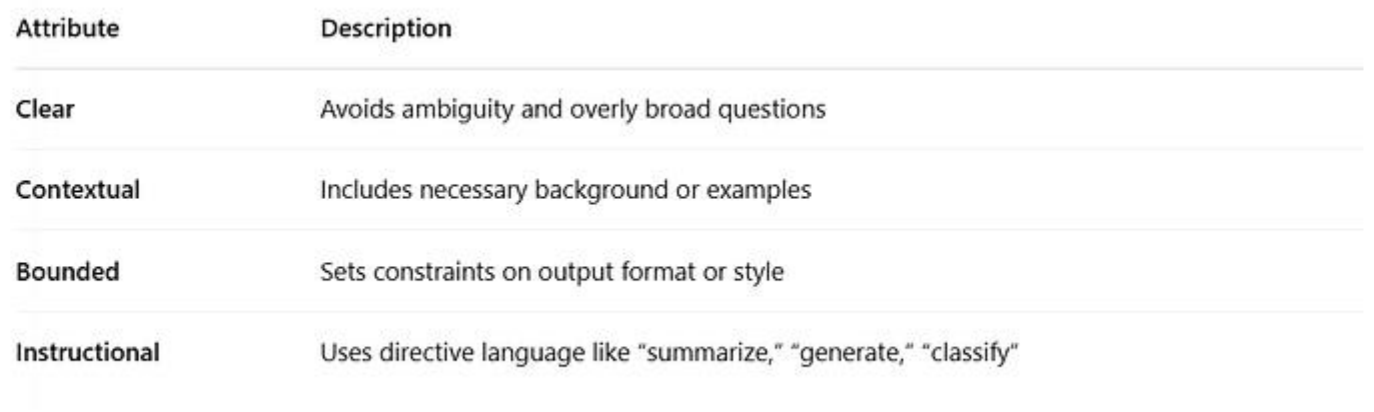
- **Single Task**: A straightforward task with a single goal.
- **Sequential Task**: Tasks that need to be performed in a specific order.
- **Graph Task**: Complex tasks that involve multiple interdependent actions.

4. **Multi-Agent Collaboration**: Multiple agents work collaboratively, each handling different aspects of a complex task.

- **Benefits**: Allows modularization and specialization.

Anatomy of an Effective Prompt

A high-quality prompt is:



Attribute	Description
Clear	Avoids ambiguity and overly broad questions
Contextual	Includes necessary background or examples
Bounded	Sets constraints on output format or style
Instructional	Uses directive language like "summarize," "generate," "classify"

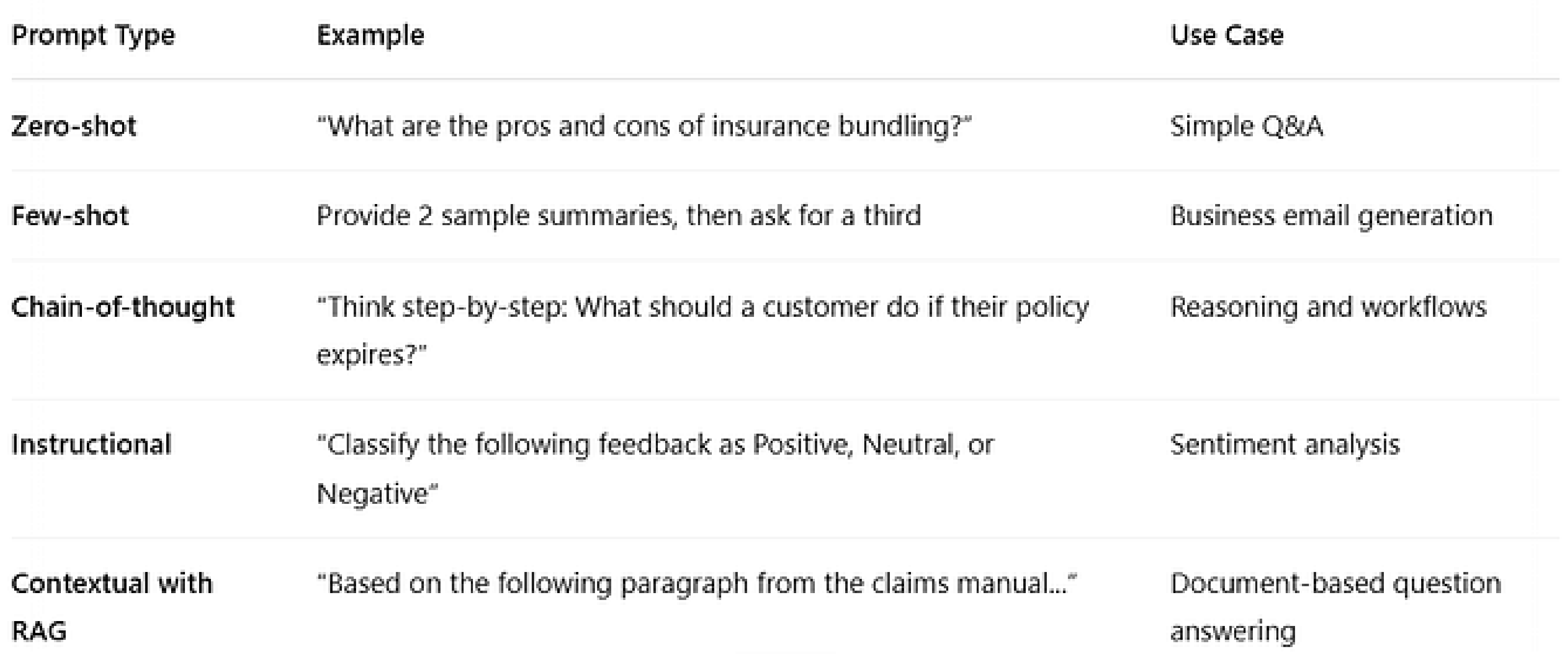
Example:

"You are a helpful assistant trained to summarize insurance policy documents. Summarize the key coverage conditions in 3 bullet points using non-technical language."

This sets:

- The **role** (a helpful assistant)
- The **task** (summarization)
- The **format** (3 bullet points)
- The **style** (non-technical)

Prompt Types in Enterprise Applications



Prompt Type	Example	Use Case
Zero-shot	"What are the pros and cons of insurance bundling?"	Simple Q&A
Few-shot	Provide 2 sample summaries, then ask for a third	Business email generation
Chain-of-thought	"Think step-by-step: What should a customer do if their policy expires?"	Reasoning and workflows
Instructional	"Classify the following feedback as Positive, Neutral, or Negative"	Sentiment analysis
Contextual with RAG	"Based on the following paragraph from the claims manual..."	Document-based question answering

Prompt Engineering on Databricks

Databricks provides two core frameworks for working with prompts:

- **LangChain**
- **Mosaic AI Prompt Playground**

LangChain Example

Create chains of prompts and responses to execute multi-step AI workflows.

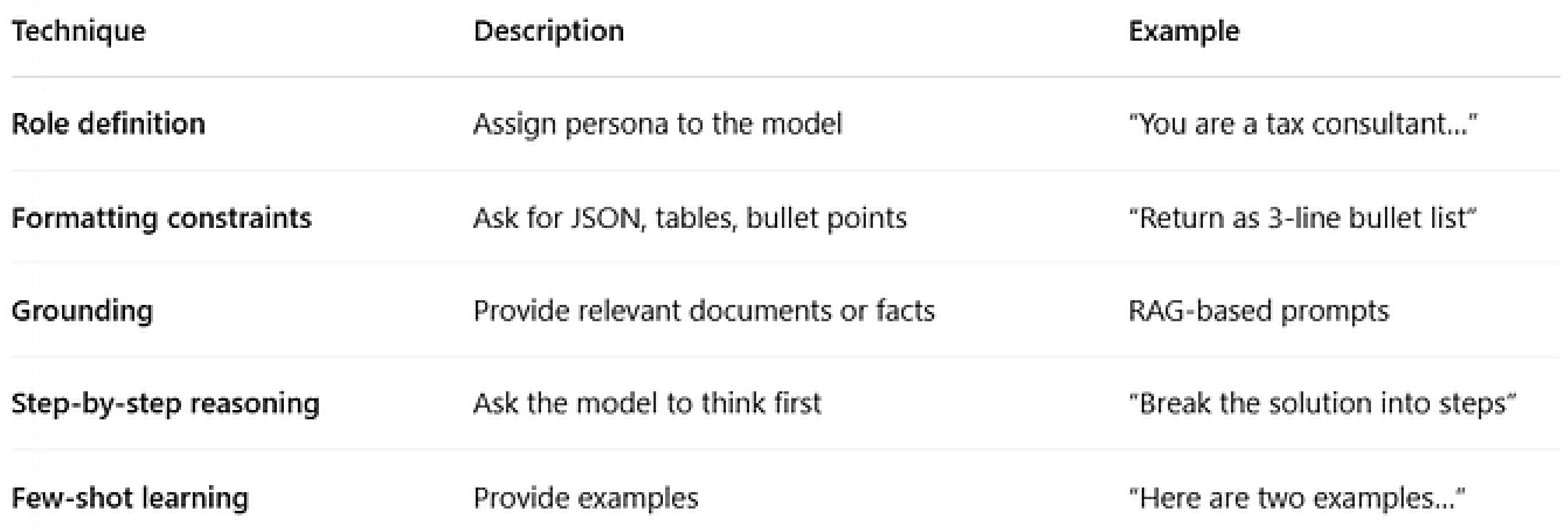
```
from langchain.prompts import PromptTemplate

prompt_template = PromptTemplate(
input_variables=["product", "features"],
template="Generate a concise product summary for {product} highlighting:
{features}."
)

prompt = prompt_template.format(product="Auto Insurance",
features="accident forgiveness, roadside assistance")
```

Mosaic AI Prompt Playground

Mosaic AI lets you:

- Experiment with prompts in a UI
- Compare model responses across providers (DBRX, GPT-4, MPT)
- Save and productionize prompts into chains or agents
- Evaluate hallucination, tone, style consistency

Techniques to Improve Prompt Performance



Technique	Description	Example
Role definition	Assign persona to the model	"You are a tax consultant..."
Formatting constraints	Ask for JSON, tables, bullet points	"Return as 3-line bullet list"
Grounding	Provide relevant documents or facts	RAG-based prompts
Step-by-step reasoning	Ask the model to think first	"Break the solution into steps"
Few-shot learning	Provide examples	"Here are two examples..."

Key Evaluation Metrics:

1. **Relevance**: Is the response on-topic?
2. **Factuality**: Are the facts accurate?
3. **Completeness**: Are all user queries addressed?
4. **Style**: Does it match tone and format expectations?

Sample Business Scenario:

You want to create a Gen AI assistant that answers customer queries about health insurance.

Bad Prompt:

“Explain health insurance.”

Good Prompt:

“You are an expert health insurance agent. A customer has asked: ‘What is the difference between co-pay and deductible?’ Provide a concise answer using customer-friendly language and examples.”

Real-World Example: Customer Support Assistant

Goal: Build a prompt for a RAG-powered assistant to explain insurance clauses to a customer.

Real-World Example: Customer Support Assistant

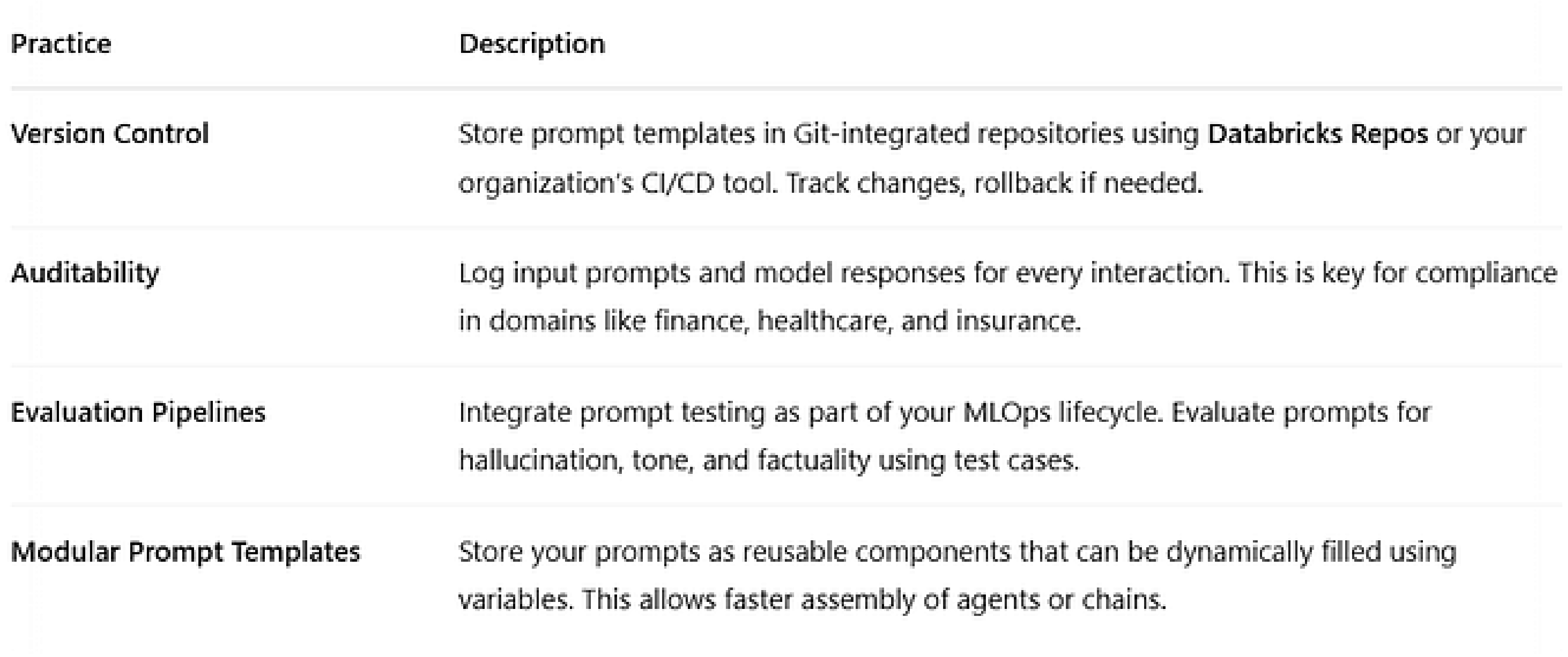
Goal: Build a prompt for a RAG-powered assistant to explain insurance clauses to a customer.

```
prompt = f"""
You are a support assistant for a health insurance company.
Use the context below to answer the user's question in a friendly, accurate manner.

Context: {retrieved_doc_chunks}

Question: {user_question}

Answer:
"""
```

Governance of Prompts in Enterprise Settings

In production environments, prompt engineering isn't just about writing better prompts — it's about **managing them at scale** with consistency, traceability, and accountability.

Key Governance Practices:



Practice	Description
Version Control	Store prompt templates in Git-integrated repositories using **Databricks Repos** or your organization's CI/CD tool. Track changes, rollback if needed.
Auditability	Log input prompts and model responses for every interaction. This is key for compliance in domains like finance, healthcare, and insurance.
Evaluation Pipelines	Integrate prompt testing as part of your MLOps lifecycle. Evaluate prompts for hallucination, tone, and factuality using test cases.
Modular Prompt Templates	Store your prompts as reusable components that can be dynamically filled using variables. This allows faster assembly of agents or chains.

Prompt Repository (A Must-Have)

A **Prompt Repository** is a central location — often a Git-backed folder structure or a managed catalog — where all prompt templates are stored, versioned, and governed.

Benefits of a Prompt Repository:

- Centralized prompt governance
- Role-based access for sensitive prompts
- Easier experimentation through branches or forks
- Collaboration between engineering, product, and compliance teams

In Databricks, you can integrate this with:

- **MLflow** to track usage per model version
- **LangChain** to load templates dynamically
- **Databricks Unity Catalog** for access controls

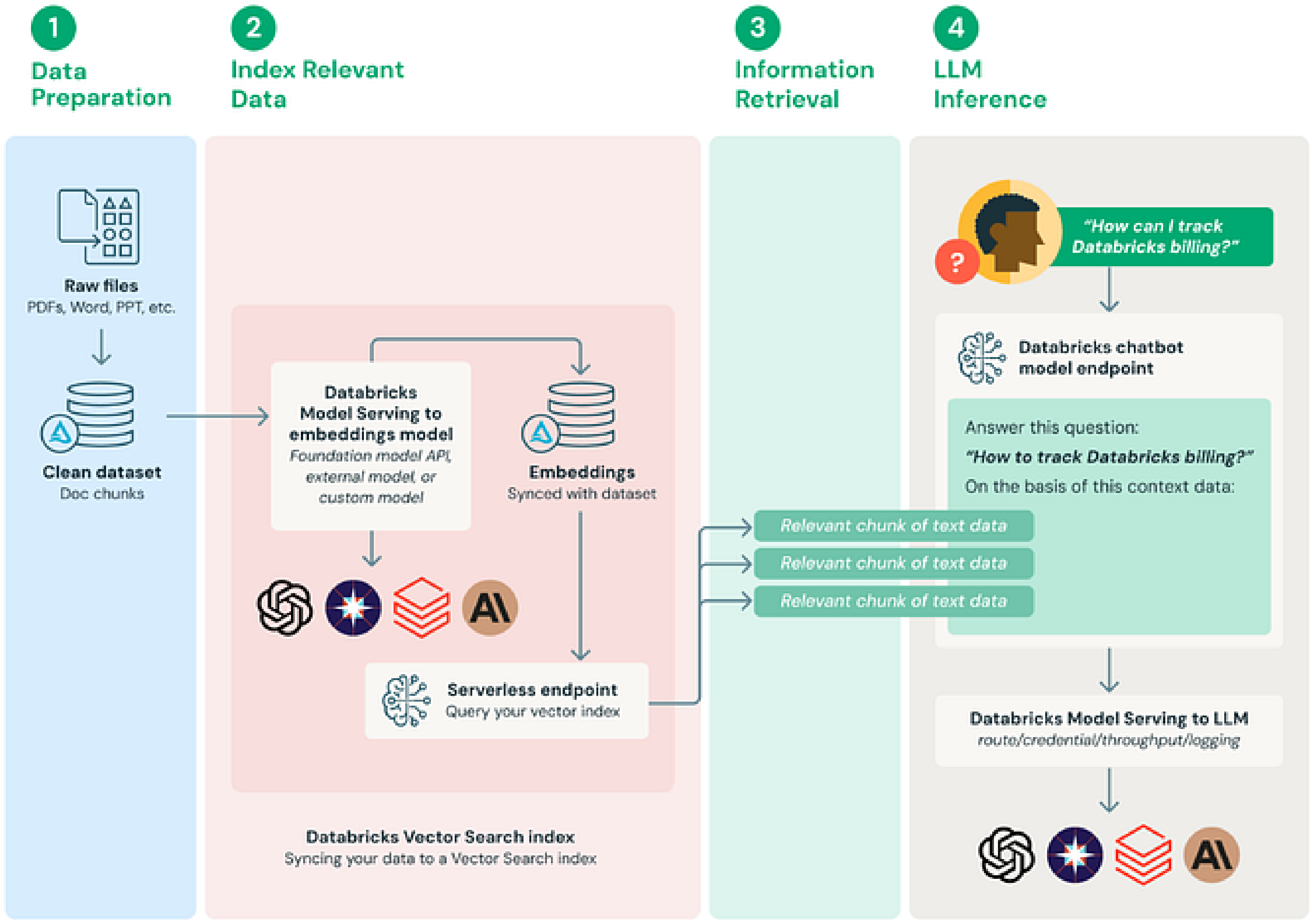
Summary

Concept	Takeaway
Prompt Engineering	The process of designing effective instructions for LLMs
Good Prompt	Clear, contextual, bounded, instructional
Prompt Types	Zero-shot, few-shot, chain-of-thought, contextual
On Databricks	Use LangChain, Mosaic AI Prompt Playground
Evaluation	Assess relevance, accuracy, style
Governance	Store, version, monitor and audit prompts
Prompt Repository	A centralized, versioned store of prompt templates

Section 2: Data Preparation

EXL

Understanding Retrieval-Augmented Generation (RAG)


1 Data Preparation
2 Index Relevant Data
3 Information Retrieval
4 LLM Inference
Raw files
PDFs, Word, PPT, etc.
Clean dataset
Doc chunks
Databricks Model Serving to embeddings model
Foundation model API, external model, or custom model
Embeddings
Synced with dataset
Serverless endpoint
Query your vector index
Databricks Vector Search index
Syncing your data to a Vector Search index
Relevant chunk of text data
Relevant chunk of text data
Relevant chunk of text data
"How can I track Databricks billing?"
Databricks chatbot model endpoint
Answer this question:
"How to track Databricks billing?"
On the basis of this context data:
Databricks Model Serving to LLM
route/credential/throughput/logging

Generative AI models, especially large language models (LLMs) like **DBRX**, **GPT-4**, or **MPT**, are trained on vast amounts of public data. They possess strong language understanding and generation capabilities. However, they are inherently **static and general-purpose**, meaning:

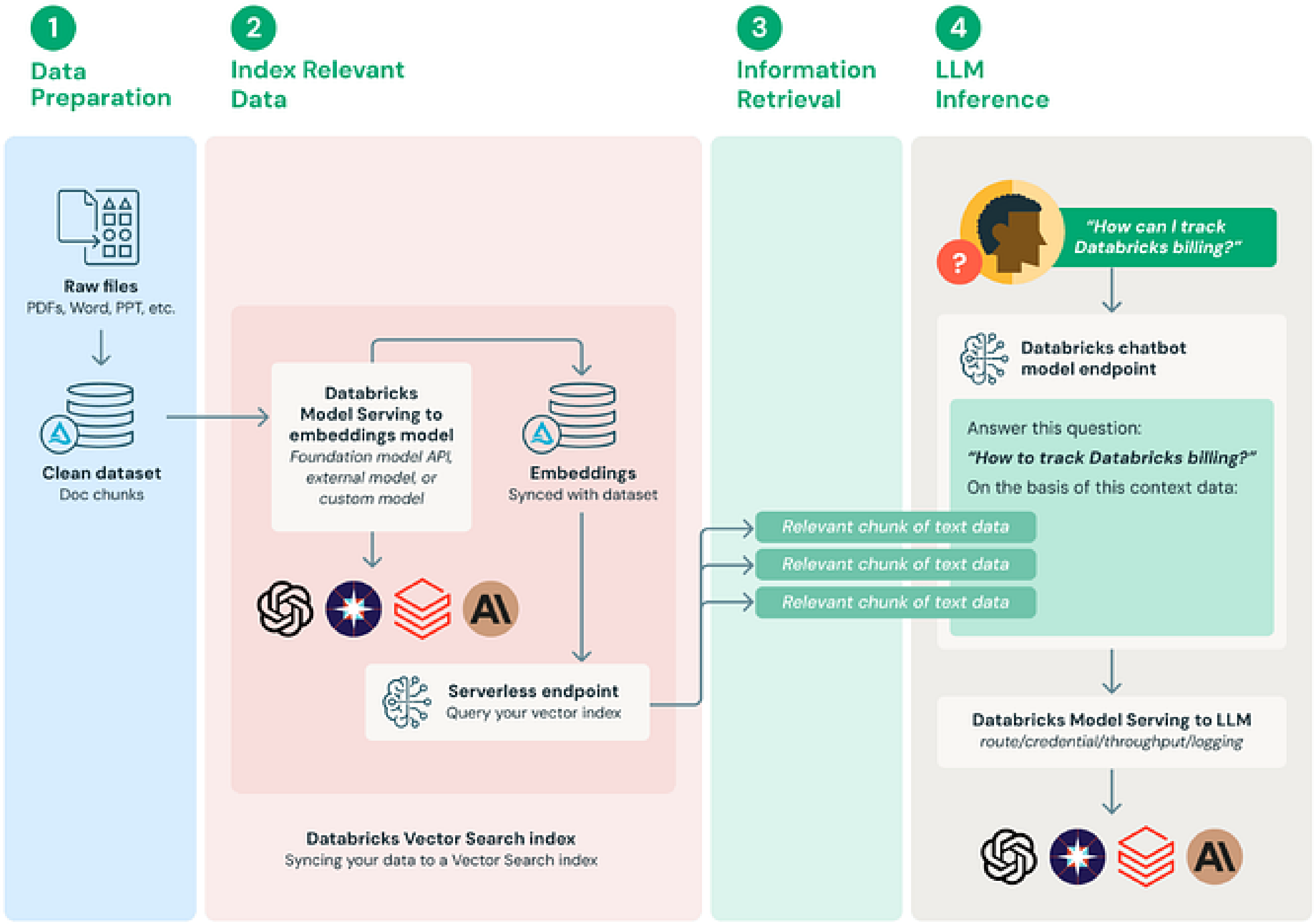
1. They don't know your **proprietary enterprise data** (internal manuals, client documents, support tickets).
2. They can't update themselves with **real-time knowledge**.
3. They sometimes **hallucinate**, i.e., generate convincing but incorrect responses.

This is where **Retrieval-Augmented Generation (RAG)** enters the picture.

- The flow illustrates four main stages: **data preparation, indexing data** as embeddings**, information retrieval**, and **LLM-based inference** based on retrieved chunks.
- This structure is for RAG systems, combining vector search over document chunks and large language model generation for context-aware Q&A.

EXL

Understanding Retrieval-Augmented Generation (RAG)


1
Data Preparation
2
Index Relevant Data
3
Information Retrieval
4
LLM Inference
Raw files
PDFs, Word, PPT, etc.
Clean dataset
Doc chunks
Databricks Model Serving to embeddings model
Foundation model API, external model, or custom model
Embeddings
Synced with dataset
Serverless endpoint
Query your vector index
Databricks Vector Search index
Syncing your data to a Vector Search index
Relevant chunk of text data
Relevant chunk of text data
Relevant chunk of text data
"How can I track Databricks billing?"
Databricks chatbot model endpoint
Answer this question:
"How to track Databricks billing?"
On the basis of this context data:
Databricks Model Serving to LLM
route/credential/throughput/logging

- Data Preparation: Raw files (PDFs, Word, PPTs, etc.) are cleaned and split into smaller document chunks for processing.
- Index Relevant Data: Each document chunk is transformed into a vector (embedding) using specialized models and stored in a vector database, making it searchable based on meaning rather than exact words.
- Information Retrieval: The system quickly finds and returns the most relevant chunks of data in response to user queries.
- LLM Inference: The retrieved relevant chunks and user question are sent to a language model, which uses this context to generate a clear, accurate natural language answer.

What is Retrieval-Augmented Generation (RAG)?

RAG is an architecture that combines two powerful capabilities:

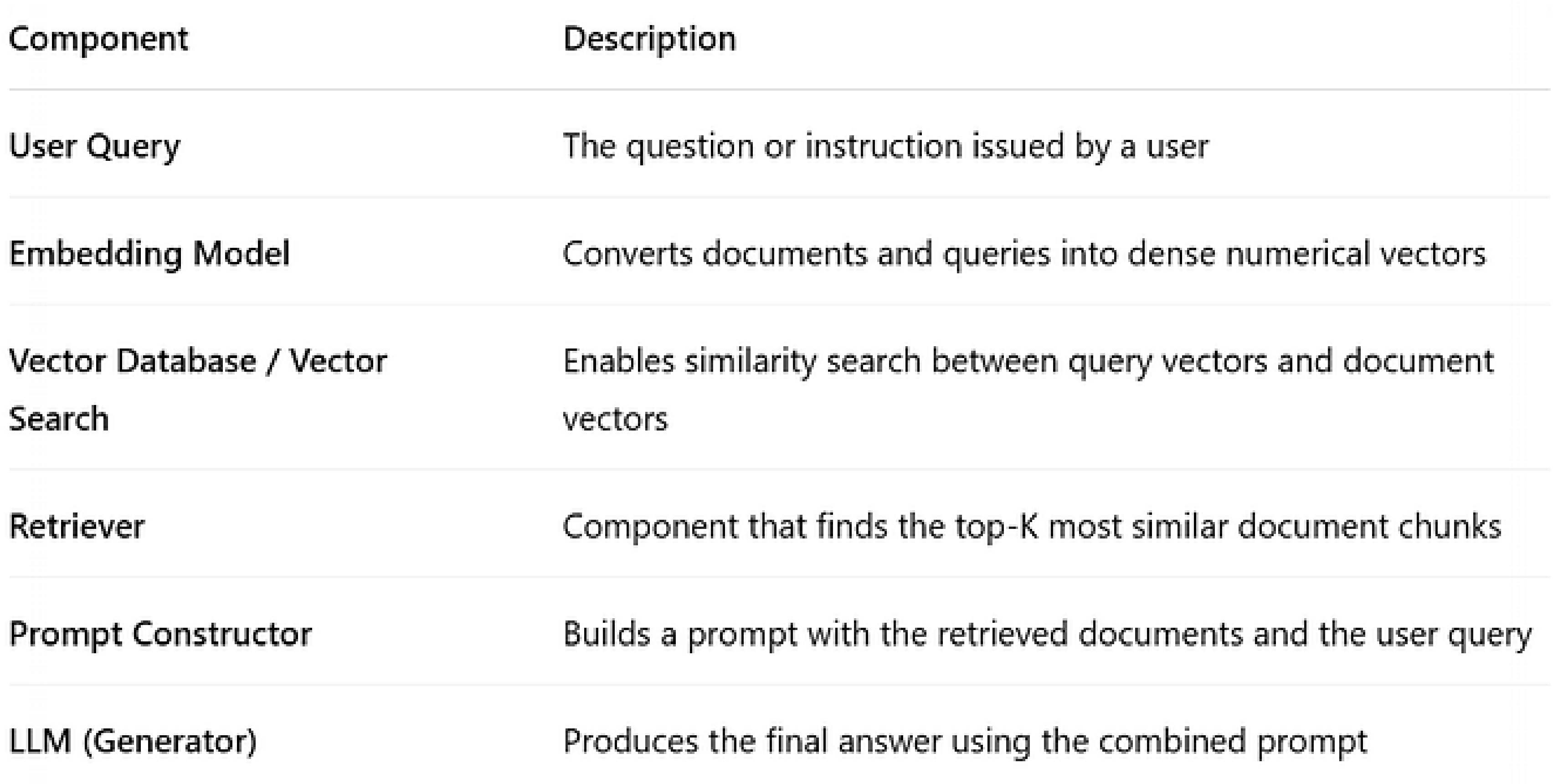
- **Retrieval**: Pulling relevant context from external data sources, typically using vector search over embedded documents.
- **Generation**: Using the retrieved context along with a user question to generate a natural language response via an LLM.

Formal Definition:

Retrieval-Augmented Generation is a hybrid approach where a language model is provided with dynamically retrieved, contextually relevant documents to improve its factual accuracy, reduce hallucinations, and adapt to private or recent knowledge.

Unlike fine-tuning, which involves retraining the model on new data (costly and slow), RAG dynamically injects real-world knowledge into the model's responses **at inference time**.

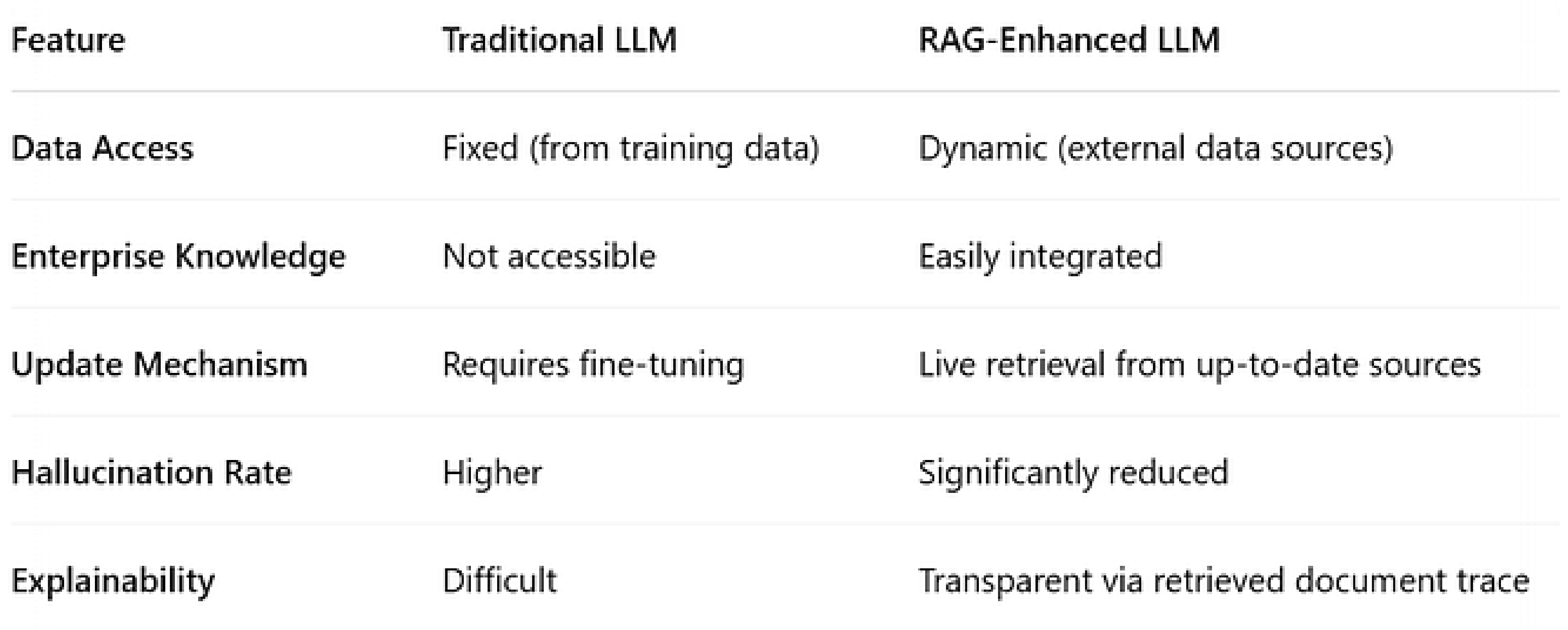
Core Components of a Retrieval-Augmented Generation (RAG)



Component	Description
User Query	The question or instruction issued by a user
Embedding Model	Converts documents and queries into dense numerical vectors
Vector Database / Vector Search	Enables similarity search between query vectors and document vectors
Retriever	Component that finds the top-K most similar document chunks
Prompt Constructor	Builds a prompt with the retrieved documents and the user query
LLM (Generator)	Produces the final answer using the combined prompt

EXL

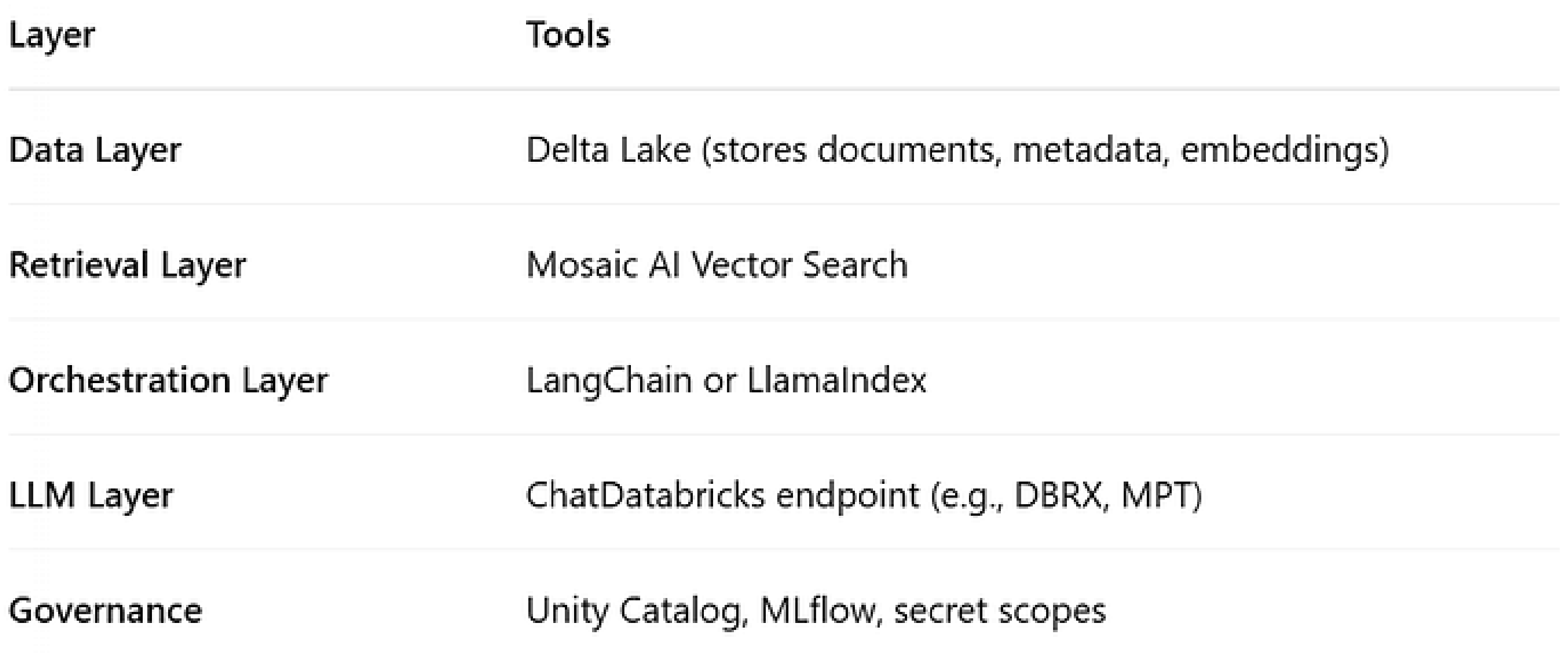
Traditional LLM vs RAG Architecture



Feature	Traditional LLM	RAG-Enhanced LLM
Data Access	Fixed (from training data)	Dynamic (external data sources)
Enterprise Knowledge	Not accessible	Easily integrated
Update Mechanism	Requires fine-tuning	Live retrieval from up-to-date sources
Hallucination Rate	Higher	Significantly reduced
Explainability	Difficult	Transparent via retrieved document trace

How RAG Works on Databricks?

Databricks provides all the building blocks for constructing a scalable RAG system:



Layer	Tools
Data Layer	Delta Lake (stores documents, metadata, embeddings)
Retrieval Layer	Mosaic AI Vector Search
Orchestration Layer	LangChain or LlamaIndex
LLM Layer	ChatDatabricks endpoint (e.g., DBRX, MPT)
Governance	Unity Catalog, MLflow, secret scopes

1. A retail company is building a question-answering bot using RAG. Sensitive customer data is stored alongside product documents. Which tool best helps ensure compliance with data governance during indexing and retrieval?

a) MLflow
b) Unity Catalog
c) Mosaic AI Vector Search
d) LangChain

1. A retail company is building a question-answering bot using RAG. Sensitive customer data is stored alongside product documents. Which tool best helps ensure compliance with data governance during indexing and retrieval?

a) MLflow
b) Unity Catalog
c) Mosaic AI Vector Search
d) LangChain

Reason - Unity Catalog enforces secure, centralized access control and auditability, essential for protecting sensitive data in RAG pipelines.

2. Your team uses Delta Lake for document and embedding storage and Mosaic AI Vector Search for fast queries. However, responses from the bot seem outdated after recent database updates. What is the most likely cause?

a) Outdated LLM weights
b) Old embeddings need re-generation
c) Secret scopes not refreshed
d) Serverless endpoint misconfigured

2. Your team uses Delta Lake for document and embedding storage and Mosaic AI Vector Search for fast queries. However, responses from the bot seem outdated after recent database updates. What is the most likely cause?

a) Outdated LLM weights
b) Old embeddings need re-generation
c) Secret scopes not refreshed
d) Serverless endpoint misconfigured

Reason - Old embeddings must be regenerated after database updates to ensure the retrieval search returns current and accurate information.

3. Consider a highly-regulated financial environment. Why would a RAG-enhanced LLM be more suitable than a traditional LLM for explainability in audit scenarios?

a) It guarantees zero hallucination
b) It disables use of enterprise knowledge
c) Training data is always up to date
d) Retrieval steps allow document traceability

3. Consider a highly-regulated financial environment. Why would a RAG-enhanced LLM be more suitable than a traditional LLM for explainability in audit scenarios?

a) It guarantees zero hallucination
b) It disables use of enterprise knowledge
c) Training data is always up to date
d) Retrieval steps allow document traceability

Reason - RAG-enhanced LLMs allow users to trace answers directly to retrieved documents, meeting high explainability demands for audits.

EXL

4. A user reports hallucinations in generated answers despite using a well-tuned LLM endpoint. Which RAG pipeline enhancement is most effective to lower hallucination rates for enterprise Q&A?

a) Use dynamic, live retrieval from enterprise sources
b) Rely on fixed training data
c) Add more metadata in Delta Lake
d) Increase orchestration steps

4. A user reports hallucinations in generated answers despite using a well-tuned LLM endpoint. Which RAG pipeline enhancement is most effective to lower hallucination rates for enterprise Q&A?

a) Use dynamic, live retrieval from enterprise sources
b) Rely on fixed training data
c) Add more metadata in Delta Lake
d) Increase orchestration steps

Reason - Live retrieval from dynamic enterprise sources ensures the model uses fresh, accurate data, thereby lowering hallucination rates.

EXL

5. Your organization must frequently update document knowledge without LLM retraining. Which combination provides the most scalable architecture?

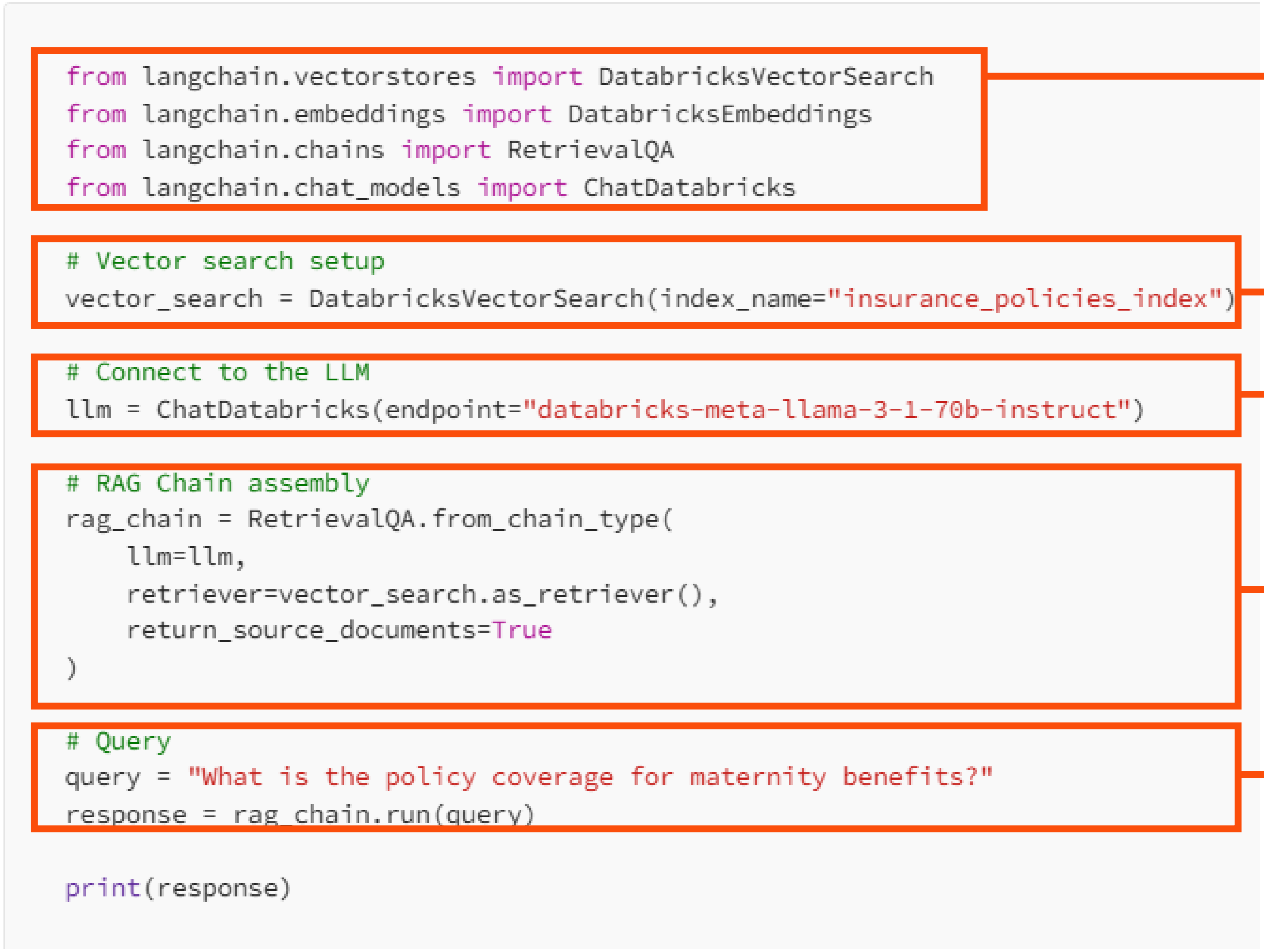
a) Fine-tuning the LLM and Unity Catalog
b) Live embedding updates in Delta Lake + Mosaic AI Vector Search
c) Static embeddings + MLflow
d) Relying solely on orchestration layer changes

5. Your organization must frequently update document knowledge without LLM retraining. Which combination provides the most scalable architecture?

a) Fine-tuning the LLM and Unity Catalog
b) Live embedding updates in Delta Lake + Mosaic AI Vector Search
c) Static embeddings + MLflow
d) Relying solely on orchestration layer changes

Reason - Live embedding updates in Delta Lake combined with Mosaic AI Vector Search allow you to add or modify knowledge instantly—any new data or documents are embedded and indexed on the fly, making them immediately searchable by the vector database without needing to retrain the LLM itself. This approach ensures your generative AI system always works with the latest information and is highly scalable for frequent updates requiring minimal operational overhead.

Code Walkthrough: Building a RAG System



```
from langchain.vectorstores import DatabricksVectorSearch
from langchain.embeddings import DatabricksEmbeddings
from langchain.chains import RetrievalQA
from langchain.chat_models import ChatDatabricks

# Vector search setup
vector_search = DatabricksVectorSearch(index_name="insurance_policies_index")

# Connect to the LLM
llm = ChatDatabricks(endpoint="databricks-meta-llama-3-1-70b-instruct")

# RAG Chain assembly
rag_chain = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vector_search.as_retriever(),
    return_source_documents=True
)

# Query
query = "What is the policy coverage for maternity benefits?"
response = rag_chain.run(query)

print(response)
```

Necessary libraries from LangChain and Databricks for embeddings, vector search, and chain assembly:

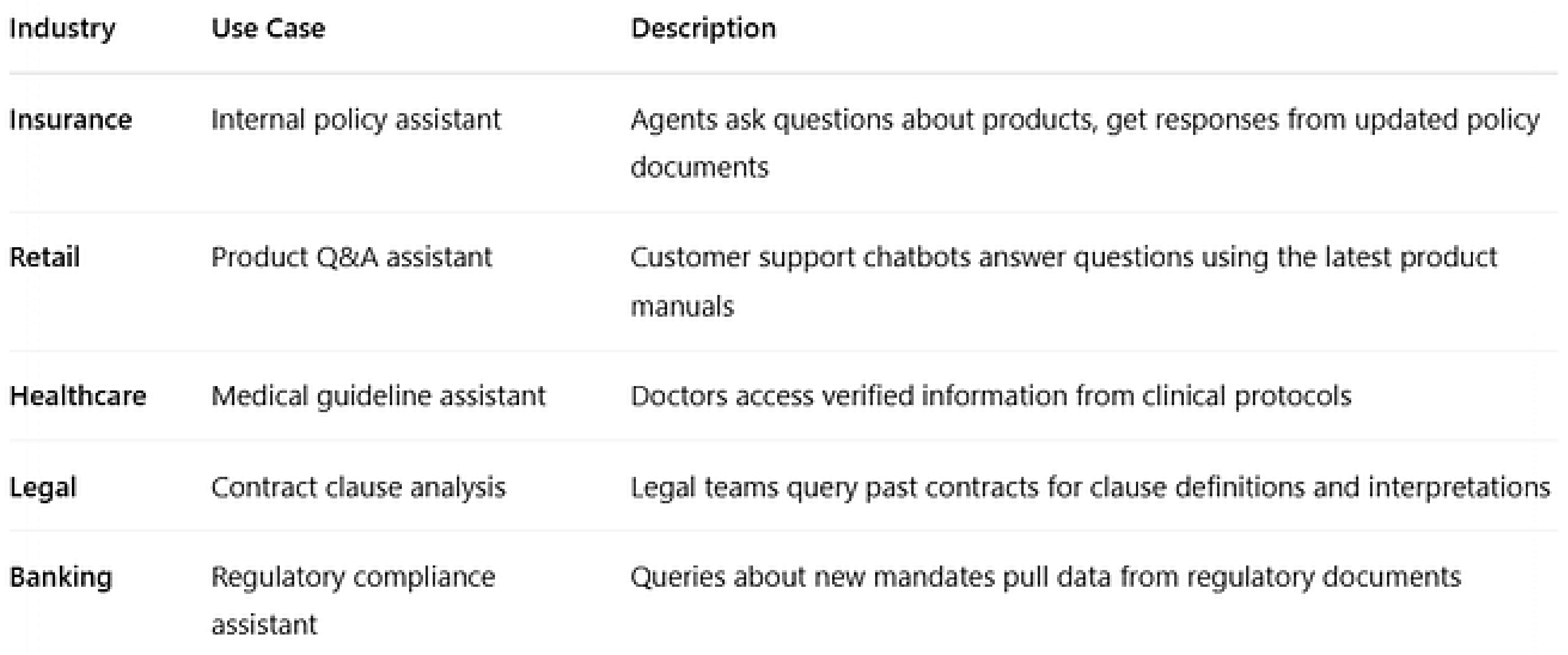
This initializes a vector search instance tied to a specific index in Databricks, enabling semantic document retrieval using MLflow-powered embeddings.

A connection is made to a language model hosted on Databricks, specifying the endpoint for transactions.

This combines the previously defined retriever (vector search) and the language model. The "RetrievalQA" chain ensures that queries are answered using both the model and the retrieved documents, with source documents returned for transparency.

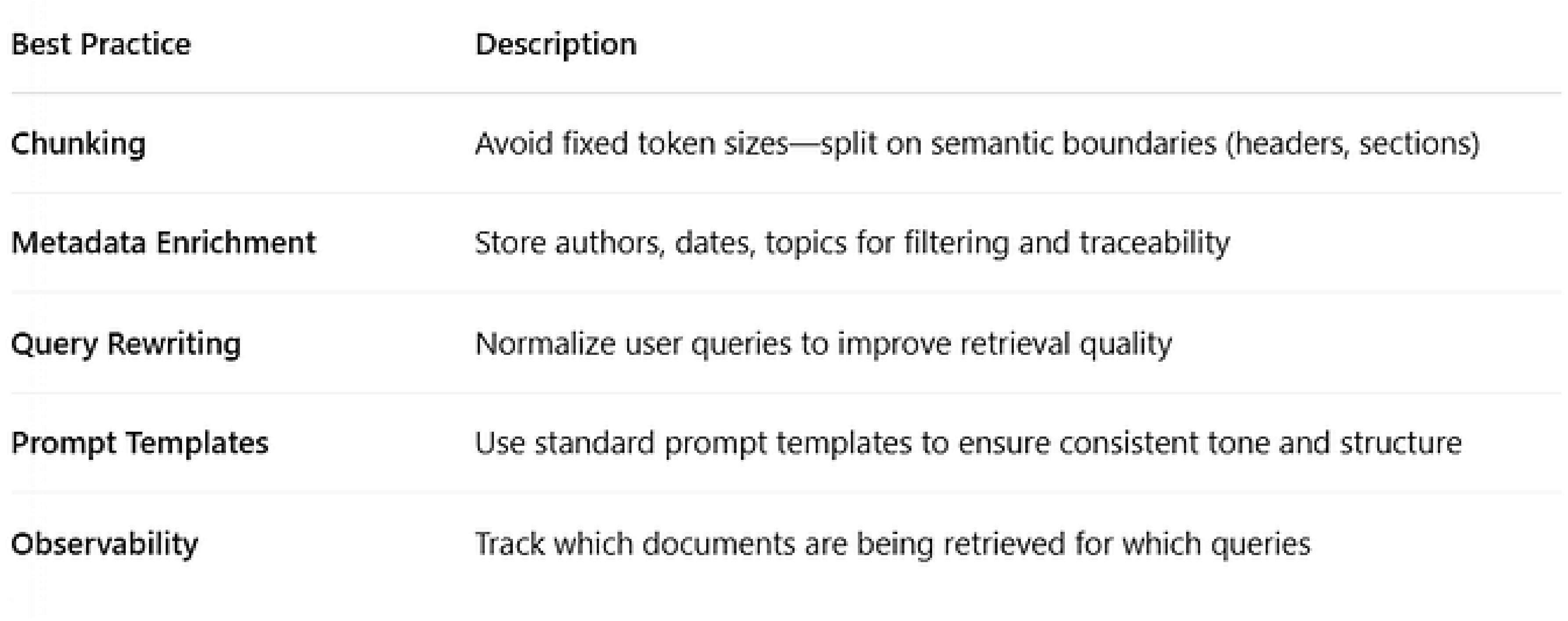
A user query is processed by the RAG chain, retrieving relevant documents and generating a detailed answer using the language model. The response is printed at the end.

Enterprise Use Cases for RAG



Industry	Use Case	Description
Insurance	Internal policy assistant	Agents ask questions about products, get responses from updated policy documents
Retail	Product Q&A assistant	Customer support chatbots answer questions using the latest product manuals
Healthcare	Medical guideline assistant	Doctors access verified information from clinical protocols
Legal	Contract clause analysis	Legal teams query past contracts for clause definitions and interpretations
Banking	Regulatory compliance assistant	Queries about new mandates pull data from regulatory documents

Best Practices in RAG systems



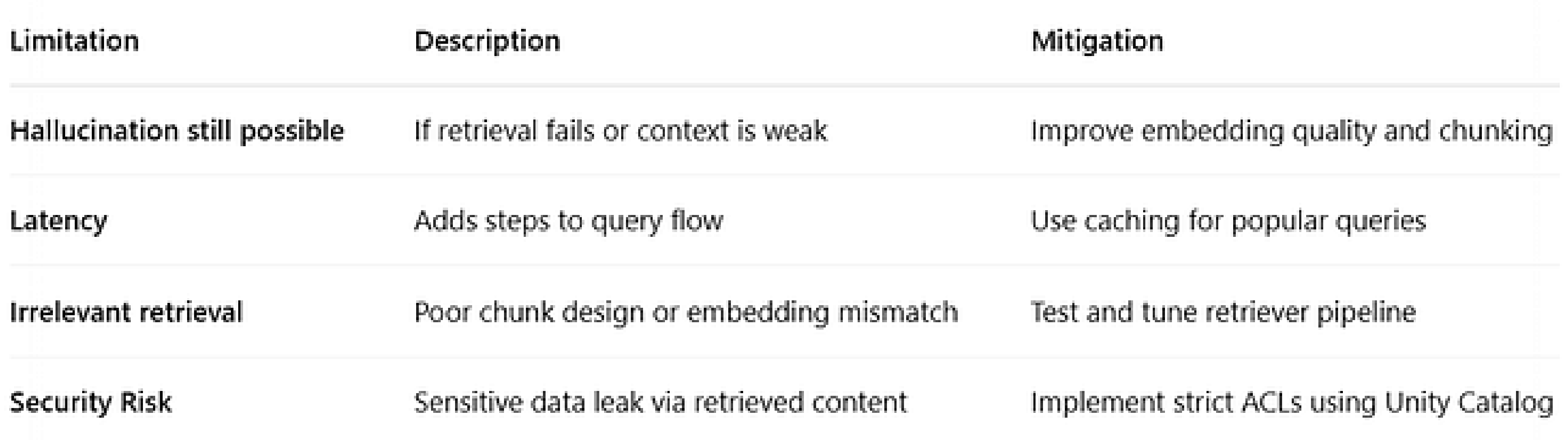
Best Practice	Description
Chunking	Avoid fixed token sizes—split on semantic boundaries (headers, sections)
Metadata Enrichment	Store authors, dates, topics for filtering and traceability
Query Rewriting	Normalize user queries to improve retrieval quality
Prompt Templates	Use standard prompt templates to ensure consistent tone and structure
Observability	Track which documents are being retrieved for which queries

Governance and Compliance in RAG

When you build RAG systems on Databricks:

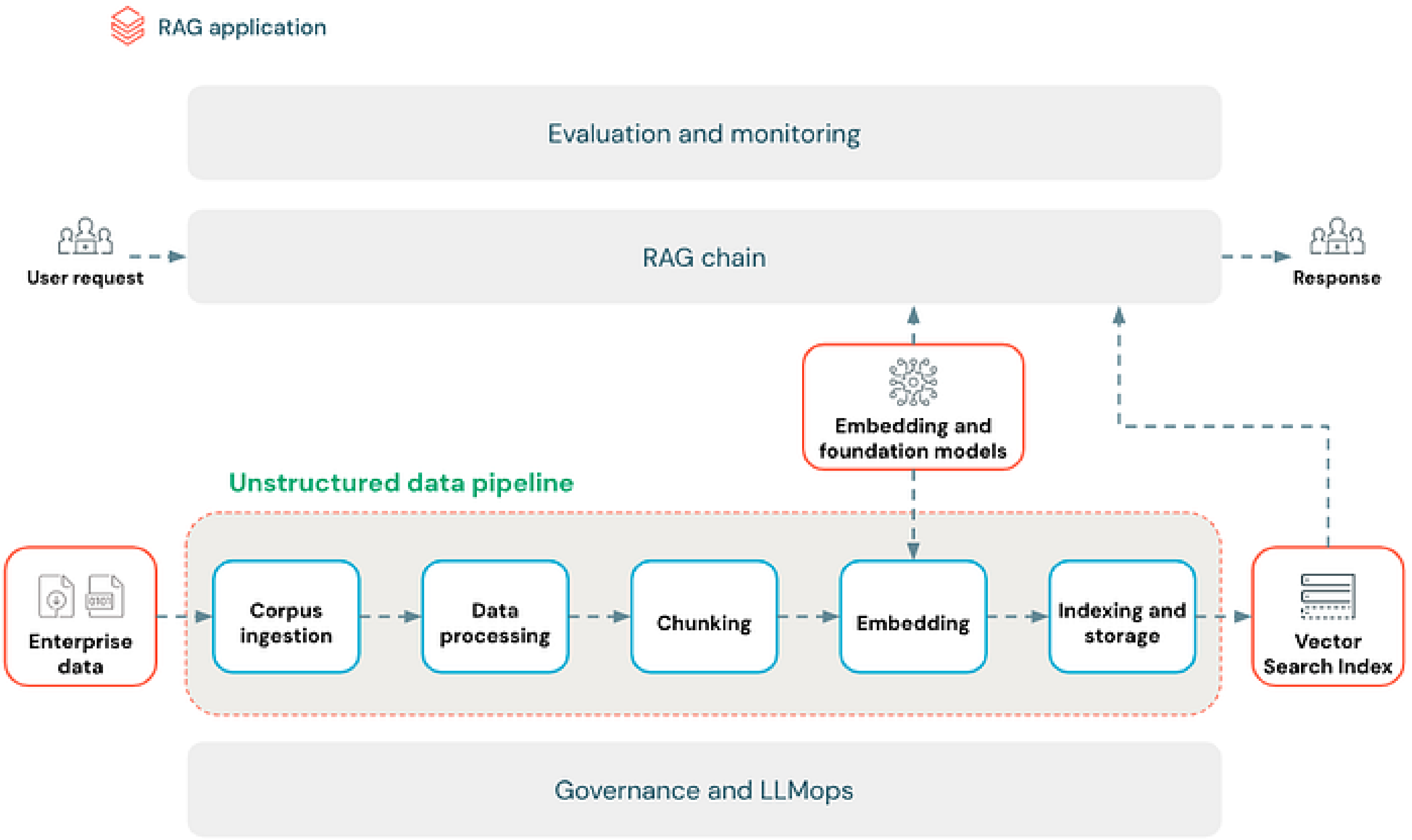
- Use **Unity Catalog** to enforce access controls on sensitive source documents.
- Store and manage prompts using a **Prompt Repository**.
- Track model versions, prompt versions, and embeddings using **MLflow**.
- Use **audit logs** to trace which documents were retrieved for any given answer.

Limitations of RAG (And Mitigations)



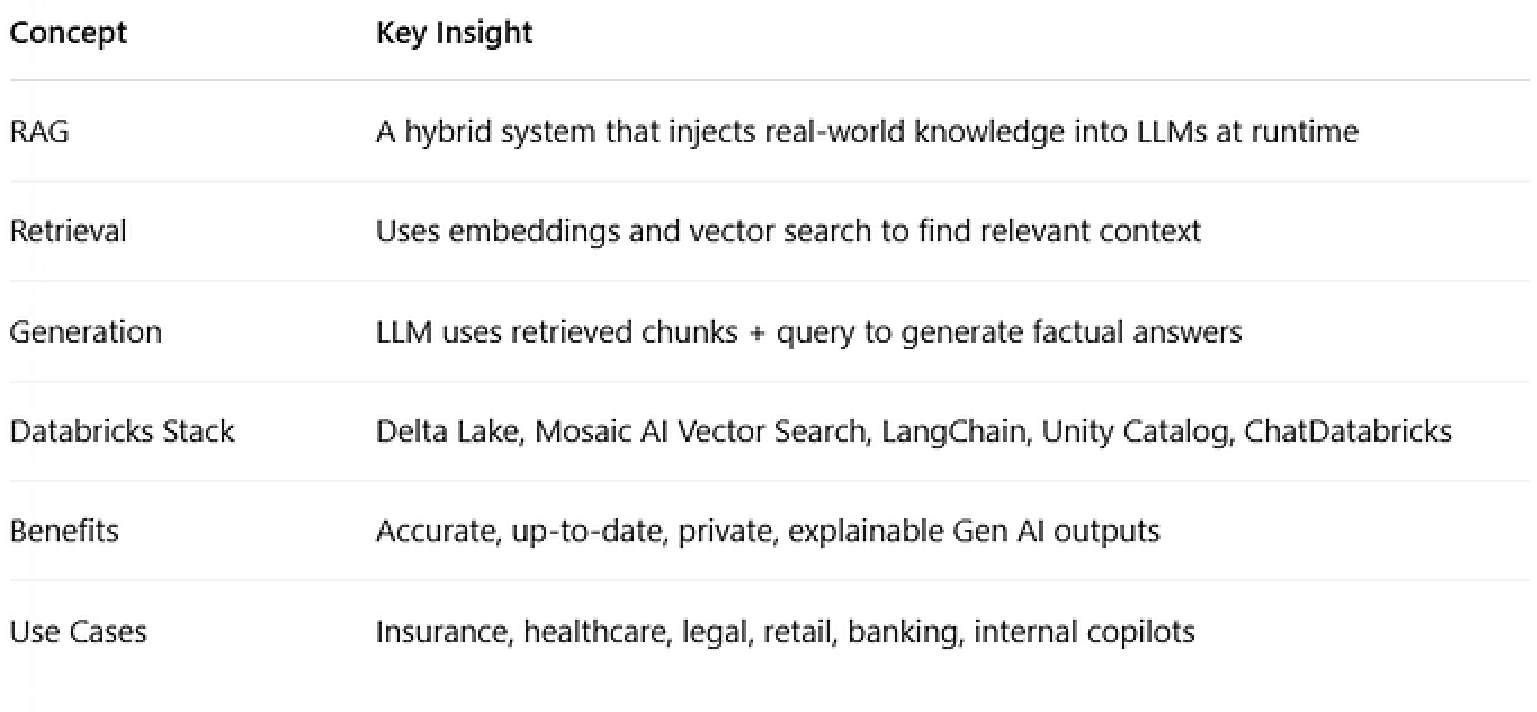
Limitation	Description	Mitigation
Hallucination still possible	If retrieval fails or context is weak	Improve embedding quality and chunking
Latency	Adds steps to query flow	Use caching for popular queries
Irrelevant retrieval	Poor chunk design or embedding mismatch	Test and tune retriever pipeline
Security Risk	Sensitive data leak via retrieved content	Implement strict ACLs using Unity Catalog

Preparing High-Quality Data for RAG Applications



EXL

Summary – Part 1



Concept	Key Insight
RAG	A hybrid system that injects real-world knowledge into LLMs at runtime
Retrieval	Uses embeddings and vector search to find relevant context
Generation	LLM uses retrieved chunks + query to generate factual answers
Databricks Stack	Delta Lake, Mosaic AI Vector Search, LangChain, Unity Catalog, ChatDatabricks
Benefits	Accurate, up-to-date, private, explainable Gen AI outputs
Use Cases	Insurance, healthcare, legal, retail, banking, internal copilots

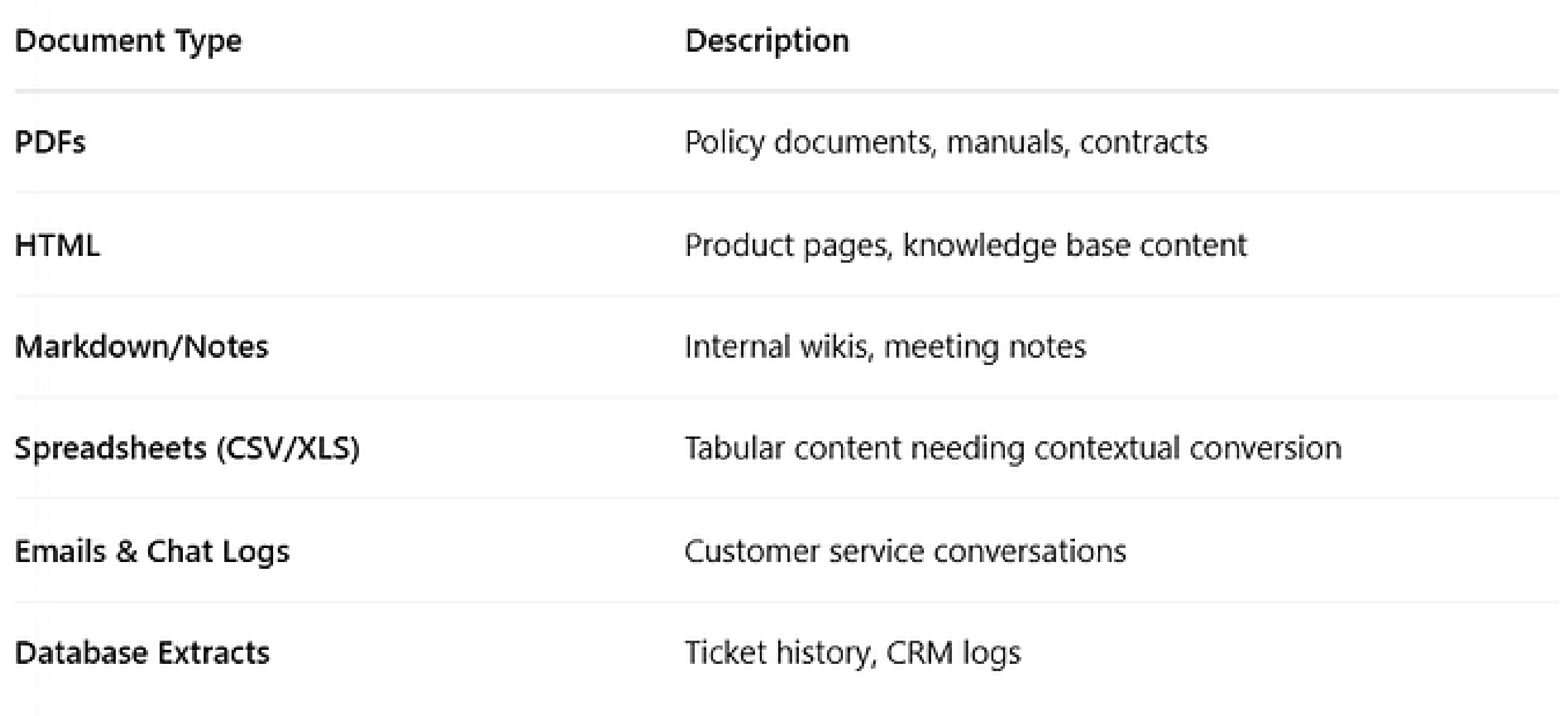
RAG has quickly become the **de facto design pattern** for enterprise LLM applications.

It enables us to:

- Ground your LLM on **proprietary knowledge**
- Avoid costly **model retraining**
- Minimize **hallucinations**
- Provide **explainable AI** with document traceability

Source Document Types in RAG Pipelines

RAG pipelines typically work with unstructured or semi-structured enterprise content, such as:



Document Type	Description
PDFs	Policy documents, manuals, contracts
HTML	Product pages, knowledge base content
Markdown/Notes	Internal wikis, meeting notes
Spreadsheets (CSV/XLS)	Tabular content needing contextual conversion
Emails & Chat Logs	Customer service conversations
Database Extracts	Ticket history, CRM logs

A preprocessing pipeline is needed to extract clean, readable text from these formats.

Text Extraction and Preprocessing PDF/Text Extraction Example:

```
from langchain.document_loaders import PyPDFLoader

loader = PyPDFLoader("/dbfs/data/insurance_policy.pdf")
documents = loader.load()
```

Best Practices:

- Normalize whitespace
- Remove headers/footers and repetitive page numbers
- Split multi-column documents into single column text
- Convert scanned images using OCR (e.g., Tesseract)

For structured formats:

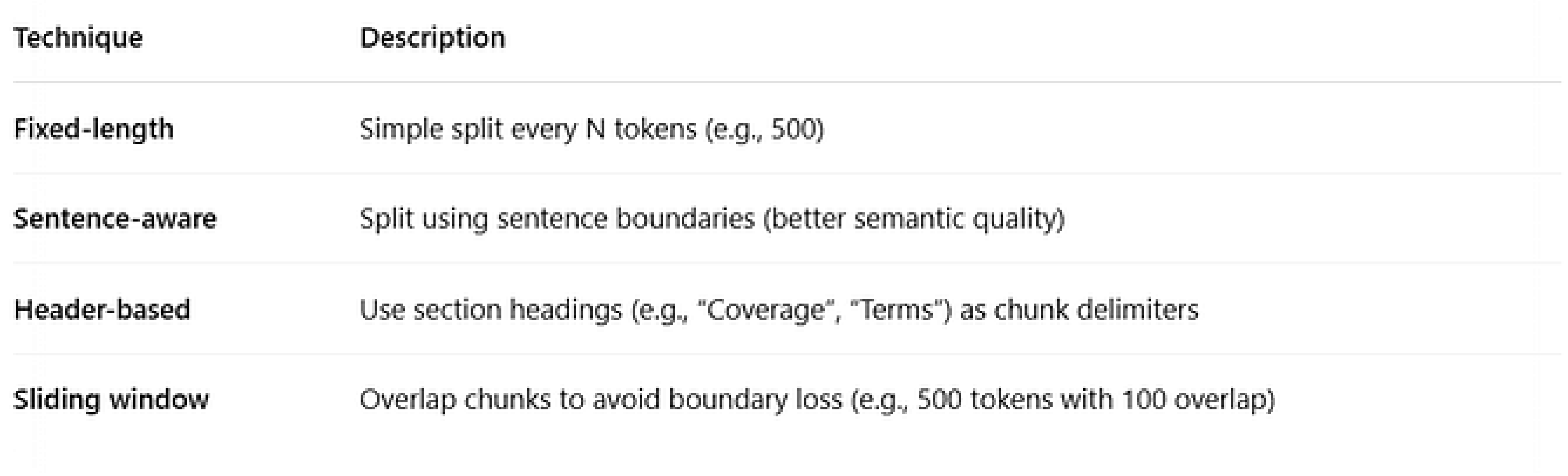
- Convert rows to narrative sentences if needed for semantic retrieval.

Chunking Strategies for Semantic Understanding

What is Chunking?
Chunking is the process of breaking down large documents into smaller, manageable semantic units that can be embedded and retrieved individually.

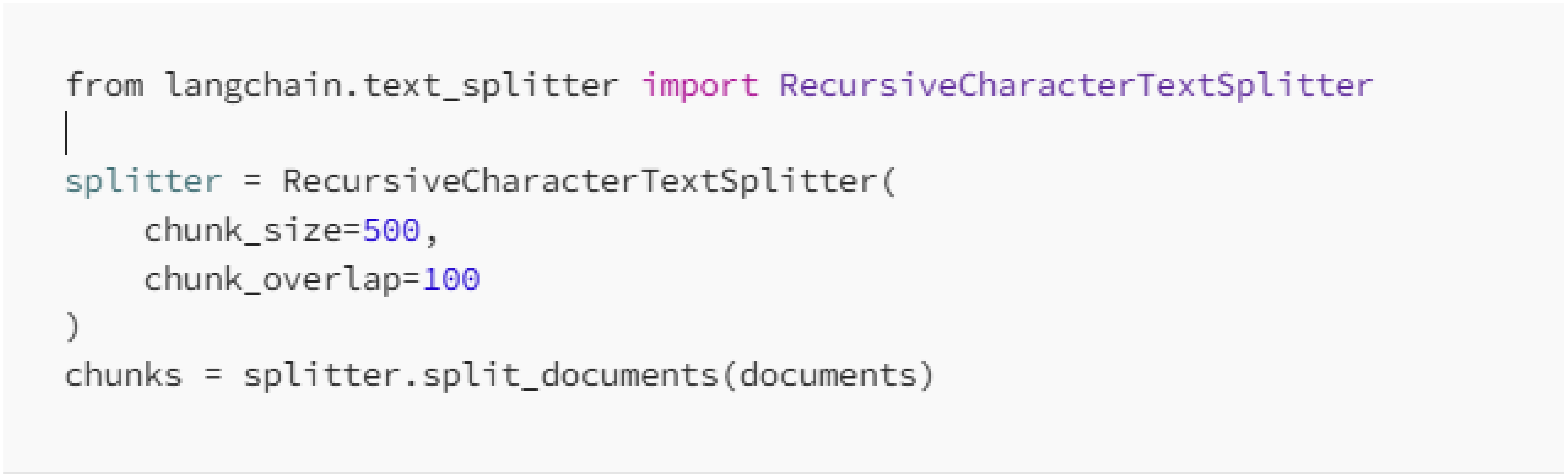
A good chunk is:
- Self-contained
- Topically consistent
- Small enough to fit in an LLM prompt

Chunking Techniques:



Technique	Description
Fixed-length	Simple split every N tokens (e.g., 500)
Sentence-aware	Split using sentence boundaries (better semantic quality)
Header-based	Use section headings (e.g., "Coverage", "Terms") as chunk delimiters
Sliding window	Overlap chunks to avoid boundary loss (e.g., 500 tokens with 100 overlap)

Example (LangChain):



```python
from langchain.text_splitter import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=100
)
chunks = splitter.split_documents(documents)
```

Tip: Don’t make chunks too small. Too small = context fragmentation. Too large = LLM can’t process it.

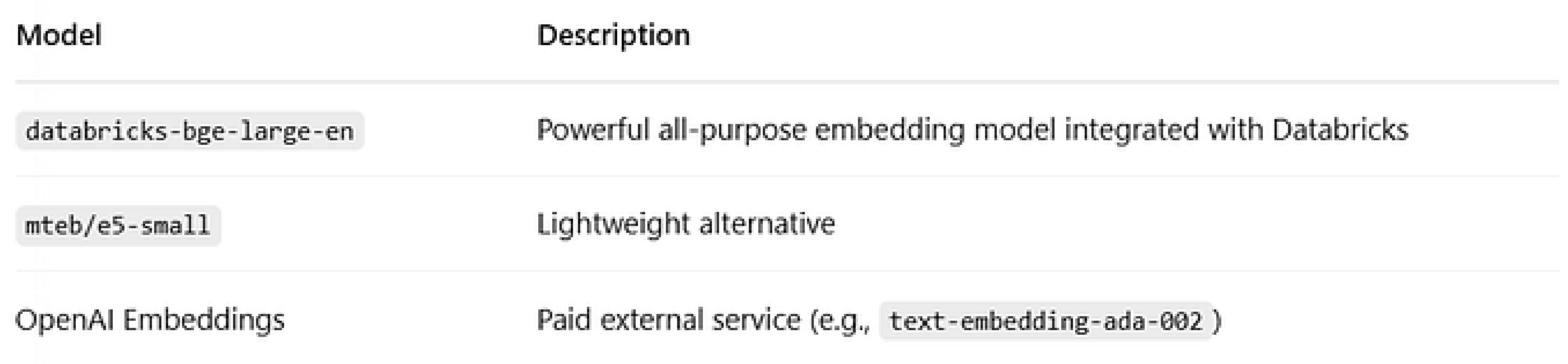
Embedding Generation — Vectorizing Your Chunks

Each chunk is converted into a **vector** using an embedding model.

What are Embeddings?

- Embeddings are **dense numerical representations** of text in high-dimensional space, where semantically similar texts are geometrically close to each other.

Embedding Models to Use:



Model	Description
`databricks-bge-large-en`	Powerful all-purpose embedding model integrated with Databricks
`mteb/e5-small`	Lightweight alternative
OpenAI Embeddings	Paid external service (e.g., `text-embedding-ada-002`)

Embedding Generation — Vectorizing Your Chunks

Embedding in LangChain:

Code:

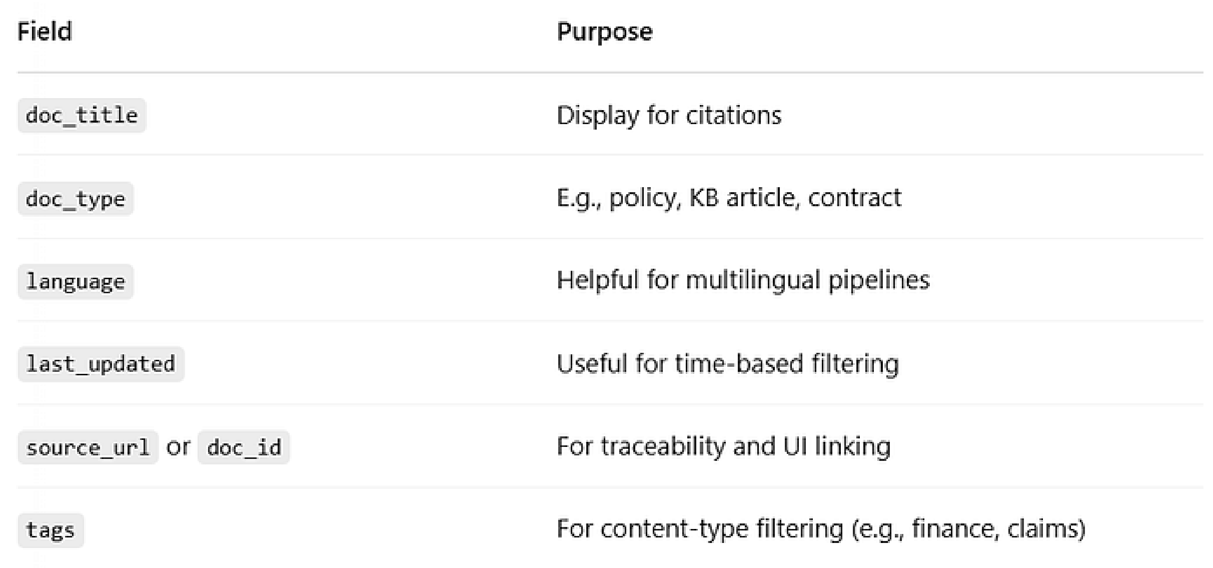
```
from langchain.embeddings import DatabricksEmbeddings

embedding_model = DatabricksEmbeddings(model="databricks-bge-large-en")
chunk_vectors = embedding_model.embed_documents([chunk.page_content for chunk in chunks])
```

Metadata Design — Traceability, Filtering, Search

- Metadata helps:
- Filter documents based on user roles
- Understand where the answer came from
- Boost relevance using custom logic

Useful Metadata Fields:

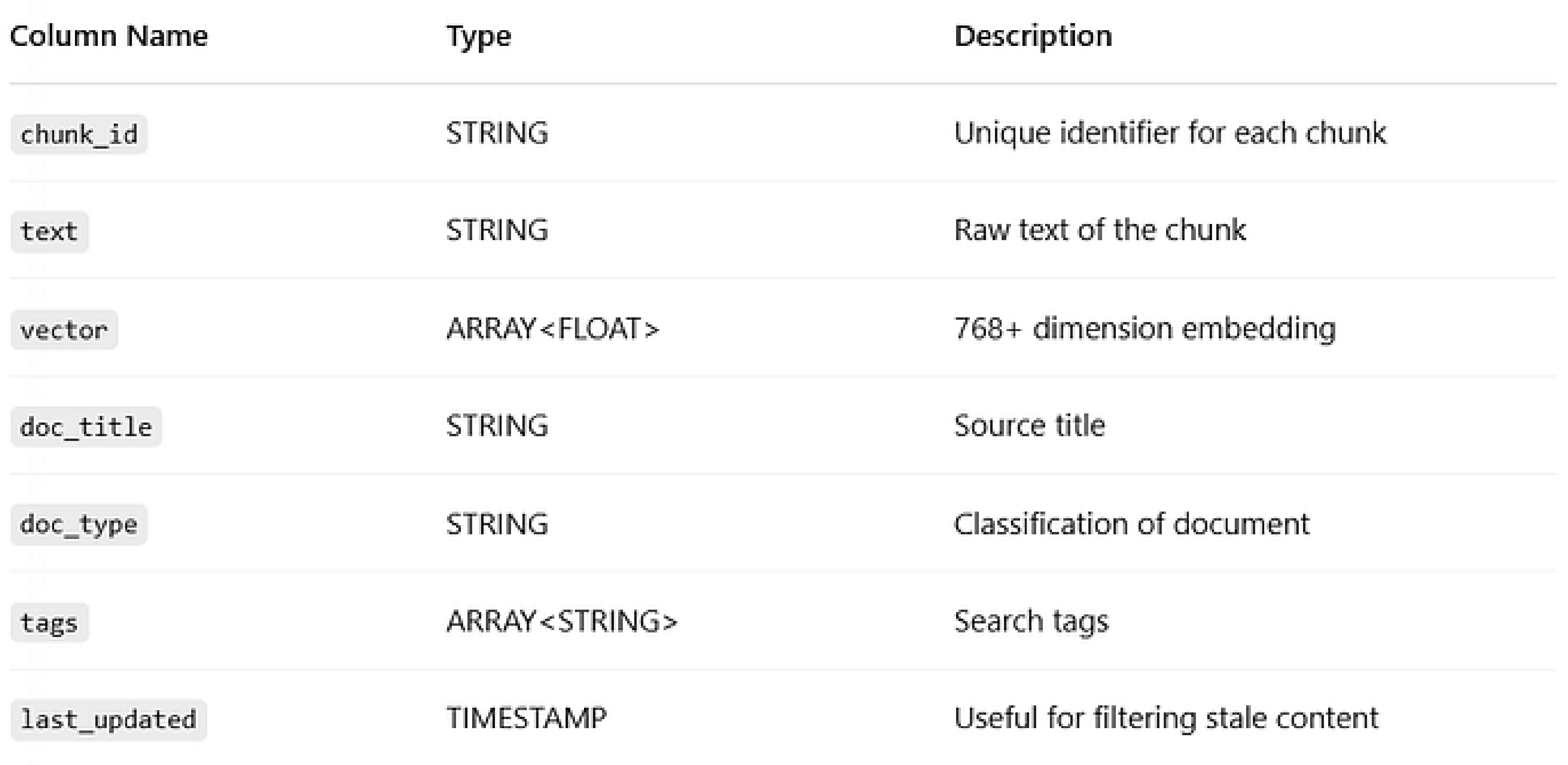
- Press enter or click to view image in full size



Field	Purpose
`doc_title`	Display for citations
`doc_type`	E.g., policy, KB article, contract
`language`	Helpful for multilingual pipelines
`last_updated`	Useful for time-based filtering
`source_url` or `doc_id`	For traceability and UI linking
`tags`	For content-type filtering (e.g., finance, claims)

Delta Table Design for RAG

Delta Lake offers versioning, ACID compliance, and performance. This makes it ideal for storing RAG content.



Column Name	Type	Description
`chunk_id`	STRING	Unique identifier for each chunk
`text`	STRING	Raw text of the chunk
`vector`	ARRAY<FLOAT>	768+ dimension embedding
`doc_title`	STRING	Source title
`doc_type`	STRING	Classification of document
`tags`	ARRAY<STRING>	Search tags
`last_updated`	TIMESTAMP	Useful for filtering stale content

Sample Code to Create Delta Table:

```python
from pyspark.sql.types import *

schema = StructType([
    StructField("chunk_id", StringType()),
    StructField("text", StringType()),
    StructField("vector", ArrayType(FloatType())),
    StructField("doc_title", StringType()),
    StructField("doc_type", StringType()),
    StructField("tags", ArrayType(StringType())),
    StructField("last_updated", TimestampType())
])
```

•Imports all data types from PySpark's SQL module.
•Defines a schema for the Delta table, specifying each field's name and data type:

- chunk_id, doc, doc_type, doc_path, embedding_model: strings for IDs, document contents, types, file paths, and model names.
- vector: array of floats to store embedding vectors.
- last_updated: timestamp indicating last update time.

```python
df = spark.createDataFrame(data, schema=schema)
df.write.format("delta").mode("overwrite").save("/mnt/rag/doc_embeddings")
```

•Creates a Spark DataFrame using the above schema and some variable data.
•Writes this DataFrame to a Delta table, overwriting any previous data at the given path.

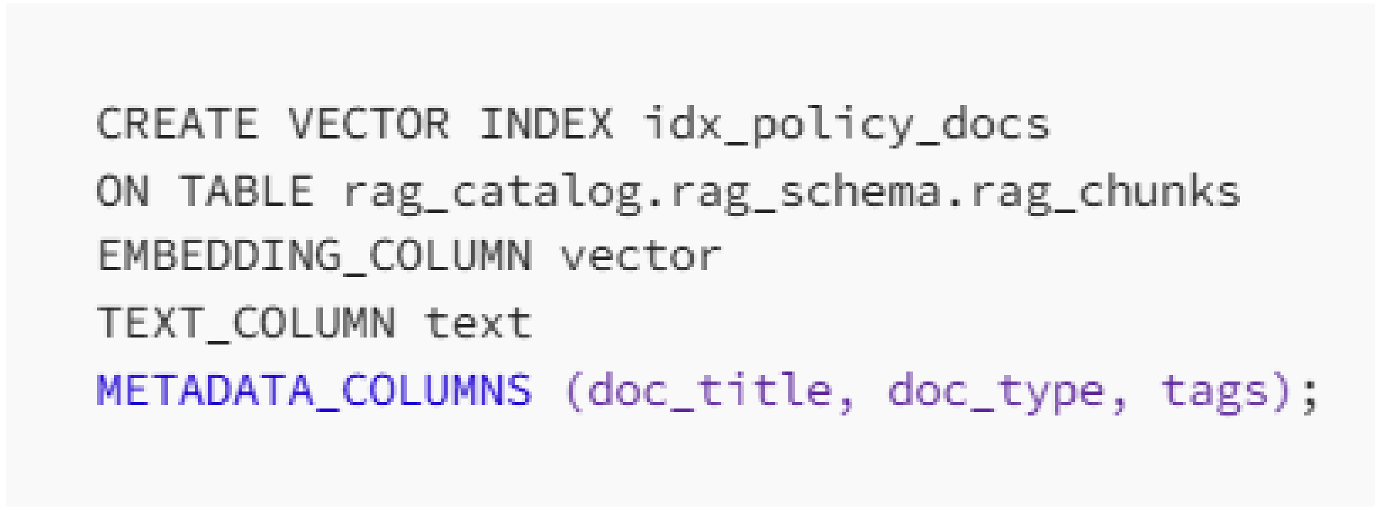
Register it in Unity Catalog for governance, supporting data lineage, access control, and compliance:

```sql
CREATE TABLE rag_catalog.rag_schema.rag_chunks
USING DELTA
LOCATION '/mnt/rag/doc_embeddings';
```

Vector Indexing Using Mosaic AI Vector Search

Once the Delta table is created, **enable Mosaic AI Vector Search**:
Steps:

1. Ensure vector column exists and is float array
2. Enable **Change Data Feed (CDF)** for updates
3. Create an index:



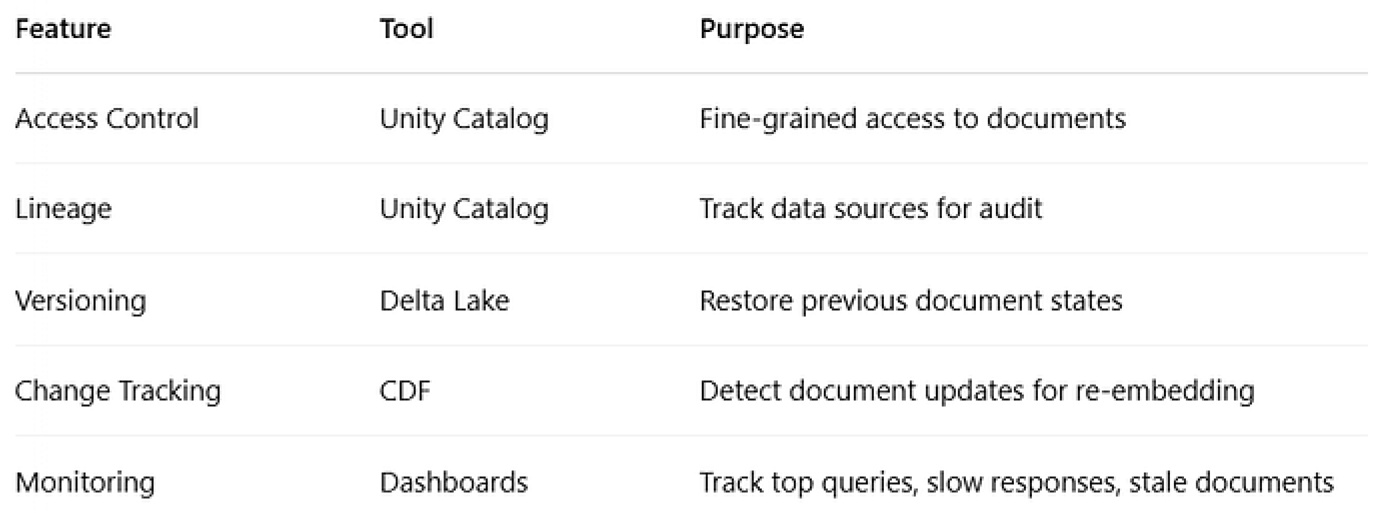
```
CREATE VECTOR INDEX idx_policy_docs
ON TABLE rag_catalog.rag_schema.rag_chunks
EMBEDDING_COLUMN vector
TEXT_COLUMN text
METADATA_COLUMNS (doc_title, doc_type, tags);
```

We can now use:

- **LangChain** to connect to the index
- **Semantic search** to retrieve top-K relevant chunks
- **Audit the source** using metadata in responses

Governance and Observability

Data governance is critical for enterprise-grade RAG systems:



Feature	Tool	Purpose
Access Control	Unity Catalog	Fine-grained access to documents
Lineage	Unity Catalog	Track data sources for audit
Versioning	Delta Lake	Restore previous document states
Change Tracking	CDF	Detect document updates for re-embedding
Monitoring	Dashboards	Track top queries, slow responses, stale documents

Section 2: Data Preparation (summary)

6. Apply a chunking strategy for a given document structure and model constraints

Applying a chunking strategy involves dividing documents into **manageable pieces that fit within the model's context window and constraints.**

Key Considerations

1. **Context Window**
2. **Chunking Strategy**:

- **Context-aware Chunking**: Divide text by sentences, paragraphs, or sections using special punctuation such as periods or newlines.
- **Fixed-size Chunking**: Divide text into chunks of a specific number of tokens.

3. Advanced Chunking Strategies like **Windowed Summarization** where Each chunk includes a summary of the previous chunks to maintain context across the document.

7. Implementation Steps:

- **Data Extraction**: Extract raw text from documents, ensuring it is clean and ready for processing.
- **Chunking Process**: Apply the chosen chunking strategy (context-aware, fixed-size, windowed summarization)

.

- **Embedding and Storage:** Embed each chunk using a model and store the embeddings in a vector store for efficient retrieval.

8. **Challenges and Solutions:**

- **Maintaining Context**: Ensure that each chunk preserves enough context to be meaningful on its own.
- **Handling Different Document Types:** Use appropriate tools and methods for different formats (e.g., .doc, .pdf, .dat, .html). Learn the basic python packages like doctr or pypdf

9. **Use Case:** Experiment with different chunk sizes and methods to find the best fit for the specific use case.

10. **Filter extraneous content in source documents that degrades quality of a RAG application**

This includes:

- **Cleaning Data:** Ensuring the text is free from irrelevant content such as advertisements, navigation bars, and footers.
- **Preprocessing Steps:** Applying preprocessing techniques like removing stop words, correcting misspellings, and normalizing text to enhance the quality of the data fed into the model

11. Choose the appropriate Python package to extract document content from provided source data and format

PyPDF, Hugging Face's Options, Doctr — where **OCR** (Optical Character Recognition) is required to extract text; learn about bigger models: **OpenAI's Models, Alphabet's Gemini 1.5, Meta's Llamma**

12. Define operations and sequence to write given chunked text into Delta Lake tables in Unity Catalog

Defining operations and sequence involves several steps:

1. **Data Ingestion:** Extract text content from documents and load it into a dataframe.
2. **Chunking and Embedding:** Apply chunking strategies and compute embeddings for each chunk.
3. **Writing to Delta Lake:** Store the chunked text and embeddings into Delta Lake tables. This process ensures the data is easily accessible for retrieval operations in Unity Catalog
4. **Governance and Metadata Management**: Ensure the tables are registered in Unity Catalog for proper governance and metadata management.

13. Continuous Integration and Data Refresh:

- **Automate Updates**: Set up workflows to continuously update the Delta tables as new data arrives or existing data is modified.
- **Delta Live Tables**: Use Delta Live Tables to automate and orchestrate these data workflows.

14. Identify needed source documents that provide the necessary knowledge and quality for a given RAG application

- **Relevance:** Selecting documents that are highly relevant to the domain and task at hand.
- **Quality Assessment:** Evaluating the accuracy, reliability, and completeness of the documents.
- **Diversity:** Ensuring a diverse set of documents to cover various aspects of the knowledge required

15. Identify prompt/response pairs that align with a given model task

Identifying suitable prompt/response pairs involves:

- **Task Alignment:** Ensuring the pairs are relevant to the specific task the model is designed to perform.
- **Contextual Relevance:** Selecting pairs that provide sufficient context for the model to generate accurate responses.

- **Quality Control:** Verifying that the prompt/response pairs are free from errors and biases
- **Tagging examples:** For sentiment analysis, tag responses as positive, negative, or neutral. And for question-answering, tag responses as factual, opinion-based, or advisory

16. **Use tools and metrics to evaluate retrieval performance**

Context Precision: Measures how much of the retrieved context is actually relevant to the query.
Context Recall: Measures how much of the relevant context was successfully retrieved.
Faithfulness: Evaluates whether the generated answer accurately reflects the retrieved or source information.
Answer Relevancy: Assesses how directly the answer addresses the user's query.
Answer Correctness: Judges whether the answer is factually and semantically accurate.

2. **Evaluation Tools and Methods**:

- **MLflow**: Facilitates the evaluation of retrievers and LLMs, supporting batch comparisons and scalable experimentation. MLflow can evaluate unstructured outputs automatically and at low cost.
- **LLM-as-a-Judge**: An approach where an LLM is used to evaluate the performance of another LLM by scoring responses based on predefined criteria. This method can be integrated with MLflow for automated and scalable evaluations.

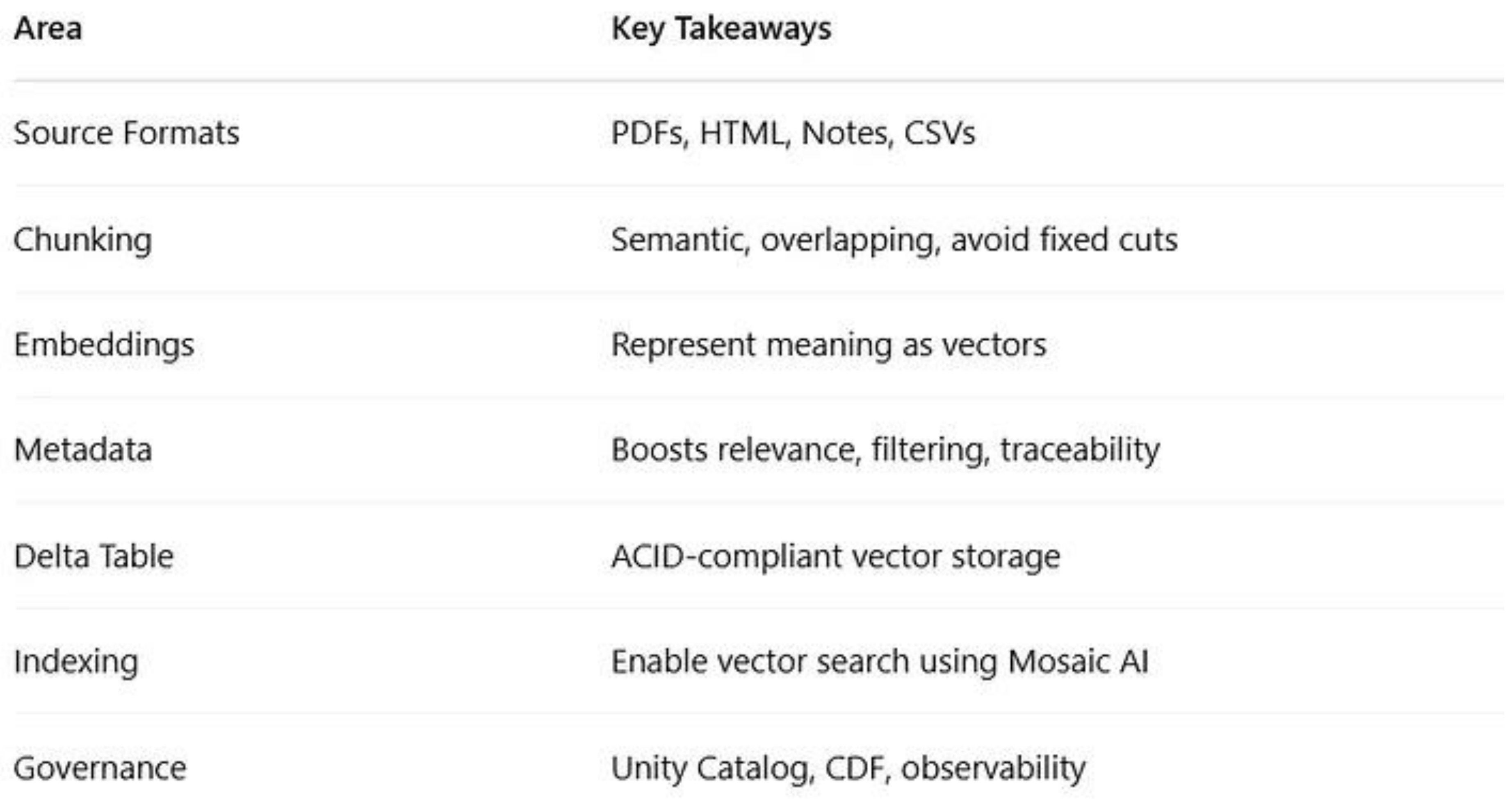
Evaluation Tools and Methods:

- **MLflow**: Facilitates the evaluation of retrievers and LLMs, supporting batch comparisons and scalable experimentation. MLflow can evaluate unstructured outputs automatically and at low cost.
- **LLM-as-a-Judge**: An approach where an LLM is used to evaluate the performance of another LLM by scoring responses based on predefined criteria. This method can be integrated with MLflow for automated and scalable evaluations.
- **Task-specific Metrics**: Metrics like BLEU for translation and ROUGE for summarization are used to evaluate LLM performance on specific tasks.

Offline vs. Online Evaluation:

- **Offline Evaluation**: Conducted before deployment using curated benchmark datasets and task-specific metrics to evaluate LLM performance.
- **Online Evaluation**: Conducted post-deployment, collecting real-time user behavior data to evaluate how well users respond to the LLM system. This approach includes metrics from A/B testing and user feedback.

Summary – Part 2



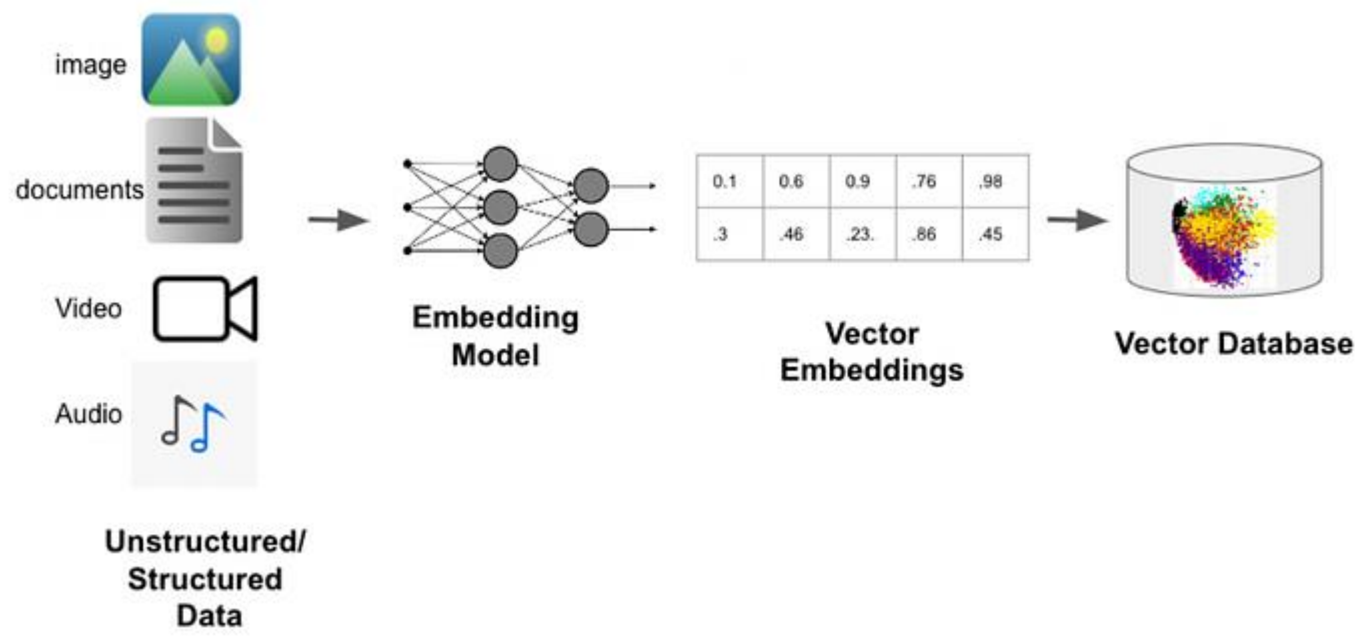
Area	Key Takeaways
Source Formats	PDFs, HTML, Notes, CSVs
Chunking	Semantic, overlapping, avoid fixed cuts
Embeddings	Represent meaning as vectors
Metadata	Boosts relevance, filtering, traceability
Delta Table	ACID-compliant vector storage
Indexing	Enable vector search using Mosaic AI
Governance	Unity Catalog, CDF, observability

Section 3: Application Development

EXL

Embedding Models and Vectorization

What Are Embeddings?



An **embedding** is a vector representation of data — specifically, a way to map words, sentences, or documents to high-dimensional numerical space.

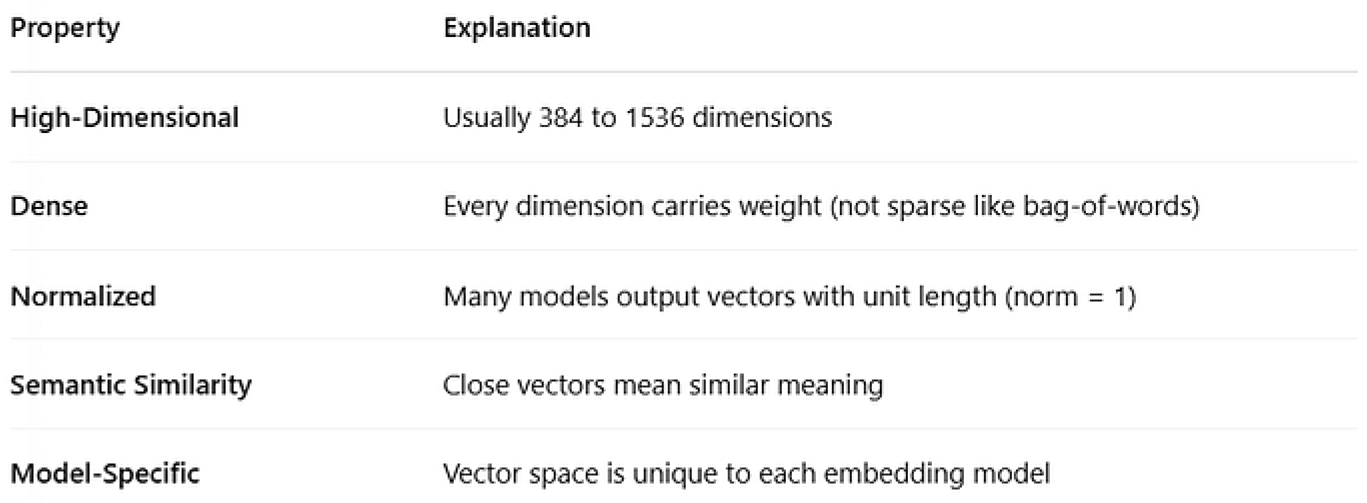
Example:

The phrase **“health insurance coverage”** might map to:
[0.234, -0.111, 0.543, ..., 0.007] → 768-dim vector
Similar phrases like “medical policy benefits” will have vectors **close** to it, while unrelated phrases will be far apart.

This geometric property allows you to:

- Search by **meaning**, not exact words
- Retrieve **semantically relevant** chunks
- Connect unstructured text to structured queries

Key Properties of Embedding Vectors



Property	Explanation
High-Dimensional	Usually 384 to 1536 dimensions
Dense	Every dimension carries weight (not sparse like bag-of-words)
Normalized	Many models output vectors with unit length (norm = 1)
Semantic Similarity	Close vectors mean similar meaning
Model-Specific	Vector space is unique to each embedding model

Cosine Similarity

Most RAG systems use **cosine similarity** to measure the closeness between a user query and document vectors:

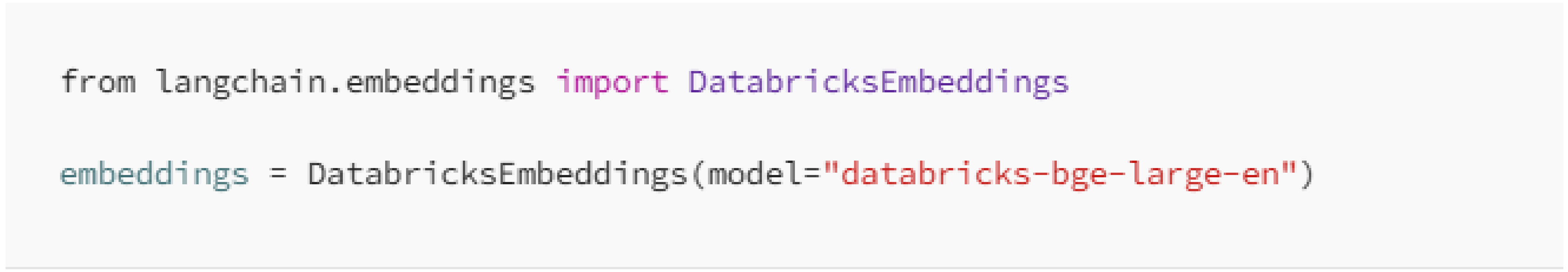
$$\text{cosine\_similarity}(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{||\vec{A}|| \cdot ||\vec{B}||}$$

Types of Embedding Models on Databricks

1. databricks-bge-large-en

The **default embedding model** used in Databricks

- 1024 dimensions, trained on **multi-domain data**
- Great for semantic search, RAG, and classification tasks
- Fine-tuned version of BAAI's BGE series



```
from langchain.embeddings import DatabricksEmbeddings

embeddings = DatabricksEmbeddings(model="databricks-bge-large-en")
```

Types of Embedding Models on Databricks

2. Open Source Models (Hugging Face)

E.g., sentence-transformers/all-MiniLM-L6-v2, mteb/e5-large

- Can be hosted via Hugging Face on Databricks
- Lighter, fast, but often lower performance in enterprise domains

Types of Embedding Models on Databricks

3. Proprietary Models (OpenAI)

- E.g., text-embedding-ada-002
- Used via external APIs
- Higher cost and latency
- May violate data residency or compliance policies

Types of Embedding Models on Databricks

4. Custom Fine-Tuned Models

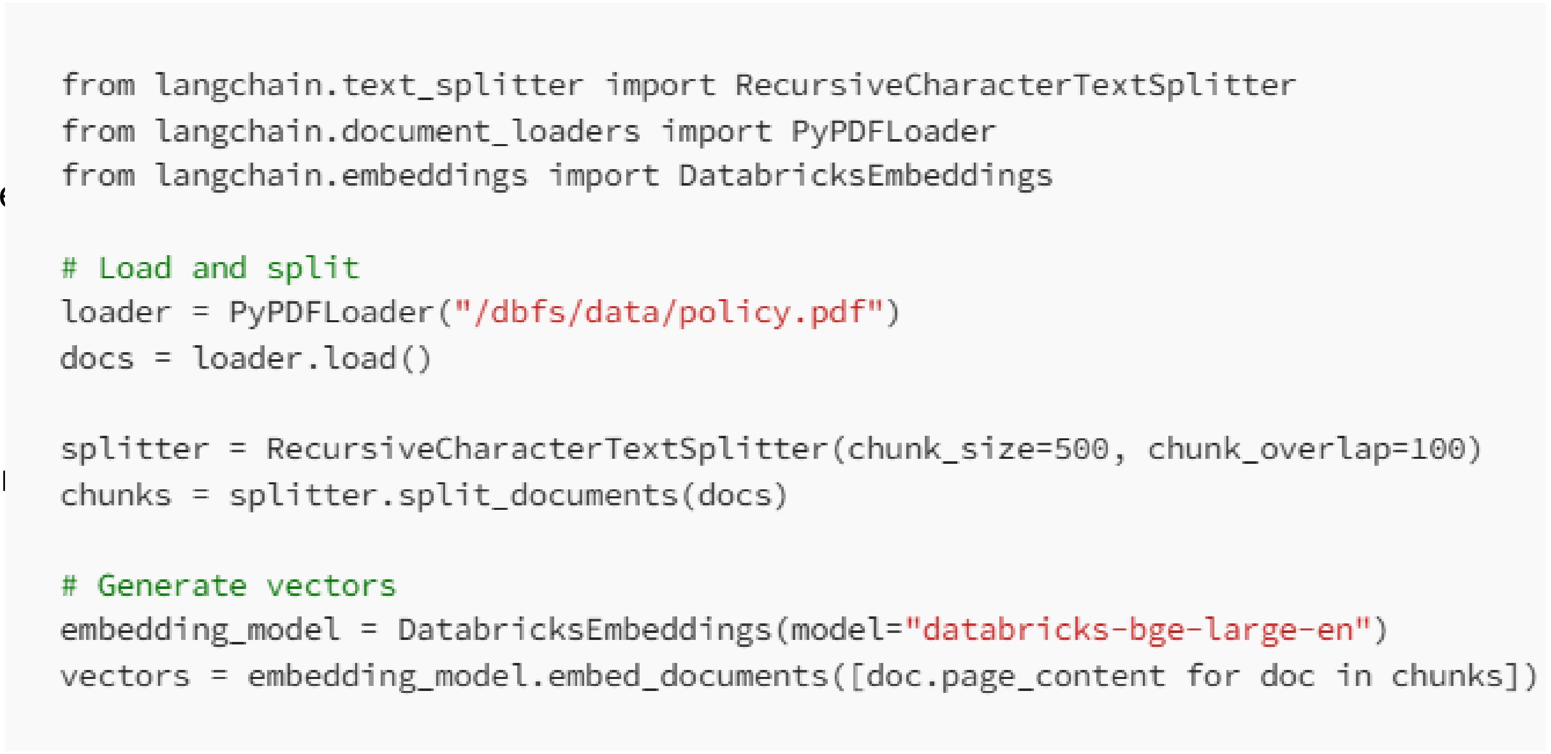
- Train on domain-specific corpora (e.g., legal clauses, financial disclosures)
- Hosted in Model Registry or Model Serving on Databricks

Vectorizing Data on Databricks

Step-by-Step Process

- **Extract Documents**
  Use loaders (PDF, HTML, CSV, etc.)
- **2. Chunk Documents**
  Semantic or recursive chunking (e.g., 500 toke
  with 100 overlap)
- **3. Embed Chunks**
  Generate embeddings for each chunk
- **4. Store Vectors in Delta Table**
  Include raw text, metadata, and the embeddi
  vector
- **5. Index with Mosaic AI Vector Search**

LangChain Example:

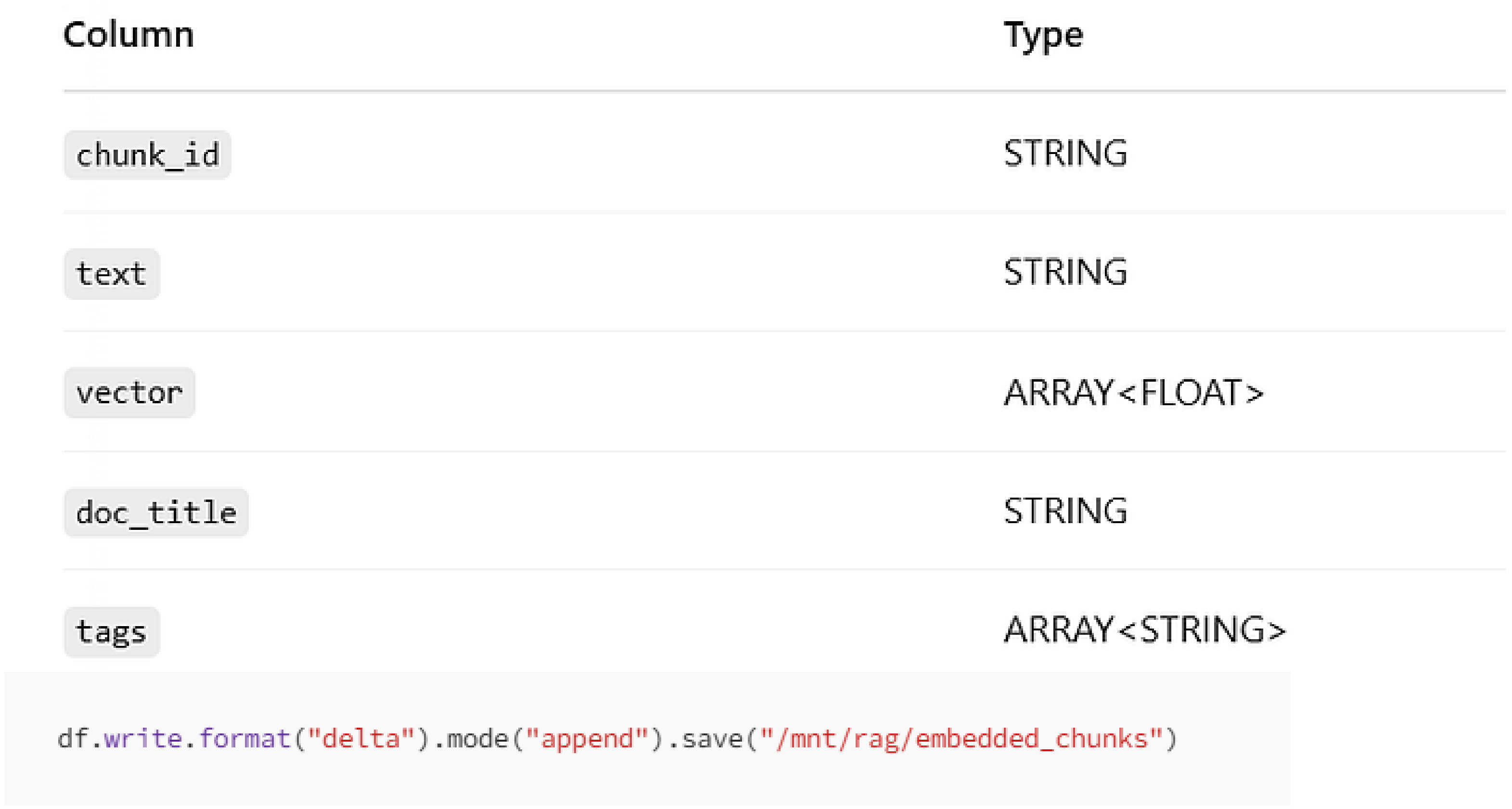


```python
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.document_loaders import PyPDFLoader
from langchain.embeddings import DatabricksEmbeddings

# Load and split
loader = PyPDFLoader("/dbfs/data/policy.pdf")
docs = loader.load()

splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
chunks = splitter.split_documents(docs)

# Generate vectors
embedding_model = DatabricksEmbeddings(model="databricks-bge-large-en")
vectors = embedding_model.embed_documents([doc.page_content for doc in chunks])
```

Storing Embeddings in Delta Lake

- Each vector is a list of float values (length = model output dimensions).

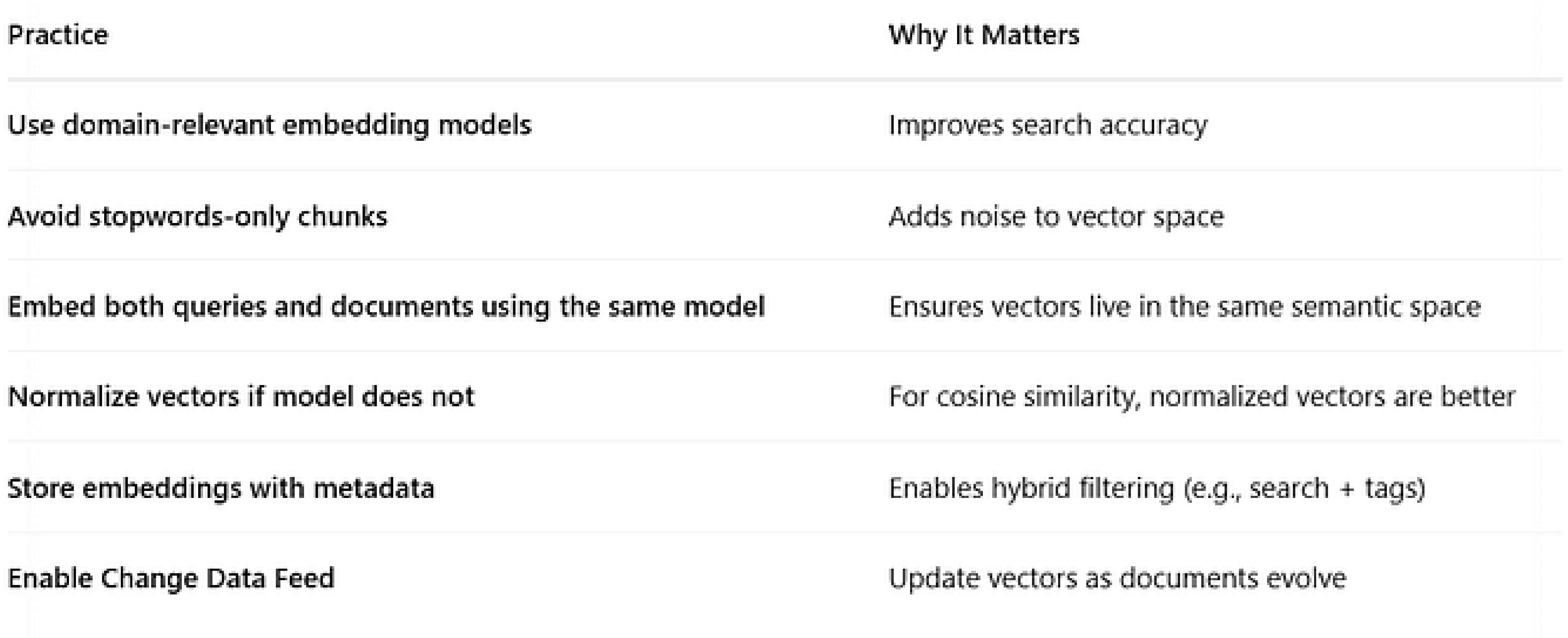
Schema Example:



Column	Type
`chunk_id`	STRING
`text`	STRING
`vector`	ARRAY<FLOAT>
`doc_title`	STRING
`tags`	ARRAY<STRING>

```
df.write.format("delta").mode("append").save("/mnt/rag/embedded_chunks")
```

Best Practices for Embedding Pipelines



Practice	Why It Matters
Use domain-relevant embedding models	Improves search accuracy
Avoid stopwords-only chunks	Adds noise to vector space
Embed both queries and documents using the same model	Ensures vectors live in the same semantic space
Normalize vectors if model does not	For cosine similarity, normalized vectors are better
Store embeddings with metadata	Enables hybrid filtering (e.g., search + tags)
Enable Change Data Feed	Update vectors as documents evolve

Evaluating Embedding Model Quality

1. Manual Testing

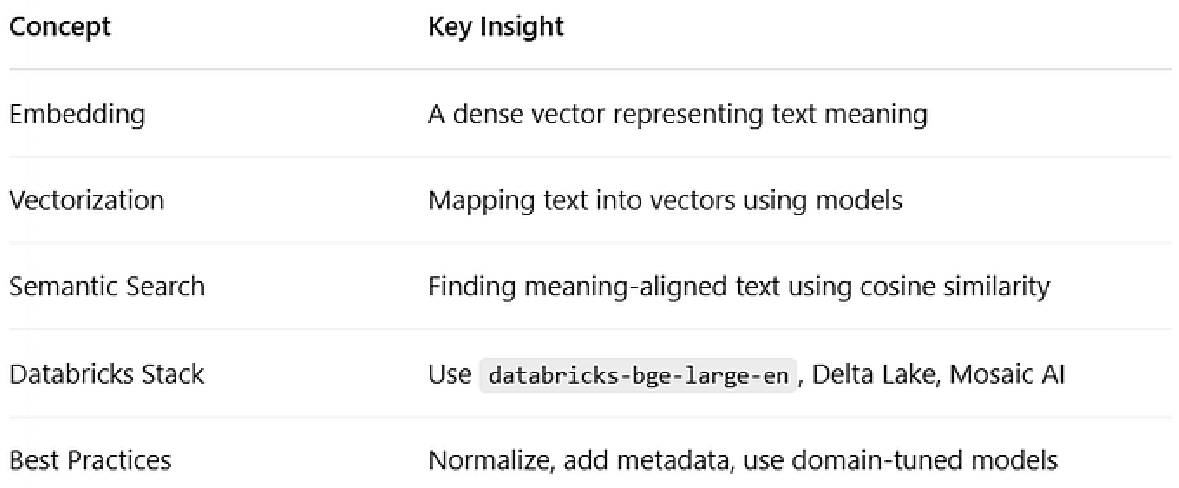
- Input: Query
- Output: Top-5 similar chunks
- Expected: Chunks should answer query without LLM inference

2. Embedding Benchmarks

- Use MTEB, BEIR, or internal Q&A datasets to compare models

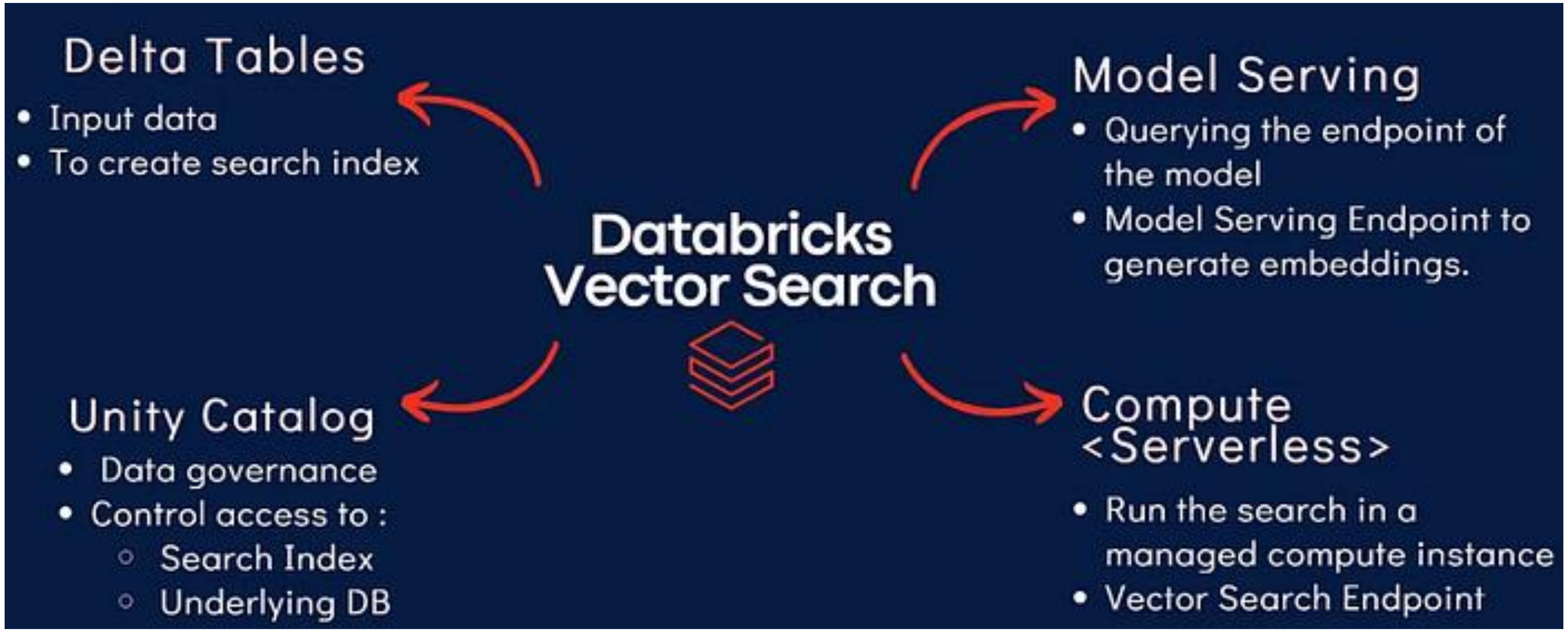
3. LLM-Assisted Feedback Loop

- RAG + LLM → Measure response quality.
- Trace poor answers to bad retrieval → Fix embeddings



Concept	Key Insight
Embedding	A dense vector representing text meaning
Vectorization	Mapping text into vectors using models
Semantic Search	Finding meaning-aligned text using cosine similarity
Databricks Stack	Use `databricks-bge-large-en`, Delta Lake, Mosaic AI
Best Practices	Normalize, add metadata, use domain-tuned models

Databricks Vector Search Fundamentals



Delta Tables
- Input data
- To create search index

Model Serving
- Querying the endpoint of the model
- Model Serving Endpoint to generate embeddings.

Unity Catalog
- Data governance
- Control access to :
    - Search Index
    - Underlying DB

Compute <Serverless>
- Run the search in a managed compute instance
- Vector Search Endpoint

Databricks Vector Search Fundamentals

What Is Vector Search?

Vector search is a way to find information by **meaning**, not by exact terms.

A **text chunk** (e.g., a paragraph from a policy document) is turned into a vector — a list of floating-point numbers (e.g., 768 or 1024-dimensional).

A **user's query** is also converted into a vector using the same model. The system then compares these vectors to find **the most similar content** using metrics like **cosine similarity**. This process is **independent of specific keywords**, allowing natural language queries to return more intelligent results.

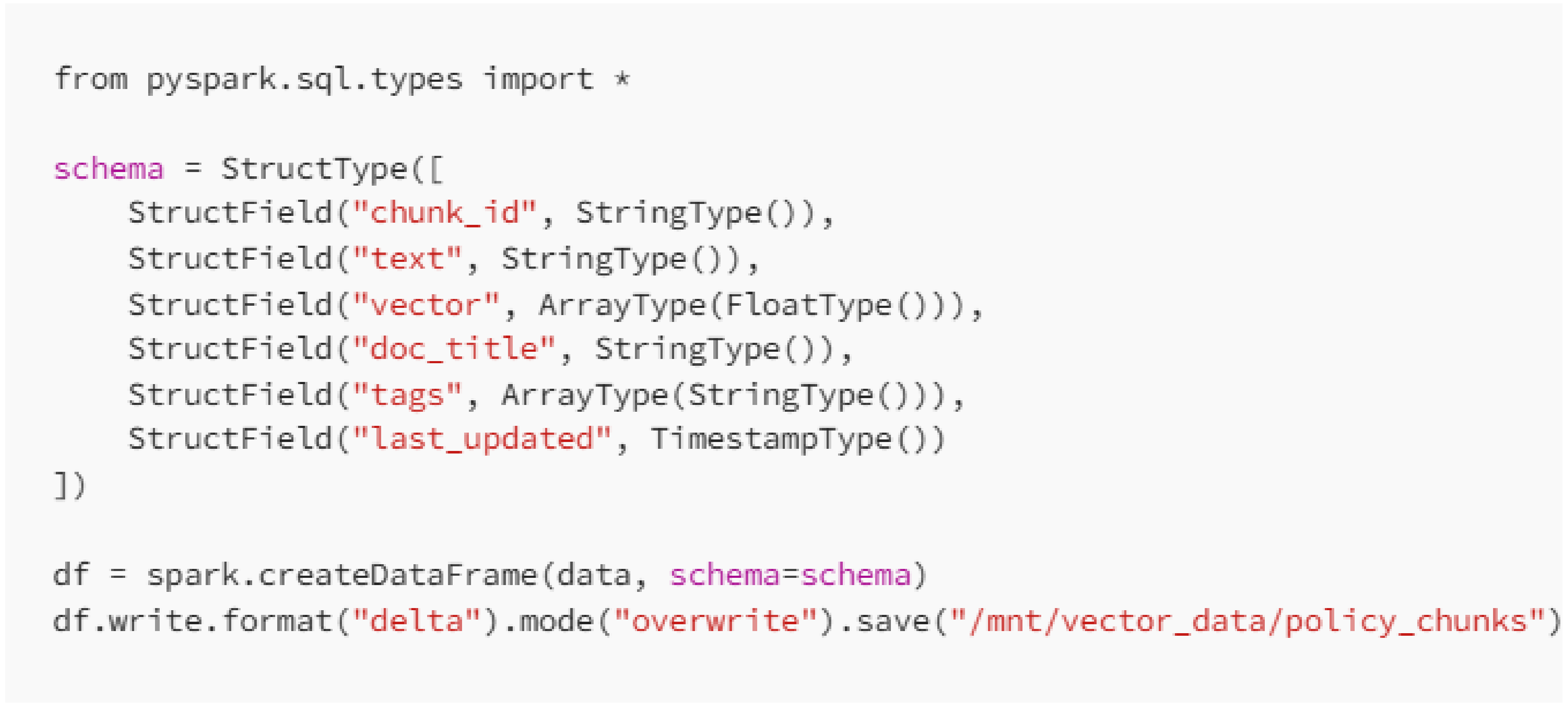
How Does Vector Search Work in Databricks?

Mosaic AI Vector Search

- Databricks provides **Mosaic AI Vector Search**, a first-party native engine designed to:
- Index billions of vectors stored in Delta Lake
- Enable fast, semantic top-K retrieval
- Support hybrid queries with structured filters
- Integrate seamlessly with LangChain and ChatDatabricks
- Govern and track access using Unity Catalog
- Support Change Data Feed for real-time updates

Step-by-Step: Creating a Vector Search Index

Step 1: Store Embeddings in a Delta Table



```
from pyspark.sql.types import *

schema = StructType([
    StructField("chunk_id", StringType()),
    StructField("text", StringType()),
    StructField("vector", ArrayType(FloatType())),
    StructField("doc_title", StringType()),
    StructField("tags", ArrayType(StringType())),
    StructField("last_updated", TimestampType())
])

df = spark.createDataFrame(data, schema=schema)
df.write.format("delta").mode("overwrite").save("/mnt/vector_data/policy_chunks")
```

Step-by-Step: Creating a Vector Search Index

Step 2: Register Table in Unity Catalog

```sql
CREATE TABLE rag_catalog.rag_schema.policy_chunks
USING DELTA
LOCATION '/mnt/vector_data/policy_chunks';
```

Step 3: Enable Change Data Feed

```sql
ALTER TABLE rag_catalog.rag_schema.policy_chunks
SET TBLPROPERTIES (delta.enableChangeDataFeed = true);
```

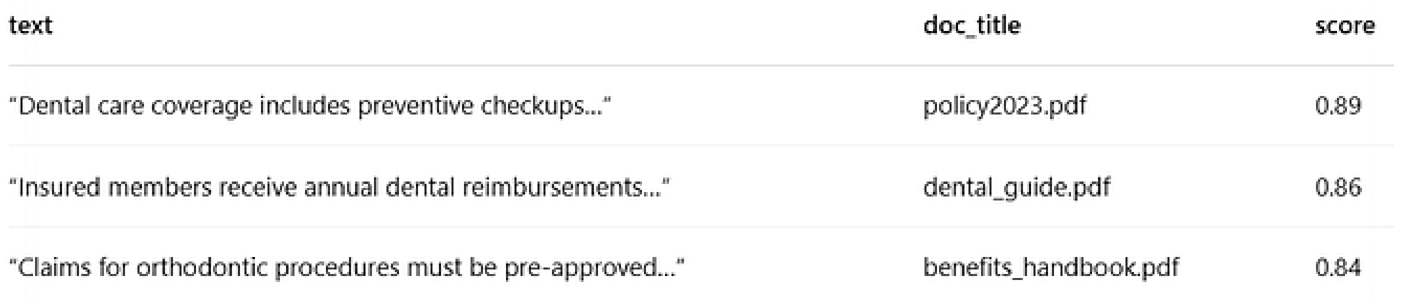
Step 4: Create Vector Index

```sql
CREATE VECTOR INDEX idx_policy_chunks
ON TABLE rag_catalog.rag_schema.policy_chunks
EMBEDDING_COLUMN vector
TEXT_COLUMN text
METADATA_COLUMNS (doc_title, tags);
```

Now your data is ready for semantic querying using SQL or APIs.

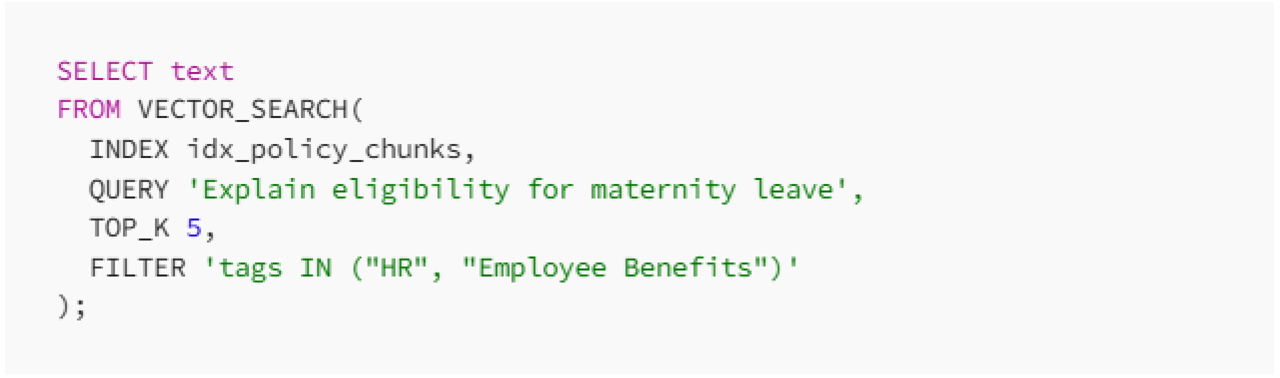
This will:
Embed the query, Search the vector index, Return the top 3 most semantically similar chunks

Sample Output:



text	doc_title	score
"Dental care coverage includes preventive checkups..."	policy2023.pdf	0.89
"Insured members receive annual dental reimbursements..."	dental_guide.pdf	0.86
"Claims for orthodontic procedures must be pre-approved..."	benefits_handbook.pdf	0.84

You can add filters to constrain semantic results based on structured metadata.

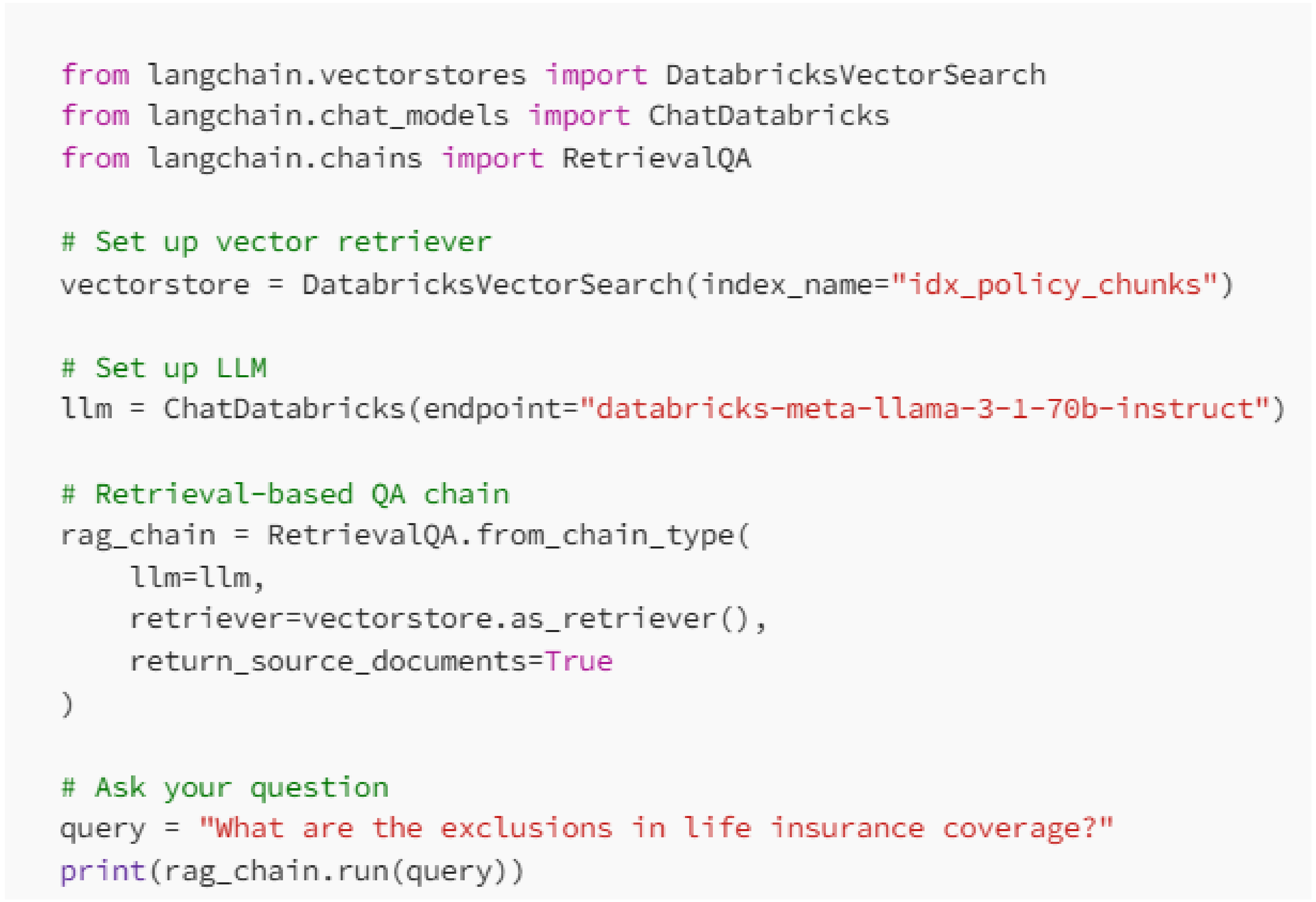


```
SELECT text
FROM VECTOR_SEARCH(
  INDEX idx_policy_chunks,
  QUERY 'Explain eligibility for maternity leave',
  TOP_K 5,
  FILTER 'tags IN ("HR", "Employee Benefits")'
);
```

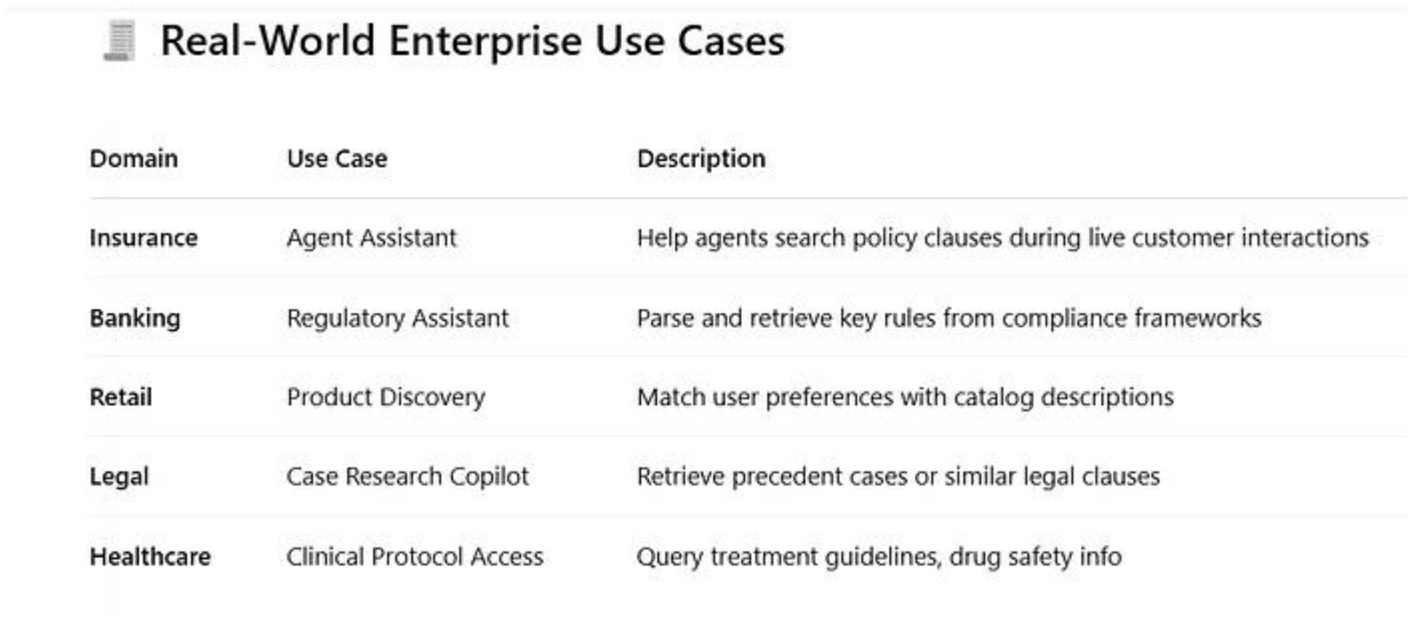
This combines **semantic similarity** with **enterprise control** — a must for domain-specific use cases.

Integration with LangChain (Python)

LangChain makes it easy to plug Mosaic Vector Search into your GenAI pipeline.



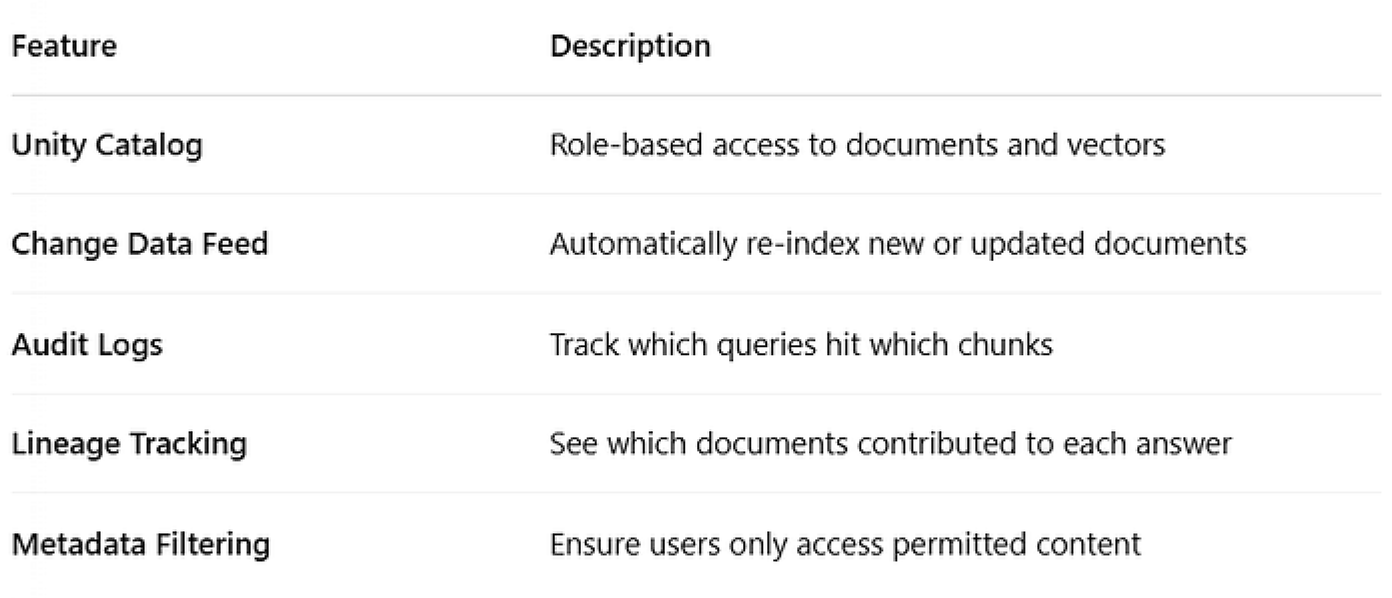
```python
from langchain.vectorstores import DatabricksVectorSearch
from langchain.chat_models import ChatDatabricks
from langchain.chains import RetrievalQA

# Set up vector retriever
vectorstore = DatabricksVectorSearch(index_name="idx_policy_chunks")

# Set up LLM
llm = ChatDatabricks(endpoint="databricks-meta-llama-3-1-70b-instruct")

# Retrieval-based QA chain
rag_chain = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=vectorstore.as_retriever(),
    return_source_documents=True
)

# Ask your question
query = "What are the exclusions in life insurance coverage?"
print(rag_chain.run(query))
```



Real-World Enterprise Use Cases

Domain	Use Case	Description
Insurance	Agent Assistant	Help agents search policy clauses during live customer interactions
Banking	Regulatory Assistant	Parse and retrieve key rules from compliance frameworks
Retail	Product Discovery	Match user preferences with catalog descriptions
Legal	Case Research Copilot	Retrieve precedent cases or similar legal clauses
Healthcare	Clinical Protocol Access	Query treatment guidelines, drug safety info

Governance, Observability & Auditability - Databricks brings **enterprise trust** to vector search:



Feature	Description
Unity Catalog	Role-based access to documents and vectors
Change Data Feed	Automatically re-index new or updated documents
Audit Logs	Track which queries hit which chunks
Lineage Tracking	See which documents contributed to each answer
Metadata Filtering	Ensure users only access permitted content

Integrating Vector Search into GenAI Pipelines

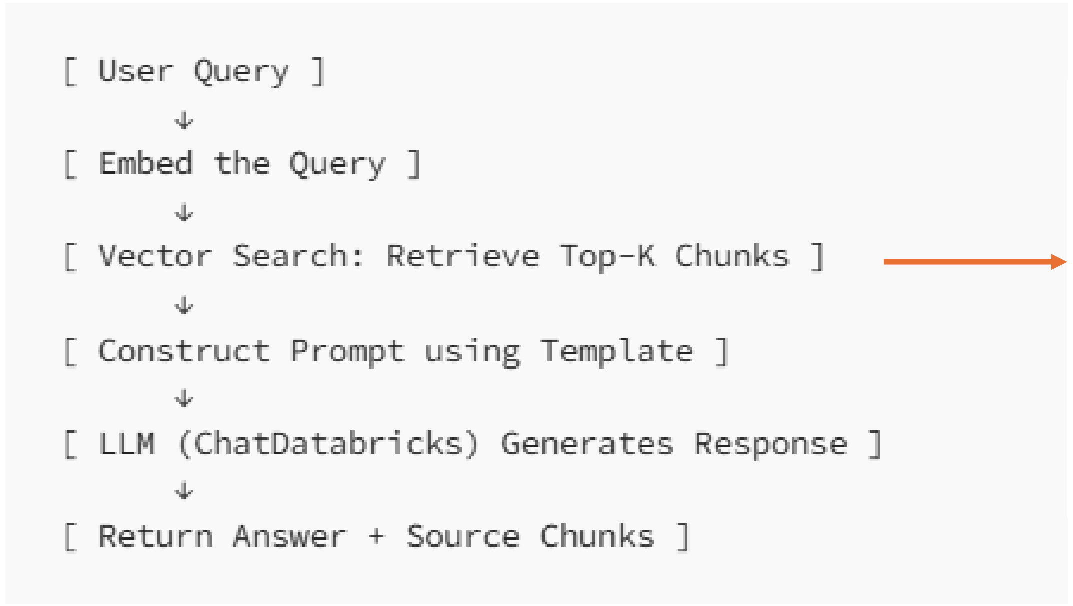
Why Integration Matters

Embedding data and creating a vector index is only part of the story. A **complete GenAI pipeline** is one where:

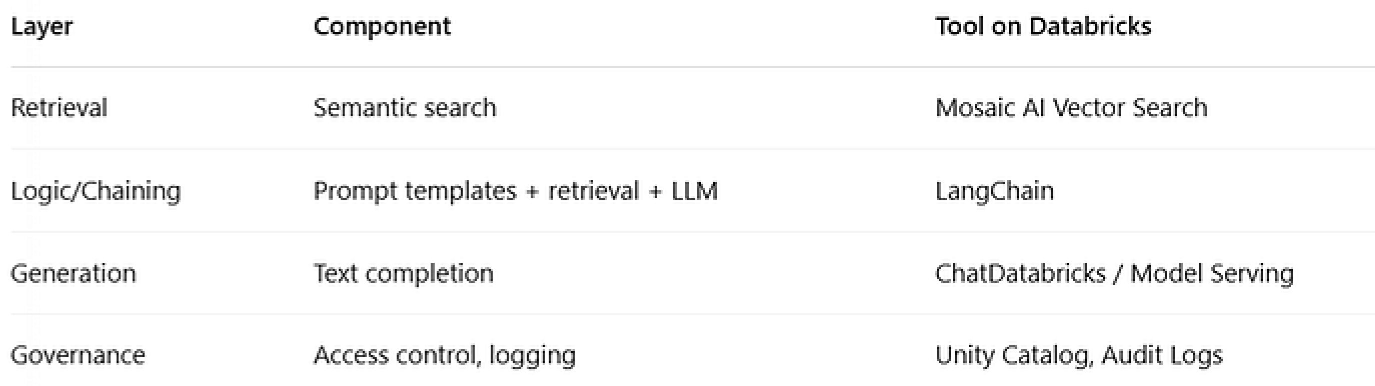
- A user asks a natural language question
- Relevant context is retrieved from vector search
- The prompt is dynamically assembled
- A large language model generates a grounded, contextual answer
- The result is explainable, secure, and production-ready

RAG Architecture Recap

Each layer has a specific function:



```
[ User Query ]
       ↓
[ Embed the Query ]
       ↓
[ Vector Search: Retrieve Top-K Chunks ]
       ↓
[ Construct Prompt using Template ]
       ↓
[ LLM (ChatDatabricks) Generates Response ]
       ↓
[ Return Answer + Source Chunks ]
```



Layer	Component	Tool on Databricks
Retrieval	Semantic search	Mosaic AI Vector Search
Logic/Chaining	Prompt templates + retrieval + LLM	LangChain
Generation	Text completion	ChatDatabricks / Model Serving
Governance	Access control, logging	Unity Catalog, Audit Logs

1. Vector Search Setup (Recap)

Assume we already have:

- A Delta table with:
- text, vector, doc_title, tags
- An index created via:

```
CREATE VECTOR INDEX idx_policy_chunks
ON TABLE rag_catalog.rag_schema.policy_chunks
EMBEDDING_COLUMN vector
TEXT_COLUMN text
METADATA_COLUMNS (doc_title, tags);
```

2. Setting Up LangChain Components

Install requirements in Databricks (if not already available):

```
from langchain.vectorstores import DatabricksVectorSearch

retriever = DatabricksVectorSearch(index_name="idx_policy_chunks").as_retriever()
```

a) Load the Vector Index as a Retriever

```
from langchain.vectorstores import DatabricksVectorSearch

retriever = DatabricksVectorSearch(index_name="idx_policy_chunks").as_retriever()
```

b) Load the LLM from Databricks Serving

```
from langchain.chat_models import ChatDatabricks

llm = ChatDatabricks(endpoint="databricks-meta-llama-3-1-70b-instruct")
```

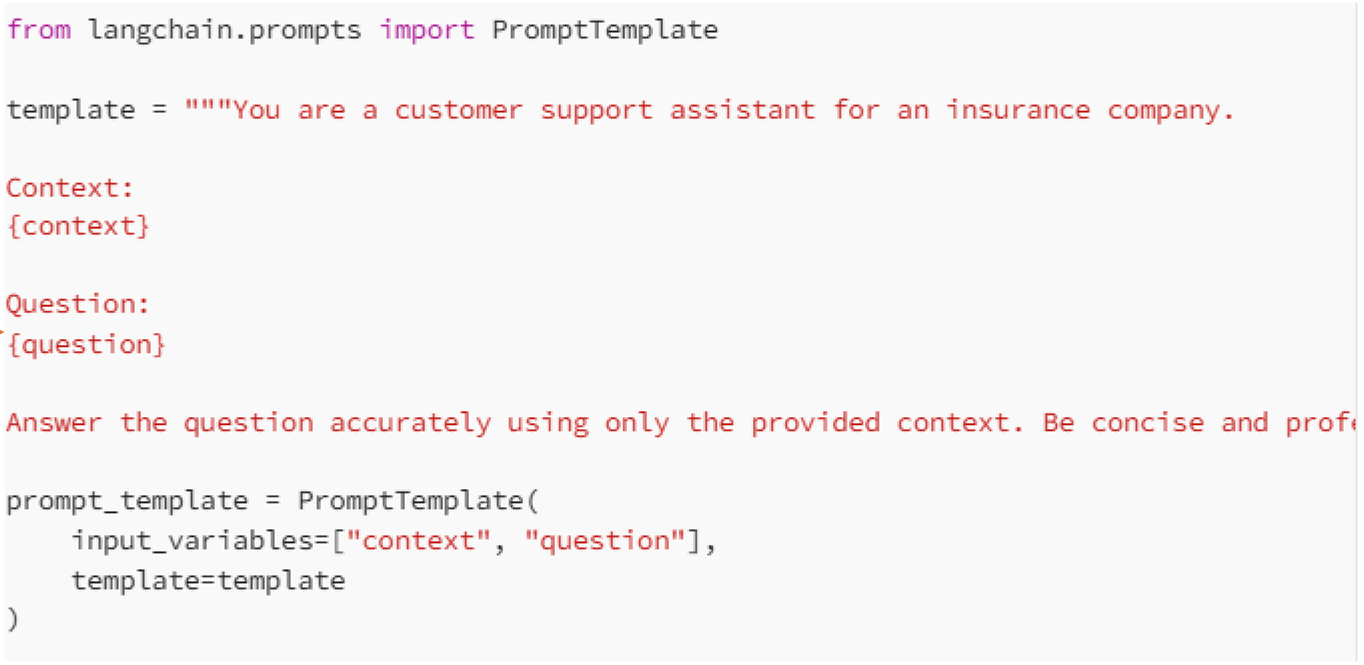
3. Designing the Prompt Template

Prompt templates help guide LLM behavior and structure.

A strong RAG prompt combines:

- User question
- Retrieved context
- Instructions (e.g., tone, format)

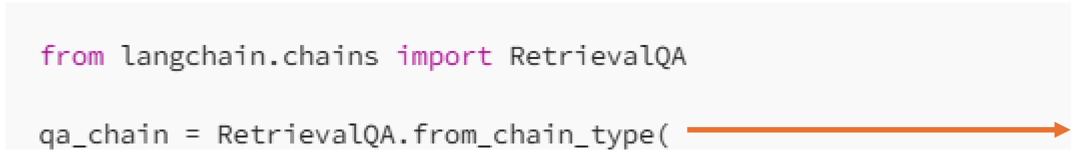
Example Template:



```python
from langchain.prompts import PromptTemplate

template = """You are a customer support assistant for an insurance company.

Context:
{context}

Question:
{question}

Answer the question accurately using only the provided context. Be concise and profe

prompt_template = PromptTemplate(
    input_variables=["context", "question"],
    template=template
)
```

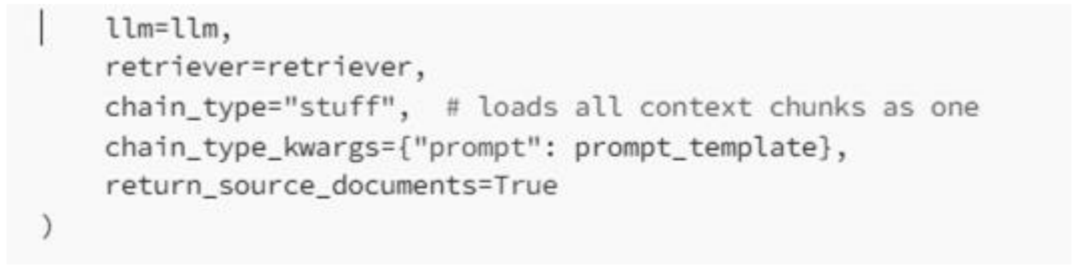
4. Assembling the Chain

Use RetrievalQA to stitch together retrieval + prompt +



```python
from langchain.chains import RetrievalQA

qa_chain = RetrievalQA.from_chain_type(
```

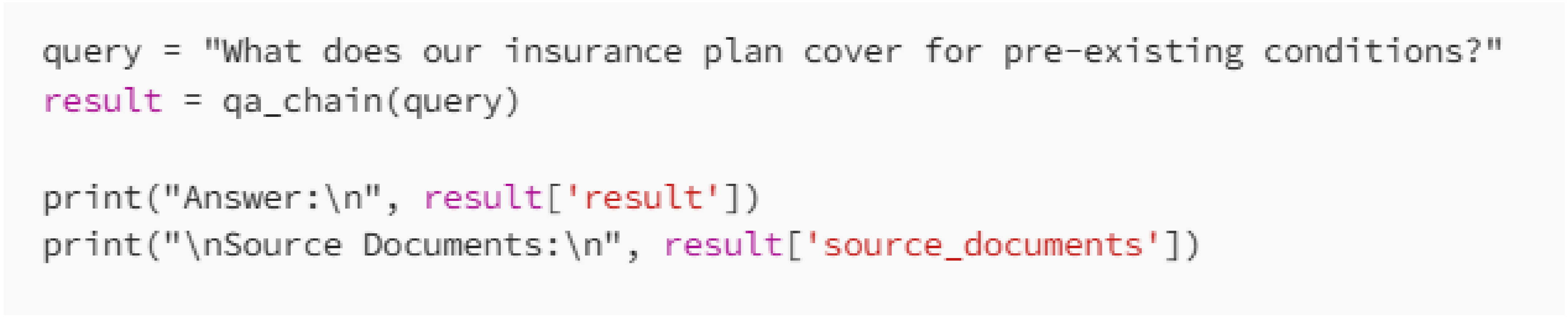
Creates an instance of RetrievalQA using the from_chain_type factory method. This lets you specify the type of retrieval and QA logic you want.



```python
    llm=llm,
    retriever=retriever,
    chain_type="stuff",  # loads all context chunks as one
    chain_type_kwargs={"prompt": prompt_template},
    return_source_documents=True
)
```

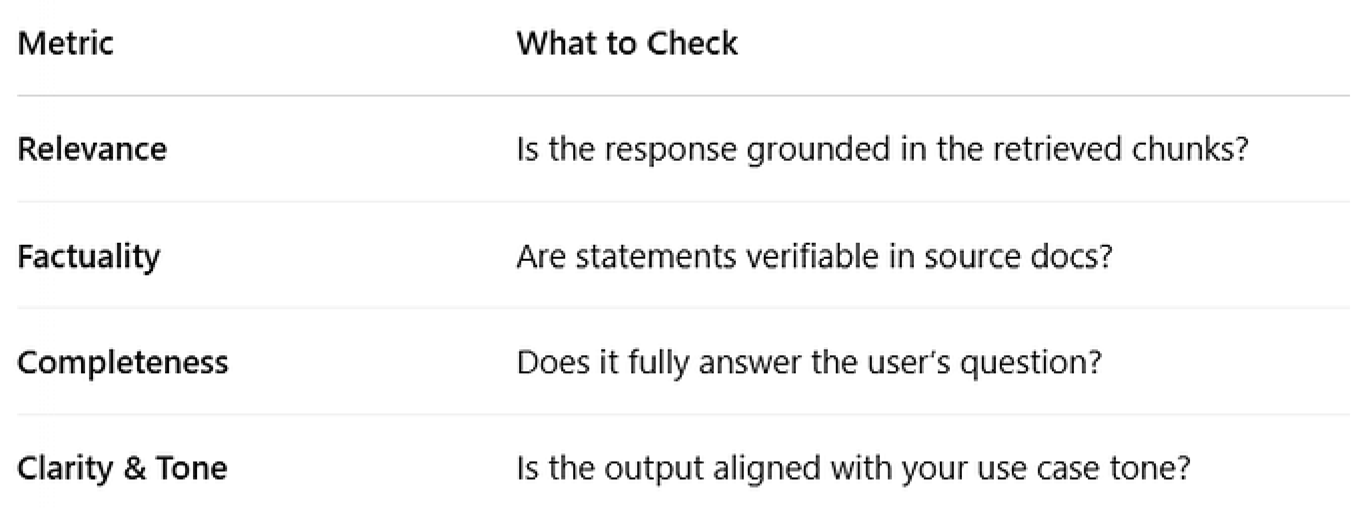
Parameter	Description
llm=llm	Language model for answering questions
retriever=retriever	Fetches relevant context documents
chain_type="stuff"	Loads all context as one chunk for LLM
chain_type_kwargs	Defines prompt used for LLM
return_source_documents	Returns answer + supporting source docs

Run the pipeline:



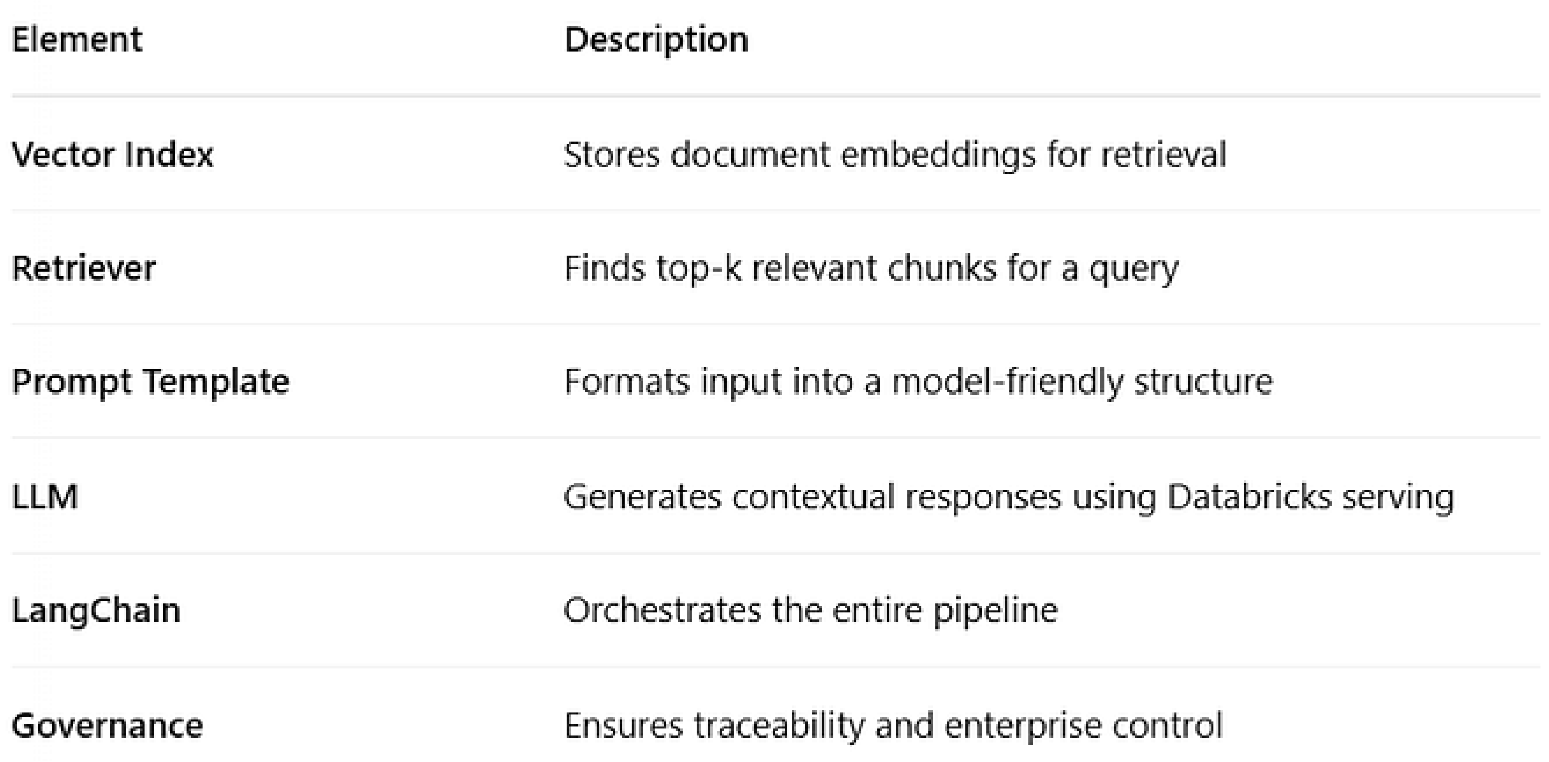
```
query = "What does our insurance plan cover for pre-existing conditions?"
result = qa_chain(query)

print("Answer:\n", result['result'])
print("\nSource Documents:\n", result['source_documents'])
```

5. Validating and Testing Your Pipeline - Functional Tests:



Metric	What to Check
Relevance	Is the response grounded in the retrieved chunks?
Factuality	Are statements verifiable in source docs?
Completeness	Does it fully answer the user's question?
Clarity & Tone	Is the output aligned with your use case tone?

Summary



Element	Description
Vector Index	Stores document embeddings for retrieval
Retriever	Finds top-k relevant chunks for a query
Prompt Template	Formats input into a model-friendly structure
LLM	Generates contextual responses using Databricks serving
LangChain	Orchestrates the entire pipeline
Governance	Ensures traceability and enterprise control

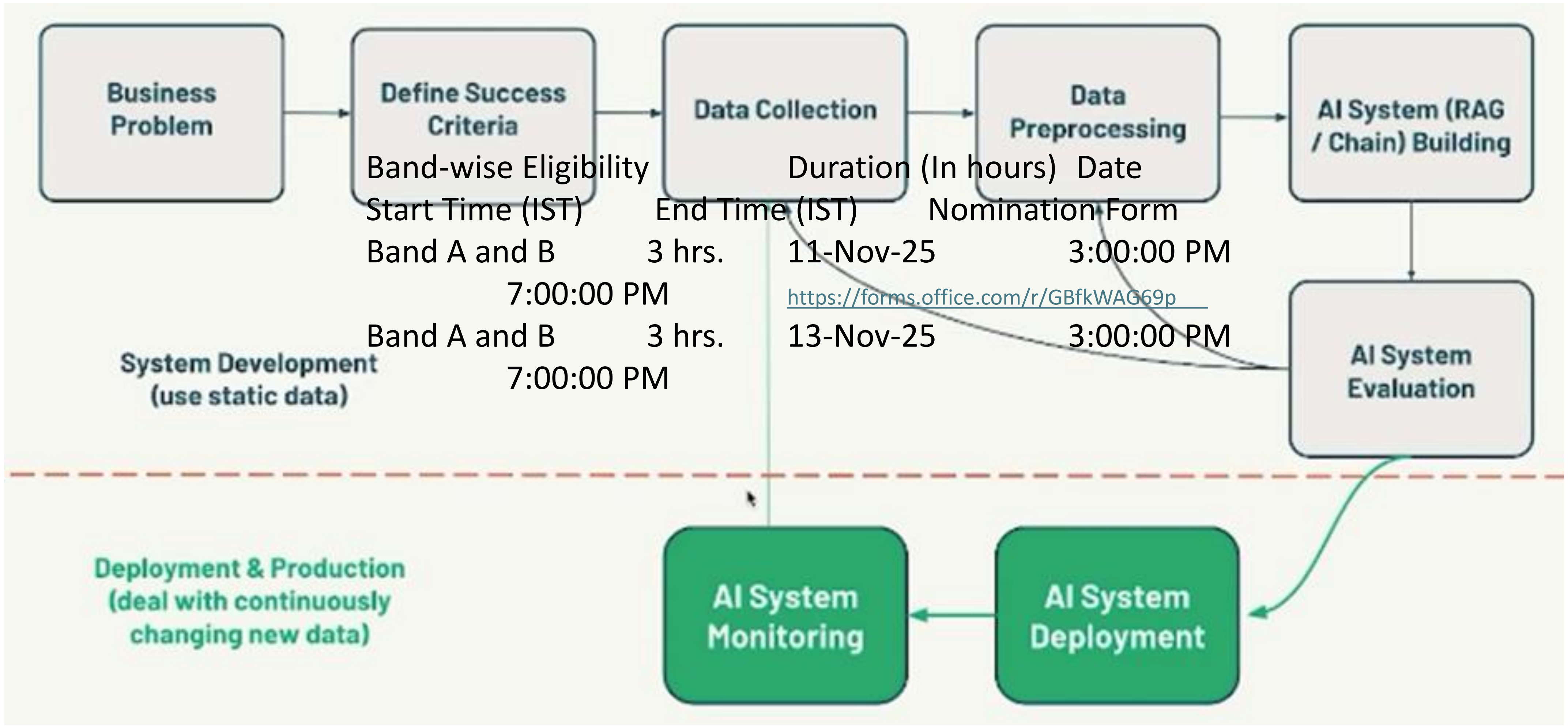
Section 4: Assembling and Deploying Applications

EXL

Section 4: Assembling and Deploying Applications

System Lifecycle:

Press enter or click to view image in full size


Business Problem
Define Success Criteria
Data Collection
Data Preprocessing
AI System (RAG / Chain) Building
AI System Evaluation
System Development (use static data)
Deployment & Production (deal with continuously changing new data)
AI System Monitoring
AI System Deployment

Band-wise Eligibility	Duration (In hours)	Date	Start Time (IST)	End Time (IST)	Nomination Form
Band A and B	3 hrs.	11-Nov-25	3:00:00 PM	7:00:00 PM	https://forms.office.com/r/GBfkWAG69p
Band A and B	3 hrs.	13-Nov-25	3:00:00 PM	7:00:00 PM	

Key Points:

- **Pyfunc Model**: MLflow's pyfunc flavor is a versatile model interface for MLflow Python models. It allows models to be loaded as Python functions for deployment.
- **Pre- and Post-Processing**: These are critical for preparing input data before it is fed into the model (pre-processing) and for handling the model's output before it is presented to the end-user or downstream applications (post-processing). Techniques can include data normalization, feature extraction, and output formatting.
- **Implementation**: Utilize the mlflow.pyfunc to log, save, and load models with necessary pre- and post-processing steps. This ensures the model can handle real-world data inputs and outputs effectively.

34. Create and query a Vector Search index

Key Points:

- **Vector Search Setup**: Create a vector search index by syncing it with a Delta table that stores embeddings. This index allows for real-time approximate nearest neighbor searches.
- Use the provided REST API or Python SDK to query the vector search index. Queries can be made using vector representations to find similar documents or data points.
- Mosaic AI Vector Search supports automatic syncing, self-managed embeddings, and CRUD operations. It integrates with Unity Catalog for governance and access control.

35. Identify how to serve an LLM application that leverages Foundation Model APIs

Key Points:

- **Foundation Model APIs**: Foundation models like OpenAI's GPT are served via Databricks Model Serving. These APIs provide a standardized way to deploy and query large language models without much effort by user.
- **Serving Process**: **Model Deployment, Query Handling, Integration** with MLflow and **Resource Management** to ensure that appropriate compute resources (CPU/GPU) are allocated for serving the models, and use Databricks' auto-scaling features to handle variable loads efficiently.

36. Identify resources needed to serve features for a RAG application

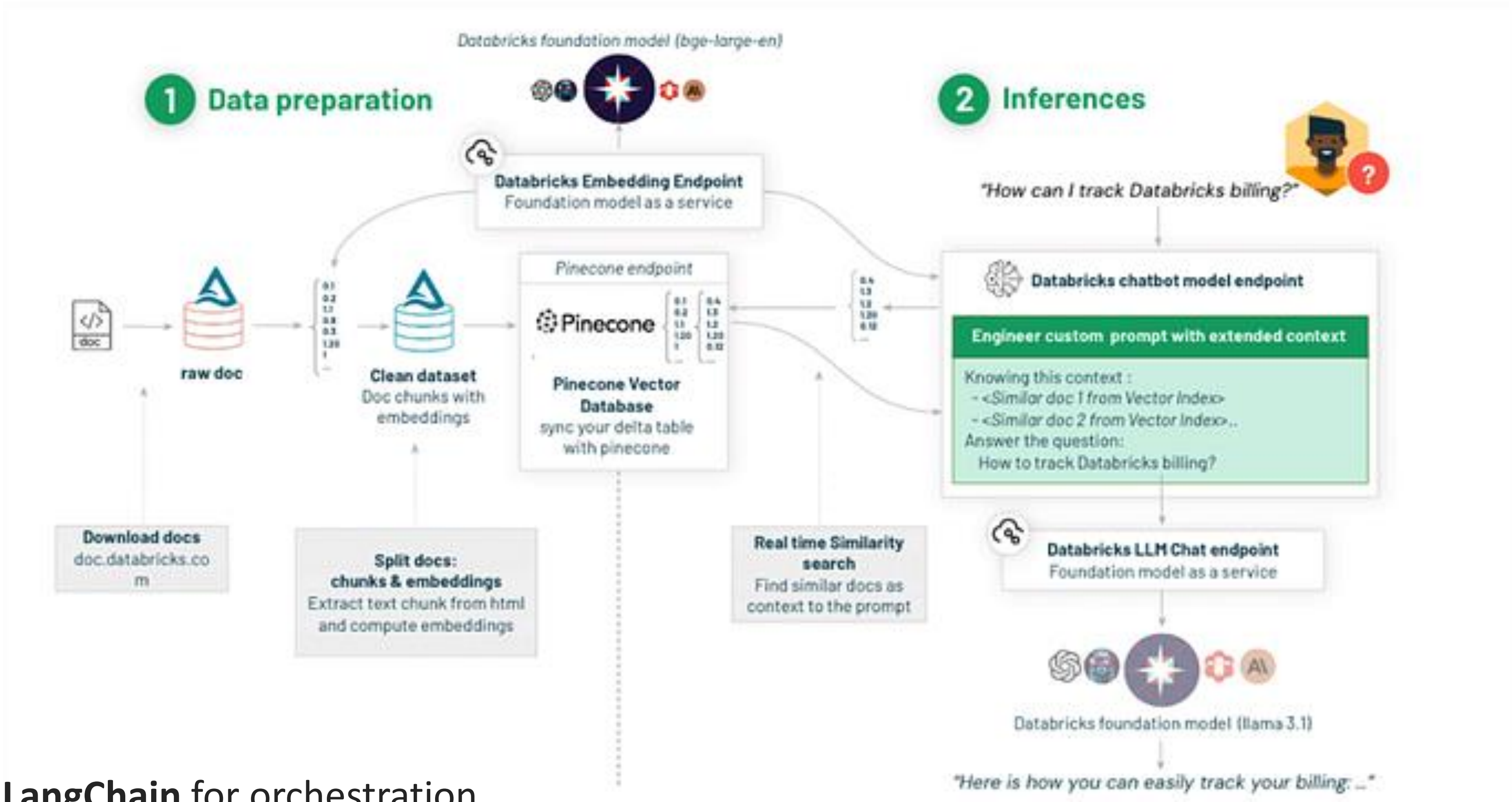
Key Points:

- **Compute Resources**: Like above. Use Databricks' scalable compute options to allocate necessary CPU/GPU resources based on the application load and performance requirements.
- **Storage and Indexing**: Utilize Delta tables for storing raw and processed text, embeddings, and vector indexes. Ensure these are properly managed and synced for efficient retrieval.

Monitoring and Logging: Implement inference logging to track model performance and diagnose issues.

Assembling a RAG Application on Databricks

LangChain + ChatDatabricks + Mosaic Vector Search = A Production-Ready RAG Pipeline


Databricks foundation model (bge-large-en)
1 Data preparation
2 Inferences
Databricks Embedding Endpoint
Foundation model as a service
Pinecone endpoint
Pinecone
raw doc
Clean dataset
Doc chunks with embeddings
Pinecone Vector Database
sync your delta table with pinecone
Download docs
doc.databricks.com
Split docs: chunks & embeddings
Extract text chunk from html and compute embeddings
Real time Similarity search
Find similar docs as context to the prompt
Databricks chatbot model endpoint
Engineer custom prompt with extended context
Databricks LLM Chat endpoint
Foundation model as a service
Databricks foundation model (llama 3.1)

"How can I track Databricks billing?"

Knowing this context :
- <Similar doc 1 from Vector Index>
- <Similar doc 2 from Vector Index>..
Answer the question:
How to track Databricks billing?

"Here is how you can easily track your billing: ..."

- **LangChain** for orchestration
- **Mosaic AI Vector Search** for retrieval
- **ChatDatabricks** for response generation

Pinecone is a fully managed vector database designed to **efficiently store, index, and search high-dimensional vector embeddings produced by AI models**, such as those from text, images, or user data.
This is ideal for use cases like semantic search, recommendation systems, and similarity matching—tasks where traditional databases are slow or inefficient due to the complexity and scale of vector data.

Assembling a RAG Application on Databricks

How Pinecone Works

- **Storing Embeddings:** Pinecone stores numerical vectors (embeddings) that capture semantic information from data such as documents or user profiles.
- **Similarity Search:** When a query is made, Pinecone computes its embedding and finds the most similar vectors by searching through the stored database using efficient similarity measures (e.g., cosine similarity).
- **Real-Time and Scalable:** Pinecone offers millisecond-level response times even for millions to billions of vectors, making it suitable for real-time applications.

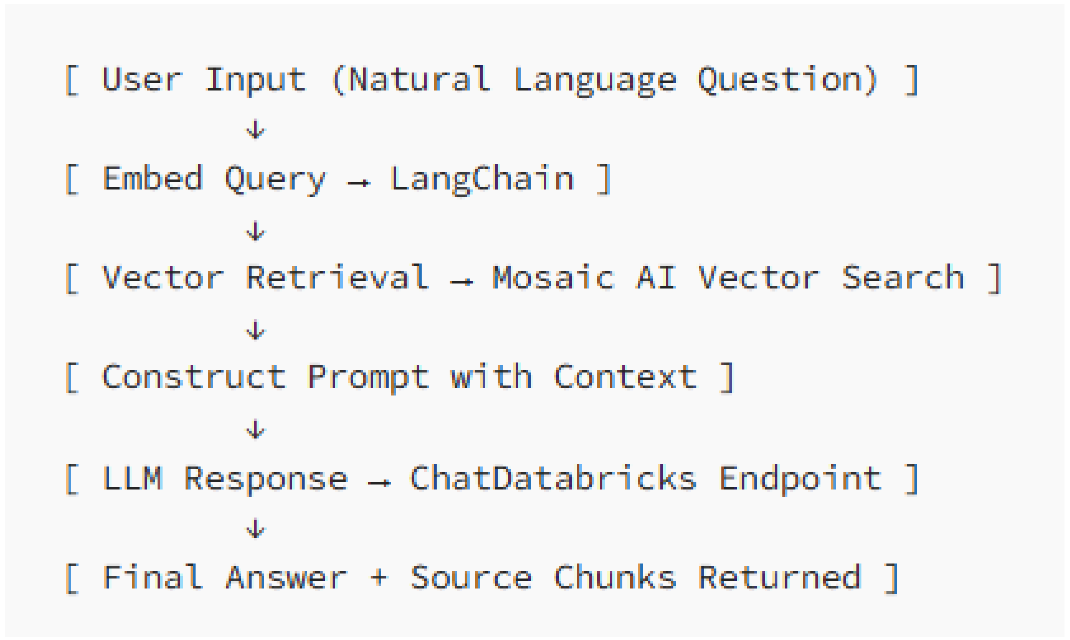
Key Features

- **Fully managed and serverless**, so users do not manage infrastructure.
- **Scales horizontally to handle large, high-dimensional datasets**.
- **Fast, low-latency** vector similarity search.
- **Real-time data ingestion and processing**, always reflecting the latest inserted or updated data.
- **Integration with cloud, ML, and data** platforms for easy application development.

Use Cases

- Retrieval-augmented generation (RAG) for AI chatbots and assistants.
- Semantic search where meaning/context matters more than keywords.
- Product, content, and user recommendation systems.
- Any ML workflow needing complex similarity matching in large datasets.

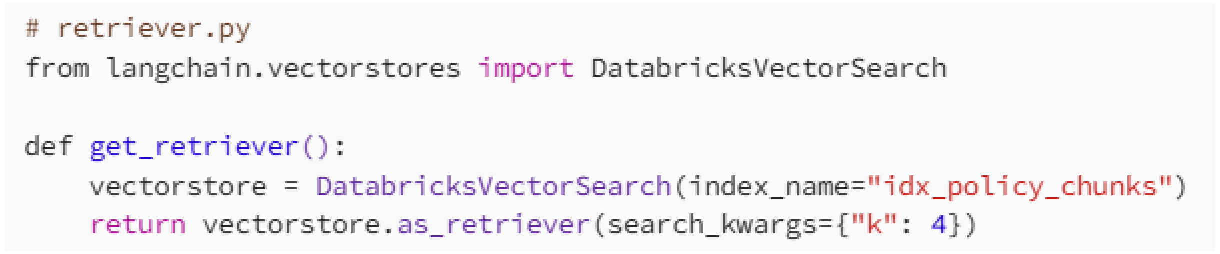
RAG Architecture Recap (Production View)



```
[ User Input (Natural Language Question) ]
         ↓
[ Embed Query → LangChain ]
         ↓
[ Vector Retrieval → Mosaic AI Vector Search ]
         ↓
[ Construct Prompt with Context ]
         ↓
[ LLM Response → ChatDatabricks Endpoint ]
         ↓
[ Final Answer + Source Chunks Returned ]
```

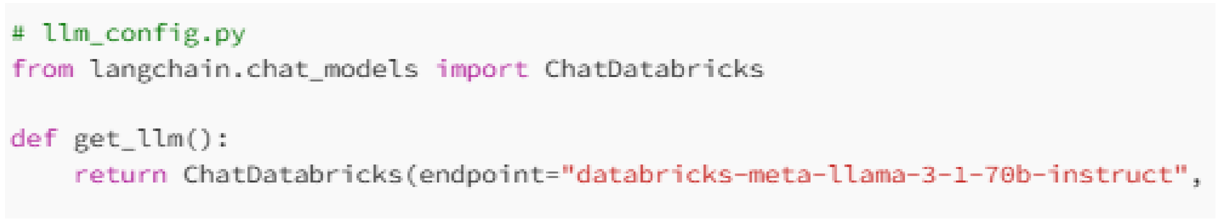
Application Structure

1. **Create the Retriever (Mosaic AI Vector Search)**



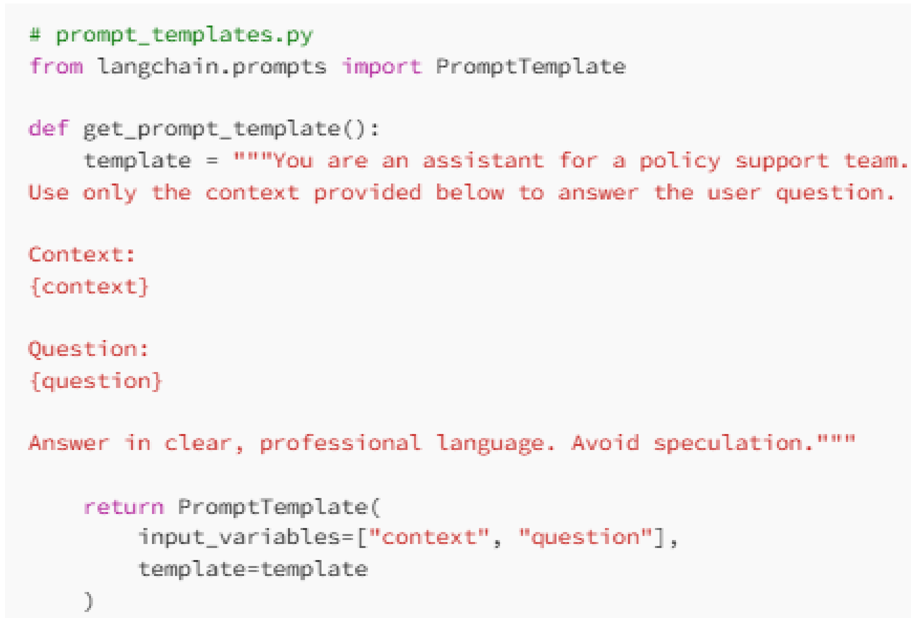
```
# retriever.py
from langchain.vectorstores import DatabricksVectorSearch

def get_retriever():
    vectorstore = DatabricksVectorSearch(index_name="idx_policy_chunks")
    return vectorstore.as_retriever(search_kwargs={"k": 4})
```

2. **Connect to the LLM (ChatDatabricks)**



```
# llm_config.py
from langchain.chat_models import ChatDatabricks

def get_llm():
    return ChatDatabricks(endpoint="databricks-meta-llama-3-1-70b-instruct",
```

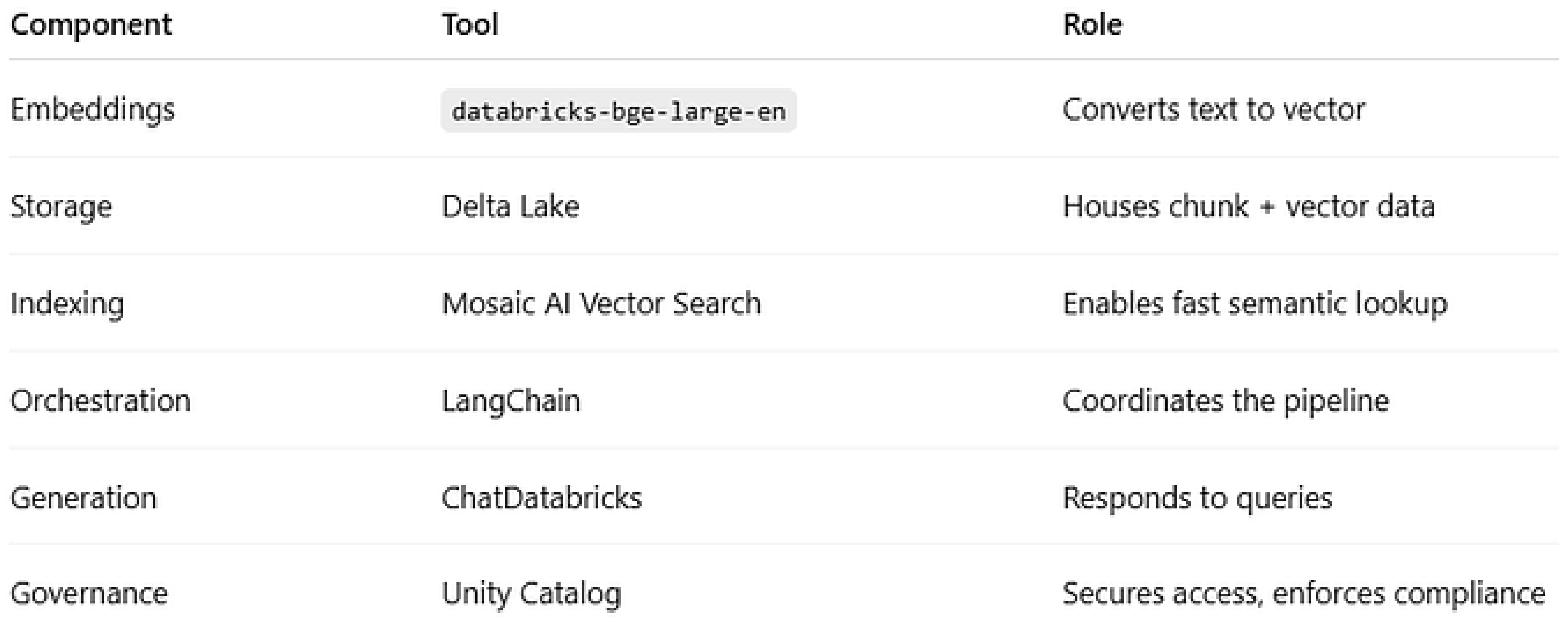
3. **Design a Structured Prompt Template**



```
# prompt_templates.py
from langchain.prompts import PromptTemplate

def get_prompt_template():
    template = """You are an assistant for a policy support team.
Use only the context provided below to answer the user question.

Context:
{context}

Question:
{question}

Answer in clear, professional language. Avoid speculation."""

    return PromptTemplate(
        input_variables=["context", "question"],
        template=template
    )
```

4. **Assemble the RAG Chain**

Summary



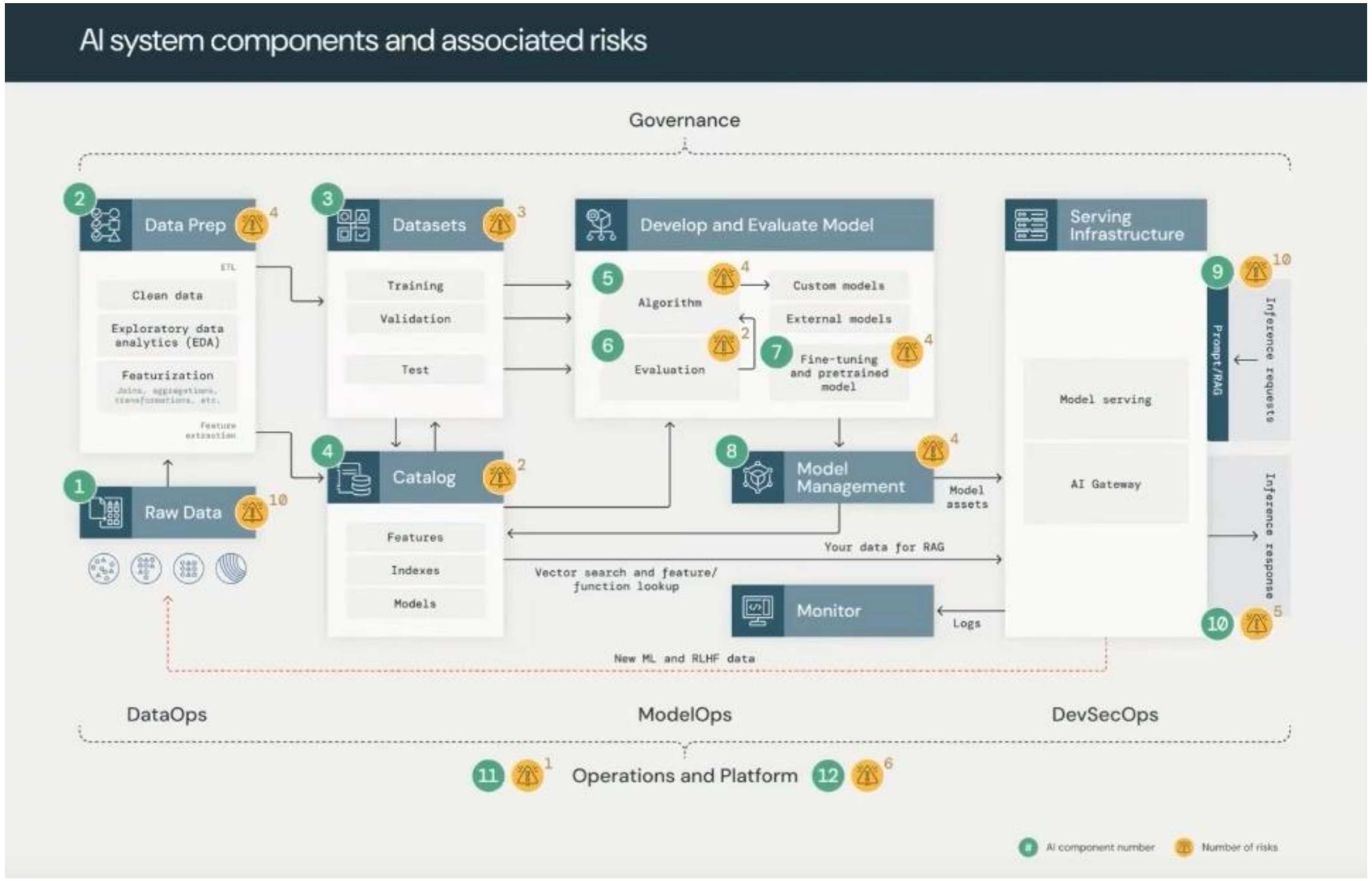
Component	Tool	Role
Embeddings	`databricks-bge-large-en`	Converts text to vector
Storage	Delta Lake	Houses chunk + vector data
Indexing	Mosaic AI Vector Search	Enables fast semantic lookup
Orchestration	LangChain	Coordinates the pipeline
Generation	ChatDatabricks	Responds to queries
Governance	Unity Catalog	Secures access, enforces compliance

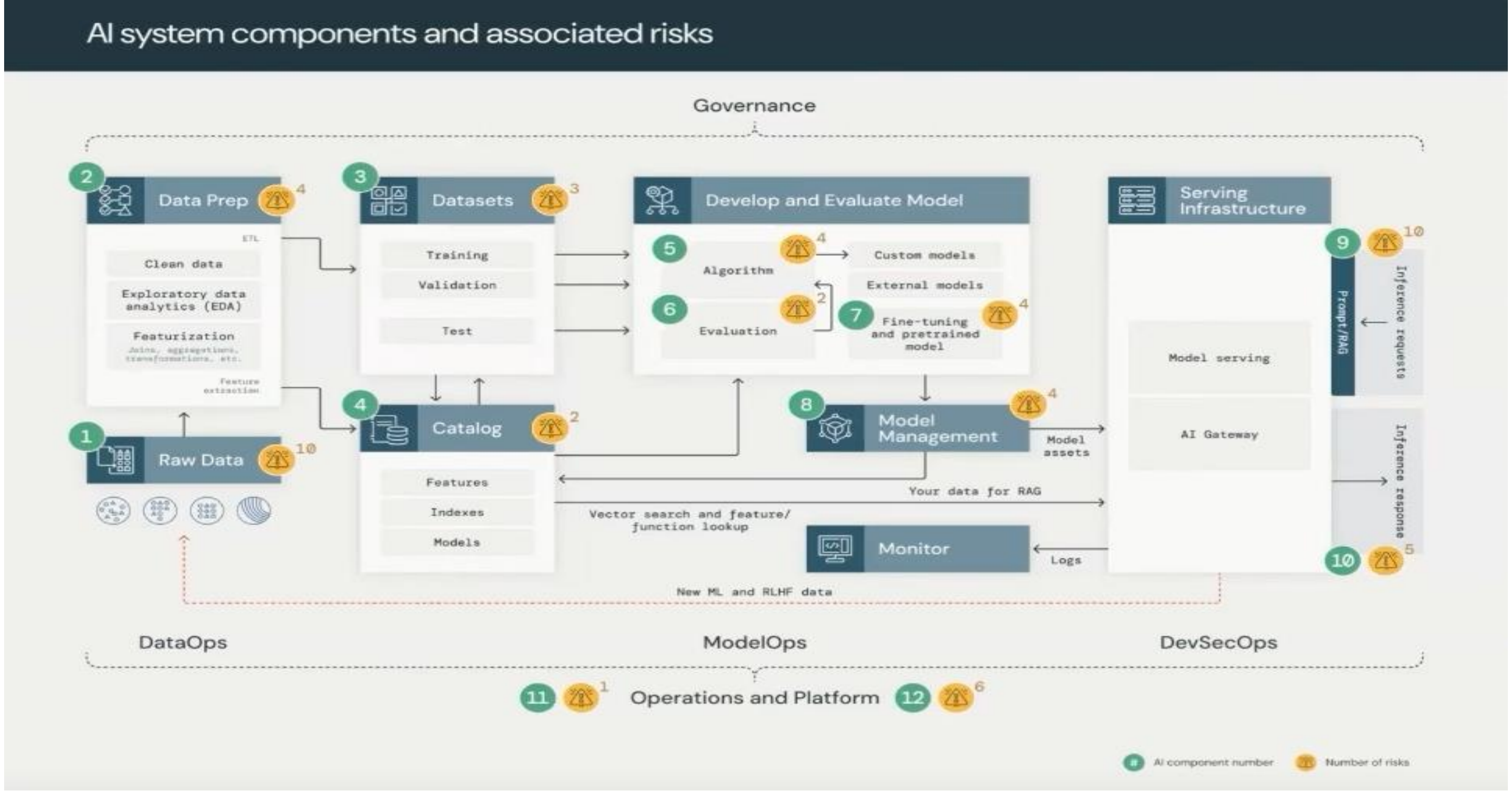
Section 5: Governance

37. Use masking techniques as guard rails to meet a performance objective

Key Points:

- **Masking Techniques**: Examples include tokenization, anonymization, and pseudonymization.
- **Implementation**: Masking can be applied at various stages of data handling, such as input processing, data storage, and during model inference to prevent leakage of private data.
- **Balance** is crucial for meeting both compliance and performance objectives.





Key Workflow Flows

- Data flows from "Raw Data" (1) → "Data Prep" (2) → "Datasets" (3).
- Features/models are catalogued (4) and referenced for model building (5, 6, 7).
- Trained models and their assets are managed (8), then deployed (9) for serving inference requests.
- Monitoring (10) ensures logging and operational stability.

The entire system is under the oversight of "Governance," spanning DataOps, ModelOps, and DevSecOps disciplines.

38. Select guardrail techniques to protect against malicious user inputs to a GenAI application

Key Points:

- **Guardrail Techniques**: Input validation, context-aware filtering, prompt sanitization to protect against harmful or malicious inputs. Implementing these helps in preventing prompt injection attacks and inappropriate content generation.
- Learn about **Llama Guard LLM** which is in Databricks Marketplace
- Continuous testing and updating of guardrails are essential to adapt to new types of malicious inputs and ensure the system's integrity and security.

39. Recommend an alternative for problematic text mitigation in a data source feeding a RAG application

Key Points:

- Addressing issues such as bias, misinformation, or inappropriate content in data sources used
- **Data Cleansing**: Techniques include preprocessing steps like text normalization, filtering out offensive terms, and leveraging domain-specific stop words lists to cleanse the data before it is fed into the model.
- **External Data Sources**: Using reputable and well-moderated external data sources to augment prompts can help mitigate the inclusion of problematic text, ensuring the generation of more accurate and safe responses.

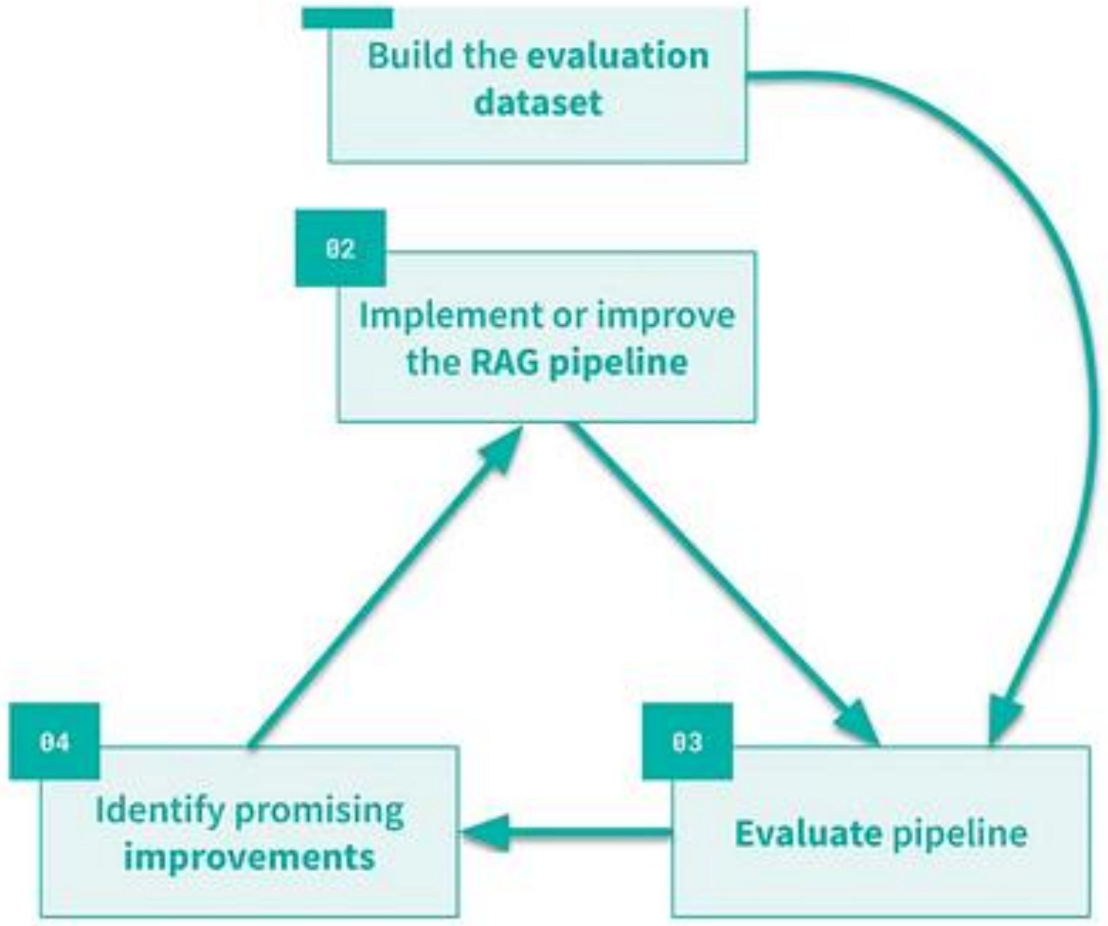
40. Use legal/licensing requirements for data sources to avoid legal risk

Key Points:

- **Legal Compliance**: Understanding and adhering to the legal and licensing requirements of data sources is crucial to avoid legal risks. This involves reviewing licenses and ensuring that the use of data complies with the terms specified.
- **Dataset Licensing**: Evaluate the licensing information of datasets available in various platforms including Databricks Marketplace, consulting with legal teams to ensure that the intended usage is permitted.
- **Regular Audits**: Conducting regular audits of data usage and maintaining detailed records of data sources and their licenses can help in managing legal risks effectively.

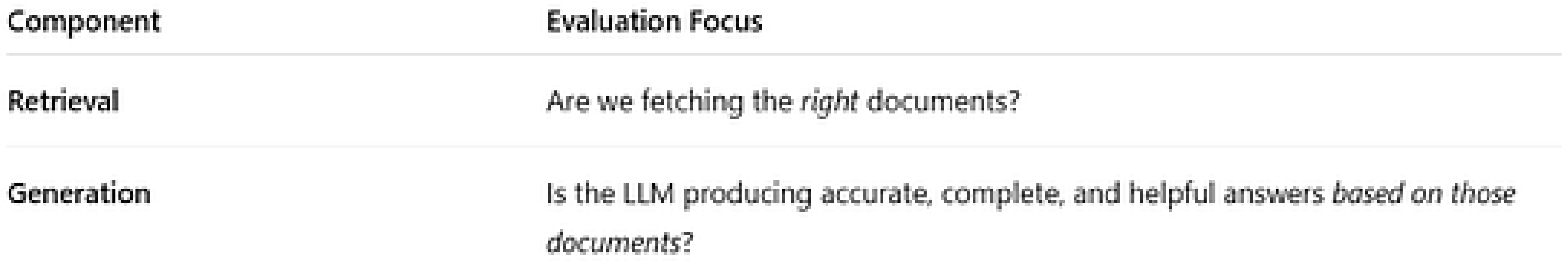
Section 6: Evaluation and Monitoring

RAG Evaluation


Build the evaluation dataset
02
Implement or improve the RAG pipeline
03
Evaluate pipeline
04
Identify promising improvements

What Are We Evaluating?

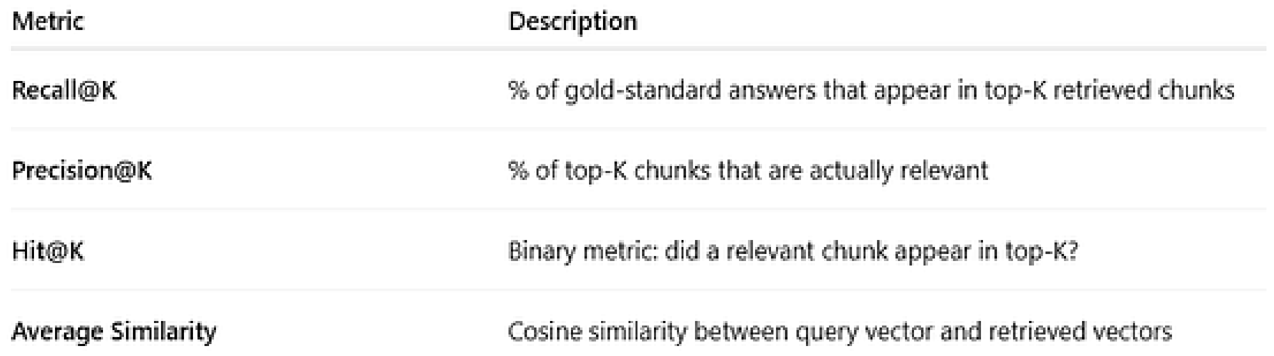
- A RAG application has **two critical sub-systems** to evaluate:



Component	Evaluation Focus
Retrieval	Are we fetching the *right* documents?
Generation	Is the LLM producing accurate, complete, and helpful answers *based on those documents?*

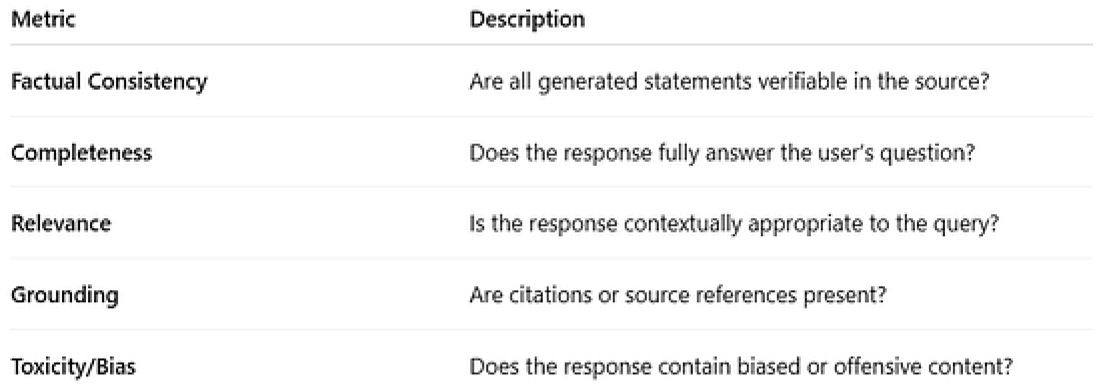
Core Evaluation Metrics

A. Retrieval Evaluation



Metric	Description
Recall@K	% of gold-standard answers that appear in top-K retrieved chunks
Precision@K	% of top-K chunks that are actually relevant
Hit@K	Binary metric: did a relevant chunk appear in top-K?
Average Similarity	Cosine similarity between query vector and retrieved vectors

B. Generation Evaluation



Metric	Description
Factual Consistency	Are all generated statements verifiable in the source?
Completeness	Does the response fully answer the user's question?
Relevance	Is the response contextually appropriate to the query?
Grounding	Are citations or source references present?
Toxicity/Bias	Does the response contain biased or offensive content?

Quantitative Evaluation Metrics

A. Using Synthetic Q&A Pairs

Create a dataset of known:

- Questions
- Expected answers
- Relevant document chunks

Then test the system for:

- Retrieval accuracy (was the expected document returned?)
- Answer correctness (was the final output within expectations?)

This enables **automated evaluation** without needing human labels every time.

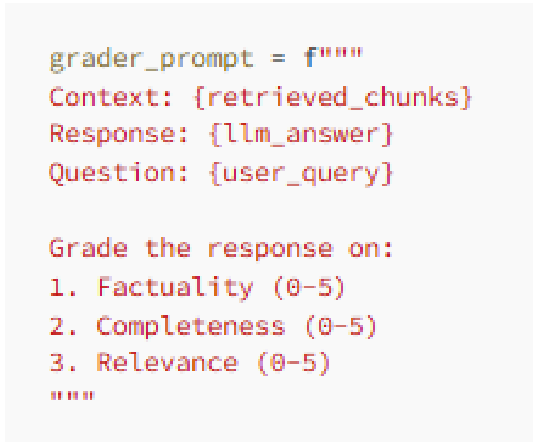
B. Embedding Similarity Check

- Use cosine similarity between:
- The query vector and the top-k chunk vectors
- The response and the retrieved chunks

Quantitative Evaluation Metrics

C. Automated LLM Grading

- Use an LLM like ChatDatabricks to **grade** another LLM's response:



```
grader_prompt = f"""
Context: {retrieved_chunks}
Response: {llm_answer}
Question: {user_query}

Grade the response on:
1. Factuality (0-5)
2. Completeness (0-5)
3. Relevance (0-5)
"""
```

4. Human-in-the-Loop Evaluation

Human evaluation is **slow but essential** for:

•New domains

•Regulated answers

•Edge-case queries

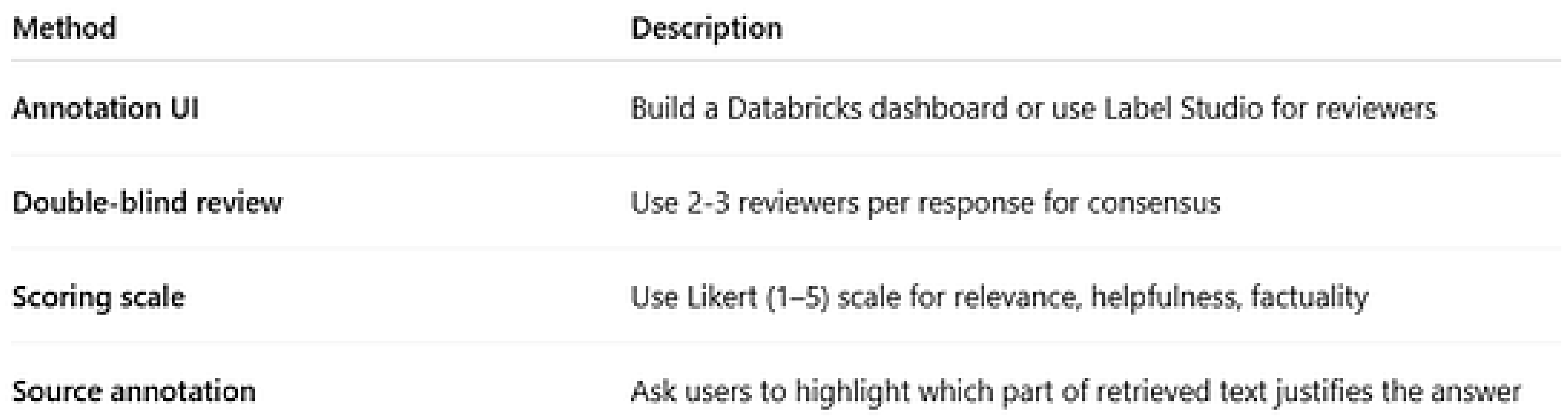
Techniques:

Human-in-the-Loop Evaluation

Human evaluation is **slow but essential** for:

- New domains
- Regulated answers
- Edge-case queries

Techniques:



Method	Description
Annotation UI	Build a Databricks dashboard or use Label Studio for reviewers
Double-blind review	Use 2-3 reviewers per response for consensus
Scoring scale	Use Likert (1–5) scale for relevance, helpfulness, factuality
Source annotation	Ask users to highlight which part of retrieved text justifies the answer

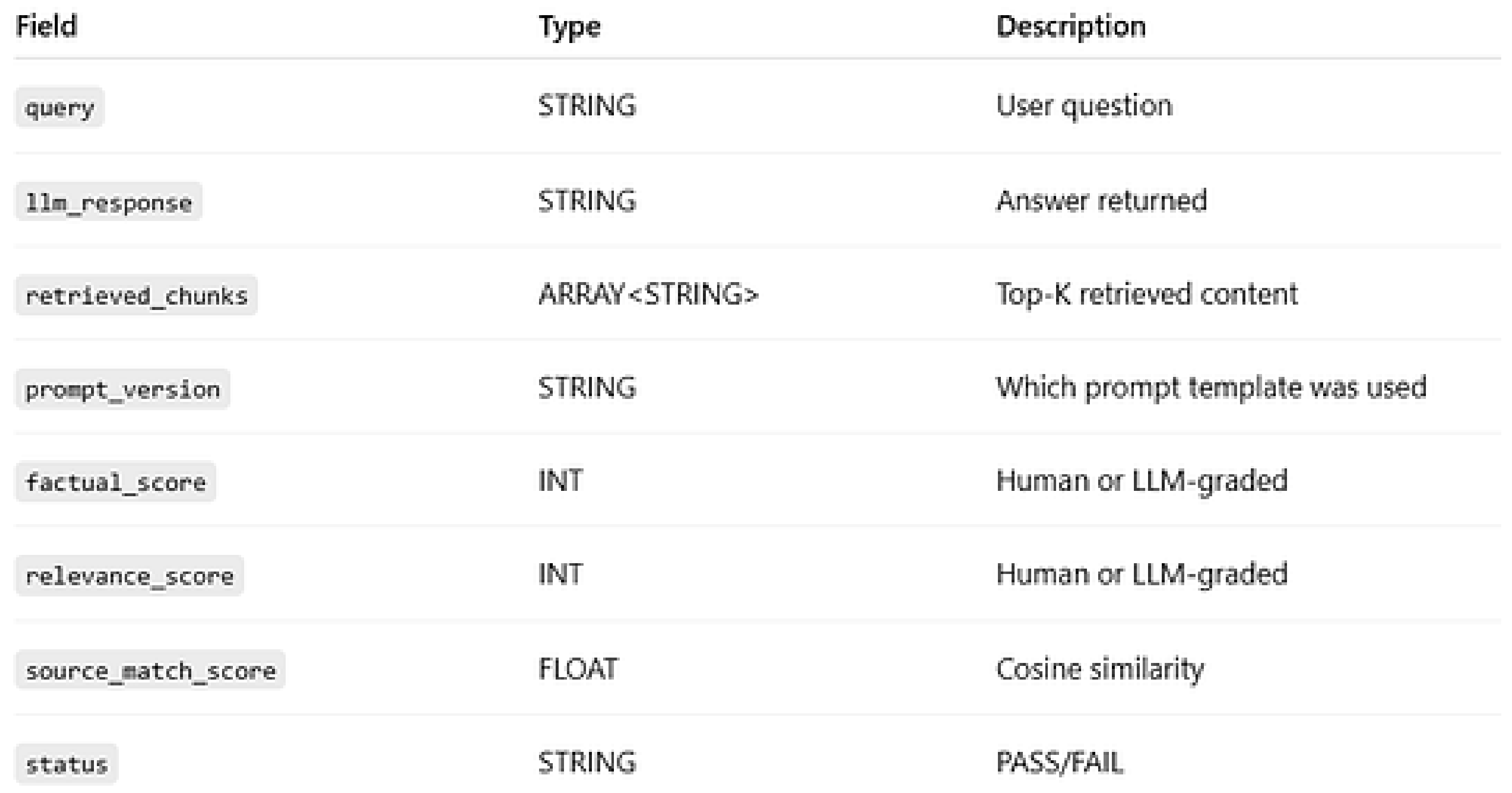
Building an Evaluation Pipeline on Databricks

Step-by-Step:

- **Log each RAG response** in a Delta
  Table: query, llm_response, retrieved_chunks, timestamp, prompt_version
- **Add synthetic ground truth** for internal tests:
  Store as expected_answer, expected_chunks
- **Compute metrics using PySpark:**
  Similarity, Recall@K, BLEU/ROUGE scores (optional)
- **Visualize metrics using Databricks SQL or dashboards:**
  Top failed queries, drift over time
- **Flag outliers for manual review:**
  Low-confidence responses → push to review UI

Building an Evaluation Pipeline on Databricks

Sample Evaluation Table Schema (Delta)



Field	Type	Description
`query`	STRING	User question
`llm_response`	STRING	Answer returned
`retrieved_chunks`	ARRAY<STRING>	Top-K retrieved content
`prompt_version`	STRING	Which prompt template was used
`factual_score`	INT	Human or LLM-graded
`relevance_score`	INT	Human or LLM-graded
`source_match_score`	FLOAT	Cosine similarity
`status`	STRING	PASS/FAIL

41. Select an LLM choice (size and architecture) based on a set of quantitative evaluation metrics

Key Points:

- Learn about various Quantitative Evaluation Metrics like Context Precision, Context Recall, Faithfulness, Answer Relevancy, Answer Correctness
- **Model Size and Architecture**: Evaluate the trade-offs between model size and architecture (e.g., smaller models for faster inference and lower costs vs. larger models for higher accuracy)
- **Performance vs. Cost**: Consider the computational resources required and the cost implications of deploying large models, especially in real-time applications VS Batch based LLM applications
- **LLMs as a Judge**: Best practices for using an LLM to evaluate another LLM:
    - Use Small Rubric Scales: Prefer scales like 1–3 or 1–5.
    - Provide Examples: Include diverse examples with detailed justifications for each score.
    - Use Additive Scales: Break evaluation into parts with additive scoring (e.g., 1 point for X, 1 for Y, 0 for Z = 2 points).
    - Use High-Token LLMs: More tokens allow for richer context in evaluations.
- **Offline VS Online Evaluation** — User response on LLM output; Detect Drifts; Learn about new **Databricks** Solutions.

42. Select key metrics to monitor for a specific LLM deployment scenario

Key Points:

- **Monitoring Metrics**: Key metrics include latency, throughput, accuracy, and resource utilization. For LLMs, additional metrics like context precision, relevancy, and faithfulness are important to ensure the quality of generated responses.
- **Real-time Monitoring**: Implement continuous monitoring of model performance using tools — MLflow to log and visualize these metrics, enabling proactive issue diagnosis and performance tuning.
- **Monitoring Solutions**: Like Alerts for automated emails and insights, metric calculations, and dashboard to maintain optimal performance and cost-efficiency.

43. Evaluate model performance in a RAG application using MLflow

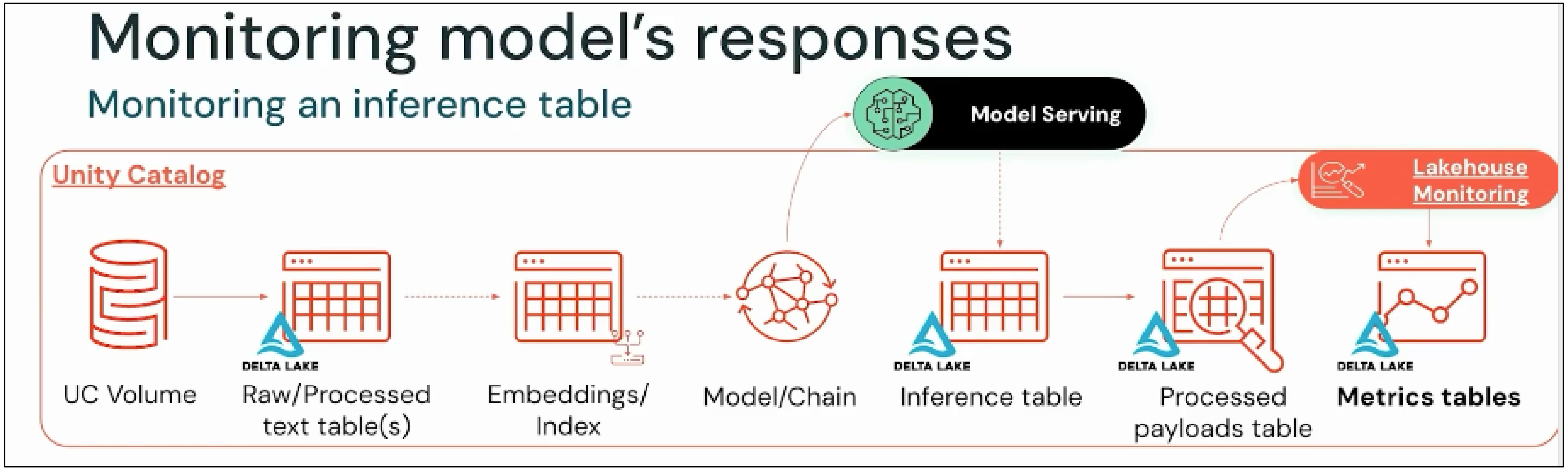
Key Points:

- **MLflow for RAG Evaluation**: MLflow supports the evaluation of RAG applications by tracking and comparing performance metrics across different model versions and configurations. MLflow to log parameters, metrics, and artifacts

44. Use inference logging to assess deployed RAG application performance

Key Points:

- **Inference Table Logging**: Capture detailed logs of inference requests and responses to analyze model behavior and performance in production. This includes logging input queries, retrieved contexts, and generated outputs.
- **Data Utilization**: Use logged data to compute key metrics such as latency, accuracy, and relevance, and to identify patterns or anomalies that may indicate issues with the deployed model.
- Log in **inference tables** to visualize performance trends and set up alerts for key metrics, ensuring continuous performance optimization. Automatic Logging — PII, Input expectation and rules.



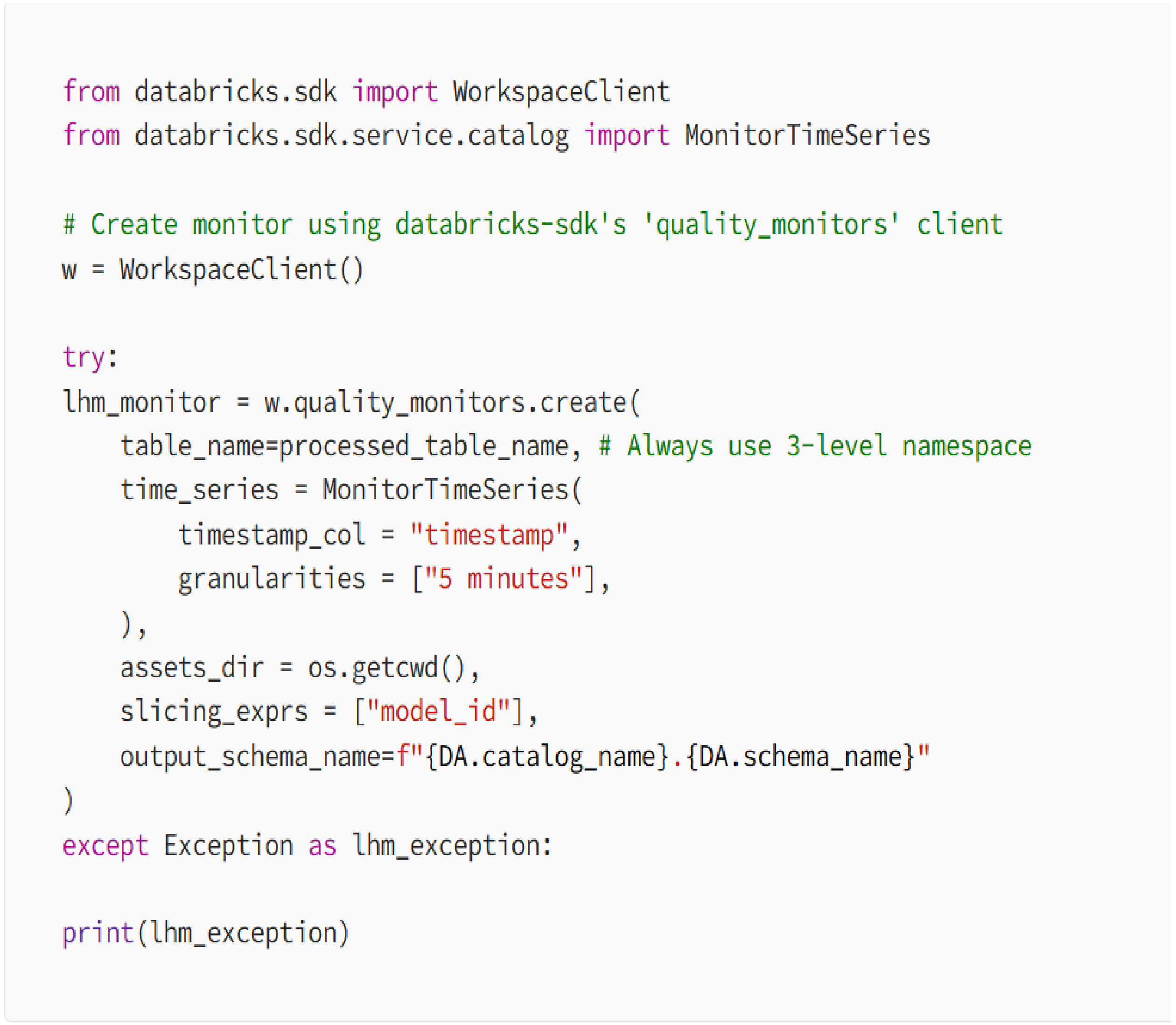
EXL

Monitoring(3 types):

1. **Time series**: It computes data quality metrics across time-based windows of the time series.
2. **InferenceLog**: This monitor is used for tables that contain the request log for a model. Each row in the table represents a request, with columns for the timestamp, the model inputs, the corresponding prediction, and optionally the ground-truth label. The monitor compares model performance and data quality metrics across time-based windows of the request log.
3. **Snapshot**: This type is used for all other types of tables. Monitoring calculates data quality metrics over all the data in the table, and the entire table is processed with every refresh.

Turn on Monitoring using:

1. Quality tab on UC Tables
2. Code:



```python
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.catalog import MonitorTimeSeries

# Create monitor using databricks-sdk's 'quality_monitors' client
w = WorkspaceClient()

try:
lhm_monitor = w.quality_monitors.create(
    table_name=processed_table_name, # Always use 3-level namespace
    time_series = MonitorTimeSeries(
        timestamp_col = "timestamp",
        granularities = ["5 minutes"],
    ),
    assets_dir = os.getcwd(),
    slicing_exprs = ["model_id"],
    output_schema_name=f"{DA.catalog_name}.{DA.schema_name}"
)
except Exception as lhm_exception:

print(lhm_exception)
```

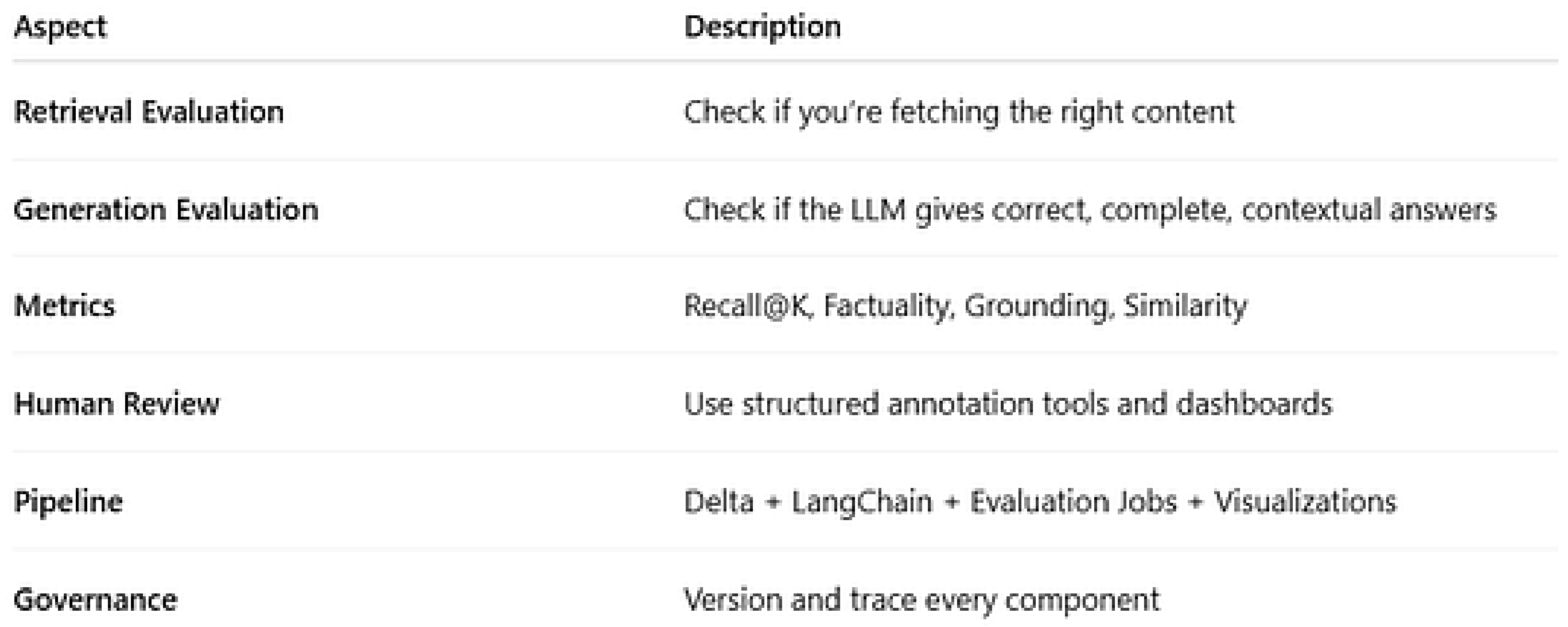
2. Code

EXL

45. Use Databricks features to control LLM costs for RAG applications

- **Cost Control Strategies**: model sizes; efficient data retrieval strategies — vector library like FAISS vs vector stores; batch processing for non-real-time tasks.
- Track resource usage
- **Resource Management**: Utilize features like auto-scaling and resource tagging to manage and optimize resource allocation dynamically

Summary



Aspect	Description
Retrieval Evaluation	Check if you're fetching the right content
Generation Evaluation	Check if the LLM gives correct, complete, contextual answers
Metrics	Recall@K, Factuality, Grounding, Similarity
Human Review	Use structured annotation tools and dashboards
Pipeline	Delta + LangChain + Evaluation Jobs + Visualizations
Governance	Version and trace every component

Thank you

EXL