RAG evaluation metrics: a guide with examples

Oct 28, 2026 · @Mikko

Introduction

A RAG system can fail in two separate places, so it needs metrics for each: retrieval (did we find the right chunks?) and generation (did the model use them well?). This guide covers the six Ragas metrics from Module 7 of *Developing Generative AI Applications on AWS*, each with a good and a bad worked example.

The evaluation sample. Every metric reads some of these four fields of a Ragas `SingleTurnSample`:

Field	What it holds	Example
`user_input`	The user's question	"What are the largest capitals in Europe?"
`retrieved_contexts`	Chunks the retriever returned, in rank order	Four chunks from the knowledge base
`response`	The answer the model generated	"Moscow (about 13 million), London…"
`reference`	The gold-standard answer, written by a person	See below

Running example. All examples use one question so the metrics can be compared directly.

- Question: "What are the largest capitals in Europe?"
- Reference: "The largest capitals in Europe are Moscow (about 13 million people), London (about 9 million), Berlin (about 3.7 million) and Madrid (about 3.4 million)."
- Population figures are rounded city-proper numbers, used only for illustration.

Reading scores. All six metrics run from 0 to 1. Neither Ragas nor AWS defines official thresholds; a common rule of thumb is ≥ 0.8 good, 0.5–0.8 investigate, < 0.5 a problem. Noise sensitivity is the exception: lower is better. High-stakes domains (medical, legal, finance) need stricter targets.

Retrieval metrics

Retrieval metrics judge the chunks before any answer is written. If they score low, no prompt or model change will fix the answer.

Context precision

Context precision asks whether the relevant chunks are ranked at the top. An irrelevant chunk at rank 1 hurts much more than one at rank 4.

- **Inputs:** `user_input`, `retrieved_contexts`, plus `reference` (with-reference variant) or `response` (without-reference variant).
- **How:** an LLM marks each chunk relevant (1) or not (0). Ragas then averages precision@k over the ranks where a relevant chunk sits.
- **Variants:** LLM-based without reference, LLM-based with reference, and non-LLM-based (string similarity against annotated reference contexts).

Formula: context precision@K = sum over k of (precision@k × v_k) ÷ number of relevant chunks in the top K

Here v_k is 1 if the chunk at rank k is relevant, else 0.

Good example — score 1.0

1. "Moscow has about 13 million residents…" (relevant)
2. "London's population is about 9 million…" (relevant)
3. "Berlin (3.7 million) and Madrid (3.4 million) are the EU's largest capitals…" (relevant)
4. "Berlin is famous for its nightlife…" (not relevant)

Pattern [1, 1, 1, 0]: precision@1, @2 and @3 are all 1.0, so the score is 1.0. The irrelevant chunk is last and costs nothing.

Bad example — score 0.25

1. "The Moscow metro opened in 1935…" (not relevant)
2. "Top 10 tourist sights in London…" (not relevant)
3. "Madrid hosted the 2019 Champions League final…" (not relevant)
4. "Population of European capitals: Moscow 13M, London 9M…" (relevant)

Pattern [0, 0, 0, 1]: only precision@4 = 0.25 counts, so the score is 0.25. The right chunk exists but is ranked last; with top-K = 3 it would be cut off. Fix with reranking or hybrid search.

Context recall

Context recall asks whether the context contains everything the reference answer says. It measures coverage, so it always needs a reference.

- **Inputs:** `user_input`, `retrieved_contexts`, `reference`.
- **How:** an LLM splits the reference into claims and checks whether each can be attributed to the context.
- **Variants:** LLM-based (claim level) or non-LLM-based (string distance such as Levenshtein against annotated reference contexts).

Formula: context recall = reference claims supported by the context ÷ total reference claims

The reference has four claims: Moscow at 13M, London at 9M, Berlin at 3.7M, Madrid at 3.4M.

Good example — score 1.0. The four chunks from the good precision example cover all four claims: 4/4.

Bad example — score 0.25. Context: "Moscow is Europe's most populous capital with about 13 million inhabitants. The city was founded in 1147." Only the Moscow claim is supported: 1/4. Even a perfect model cannot write a complete answer from this. Typical causes: top-K too small, or the list split across chunks.

Context entity recall

Context entity recall asks whether the key names, places, dates and numbers from the reference appear in the context. It is stricter and more literal than context recall.

- **Inputs:** `retrieved_contexts`, `reference`.
- **How:** an LLM extracts entities from both, then counts the overlap.
- **Best for:** fact-heavy domains where one missing name or number breaks the answer: tourism, history, product catalogs, legal, finance.

Formula: context entity recall = entities found in both context and reference ÷ entities in the reference

The reference has eight entities: Moscow, London, Berlin, Madrid, 13M, 9M, 3.7M, 3.4M.

Good example — score 1.0. Context: "Moscow (13M), London (9M), Berlin (3.7M), Madrid (3.4M)." All eight entities match: 8/8.

Bad example — score 0.25. Context: "Moscow is the largest capital in Europe, ahead of London. Germany's and Spain's capitals are the biggest in the EU." Only Moscow and London appear by name, and no numbers: 2/8. An LLM may still infer Berlin and Madrid, so context recall can score higher here; entity recall exposes that the hard facts are missing.

Two cases where the recall metrics disagree

Context	Context recall	Entity recall	What it means
"The Russian capital has about 13 million residents… the British capital is second at 9 million, ahead of the German (3.7M) and Spanish (3.4M) capitals."	1.0	0.5	Facts present, names paraphrased; the model must infer them
"Moscow, London, Berlin and Madrid have all hosted major football finals."	0.0	0.5	Names matched, facts missing; typical keyword-retrieval miss

Generation metrics

Generation metrics judge the answer, assuming retrieval has done its job. All examples below use the four good chunks from the context precision example.

Faithfulness

Faithfulness asks whether every claim in the answer is supported by the retrieved context. It is the main hallucination metric.

- **Inputs:** `user_input`, `retrieved_contexts`, `response`. No reference needed.
- **How:** an LLM splits the response into claims and checks each against the context.
- **Variants:** standard LLM-based (Ragas `Faithfulness`), Vectara HHEM hallucination model, or Bedrock's native faithfulness metric.

Formula: faithfulness = response claims supported by the context ÷ total response claims

Good example — score 1.0. Response: "The largest European capitals are Moscow (about 13 million), London (about 9 million), Berlin (3.7 million) and Madrid (3.4 million)." Four claims, all in the context: 4/4.

Bad example — score 0.5. Response: "The largest are Moscow (13 million) and London (9 million), followed by Rome (2.8 million) and Paris (2.1 million)." The Rome and Paris claims are not in the context: 2/4.

Those figures are roughly true; the model took them from its training data. Faithfulness still drops, because it measures whether claims are supported, not whether they are true. That is what you want when answers must come from your own documents. The reverse

also holds: a faithful answer is wrong if the source documents are wrong.

Response relevancy

Response relevancy asks whether the answer addresses the question that was actually asked. It ignores factual accuracy.

- **Inputs:** `user_input`, `response` (Ragas also accepts `retrieved_contexts`). Needs an evaluator LLM and an embedding model.
- **How:** the LLM writes artificial questions (three by default) that the response would answer. Each is embedded and compared with the original question by cosine similarity, then averaged.
- **Penalizes:** incomplete answers, padding unrelated to the question, and off-topic or evasive replies.

Formula: response relevancy = average over the N generated questions of $\cos(E_g_i, E_o)$

E_o is the embedding of the original question, E_g_i of generated question i.

Good example — about 0.89. Response: the good faithfulness answer.

Generated question	Similarity to original
"Which European capitals have the most inhabitants?"	0.95
"What is the largest capital city in Europe?"	0.90
"What are the populations of Moscow, London, Berlin and Madrid?"	0.82

Bad example — about 0.5. Response: "Europe has around 44 countries, each with its own capital. Capitals usually host the government and are important cultural centers."

Generated question	Similarity to original
"How many countries are there in Europe?"	0.55
"What role do capital cities play?"	0.48

Everything in it is true and on topic, but it does not answer the question. Similarity values are illustrative; real ones depend on the embedding model.

Noise sensitivity

Noise sensitivity asks how often irrelevant context leads the model into wrong claims. Lower is better.

- **Inputs:** `user_input`, `retrieved_contexts`, `response`, `reference`.
- **How:** an LLM splits the response into claims, checks which are incorrect against the reference, and checks whether those incorrect claims trace back to the retrieved chunks (relevant or irrelevant mode).

Formula: noise sensitivity = incorrect claims in the response ÷ total claims in the response

For these examples a distracting chunk is added to the context: "Istanbul, with 15.5 million residents, is the largest city in Europe."

Good example — score 0.0. Response: "Moscow (13M), London (9M), Berlin (3.7M), Madrid (3.4M)." The model ignored the Istanbul chunk, correctly, since Istanbul is not a capital: 0/4 incorrect.

Bad example — score 0.5. Response: "Istanbul is the largest capital (15.5 million), followed by Moscow (13M), London (9M) and Berlin (3.7M)." Two of four claims are wrong (Istanbul is a capital; Istanbul is the largest capital): 2/4. Fix by cutting noise at retrieval (better precision, reranking) and by prompting the model to answer strictly what was asked.

Reading the metrics together

One metric alone rarely tells you what to fix; the pattern across retrieval and generation does. Check retrieval first, because a generation fix cannot recover information that was never retrieved.

Metric	Stage	Needs reference	Good example	Bad example	A bad score usually means
Context precision	Retrieval	Optional	1.0	0.25	Right chunks ranked too low
Context recall	Retrieval	Yes	1.0	0.25	Information missing from the context
Context entity recall	Retrieval	Yes	1.0	0.25	Specific names or numbers missing
Faithfulness	Generation	No	1.0	0.5	Hallucination or reliance on training data
Response relevancy	Generation	No	~0.89	~0.5	Off-topic, incomplete or evasive answer

Metric	Stage	Needs reference	Good example	Bad example	A bad score usually means
Noise sensitivity (lower is better)	Generation	Yes	0.0	0.5	Model misled by irrelevant chunks

Diagnosis patterns

Pattern	Likely cause	Fixes to try
Low recall, any precision	Retriever misses information	Raise top-K, smaller or overlapping chunks, better embedding model, query rewriting
Good recall, low precision	Right chunks found but buried	Reranking (e.g. a Bedrock rerank model), metadata filters, lower top-K after reranking
Good recall, low entity recall	Topic found, exact facts missing	Hybrid search (keyword + semantic), metadata on entities, smaller chunks
Good retrieval, low faithfulness	Model adds outside knowledge	Stricter grounding instructions, ask for citations, contextual grounding check in Guardrails
Good retrieval, low relevancy	Model drifts from the question	Clearer prompt template, few-shot examples, restate the question in the prompt
High noise sensitivity	Model trusts every chunk	Improve precision, rerank, tell the model to ignore unrelated context

Where the reference comes from. Context recall, entity recall and noise sensitivity need a person-written reference, so they run on a fixed golden test set. Faithfulness and response relevancy need no reference, so they can also score sampled production traffic.

On AWS

You can run these metrics with Ragas in your own code, or use Amazon Bedrock's managed RAG evaluation, which offers close equivalents for most of them.

Ragas vs Bedrock RAG evaluation (approximate mapping)

Ragas metric	Closest Bedrock RAG evaluation metric	Notes
Context precision	Context relevance (retrieve-only job)	Bedrock scores relevance per chunk, not rank-weighted
Context recall	Context coverage (retrieve-only job)	Needs ground-truth answers
Context entity recall	None	Run Ragas yourself
Faithfulness	Faithfulness (retrieve-and-generate job)	Also citation precision and citation coverage
Response relevancy	Helpfulness, completeness, correctness	Bedrock judges these directly with an LLM judge
Noise sensitivity	None	Run Ragas yourself

Bedrock's metric list changes over time; check the current docs before relying on a specific name.

Running Ragas with Bedrock models as the judge

```
from langchain_aws import ChatBedrockConverse, BedrockEmbeddings
from ragas import evaluate, EvaluationDataset, SingleTurnSample
from ragas.llms import LangchainLLMWrapper
from ragas.embeddings import LangchainEmbeddingsWrapper
from ragas.metrics import (LLMContextPrecisionWithReference,
LLMContextRecall,
    ContextEntityRecall, Faithfulness, ResponseRelevancy, NoiseSensitivity)

evaluator_llm = LangchainLLMWrapper(ChatBedrockConverse(model="<judge model
id>", temperature=0))
evaluator_embeddings =
LangchainEmbeddingsWrapper(BedrockEmbeddings(model_id="amazon.titan-embed-
text-v2:0"))

sample = SingleTurnSample(
    user_input="What are the largest capitals in Europe?",
    retrieved_contexts=["Moscow has about 13 million residents...", "London's
population is about 9 million..."],
    response="The largest are Moscow (13M) and London (9M)...",
    reference="Moscow (about 13M), London (about 9M), Berlin (about 3.7M) and
Madrid (about 3.4M).",
)
```

```
result = evaluate(
    dataset=EvaluationDataset(samples=[sample]),
    metrics=[LLMContextPrecisionWithReference(), LLMContextRecall(),
ContextEntityRecall(),
             Faithfulness(), ResponseRelevancy(), NoiseSensitivity()],
    llm=evaluator_llm, embeddings=evaluator_embeddings,
)
print(result)
```

Pin the judge model ID and use temperature 0, so score changes come from your system and not the judge.

At runtime: ApplyGuardrail. The contextual grounding check in Amazon Bedrock Guardrails scores grounding and relevance on live answers, a runtime counterpart to faithfulness and response relevancy. With `ApplyGuardrail` you mark the retrieved text as `grounding_source`, the question as `query` and the answer as `guard_content`.

Notes on the Module 7 deck

- **Slide 18:** the notes average precision@k over all positions ([1, 0, 1, 1] → 0.73). Ragas averages only over relevant positions, which gives 0.81.
- **Slide 23:** the diagram box for context entity recall is labeled "Context recall".
- **Slide 26:** the notes say only the user input and response feed noise sensitivity; the diagram and Ragas also use the reference and retrieved contexts.

Sources

- Module 7 deck, *Evaluating Generative AI Application Components* (v2.1.16), slides 13–31
- AWS docs linked in the deck: Amazon Bedrock RAG evaluation, LLM as a judge