**RAG Evaluation Dashboard**

**Overview**

The **RAG Evaluation Dashboard** is a Streamlit-based web application designed to evaluate Retrieval-Augmented Generation (RAG) systems. It allows users to test RAG performance by generating answers to predefined questions, evaluate them using the RAGAS framework, and visualize results through interactive charts and metrics. Built with AWS Bedrock integration, it supports models like Claude-3.5-Sonnet and Amazon-Nova-Pro-v1, with a focus on financial compliance scenarios (e.g., CCAR regulations).

**Key Features**

* **Run RAG Tests**: Upload test questions in CSV format, generate answers using a selected model, and save results.
* **Evaluate RAG**: Assess generated answers using RAGAS metrics (faithfulness, answer relevancy, etc.), with fallbacks to LLM scoring and text similarity.
* **View Results**: Visualize performance via gauges, histograms, radar charts, and per-question analysis, with filtering and correlation insights.
* **Model Support**: Integrates with AWS Bedrock models, configurable per program.
* **Guardrails**: Ensures security by validating prompts and responses for PII or malicious content.
* **Customizable**: Supports program-specific configurations (e.g., CCAR) via a JSON config file.

**Prerequisites**

* **Python**: Version 3.8 or higher.
* **AWS Credentials**: Configured for Bedrock and S3 access (e.g., via aws configure or environment variables).
* **Dependencies**: Listed in requirements.txt (see Installation).

**Installation**

1. **Clone the Repository**:

bash

CollapseWrapCopy

git clone https://github.com/your-repo/rag-evaluation-dashboard.git

cd rag-evaluation-dashboard

1. **Set Up a Virtual Environment** (optional but recommended):

bash

CollapseWrapCopy

python -m venv venv

source venv/bin/activate *# On Windows: venv\Scripts\activate*

1. **Install Dependencies**: Create a requirements.txt file with the following content:

text

CollapseWrapCopy

streamlit

pandas

plotly

boto3

langchain-aws

datasets

ragas

numpy

scikit-learn

Then install:

bash

CollapseWrapCopy

pip install -r requirements.txt

1. **Configure AWS**:
   * Ensure AWS credentials are set up with permissions for Bedrock (bedrock-runtime) and S3.
   * Example ~/.aws/credentials:

text

CollapseWrapCopy

[default]

aws\_access\_key\_id = YOUR\_ACCESS\_KEY

aws\_secret\_access\_key = YOUR\_SECRET\_KEY

region = us-east-1

1. **Set Up Configuration**:
   * Create a config directory in the project root.
   * Add a common\_config.json file with your settings (example below):

json

CollapseWrapCopy

{

"s3\_config": {

"bucket\_name": "your-s3-bucket"

},

"knowledge\_base\_config": {

"index\_name": "your-index"

},

"model\_config": {

"region": "us-east-1"

},

"programs": {

"ccar": {

"name": "CCAR Compliance",

"prompt\_template\_file": "ccar\_prompt.txt",

"models": {

"allowed": ["Claude-3.5-Sonnet", "Amazon-Nova-Pro-v1"],

"default": "Claude-3.5-Sonnet"

}

}

}

}

* + Adjust paths and settings as needed.

1. **Directory Structure**: Ensure the src/services directory exists with the required service files (s3\_service.py, knowledge\_base\_service.py, bedrock\_service.py, guardrails\_service.py).

**Running the Application**

1. **Start the App**:

bash

CollapseWrapCopy

streamlit run app.py

* + Replace app.py with the filename of your script if different.

1. **Access the Dashboard**:
   * Open your browser to http://localhost:8501.

**Usage**

**1. Run RAG Test**

* **Navigate**: Select "Run RAG Test" from the sidebar.
* **Select Program**: Choose a program (e.g., "CCAR Compliance") and model (e.g., Claude-3.5-Sonnet).
* **Upload CSV**: Provide a CSV file with columns: Question ID, Question, Context, Ground Truth.
  + Example:

text

CollapseWrapCopy

Question ID,Question,Context,Ground Truth

Q1,What are the 2023 CCAR capital requirements?,In 2023, the Federal Reserve updated CCAR rules, requiring Tier 1 capital ratios of at least 4.5% for all banks.,The latest CCAR capital requirements, updated in 2023, mandate a minimum Tier 1 capital ratio of 4.5% for all banks.

* **Run Test**: Click "Run RAG Test" to generate answers and download results as a CSV.

**2. Evaluate RAG**

* **Navigate**: Select "Evaluate RAG" from the sidebar.
* **Select Results**: Choose a saved test result file (e.g., question\_answer\_YYYYMMDD\_HHMMSS.csv).
* **Evaluate**: Click "Evaluate with RAGAS" to compute metrics. Results are saved (e.g., RAG\_Results\_YYYYMMDD\_HHMMSS.csv).

**3. View Results**

* **Navigate**: Select "View RAG Results" from the sidebar.
* **Select Evaluation**: Pick an evaluation result file.
* **Analyze**:
  + **Gauges**: View average scores for each metric and overall performance.
  + **Radar Chart**: Compare metric averages.
  + **Histograms**: See score distributions.
  + **Per-Question**: Dive into individual question details.
  + **Correlation**: Explore metric relationships.
  + **Filter**: Apply score thresholds and download filtered results.

**File Structure**

text

CollapseWrapCopy

rag-evaluation-dashboard/

├── app.py # Main application script

├── config/ # Configuration directory

│ └── common\_config.json # Common configuration file

├── src/ # Source directory for custom services

│ └── services/

│ ├── s3\_service.py # S3 interaction logic

│ ├── knowledge\_base\_service.py # Knowledge base management

│ ├── bedrock\_service.py # Bedrock model integration

│ └── guardrails\_service.py # Security and validation

├── rag\_evaluation\_files/ # Auto-generated directory for test and evaluation CSVs

├── requirements.txt # Dependency list

└── README.md # This file

**Key Components**

* **RAG Pipeline**:
  + **Retrieval**: KnowledgeBaseService fetches context from S3 using S3Service.
  + **Generation**: BedrockService integrates with AWS Bedrock models.
  + **Security**: GuardrailsService validates inputs/outputs.
* **Evaluation**:
  + **RAGAS**: Primary framework with metrics like faithfulness, answer\_relevancy, etc.
  + **Fallbacks**: LLM scoring (\_evaluate\_with\_llm) and text similarity (\_fallback\_scoring) ensure robustness.
* **Visualization**:
  + Built with Plotly for interactive charts (gauges, histograms, radar, correlation matrix).

**Customization**

* **Add Programs**: Extend common\_config.json with new programs and their settings.
* **Model Options**: Modify allowed/default models in the config.
* **Prompt Templates**: Define custom templates in prompt\_template\_file for each program.

**Troubleshooting**

* **Import Errors**: Ensure src/services files are present and correctly implemented.
* **AWS Issues**: Verify credentials and Bedrock/S3 permissions.
* **File Not Found**: Check common\_config.json exists in config/.
* **Evaluation Fails**: Review logs for RAGAS/LLM errors; ensure test CSV format is correct.

----------------------------------------------

**RAG Evaluation Dashboard**

**Overview**

The **RAG Evaluation Dashboard** is a Streamlit-based web application designed to evaluate Retrieval-Augmented Generation (RAG) systems. It allows users to test RAG performance by generating answers to predefined questions, evaluate them using the RAGAS framework, and visualize results through interactive charts and metrics. Built with AWS Bedrock integration, it supports models like Claude-3.5-Sonnet and Amazon-Nova-Pro-v1, with a focus on financial compliance scenarios (e.g., CCAR regulations).

**Key Features**

* **Run RAG Tests**: Upload test questions in CSV format, generate answers using a selected model, and save results.
* **Evaluate RAG**: Assess generated answers using RAGAS metrics (faithfulness, answer relevancy, etc.), with fallbacks to LLM scoring and text similarity.
* **View Results**: Visualize performance via gauges, histograms, radar charts, and per-question analysis, with filtering and correlation insights.
* **Model Support**: Integrates with AWS Bedrock models, configurable per program.
* **Guardrails**: Ensures security by validating prompts and responses for PII or malicious content.
* **Customizable**: Supports program-specific configurations (e.g., CCAR) via a JSON config file.

**Prerequisites**

* **Python**: Version 3.8 or higher.
* **AWS Credentials**: Configured for Bedrock and S3 access (e.g., via aws configure or environment variables).
* **Dependencies**: Listed in requirements.txt (see Installation).

**Installation**

1. **Clone the Repository**:

bash

CollapseWrapCopy

git clone https://github.com/your-repo/rag-evaluation-dashboard.git

cd rag-evaluation-dashboard

1. **Set Up a Virtual Environment** (optional but recommended):

bash

CollapseWrapCopy

python -m venv venv

source venv/bin/activate *# On Windows: venv\Scripts\activate*

1. **Install Dependencies**: Create a requirements.txt file with the following content:

text

CollapseWrapCopy

streamlit

pandas

plotly

boto3

langchain-aws

datasets

ragas

numpy

scikit-learn

Then install:

bash

CollapseWrapCopy

pip install -r requirements.txt

1. **Configure AWS**:
   * Ensure AWS credentials are set up with permissions for Bedrock (bedrock-runtime) and S3.
   * Example ~/.aws/credentials:

text

CollapseWrapCopy

[default]

aws\_access\_key\_id = YOUR\_ACCESS\_KEY

aws\_secret\_access\_key = YOUR\_SECRET\_KEY

region = us-east-1

1. **Set Up Configuration**:
   * Create a config directory in the project root.
   * Add a common\_config.json file with your settings (example below):

json

CollapseWrapCopy

{

"s3\_config": {

"bucket\_name": "your-s3-bucket"

},

"knowledge\_base\_config": {

"index\_name": "your-index"

},

"model\_config": {

"region": "us-east-1"

},

"programs": {

"ccar": {

"name": "CCAR Compliance",

"prompt\_template\_file": "ccar\_prompt.txt",

"models": {

"allowed": ["Claude-3.5-Sonnet", "Amazon-Nova-Pro-v1"],

"default": "Claude-3.5-Sonnet"

}

}

}

}

* + Adjust paths and settings as needed.

1. **Directory Structure**: Ensure the src/services directory exists with the required service files (s3\_service.py, knowledge\_base\_service.py, bedrock\_service.py, guardrails\_service.py).

**Running the Application**

1. **Start the App**:

bash

CollapseWrapCopy

streamlit run app.py

* + Replace app.py with the filename of your script if different.

1. **Access the Dashboard**:
   * Open your browser to http://localhost:8501.

**Usage**

**1. Run RAG Test**

* **Navigate**: Select "Run RAG Test" from the sidebar.
* **Select Program**: Choose a program (e.g., "CCAR Compliance") and model (e.g., Claude-3.5-Sonnet).
* **Upload CSV**: Provide a CSV file with columns: Question ID, Question, Context, Ground Truth.
  + Example:

text

CollapseWrapCopy

Question ID,Question,Context,Ground Truth

Q1,What are the 2023 CCAR capital requirements?,In 2023, the Federal Reserve updated CCAR rules, requiring Tier 1 capital ratios of at least 4.5% for all banks.,The latest CCAR capital requirements, updated in 2023, mandate a minimum Tier 1 capital ratio of 4.5% for all banks.

* **Run Test**: Click "Run RAG Test" to generate answers and download results as a CSV.

**2. Evaluate RAG**

* **Navigate**: Select "Evaluate RAG" from the sidebar.
* **Select Results**: Choose a saved test result file (e.g., question\_answer\_YYYYMMDD\_HHMMSS.csv).
* **Evaluate**: Click "Evaluate with RAGAS" to compute metrics. Results are saved (e.g., RAG\_Results\_YYYYMMDD\_HHMMSS.csv).

**3. View Results**

* **Navigate**: Select "View RAG Results" from the sidebar.
* **Select Evaluation**: Pick an evaluation result file.
* **Analyze**:
  + **Gauges**: View average scores for each metric and overall performance.
  + **Radar Chart**: Compare metric averages.
  + **Histograms**: See score distributions.
  + **Per-Question**: Dive into individual question details.
  + **Correlation**: Explore metric relationships.
  + **Filter**: Apply score thresholds and download filtered results.

**File Structure**

text

CollapseWrapCopy

rag-evaluation-dashboard/

├── app.py # Main application script

├── config/ # Configuration directory

│ └── common\_config.json # Common configuration file

├── src/ # Source directory for custom services

│ └── services/

│ ├── s3\_service.py # S3 interaction logic

│ ├── knowledge\_base\_service.py # Knowledge base management

│ ├── bedrock\_service.py # Bedrock model integration

│ └── guardrails\_service.py # Security and validation

├── rag\_evaluation\_files/ # Auto-generated directory for test and evaluation CSVs

├── requirements.txt # Dependency list

└── README.md # This file

**Key Components**

* **RAG Pipeline**:
  + **Retrieval**: KnowledgeBaseService fetches context from S3 using S3Service.
  + **Generation**: BedrockService integrates with AWS Bedrock models.
  + **Security**: GuardrailsService validates inputs/outputs.
* **Evaluation**:
  + **RAGAS**: Primary framework with metrics like faithfulness, answer\_relevancy, etc.
  + **Fallbacks**: LLM scoring (\_evaluate\_with\_llm) and text similarity (\_fallback\_scoring) ensure robustness.
* **Visualization**:
  + Built with Plotly for interactive charts (gauges, histograms, radar, correlation matrix).

**Customization**

* **Add Programs**: Extend common\_config.json with new programs and their settings.
* **Model Options**: Modify allowed/default models in the config.
* **Prompt Templates**: Define custom templates in prompt\_template\_file for each program.

**Troubleshooting**

* **Import Errors**: Ensure src/services files are present and correctly implemented.
* **AWS Issues**: Verify credentials and Bedrock/S3 permissions.
* **File Not Found**: Check common\_config.json exists in config/.
* **Evaluation Fails**: Review logs for RAGAS/LLM errors; ensure test CSV format is correct.

**RAG Evaluation Approaches in RAGEvaluationApp**

The RAGEvaluationApp implements a multi-tiered evaluation framework to assess Retrieval-Augmented Generation (RAG) systems, leveraging three distinct approaches: **RAGAS**, **LLM-based evaluation**, and **ML-based fallback scoring**. These methods work in a cascading hierarchy—RAGAS as the primary evaluator, followed by LLM scoring, and finally ML-based scoring as a fallback—to ensure robust and reliable assessment of RAG performance. This document explains each approach in detail, including their mechanics, implementation in the code, and practical implications.

**1. RAGAS Evaluation**

**Overview**

RAGAS (Retrieval-Augmented Generation Assessment Suite) is a specialized framework for evaluating RAG systems by measuring both retrieval and generation quality. It provides seven key metrics, each scored from 0 to 1, where 1 represents optimal performance. These metrics assess various dimensions of RAG output, from factual consistency to semantic alignment with ground truth.

**Metrics**

1. **Faithfulness**: Measures how factually consistent the generated answer is with the retrieved context.
2. **Answer Relevancy**: Evaluates how well the answer addresses the question.
3. **Context Precision**: Assesses the relevance of retrieved context to the question.
4. **Context Recall**: Checks if the retrieved context contains all necessary information for the ground truth.
5. **Context Entity Recall**: Focuses on the presence of key entities from the ground truth in the context.
6. **Answer Similarity**: Compares the semantic meaning of the answer to the ground truth.
7. **Answer Correctness**: Evaluates factual accuracy and completeness against the ground truth.

**Implementation in Code**

* **Method**: \_evaluate\_with\_ragas
* **Process**:
  1. **Setup**: Configures an AWS Bedrock client with a 120-second timeout and no retries, using ChatBedrock (Nova model) for generation and BedrockEmbeddings (Titan model) for embeddings.
  2. **Data Preparation**: Converts input data (question, context, ground truth, answer) into a Hugging Face Dataset object.
  3. **Evaluation**: Calls ragas.evaluate with the dataset, metrics list, LLM, and embeddings. Results are converted to a pandas DataFrame and mapped to a dictionary with metric scores.
  4. **Fallback**: If any input is empty, returns default scores of 0.5 for all metrics.
* **Key Code**:

python

CollapseWrapCopy

bedrock\_client = boto3.client('bedrock-runtime', config=bedrock\_config)

llm = ChatBedrock(model\_id="amazon.nova-pro-v1:0", client=bedrock\_client)

embeddings = BedrockEmbeddings(model\_id="amazon.titan-embed-text-v2:0", client=bedrock\_client)

data = {"question": [question], "contexts": [[str(context)]], "ground\_truth": [ground\_truth], "answer": [answer]}

hf\_dataset = HFDataset.from\_dict(data)

result = evaluate(dataset=hf\_dataset, metrics=metrics, llm=llm, embeddings=embeddings)

**Mechanics**

* **Faithfulness**: Breaks the answer into statements, verifies each against the context using the LLM, and computes the supported proportion.
* **Answer Relevancy**: Uses embeddings to measure semantic similarity between question and answer, penalizing irrelevance.
* **Context Precision/Recall**: Analyzes context relevance and completeness via LLM reasoning and embeddings.
* **Entity Recall**: Extracts entities from ground truth and checks their presence in context.
* **Similarity/Correctness**: Computes cosine similarity of embeddings and assesses factual alignment.

**Strengths**

* **Comprehensive**: Covers both retrieval (context metrics) and generation (answer metrics).
* **Standardized**: Built on a research-backed framework, ensuring consistency.
* **Granular**: Provides detailed, metric-specific insights.

**Weaknesses**

* **Dependency**: Relies on LLM and embedding quality, which can fail with complex inputs.
* **Resource-Intensive**: Requires API calls to Bedrock, potentially slow or costly.
* **Fragility**: May raise exceptions (e.g., timeouts), necessitating fallbacks.

**Role in the App**

RAGAS is the primary evaluation method, invoked first in \_render\_evaluate\_rag\_page. It offers a robust baseline for scoring RAG performance, with results visualized in the "View Results" page.

**2. LLM-Based Evaluation**

**Overview**

When RAGAS fails (e.g., due to API errors or timeouts), the app falls back to an LLM-based evaluation approach. This method uses an LLM (Amazon-Nova-Pro-v1) to directly score each RAGAS metric by prompting it with specific questions about the input data.

**Metrics**

The same seven RAGAS metrics are evaluated, but scores are derived from LLM responses rather than RAGAS’s internal logic.

**Implementation in Code**

* **Method**: \_evaluate\_with\_llm
* **Process**:
  1. **Setup**: Configures a Bedrock client and instantiates ChatBedrock with the Nova model.
  2. **Prompt Design**: Defines a prompt for each metric, asking the LLM to provide a score from 0 to 1. Example:
     + Faithfulness: "On a scale from 0 to 1, how factually consistent is the answer with the context?"
  3. **Execution**: Invokes the LLM for each prompt, extracts a numeric score from the response using \_extract\_score.
  4. **Fallback**: Returns 0.5 for any metric if the LLM fails or inputs are empty.
* **Key Code**:

python

CollapseWrapCopy

prompts = {

"faithfulness": f"Question: {question}\nContext: {context\_text}\nAnswer: {answer}\nOn a scale from 0 to 1, how factually consistent is the answer with the context? Provide a single number between 0 and 1.",

*# ... other prompts ...*

}

for metric, prompt in prompts.items():

response = llm.invoke(prompt)

scores[metric] = self.\_extract\_score(response.content)

**Mechanics**

* **Prompting**: The LLM interprets the question, context, ground truth, and answer, then assigns a score based on its reasoning.
* **Score Extraction**: Uses regex (re.findall) to find numbers in the response, clamping them to [0, 1]. Defaults to 0.5 if no valid number is found.
* **Context Handling**: Truncates context to 4000 characters to avoid exceeding LLM token limits.

**Strengths**

* **Flexibility**: Adapts to any input without requiring a specialized framework like RAGAS.
* **Resilience**: Works even if RAGAS fails, leveraging the LLM’s general reasoning capabilities.
* **Directness**: Provides a model-driven perspective, potentially capturing nuances missed by automated metrics.

**Weaknesses**

* **Subjectivity**: Scores depend on the LLM’s interpretation, which may vary or be inconsistent.
* **Prompt Sensitivity**: Poorly worded prompts or ambiguous responses can skew results.
* **Resource Use**: Still relies on Bedrock API calls, though less complex than RAGAS.

**Role in the App**

LLM-based evaluation serves as the first fallback in \_render\_evaluate\_rag\_page, triggered if RAGAS raises an exception. It ensures evaluation continuity and provides a secondary layer of insight into model performance.

**3. ML-Based Fallback Scoring**

**Overview**

As a final fallback, the app employs an ML-based approach using text similarity techniques (Jaccard similarity and TF-IDF with cosine similarity) when both RAGAS and LLM evaluation fail. This method is lightweight and deterministic, relying on statistical text analysis rather than LLM reasoning.

**Metrics**

The same seven RAGAS metrics are approximated using similarity scores between pairs of text inputs (question, context, ground truth, answer).

**Implementation in Code**

* **Method**: \_fallback\_scoring
* **Process**:
  1. **Input Validation**: Converts all inputs to strings, handling None values, and returns 0.5 scores if all are empty.
  2. **Jaccard Similarity**:
     + Computes word overlap between two strings (e.g., answer vs. context) as a ratio of intersection to union.
     + Used as a standalone metric if TF-IDF fails or fewer than two valid texts are available.
  3. **TF-IDF and Cosine Similarity**:
     + Vectorizes texts (question, answer, ground truth, context) using TfidfVectorizer.
     + Calculates cosine similarity between vectors to measure semantic overlap.
  4. **Scoring**:
     + Maps similarity scores to metrics (e.g., answer vs. context for faithfulness, answer vs. ground truth for answer\_similarity).
     + Combines TF-IDF and Jaccard for answer\_correctness.
  5. **Fallback**: Uses Jaccard-only scoring if TF-IDF fails, ensuring some result is returned.
* **Key Code**:

python

CollapseWrapCopy

def jaccard\_similarity(str1, str2):

set1, set2 = set(str1.lower().split()), set(str2.lower().split())

union\_len = len(set1 | set2)

return len(set1 & set2) / union\_len if union\_len > 0 else 0.0

vectorizer = TfidfVectorizer()

tfidf\_matrix = vectorizer.fit\_transform([question, answer, ground\_truth, context])

cosine\_sim = cosine\_similarity(tfidf\_matrix, tfidf\_matrix)

scores = {

"faithfulness": cosine\_sim[1, 3], *# Answer vs Context*

"answer\_relevancy": cosine\_sim[1, 0], *# Answer vs Question*

*# ... other mappings ...*

}

**Mechanics**

* **Jaccard Similarity**: Measures lexical overlap (e.g., shared words), simple but effective for basic alignment.
* **TF-IDF**: Weights words by importance (term frequency-inverse document frequency), capturing semantic relevance beyond raw overlap.
* **Cosine Similarity**: Computes the angle between TF-IDF vectors, indicating how similar two texts are in meaning.
* **Metric Approximation**: Assigns similarity scores to RAGAS metrics based on logical pairings (e.g., context vs. ground truth for context\_precision).

**Strengths**

* **Deterministic**: No reliance on external APIs or LLM variability, ensuring consistent results.
* **Lightweight**: Runs locally with minimal computational overhead.
* **Fallback Reliability**: Guarantees scores even in complete failure scenarios.

**Weaknesses**

* **Simplistic**: Lacks the nuanced reasoning of RAGAS or LLM methods, missing context or intent.
* **Lexical Bias**: Focuses on word overlap, potentially misjudging semantic differences (e.g., synonyms).
* **Limited Scope**: Approximates rather than directly measures complex metrics like faithfulness.

**Role in the App**

ML-based scoring is the last resort in \_render\_evaluate\_rag\_page, used when both RAGAS and LLM evaluation fail. It ensures the app always provides some evaluation output, albeit less sophisticated.

**Comparative Analysis**

| **Aspect** | **RAGAS** | **LLM-Based** | **ML-Based** |
| --- | --- | --- | --- |
| **Primary Use** | Main evaluation | First fallback | Final fallback |
| **Metrics** | 7 RAGAS metrics | Same 7, via prompts | Same 7, via similarity |
| **Dependency** | Bedrock LLM + embeddings | Bedrock LLM | None (local ML) |
| **Complexity** | High (framework-based) | Medium (prompt-based) | Low (statistical) |
| **Accuracy** | High (research-backed) | Moderate (subjective) | Low (approximate) |
| **Speed** | Slow (API calls) | Moderate (fewer calls) | Fast (local) |
| **Robustness** | Fragile (API-dependent) | More resilient | Very robust |

**Integration in the Workflow**

**Evaluation Cascade**

* **Step 1**: \_evaluate\_with\_ragas attempts RAGAS evaluation.
  + Success: Returns detailed scores.
  + Failure: Raises exception, triggers LLM fallback.
* **Step 2**: \_evaluate\_with\_llm uses Nova to score metrics.
  + Success: Returns LLM-assigned scores.
  + Failure: Raises exception, triggers ML fallback.
* **Step 3**: \_fallback\_scoring computes similarity-based scores.
  + Always succeeds, providing a baseline.

**Code Flow**

* **Trigger**: Initiated in \_render\_evaluate\_rag\_page when "Evaluate with RAGAS" is clicked.
* **Error Handling**: Wrapped in try-except blocks, with warnings displayed for each fallback transition.
* **Output**: Results are stored in evaluation\_results and saved as a CSV (e.g., RAG\_Results\_YYYYMMDD\_HHMMSS.csv).

**Practical Example**

* **Input**:
  + Question: "What are the 2023 CCAR capital requirements?"
  + Context: "In 2023, CCAR rules set Tier 1 capital at 4.5%."
  + Ground Truth: "2023 CCAR requires a 4.5% Tier 1 capital ratio."
  + Answer: "CCAR 2023 mandates 4.5% Tier 1 capital."
* **RAGAS**: Scores 1.0 for faithfulness, answer\_similarity, etc., using embeddings and LLM checks.
* **LLM Fallback**: Nova rates faithfulness as 1.0 via prompt response: "1.0".
* **ML Fallback**: Cosine similarity between answer and context yields 0.95 for faithfulness.

**Conclusion**

The RAGEvaluationApp employs a tiered evaluation strategy to balance accuracy, robustness, and resilience:

* **RAGAS** provides a gold standard for detailed, framework-based assessment.
* **LLM-Based Evaluation** offers a flexible, reasoning-driven alternative.
* **ML-Based Scoring** ensures a fail-safe, lightweight backup.

Together, these approaches enable comprehensive analysis of RAG systems, from retrieval quality to model performance, supporting applications like CCAR compliance with actionable insights. Users can refine their RAG pipeline by identifying weaknesses (e.g., low context\_recall for retrieval tuning) and comparing models (e.g., Claude vs. Nova) through the app’s visualizations.