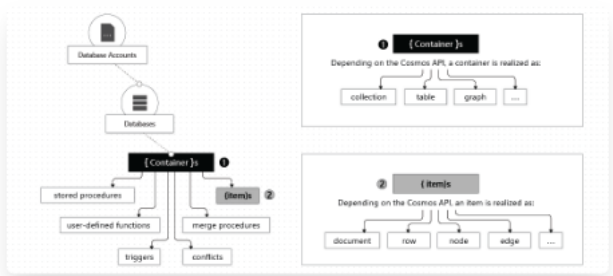
Hands-On Session



Build Scalable LLM Evaluation Pipeline

with Azure Cosmos DB

Featuring RAG Evaluation, Semantic Caching, and Scalable Architecture

PRESENTED BY

Farah Abdou

Agenda

Today's hands-on session roadmap

1. Introduction
2. Foundation Knowledge
3. System Architecture Deep Dive
4. Implementation Walkthrough
5. Hands-on Exercises
6. Advanced & Scaling Topics
7. Q&A and Next Steps

Session Objectives

By the end of this session, you will be able to:

- Understand LLM evaluation metrics and their significance in **production environments**
- Learn how to utilize Azure Cosmos DB for building **scalable evaluation pipelines**
- Explore multi-metric evaluation approaches using **RAGAS**, **ROUGE**, and semantic similarity
- Gain hands-on experience implementing a **real-world pipeline** with semantic caching and async processing

You'll implement these concepts through **hands-on code examples** throughout the session.

Prerequisites

What you'll need for today's hands-on session

Azure Resources

Azure subscription with access to:

- Azure OpenAI Service
- Azure Cosmos DB (NoSQL API)

Development Environment

Python 3.8+ environment with packages:

- azure-cosmos, openai, pandas, numpy, nltk, rouge_score

Code editor with sample code repository

- VS Code, PyCharm, or Jupyter notebooks

Prior Knowledge

Basic familiarity with:

- Python programming
- RAG (Retrieval-Augmented Generation) concepts
- LLMs and evaluation fundamentals

Introduction to LLM Evaluation

Why Evaluation Matters

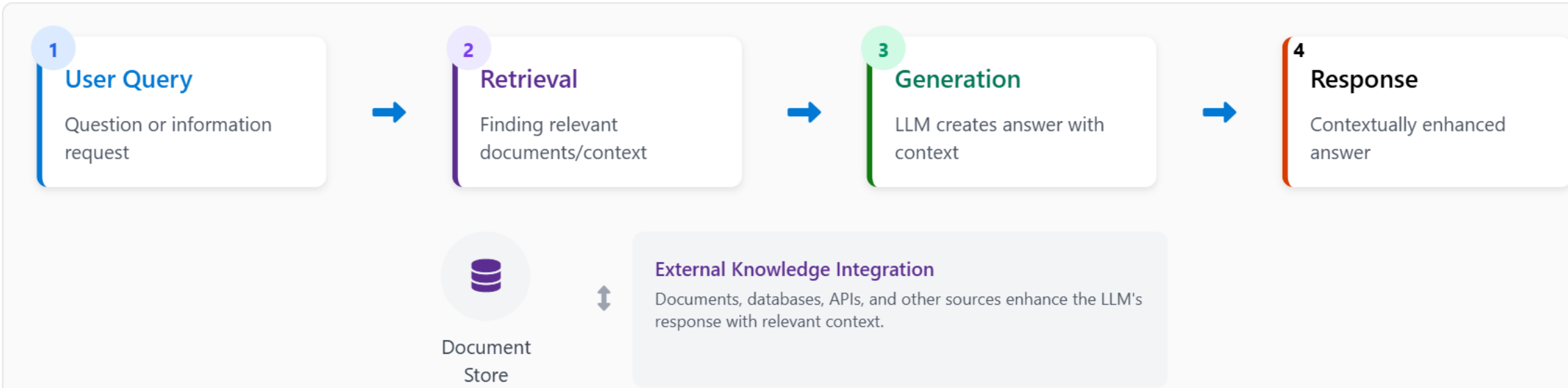
LLM outputs directly impact business decisions, customer experiences, and critical operations. Without rigorous evaluation, organizations risk exposing unreliable, biased, or hallucinated content to end users.

Key Evaluation Challenges

- **Stochasticity**: Different outputs for identical prompts make deterministic testing difficult
- **Hallucinations**: Detecting factual inaccuracies requires contextual verification
- **Multidimensional Quality**: Need to measure accuracy, relevance, coherence, and factuality simultaneously
- **Scale**: Production evaluation requires processing thousands of responses efficiently

Traditional metrics like BLEU or ROUGE alone are **insufficient** for RAG systems - we need specialized evaluation approaches.

Overview: Retrieval-Augmented Generation (RAG)



The RAG architecture combines retrieval of relevant information with generative capabilities of LLMs

Why RAG?

- Reduces hallucinations by grounding in facts
- Provides up-to-date information beyond training data
- Enables domain-specific knowledge without full retraining

Evaluation Needs

- Retrieval accuracy & relevance
- Answer faithfulness to retrieved context
- Response quality & relevance to query

RAG Evaluation Metrics: Why Go Beyond BLEU & ROUGE?

Traditional metrics fall short for RAG systems that require context-aware evaluation

Traditional Metrics

ROUGE & BLEU

Focus on **text similarity** but miss factual accuracy and context relevance

Perplexity

Measures **statistical likelihood** of text but not correctness or usefulness

Key Limitations

- No factual accuracy assessment
- Context utilization not measured
- Retrieval quality ignored

RAG-Specific Metrics

Faithfulness

Measures if responses are **grounded in retrieved context**, preventing hallucinations

Relevance Metrics

Evaluates both **answer relevancy** to query and **context relevancy** to information need

Context Quality

Measures **precision** (relevance of retrieved docs) and **recall** (coverage of relevant info)

Our pipeline uses RAGAS framework to measure these specialized metrics automatically at scale.

The RAGAS Evaluation Framework

RAGAS: Retrieval Augmented Generation Assessment

Query → Context → Answer

Context Relevancy

How relevant retrieved context is to the query

Context Precision

Proportion of relevant retrieved passages

Context Recall

Coverage of information needed for the answer

Faithfulness

Answer's grounding in the retrieved context

Answer Relevancy

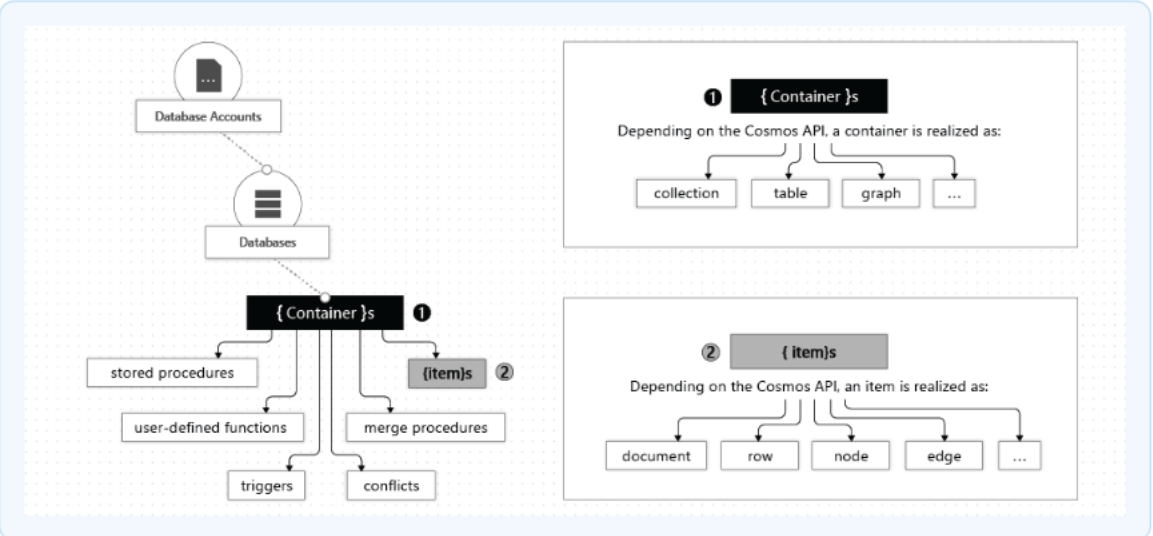
How well the answer addresses the query

RAGAS provides a comprehensive evaluation framework specifically designed for RAG systems

Introduction to Azure Cosmos DB

A fully managed, globally distributed NoSQL database service designed for scalable applications

- **Global Distribution**: Single-digit millisecond latency at the 99th percentile anywhere in the world
- **Multi-model**: Supports SQL, MongoDB, Cassandra, Gremlin, and Table APIs
- **Elastic Scalability**: Auto-scales throughput and storage on demand
- **SLA-backed**: 99.999% availability with comprehensive SLAs



Why Cosmos DB for ML & Evaluation Pipelines?

- Seamless handling of JSON documents for ML metrics
- Fast queries across large evaluation datasets
- Transactional batch operations for efficient data processing
- Built-in TTL support for semantic caching

Why Cosmos DB for LLM Evaluation?

Azure Cosmos DB provides critical capabilities for building scalable LLM evaluation pipelines:

- **Unlimited scalability** to handle millions of evaluation records and multi-TB embeddings with automatic scaling
- **Semantic caching** with TTL support reduces embedding API costs and improves response times
- **Global distribution** for low-latency evaluation across regions with multi-master replication
- **Transactional batch operations** for efficient processing of evaluation workloads at scale
- **Time-series analytics** for tracking evaluation metrics over time and across model versions

> 💡 Our implementation uses **dedicated containers** for evaluations and semantic cache with optimized partition keys.

Semantic Caching with Cosmos DB

Why caching matters for LLM evaluation

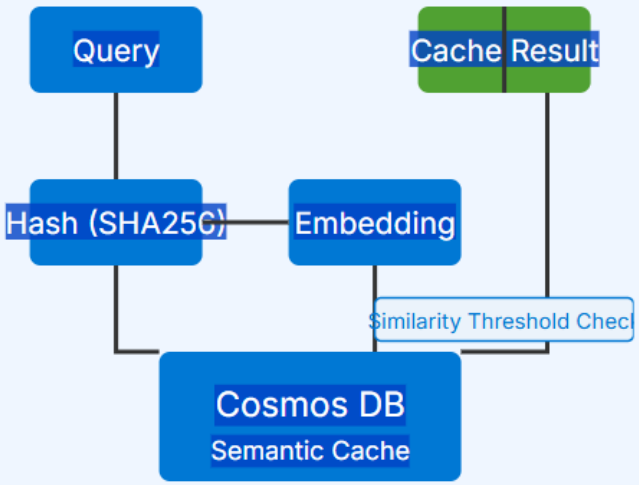
Reduces redundant API calls, lowers costs, and dramatically improves response times for similar evaluation queries

Vector search vs. traditional key-value

Traditional cache requires exact matches; semantic cache uses vector similarity to find close-enough results

SHA256 + vector similarity reduces costs

Fast hash lookups combined with similarity thresholds offer both speed and semantic flexibility



System Design: Our Scalable Evaluation Pipeline

1 **Query & Context**
RAG system input & retrieved passages

2 **Embedding**
Azure OpenAI with semantic cache

3 **Evaluation**
Multi-metric assessment

4 **Storage**
Cosmos DB persistence

Azure OpenAI
- Text embeddings for semantic analysis
- GPT models for answer generation
- Cached embeddings to reduce API calls

Evaluation Engine
- RAGAS metrics (faithfulness, relevancy)
- ROUGE scores for text comparison
- Semantic similarity for context/answer

Azure Cosmos DB
- Evaluation results container
- Semantic cache container
- Partitioned by query_id & cache_id

Reporting & Export
- Comprehensive summary reports
- Metrics aggregation & analysis
- CSV export for further processing

Async Batch Processing
Efficiently process multiple evaluation requests in parallel with error handling and backoff strategies

End-to-End architecture for scalable LLM evaluation with Azure Cosmos DB for persistence and semantic caching

Key Components Overview

Core classes and their responsibilities in our evaluation pipeline:

EvaluationConfig

Centralized configuration dataclass for all settings: Azure OpenAI credentials, Cosmos DB endpoints, model parameters, cache settings, and evaluation thresholds

ModernRAGEvaluator

Main orchestrator class that manages metrics calculation, semantic caching, database interactions, batch processing, and comprehensive reporting

Cosmos DB Containers

Two container types: `evaluations` for storing assessment results and `semantic_cache` for efficient embedding storage and retrieval

Evaluation Metrics and RAGAS

Comprehensive Metric Suite

Faithfulness

Measures how factually consistent the generated answer is with the retrieved context. Calculated using embedding similarity between answer and context vectors.

Answer Relevancy

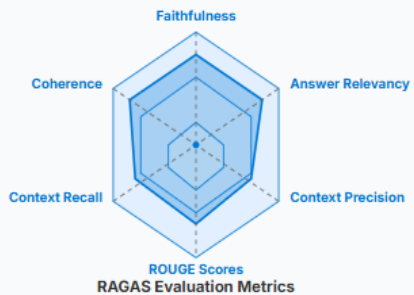
Evaluates how well the answer addresses the user's question. Computed as semantic similarity between query and answer embeddings.

Context Precision & Recall

Precision measures how many retrieved documents are relevant. Recall evaluates if all necessary context was retrieved for answering.

ROUGE Scores

Text similarity metrics comparing generated answers with expected answers. Includes ROUGE-1 (unigrams), ROUGE-2 (bigrams), and ROUGE-L (longest common subsequence).



Comprehensive evaluation combines multiple metrics
Higher scores in all dimensions indicate better RAG performance

Component Deep Dive: EvaluationConfig

Central configuration dataclass that powers the entire evaluation pipeline

```python
@dataclass
class EvaluationConfig:
    # Azure Cosmos DB Configuration
    cosmos_endpoint: str
    cosmos_key: str
    database_name: str = "llm_evaluation"
    container_name: str = "evaluations"
    cache_container: str = "semantic_cache"

    # Azure OpenAI Configuration
    azure_openai_endpoint: str
    azure_openai_api_key: str
    azure_openai_api_version: str
    embedding_deployment_name: str

    # LLM Parameters
    llm_deployment_name: str
    llm_temperature: float
    llm_top_p: float
    llm_max_tokens: int

    # Pipeline Configuration
    similarity_threshold: float = 0.85
    cache_ttl_hours: int = 24
    batch_size: int = 10
    chunk_size: int = 512
    chunk_overlap: int = 50
```

Key Parameter Groups

Cosmos DB Configuration

Database connection and container settings for evaluation storage and semantic caching

Azure OpenAI Configuration

API endpoints, credentials, and model deployment names for embedding generation

LLM Parameters

Model selection and generation settings that affect output quality and consistency

Pipeline Configuration

Performance tuning, batch processing, and caching settings for scalability

💡 The `EvaluationConfig` dataclass centralizes all settings, making it easy to modify behavior without changing code.

Component Deep Dive: ModernRAGEvaluator Class

Main orchestrator for evaluation with multiple metrics and async batch processing

```python
class ModernRAGEvaluator:
    def __init__(self, config: EvaluationConfig):
        self.config = config
        self.azure_client = AzureOpenAI(...)
        self.cosmos_client = CosmosClient(...)
        self._setup_evaluation_database()
        self.semantic_cache_container = self._setup_semantic_cache()

    async def evaluate_rag_response(self, query_id,
                                    query, retrieved_context,
                                    generated_answer,
                                    expected_answer=None):
        retrieval_metrics = self.calculate_retrieval_metrics(...)
        generation_metrics = self.calculate_generation_metrics(...)
        ragas_metrics = self.calculate_ragas_metrics(...)
        rouge_scores = self.calculate_rouge_scores(...)

        result = EvaluationResult(...)
        await self._store_evaluation_result(result)
        return result

    async def batch_evaluate(self, evaluation_data: List[Dict]):
        results = []
        for idx, item in enumerate(evaluation_data):
            try:
                logger.info(f"Evaluating item {idx + 1}/{len(evaluation_data)}")
                result = await self.evaluate_rag_response(...)
                if result:
                    results.append(result)
            except Exception as e:
                logger.error(f"Error evaluating item {idx}: {e}")
        return results
```

Key Evaluation Methods

Retrieval Evaluation

Evaluates how effectively documents are retrieved using semantic similarity and precision/recall metrics

Generation Evaluation

Assesses LLM-generated answers for relevance to query, coherence, and length metrics

RAGAS Metrics Calculation

Provides specialized RAG metrics: faithfulness, answer relevancy, context precision/recall

Async Batch Processing

Enables parallel evaluation of multiple queries with error isolation and robust logging

⚡ The `async` methods enable high throughput by processing multiple evaluations concurrently while handling errors gracefully.

Best Practices & Troubleshooting

Best Practices

Secure Configuration Management
Store API keys in Azure Key Vault and use managed identities

Optimize Cosmos DB Throughput
Configure appropriate RU/s based on evaluation volume and adjust partition keys

Monitor Cache Performance
Track hit/miss ratios and adjust similarity thresholds accordingly

Version Evaluation Results
Track model versions and evaluation metrics over time for A/B testing

Troubleshooting

Cosmos DB Throttling
Implement backoff strategies and increase RU/s allocation if receiving 429 errors

Slow Batch Processing
Adjust batch sizes and add more async workers for parallel processing

Memory Pressure
Process large datasets in smaller chunks and implement proper garbage collection

API Key Rate Limits
Implement token bucket rate limiting and monitor usage quotas for Azure OpenAI

Key Takeaway
Monitor your evaluation pipeline regularly and implement proper error handling and logging for production environments. Use structured logging to track performance metrics and establish baselines for comparison.

Thanks !