

# Evaluating RAG Performance

---

Congratulations on building your first RAG! RAG – short for Retrieval-Augmented Generation – is one of the most popular ways to build useful LLM applications. It lets you bring in external knowledge so your AI assistant or chatbot can work with more than just what the LLM memorized during training.

But adding retrieval functionality makes things more complex.

Suddenly you are not just prompting an LLM – you're building a pipeline. There is chunking, search, context assembly, and generation. And when something goes wrong, it's not always clear where the problem is. Did the model hallucinate? Or did it never get the right information to begin with?

So how can you tell how well your RAG system is working, before releasing it to your users? This is where evaluation mechanisms come in.

## The story so far...

You've successfully implemented RAG into **TaskFriend**, and now it can obtain information stored elsewhere on your app. But – you have a nagging feeling that something, somewhere is not quite right. So you did a check between your app data, and what **TaskFriend** is telling you. Lo and behold – **TaskFriend** gets some information wrong. You're worried that this might be a symptom of a larger problem, but manually combing through the data is not feasible, at all.

So, you decide to build an evaluation mechanism to help you automatically evaluate how well the RAG functionality on your **TaskFriend** is doing.

## Goals

- Understand why manual evaluation is useful for debugging but insufficient at scale.
- Learn how to inspect a RAG pipeline step-by-step to identify failure points.
- Grasp the core dimensions of RAG evaluation: context recall, context precision, faithfulness, and answer correctness.
- Set up and use Ragas to automate the evaluation of a RAG system.
- Interpret evaluation metrics to gain actionable insights and improve your RAG pipeline.
- Compare popular RAG evaluation frameworks and understand when to use each.
- Establish best practices for building a sustainable, expert-informed evaluation process.

## Intitializing the environment

### Setting up the API key

Before we start work on in any notebook, we'll need to load the [API key for Model Studio](#). This ensures that we can call APIs of Qwen models we'll be using throughout this course.

If you're unsure about how to find your **Model Studio** API key, refer to the [00 Setting Up the Environment](#) file.

```
# Load Model Studio API key
import os
from config.load_key import load_key
load_key(
    confirmation=False
)
```

## Setting up the LLM and embedding model

Like the previous lesson, we'll be using Alibaba Cloud's `qwen-plus` as the LLM and DashScope's `text-embedding-v3` embedding model.

```
# Set global settings
import time
import logging
import dashscope
from llama_index.core import Settings, VectorStoreIndex,
SimpleDirectoryReader
from llama_index.embeddings.dashscope import DashScopeEmbedding
from llama_index.llms.openai_like import OpenAILike
from pathlib import Path

logging.getLogger().setLevel(logging.ERROR)

# Dashscope uses https://dashscope-intl.aliyuncs.com/api/v1
# instead of https://dashscope-intl.aliyuncs.com/compatible-mode/v1
dashscope.base_http_api_url = "https://dashscope-intl.aliyuncs.com/api/v1"

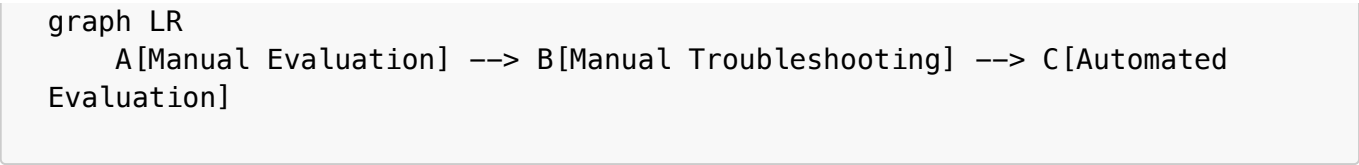
Settings.llm=OpenAILike(
    model="qwen-plus",
    api_base="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    is_chat_model=True
)

Settings.embed_model = DashScopeEmbedding(
    model_name="text-embedding-v2",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    encoding_format="float"
)

print("✅ Global parameters set!")
```

## RAG Evaluation Pipeline


---




In a typical real-world scenario, evaluating RAG systems are best left to **automated evaluation** methods - due to the sheer size and volume of data that you'll need to deal with. However, today we'll be looking at a smaller scope - our newly built **TaskFriend** app.


## Manual evaluation: Identifying the issue

In the previous example, you may remember something like this:


 TaskFriend Conversation



You: How many tasks do I need to complete today?



TaskFriend is thinking...



TaskFriend: You have one task that is due today.

But when we look into our `tasks.pdf`, we find out that this is not the case:

ID	Description	Type	Priority	Due	Status	Stakeholders	Notes
01	Finalize Q3 OKRs by 3pm	One-off	High	★ Today	Not Started	Department Leads, Executive Team, HRBP	Collaborate with department heads to align on measurable objectives. Ensure KPIs are SMART (Specific, Measurable, Achievable, Relevant, Time-bound). Submit final version to leadership for approval before 3 PM.
03	Onboard new team member	One-off	High	★ Today	Not Started	HR, IT Support, Team Mentor, New Hire	Complete HR onboarding checklist: assign laptop, set up email, grant access to

							DingTalk, Jira, and internal wikis. Schedule intro meetings with team members. Assign mentor for first 30 days. Send welcome email with onboarding schedule.
05	Update project roadmap in Dingtalk	Recurring	Med	★ Today	Not Started	Project Leads, Engineering, Design, QA	Sync with project leads to reflect latest timelines, milestones, and resource allocations. Highlight any delays or risks. Ensure all stakeholders are tagged and notified upon update. Use Gantt view for clarity.
...							
14	Write thank-you letter to penpal in Korea	One-off	Low	★ Today	Started	Penpal	Thank penpal for the help they provided when you needed some advice on whether you should go to Norway. They provided a lot of useful information that aided in your decision to bring your family to Norway for holiday.

So what happened? What is causing this issue?

Troubleshooting your RAG: Getting hands-on

Let's try out our RAG-powered **TaskFriend** again. But this time, we're going to also **print the chunks (or nodes)** that our RAG retrieves from our index.

We'll also add a custom `highlight_words` function for better visibility. You'll see why in a bit.

```
def highlight_words(text, words_to_highlight, emoji="★"):
    for word in words_to_highlight:
        if word in text:
            text = text.replace(word, f"{emoji}{word}{emoji}")
    return text

def get_rag_response_with_info(query, query_engine, highlight=None):
    print("🔥 TaskFriend Conversation (single-round)")
    print("-" * 50)
    print(f'👤 You: {query}')

    try:
        # 🔍 Query the RAG engine
        response = query_engine.query(query)

        # 🧠 Extract the answer
        if hasattr(response, 'response'):
            answer = response.response
        else:
            answer = str(response)

        # 📖 Show source references AFTER the answer
        print("🤖 TaskFriend:", answer)
        print('\n\n' + '=' * 50)
        print('📖 References\n')

        highlight = highlight or []
        for i, source_node in enumerate(response.source_nodes, start=1):
            print(f'Chunk {i}:')

            # Highlight words in chunk
            highlighted_text = highlight_words(source_node.text,
highlight)
            print(highlighted_text)
            print()
            print('=' * 50)

        return answer

    except Exception as e:
        print(f"[RAG Error] {e}")
        return "[Error retrieving response]"
```

Since we already built and saved our index in a previous chapter, all we need to do is load it from where we saved it (`./knowledge_base/taskfriend`).

Here, we're going to use some functions from `llama_index`:

- **StorageContext**: Stores the saved nodes, embeddings, and vector store from the previous chapter, letting your code find and connect to this saved data, so you can reload your entire knowledge base without having to re-process all your documents from scratch.
- **load\_index\_from\_storage**: Rebuilds your RAG index from the saved files stored in **StorageContext**. It's the most efficient way to get your RAG system up and running again.

```
from llama_index.core import StorageContext, load_index_from_storage

persist_path="./knowledge_base/taskfriend"

# Import index ("knowledgebase") we built last chapter,
storage_context = StorageContext.from_defaults(
    persist_dir=persist_path,
)

index = load_index_from_storage(
    storage_context,
    embed_model=Settings.embed_model
)
print(f"✅ Index loaded from `{persist_path}`!")

# Build the query engine (used to implement RAG)
query_engine = index.as_query_engine(
    streaming=False,
    llm=Settings.llm,
)
print("✅ Query engine built!")
```

And finally, we're ready to run the same query, but this time with the info on where **TaskFriend** is getting its information from.

```
# User query
query = "How many tasks do I need to complete today?"


# Choose words to highlight
highlight = ["Today"]

response = get_rag_response_with_info(query, query_engine=query_engine,
highlight=highlight)
```

Now, we'll see something like this:



## TaskFriend Conversation (single-round)

 You: How many tasks do I need to complete today?

 TaskFriend: You need to complete one task today. Task 01, "Finalize Q3 OKRs by 3pm," is due today and is marked as "Not Started."

## References

Chunk 1:

...ID Descrip)on Type Priority Due Status Stakeholders Notes 01 Finalize Q3 OKRs by 3pm One-off High  Today  Not Started Department Leads, ExecuHve Team, HRBP Collaborate with department heads to align on measurable...

Chunk 2:

...automaHng repeHHve tasks (e.g., email sorHng, report generaHon). Evaluate cost, integraHon ease, and security compliance. Prepare comparison matrix and present by Q4. 08 Clean up email inbox Recurring ...

There are a few things that may jump out at you:

- The word **Today** only appears once, and in one chunk only. (Thanks **highlight\_words** for helping us there!)
- The RAG is retrieving *only 2 chunks*.
- It seems like any word with **ti** in it has had it replaced with **H**.
  - Executive → ExecuHve
  - automating → automaHng

With this, you've successfully determined the root cause of the issues in your RAG system.

## Automated Evaluation

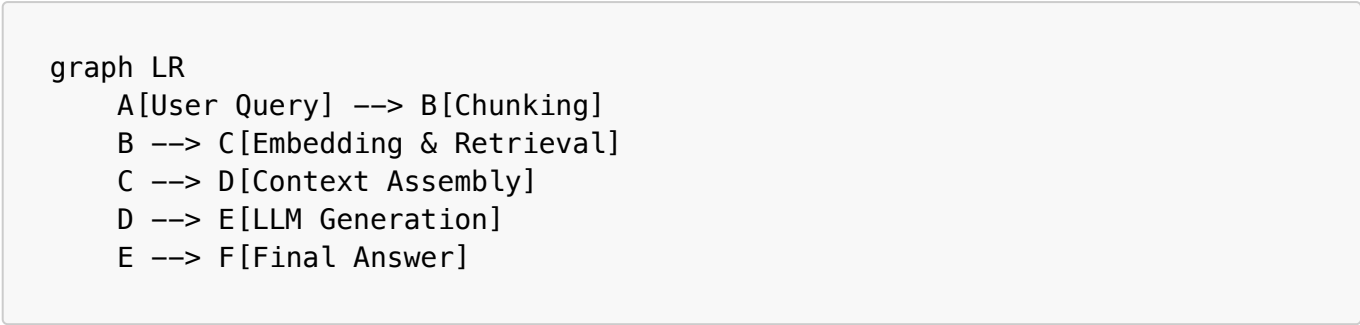
For one-off cases or applications like TaskFriend in the early development stage, manual evaluation and troubleshooting is fine — and will even help you as a learner to grow as you dive deep into the nitty-gritty of your development process. However, once you've reached a certain scale, manually looking for issues becomes less and less feasible.

Now, let's look into how we can automate this process.

But before that, let's talk about the how. We need to know what we're looking for before we can automate looking for it. There are a few basic ideas behind evaluating LLMs and their extended pipelines:

## What Are We Evaluating?

A RAG system is a pipeline, not a single model. It has multiple stages, and each can fail independently:



When the final answer is wrong, the problem could be at any of these stages:

- ✗ **Chunking:** The relevant info was split across chunks.
- ✗ **Retrieval:** The right chunk wasn't retrieved.
- ✗ **Generation:** The LLM ignored or misinterpreted the context.

So, to evaluate RAG properly, we need to assess multiple dimensions:

Metric	What it measures?	Why it matters?
Context Recall	Did the retrieved context contain the correct answer?	If not, no matter how good your LLM is, it can't answer correctly.
Context Precision	How many retrieved chunks were actually relevant?	Low precision means the LLM is distracted by noise.
Answer Correctness	Is the final answer factually correct and complete?	This is the user-facing result — it's what ultimately matters.
Faithfulness	Does the answer stick to the retrieved context?	Prevents hallucinations and ensures grounding.

These are a few metrics that we can use to evaluate RAGs. For more mertics, the [Ragas documentation](#) proves to be a great source of information.

## Introducing Ragas: Automated RAG evaluation

To automate evaluation, we'll use [Ragas](#), a powerful open-source framework designed specifically for evaluating RAG systems.

Ragas computes the metrics we talked about automatically, using another LLM as a judge (called an "evaluator LLM"). You don't need human labels for every test case — just a set of questions and ground-truth answers.

However, Ragas internally uses **LangChain's** evaluation framework, that means you'll have to build a wrapper around our RAG pipeline.

Let's go through it step by step.



```

from langchain_openai import ChatOpenAI
from langchain_dashscope import DashScopeEmbeddings
import os
import time

dashscope.base_http_api_url = "https://dashscope-intl.aliyuncs.com/api/v1"

# LangChain LLM for Ragas
ragas_llm = ChatOpenAI(
    model="qwen-plus",
    base_url="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    timeout=60,
    max_retries=3
)

# LangChain Embeddings for Ragas
ragas_embeddings = DashScopeEmbeddings(
    model="text-embedding-v2",
    api_key=os.getenv("DASHSCOPE_API_KEY")
)

print("✅ LLM and Embeddings pipeline for Ragas successfully built!")

```

```

# Create a new query engine for evaluation
query_engine = index.as_query_engine(
    llm=ragas_llm,
    embeddings=ragas_embeddings,
    streaming=False # Disable streaming for evaluation
)

print("✅ Query engine built!")

```

```

# Define test cases
test_cases = [
    {
        "question": "How many tasks are due today?",
        "ground_truth": "You have 4 tasks due today: Finalize Q3 OKRs, Onboard new team member, Update project roadmap, and Write thank-you letter to penpal."
    },
    {
        "question": "What is the status of the 'Plan for trip to Norway' task?",
        "ground_truth": "The status is Started."
    },
    {
        "question": "Who are the stakeholders for updating the project

```

```
roadmap?",
    "ground_truth": "Project Leads, Engineering, Design, QA"
},
{
    "question": "When is the 'Develop 3-year plan' task due?",
    "ground_truth": "This Year"
},
{
    "question": "What is 'Project Phoenix'?",
    "ground_truth": "'Project Phoenix' is a "
},
]

print("✅ Test cases defined!")
```

```
from functions.html_table import create_html_table

# Define function to run test cases with evaluator LLM
def run_test_cases(query_engine, test_cases):
    results = {
        "question": [],
        "answer": [],
        "contexts": [], # Store raw contexts for Dataset
        "contexts_display": [], # Store formatted contexts for display
        "ground_truth": []
    }

    print("Generating answers based on test cases...", end="", flush=True)
    start_time = time.time()

    for case in test_cases:
        # Get response
        response = query_engine.query(case["question"])

        # Extract answer and context
        answer = str(response).strip()
        # Ensure contexts is a list of strings
        contexts = [node.get_content().strip() for node in
response.source_nodes]

        # Format contexts for display with [] and line breaks
        contexts_display = [f"[{ctx}]" for ctx in contexts]

        # Store results
        results["question"].append(case["question"])
        results["answer"].append(answer)
        results["contexts"].append(contexts)
        results["contexts_display"].append("★".join(contexts_display)) #
Use this to better visualize breaks between contexts
        results["ground_truth"].append(case["ground_truth"])
```

```

    load_time = time.time() - start_time
    print(f"✅ Done ({load_time:.1f} seconds)")

    return results

# Run the test
test_results = run_test_cases(query_engine, test_cases)

table_data = {
    "question": test_results["question"],
    "answer": test_results["answer"],
    "contexts": test_results["contexts_display"],
    "ground_truth": test_results["ground_truth"]
}

# Use 5:15:15:50:15 percentage distribution (index + 4 columns)
create_html_table(
    table_data,
    title="RAG Evaluation Results",
    column_widths=[5, 15, 15, 50, 15] #
index:question:contexts:ground_truth:answer
)

```

```

from datasets import Dataset

# Convert to Dataset
eval_dataset = Dataset.from_dict({
    "question": test_results["question"],
    "answer": test_results["answer"],
    "contexts": test_results["contexts"],
    "ground_truth": test_results["ground_truth"]
})

```

```

from ragas import evaluate
from ragas.metrics import (
    context_recall,
    context_precision,
    faithfulness,
    answer_correctness
)
from functions.rag_eval_table import create_rag_evaluation_table

# Use the LangChain-compatible LLM for Ragas
results = evaluate(
    dataset=eval_dataset,
    metrics=[
        context_recall,
        context_precision,
        faithfulness,

```

```
        answer_correctness
    ],
    llm=ragas_llm,
    embeddings=ragas_embeddings
)

# Convert results to pandas DataFrame
results_df = results.to_pandas()

# Extract the input fields from the dataset to add context
results_df["question"] = eval_dataset["question"]
results_df["ground_truth"] = eval_dataset["ground_truth"]
results_df["answer"] = eval_dataset["answer"]

# Reorder columns for better readability
results_df = results_df[[
    "question",
    "answer",
    "ground_truth",
    "answer_correctness",
    "context_recall",
    "context_precision",
    "faithfulness",
]]

# print("\n📊 Evaluation Results:")
# print(results_df)
create_rag_evaluation_table(results_df)
```

## Dissecting Ragas Evaluation Metrics

We've run our evaluation and seen the numbers. Now it's time to open the hood and understand what each metric really means. Just like a doctor uses different blood tests to diagnose an illness, we use these metrics to diagnose the health of our RAG pipeline.

Remember, these scores aren't just a final grade—they're a diagnostic report. A low **answer\_correctness** score tells us the patient is sick, but it's the other metrics that tell us why.

### Note:

Each Ragas evaluation metric has two variants: LLM-based and non-LLM-based. Ragas uses the LLM-based evaluation metrics by default, and that's what we'll cover.

For more information, see the [Ragas documentation](#).

Let's dive into the four core metrics we used, one by one.

### Context recall

Did we find the right information?

**What it measures:**

Context Recall answers the critical question: Did our retriever find all the information needed to answer the question correctly? It measures the fraction of relevant facts from the ground truth that were present in the retrieved context.

**Why it matters:**

This is the foundation of a reliable RAG system. If the right information isn't retrieved, the LLM is working blind. No matter how good your LLM is, it cannot generate a correct answer from missing facts. A high recall score means your retrieval stage is doing its job well.

Ragas calculates this by breaking down the **ground\_truth** into atomic "claims" and checking if each claim can be supported by the **retrieved\_contexts**. The formula is:

$$\text{Context Recall} = \frac{\text{Number of claims in the reference supported by the retrieved context}}{\text{Total number of claims in the reference}}$$

**Seeing it in action**

Let's create a simple test case to see how the retrieved context directly impacts the recall score. We'll use the same `ragas_llm` and `ragas_embeddings` from our setup.

```
from functions.metric_barchart import plot_metric_comparison

# Define the test scenario
question = "How many tasks are due today?"

# The ground truth contains 4 distinct claims
ground_truth = (
    "1. Finalize Q3 OKRs is due today, "
    "2. Onboard new team member is due today, "
    "3. Update project roadmap is due today, "
    "4. Write thank-you letter is due today."
)

# Test Case 1: Perfect Retrieval - All 4 claims are present
contexts_perfect = [
    "ID 01: Finalize Q3 OKRs - Due: Today",
    "ID 03: Onboard new team member - Due: Today",
    "ID 05: Update project roadmap - Due: Today",
    "ID 14: Write thank-you letter - Due: Today"
]

# Test Case 2: Partial Retrieval - Only 2 claims are present
contexts_partial = [
    "ID 01: Finalize Q3 OKRs - Due: Today",
    "ID 05: Update project roadmap - Due: Today",
]
```

```

# Test Case 3: No Retrieval – 0 claims are present
contexts_none = [
    [
        "ID 02: Prepare presentation – Due: This Week",
        "ID 08: Clean up email – Due: This Year"
    ]
]

# Create datasets
data_perfect = {"question": [question], "ground_truth": [ground_truth],
"contexts": contexts_perfect}
data_partial = {"question": [question], "ground_truth": [ground_truth],
"contexts": contexts_partial}
data_none = {"question": [question], "ground_truth": [ground_truth],
"contexts": contexts_none}

dataset_perfect = Dataset.from_dict(data_perfect)
dataset_partial = Dataset.from_dict(data_partial)
dataset_none = Dataset.from_dict(data_none)

# Evaluate all three cases
result_perfect = evaluate(dataset_perfect, metrics=[context_recall],
llm=ragas_llm, embeddings=ragas_embeddings)
result_partial = evaluate(dataset_partial, metrics=[context_recall],
llm=ragas_llm, embeddings=ragas_embeddings)
result_none = evaluate(dataset_none, metrics=[context_recall],
llm=ragas_llm, embeddings=ragas_embeddings)

# Extract scores
scores = {
    "Perfect Retrieval\n(4/4 claims)": result_perfect["context_recall"],
    "Partial Retrieval\n(2/4 claims)": result_partial["context_recall"],
    "No Relevant Retrieval\n(0/4 claims)": result_none["context_recall"]
}

# Generate comparison chart
plot_metric_comparison(
    scores_dict=scores,
    title="Context Recall Score vs. Retrieved Information Quality",
    ylabel="Context Recall Score",
)

```

### What the chart shows:

The recall score drops linearly with the amount of relevant information retrieved. This proves that recall is fundamentally about completeness of retrieval.

### Improving context recall

- **Adjust Chunk Size:** If your chunks are too small, a single fact might be split across multiple chunks. Try increasing chunk\_size (e.g., from 128 to 512).

- **Use Chunk Overlap:** Add `chunk_overlap=100` to your text splitter to preserve context at the boundaries.
- **Improve Embeddings:** Experiment with different embedding models. A model better suited to your domain data will create more meaningful vector representations.
- **Implement Re-ranking:** Use a cross-encoder model to re-rank the initial retrieval results, boosting the most relevant chunks to the top.
- **Hybrid Search:** Combine vector search with keyword-based search (like BM25) to catch documents that might be missed by embeddings alone.

## Context precision

How much noise did we retrieve?

### What it measures:

Context Precision answers a key question: **Of all the chunks we retrieved, how many were actually relevant to the query?**

It measures the signal-to-noise ratio of your retrieval.

### Why it matters:

A low precision score means your LLM is being fed a lot of irrelevant information. This can distract the model, increase processing costs, and potentially lead to incorrect answers if the LLM latches onto a red herring.

Ragas calculates this metric using an LLM-as-a-judge approach. It doesn't perform a simple arithmetic calculation. Instead, it uses a separate "evaluator" LLM to analyze the `question`, the `ground_truth`, and the `retrieved_contexts`. The evaluator LLM assesses how relevant the retrieved chunks are to the question and whether the most useful information is prioritized at the top of the list.

This means the score reflects a nuanced judgment about the quality of the retrieval, considering both the relevance of the chunks and their order.

$$\text{Precision@k} = \frac{\text{true positives@k}}{\text{true positives@k} + \text{false positives@k}}$$

### Seeing it in action

Let's see how the order of retrieved chunks affect the `context_precision` score. We'll use a simple query and three different retrieval results.

```
# Define the test scenario
question = "When is the project roadmap due?"

# Ground truth (based on your data, it's "Project Leads")
ground_truth = "The project roadmap is due today."

# Test Case 1: High Precision
contexts_high = [
    "Update project roadmap is due today"
```

```

]

# Test Case 2: Med Precision
contexts_med = [
    [
        "Trip to Norway is in November",
        "Update project roadmap is due today",          # Correct
        "Write to penpal needs to be completed within 24 hours"
    ]
]
answer is preceded by one false answer

# Test Case 3: Low Precision
contexts_low = [
    [
        "Trip to Norway is in November",
        "Write to penpal needs to be completed within 24 hours",
        "Update project roadmap is due today"          # Correct
    ]
]
answer is preceded by two false answers

# Create and evaluate datasets
def create_and_evaluate(contexts_list, question, ground_truth):
    dataset = Dataset.from_dict({
        "question": [question],
        "ground_truth": [ground_truth],
        "contexts": contexts_list
    })
    result = evaluate(dataset, metrics=[context_precision], llm=ragas_llm,
embeddings=ragas_embeddings)
    return result["context_precision"]

scores = {
    "High Precision\n(Correct answer at top)":
create_and_evaluate(contexts_high, question, ground_truth),
    "Med Precision\n(Correct answer buried once)":
create_and_evaluate(contexts_med, question, ground_truth),
    "Low Precision\n(Correct answer buried twice)":
create_and_evaluate(contexts_low, question, ground_truth)
}

# Generate comparison chart
plot_metric_comparison(
    scores_dict=scores,
    title="Context Precision Score: When Answers are Buried",
    ylabel="Context Precision Score",
)

```

**What the chart shows:**

- **High (1.000):** The evaluator LLM sees a perfect result. The only chunk retrieved is highly relevant and is presented first.
- **Medium (0.500):** The evaluator LLM sees that the correct information is present but is preceded by an irrelevant chunk. This "noise" is penalized, significantly lowering the score.
- **Low (0.333):** The evaluator LLM sees that the correct information is buried under two irrelevant chunks. This poor ranking results in a very low score, indicating a high level of noise.

This experiment proves that **context\_precision** is not just about what you retrieve, but in what order. A high precision score means your retrieval system is not only finding the right information but also presenting it in a way that the LLM can easily use.

## Improving context precision

- **Tune similarity threshold:** Filter out retrieved chunks with a similarity score below a certain threshold.
- **Use query rewriting:** Employ a technique like HyDE (Hypothetical Document Embeddings) to generate a better query before retrieval.
- **Leverage metadata filtering:** If your data has metadata (e.g., task\_type, priority), use it to filter the search space before the vector search.
- **Implement re-ranking:** A re-ranker can push irrelevant but vectorially-close chunks down the list.

By optimizing for context precision, you ensure that your LLM is working with a clean, focused, and well-ordered set of information—maximizing its chances of generating a correct and helpful answer.

## Faithfulness

### What it measures:

Faithfulness answers the question: *Did the LLM hallucinate, or did it base its answer strictly on the provided context?*

It measures the fraction of claims in the generated answer that can be directly supported by the retrieved context.

### Why it matters:

This is crucial for building trustworthy AI. A high faithfulness score means your LLM is not making up information. It's a direct measure of the model's grounding in the provided evidence.

The calculation is similar to recall, but flipped:

$$\text{Faithfulness Score} = \frac{\text{Number of claims in the response supported by the retrieved context}}{\text{Total number of claims in the response}}$$

## Seeing it in action

Let's create an answer that contains a hallucination.

```
# Define the test case
question = "When is the 'Develop 3-year plan' task due?"

# The ground truth
```

```
ground_truth = "This Year"

# The retrieved context (our source of truth)
contexts = [
    [
        "The 'Develop 3-year plan' task is due 'This Year'.",
        "The 'Finalize Q3 OKRs' task is due 'Today'."
    ]
]

# Test Case 1: Faithful Answer
answer_faithful = "The 'Develop 3-year plan' task is due 'This Year'."

# Test Case 2: Hallucinated Answer
answer_hallucinated = "The 'Develop 3-year plan' task is due 'This Year,' and it must be approved by the CFO before submission."

# Create datasets
dataset_faithful = Dataset.from_dict({
    "question": [question],
    "answer": [answer_faithful],
    "contexts": contexts
})

dataset_hallucinated = Dataset.from_dict({
    "question": [question],
    "answer": [answer_hallucinated],
    "contexts": contexts
})

# Evaluate
result_faithful = evaluate(dataset_faithful, metrics=[faithfulness],
                           llm=ragas_llm, embeddings=ragas_embeddings)
result_hallucinated = evaluate(dataset_hallucinated, metrics=[faithfulness],
                                llm=ragas_llm, embeddings=ragas_embeddings)

# Prepare data for comparison
scores = {
    "Faithful Answer\n(Claims supported by context)":
    result_faithful["faithfulness"],
    "Hallucinated Answer\n(Claim about 'CFO approval' unsupported)":
    result_hallucinated["faithfulness"]
}

# Generate comparison chart
plot_metric_comparison(
    scores_dict=scores,
    title='Faithfulness Score: Accurate vs. Hallucinated Answer',
    ylabel='Faithfulness Score'
)
```

**What the chart shows:**

The hallucinated answer receives a much lower faithfulness score. The claim about "CFO approval" is unsupported by the context, which drags the overall score down.

**Improving faithfulness**

- **Strengthen the prompt:** Use explicit instructions like "Answer only using the information in the context below. If the answer is not in the context, say 'I don't know.'"
- **Use a different LLM:** Some LLMs are inherently more prone to hallucination than others. Consider using a model known for its grounding.
- **Post-process the answer:** Implement a verification step where another LLM checks if the answer is supported by the context.

**Answer correctness****What it measures:**

Answer Correctness is the *ultimate* user-facing metric.

It answers: *How correct and complete is the final answer compared to the ground truth?*

**Why it matters:** This is what the user sees. A high answer correctness score means your system is delivering value.

Ragas combines semantic similarity (does the answer mean the same thing?) and factual correctness (are the specific facts right?) into a single score. It uses an F1-like calculation over overlapping facts. Factual correctness quantifies the factual overlap between the generated answer and the ground truth answer. This is done using the concepts of:

Answer correctness is calculated as follows:

$$\text{F1 Score} = \frac{|\text{TP}|}{(|\text{TP}| + 0.5 \times (|\text{FP}| + |\text{FN}|))}$$

Where:

- **TP (True Positive):** Facts or statements that are present in both the ground truth and the generated answer.
- **FP (False Positive):** Facts or statements that are present in the generated answer but not in the ground truth.
- **FN (False Negative):** Facts or statements that are present in the ground truth but not in the generated answer.

**Seeing it in action**

Let's compare a perfect answer, a partially correct answer, and a wrong answer.

```
# Ground truth
ground_truth = "4 tasks are due today: Finalize Q3 OKRs, Onboard new team member, Update project roadmap, Write thank-you letter."

# Define test cases
test_cases = [
```

```

    {
        "answer": "There are 4 tasks due today: Finalize Q3 OKRs, Onboard
new team member, Update project roadmap, and Write thank-you letter to
penpal.",
        "label": "Perfect Answer\n(All facts correct)"
    },
    {
        "answer": "One task is due today: Finalize Q3 OKRs.",
        "label": "Partially Correct\n(1 fact correct, 3 missing)"
    },
    {
        "answer": "No tasks are due today.",
        "label": "Wrong Answer\n(0 facts correct)"
    }
]

# Evaluate each case
scores = {}
for case in test_cases:
    dataset = Dataset.from_dict({
        "question": [question],
        "answer": [case["answer"]],
        "ground_truth": [ground_truth]
    })
    result = evaluate(dataset, metrics=[answer_correctness],
llm=ragas_llm, embeddings=ragas_embeddings)
    scores[case["label"]] = result["answer_correctness"]

# Generate the chart using our reusable function
plot_metric_comparison(
    scores_dict=scores,
    title='Answer Correctness Score for Different Answer Qualities',
    ylabel='Answer Correctness Score'
)

```

### What the chart shows:

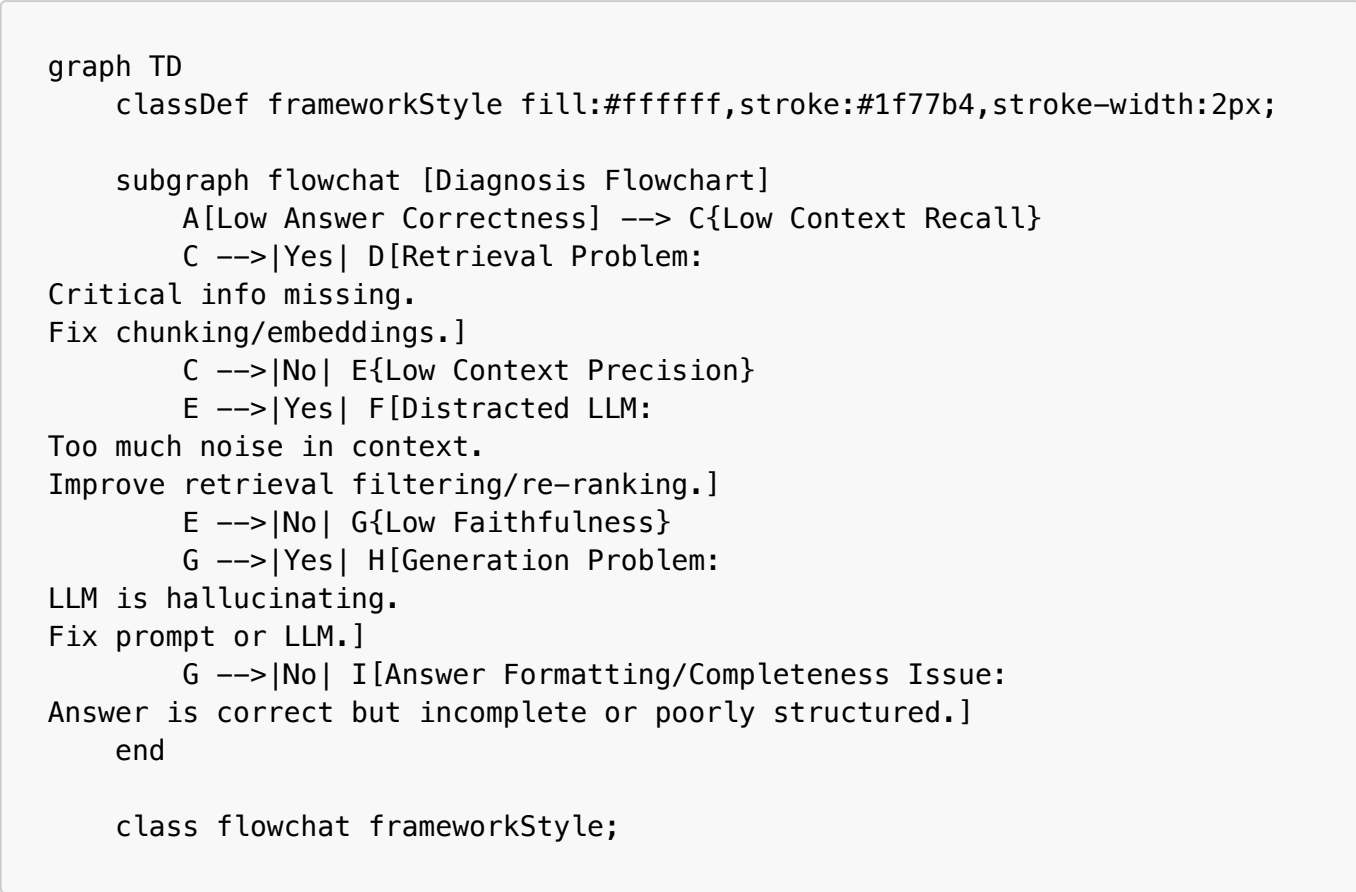
The score directly reflects the quality of the final output, rewarding completeness and penalizing omissions or errors.

### Improving answer correctness

- **Fix upstream issues:** Answer correctness is often a symptom. Use the other metrics to diagnose the root cause (e.g., low recall).
- **Improve prompting:** Use prompts that encourage structured, complete responses (e.g., "List all tasks due today.").
- **Enable multi-step reasoning:** Use a more powerful LLM or prompt it to "think step by step" to ensure it doesn't miss details.

### Actionable insights from metrics

The real power of evaluation comes from combining these metrics. Looking at them in isolation can be misleading. The magic is in the diagnosis.



So, in summary, we can distill what we've learnt into a simple table.

Metric	High Value (≥0.8)	Medium Value (0.5–0.8)	Low Value (<0.5)< th>
Context Recall	Retriever found most relevant info	Some key info was missed	Critical info missing – fix retrieval
Context Precision	Retrieved chunks are highly relevant	Some noise in retrieved context	Lots of irrelevant context – LLM distracted
Faithfulness	Answer sticks to retrieved context	Minor hallucinations or omissions	Major hallucinations – fix prompt or LLM
Answer Correctness	Answer is factually correct and complete	Partially correct or incomplete	Answer is wrong – check entire pipeline
Key Insight: If <b>Context Recall</b> is low but <b>Faithfulness</b> is high, the LLM isn't lying — it's just misinformed. Fix the retrieval (chunking, embeddings), not the generation.			

## Other Evaluation Frameworks

While Ragas is excellent for automated, LLM-powered evaluation, it’s not the only tool in the box. Let’s explore a few other popular frameworks — each with its own strengths.

## Ragas (Our choice)

```
graph TD
    classDef frameworkStyle fill:#ffffff,stroke:#1f77b4,stroke-width:2px;

    subgraph Ragas_Framework [Ragas Framework]
        direction TB

        subgraph Ragas_Evaluator [Ragas Evaluator]
            direction LR
            Judge[LLM Judge] -->|Scores| CR[Context Recall]
            Judge -->|Scores| CP[Context Precision]
            Judge -->|Scores| F[Faithfulness]
            Judge -->|Scores| AC[Answer Correctness]
        end

        end

        Q[User Question] --> Ragas
        A[Generated Answer] --> Ragas
        C[Retrieved Context] --> Ragas
        GT[Ground Truth] --> Ragas
        Ragas --> Judge
    end

    class Ragas_Framework frameworkStyle;

    style Judge fill:#e1f5fe,stroke:#1f77b4
    style Ragas stroke:#1f77b4,stroke-width:2px
```

Ragas uses a separate LLM as a "judge" to automatically score your RAG system's output. It requires a ground truth for comparison and provides a quick, holistic performance assessment.

Ragas is a popular, open-source framework specifically designed for evaluating RAG applications. Its primary strength lies in its suite of RAG-tailored metrics, such as context recall, context precision, faithfulness, and answer relevancy, which directly address the common failure points in a retrieval-augmented system. By using another LLM as a judge, Ragas can provide nuanced scores on the quality of the generated answer and its grounding in the retrieved context. This makes it an excellent choice for getting a quick, automated assessment of your RAG pipeline's overall health. However, because it relies on an LLM evaluator, the scoring can sometimes feel like a "black box," making it difficult to understand precisely why a particular score was given, which can complicate debugging efforts.

## TruLens

```
graph LR
    classDef frameworkStyle fill:#ffffff,stroke:#1f77b4,stroke-width:2px;

    subgraph TruLens_Framework [TruLens Framework]
        A[User Query] --> B[TruLens Monitor]
        B --> C[Chunking & Retrieval]
    end

    class TruLens_Framework frameworkStyle;
```

```

B --> D[Prompt Assembly]
B --> E[LLM Generation]
B --> F[Final Answer]

subgraph TruLens Dashboard
  G[Log: Query]
  H[Log: Retrieved Chunks]
  I[Log: Final Prompt]
  J[Log: LLM Call]
  K[Log: Answer]
  G --> H --> I --> J --> K
end

B --> G
B --> H
B --> I
B --> J
B --> K

end

class TruLens_Framework frameworkStyle;

%% style TruLens Monitor stroke:#2e7d32,stroke-width:2px
%% style TruLens Dashboard fill:#f1f8e9,stroke:#2e7d32

```

TruLens acts as a flight recorder, capturing every step of your RAG pipeline. This makes it ideal for deep debugging and understanding exactly where a failure occurred.

TruLens distinguishes itself by focusing on observability and traceability. Instead of just providing a final score, TruLens acts like a flight recorder for your LLM application, capturing and logging every step of the process, from the initial query to the final answer, including the intermediate prompts, retrieved context, and model calls. This level of detail is invaluable for debugging complex issues. Its integration with frameworks like LlamaIndex allows developers to set up feedback functions to programmatically evaluate qualities like groundedness and relevance at various stages. While it may have fewer built-in metrics than some competitors, its ability to provide a comprehensive view of the entire pipeline makes it a powerful tool for developers who need to understand not just if something is wrong, but exactly where and why.

## DeepEval

```

graph TD
  subgraph DeepEval_Framework
    T[Define Test Cases] -->|Question, Ground Truth| DE[DeepEval]
    DE -->|Run Tests| A1[Test: Answer Correctness]
    DE -->|Run Tests| A2[Test: Faithfulness]
    DE -->|Run Tests| A3[Test: Context Recall]

    A1 -->|Pass/Fail| R[CI/CD Pipeline]
    A2 -->|Pass/Fail| R
    A3 -->|Pass/Fail| R
  end

```

```
end

style DE fill:#ffecb3,stroke:#f57c00,stroke-width:2px
style R fill:#e8f5e8,stroke:#2e7d32
```

DeepEval treats RAG evaluation like software testing. You define assertions, and it runs automated tests, making it ideal for preventing regressions in production.

DeepEval takes a software engineering approach to LLM evaluation, positioning itself as "Pytest for LLMs". It encourages a test-driven development (TDD) mindset, where you write evaluation tests before or alongside your RAG application. This framework excels in CI/CD environments, allowing you to define strict pass/fail criteria (e.g., "faithfulness > 0.9") to prevent regressions. DeepEval supports a wide range of metrics for various LLM tasks, not just RAG, making it a versatile choice for teams building multiple types of LLM-powered features. Its structured, assertion-based testing model provides a high degree of control and automation for maintaining quality in a production setting.

### Custom evaluation frameworks

Custom evaluation frameworks are necessary when the standard metrics don't capture your application's unique requirements. As discussed, this could involve checking for the inclusion of specific links, ensuring a response follows a mandated workflow, or even assessing the emotional tone of the answer. While building a custom solution requires the most effort, it ensures that your evaluation is perfectly aligned with your business goals and user experience standards. The key insight is that these frameworks are not mutually exclusive; a robust evaluation strategy often involves using a combination, such as Ragas for broad automated scoring, TruLens for deep dives into problematic cases, and custom rules to enforce critical business logic.

### Framework comparison

Feature	Ragas	TruLens	DeepEval	Custom Frameworks
Best for	Automated, end-to-end metric scoring for RAG systems.	Observability, debugging, and tracing the entire RAG pipeline.	Test-driven development (TDD) and CI/CD integration.	Enforcing specific business logic and domain-specific rules.
Strengths	Lightweight, easy integration, metrics specifically designed for RAG (e.g., context recall, faithfulness), uses LLM-as-a-judge for nuanced scoring.	Excellent observability with a visual dashboard, tracks every step of the pipeline, strong integration with LangChain and LlamaIndex for real-time feedback.	Designed like a software testing framework (e.g., pytest), allows for pass/fail assertions and automated regression testing in production.	Maximum flexibility to define unique metrics (e.g., policy compliance, emotional tone) that off-the-shelf tools cannot capture.

Feature	Ragas	TruLens	DeepEval	Custom Frameworks
Weaknesses	Requires a capable evaluator LLM; can be a "black box" making it hard to debug low scores; metrics may not be self-explanatory.	Has fewer built-in metrics compared to other frameworks; more setup overhead than simpler tools.	May require more configuration for complex custom logic; focused on validation rather than deep pipeline analysis.	High development and maintenance cost; requires significant engineering effort to build and scale.
Use Case	Quickly getting an automated score on a RAG pipeline's performance to identify high-level issues.	Deeply debugging a failing pipeline by inspecting intermediate steps, prompts, and retrieved context.	Automating quality checks in a production environment, ensuring new updates don't break existing functionality.	Validating that an AI response adheres to strict business rules, such as including a required link or following a specific customer service protocol.

## Best practices for Establishing Evaluation Frameworks

You now have a powerful toolkit: manual inspection to diagnose problems, automated metrics from Ragas to measure performance at scale, and a suite of frameworks to choose from. But the real challenge isn't just using these tools—it's building a sustainable, effective evaluation practice that drives continuous improvement.

Think of it like maintaining a high-performance engine. You can have the best diagnostic tools in the world, but if you don't have a regular maintenance schedule, skilled mechanics, and a clear log of what's been fixed, the engine will eventually fail.

Here's how to build a robust evaluation system for your RAG applications.

### Involve domain experts early

The single most important factor in a successful evaluation strategy is human expertise. No LLM judge, no matter how sophisticated, can replace the nuanced understanding of a domain expert.

**Why it matters:**

Only a person who lives and breathes your business can define what "correct" truly means. Is a "high-priority" task more urgent than a "due today" task? Does the answer need to include a link, a deadline, or an approval process?

How to do it:

- **Co-create ground truth:** Don't write your ground\_truth answers alone. Sit down with an expert (e.g., your HR manager for onboarding tasks, your project lead for roadmap questions) and craft the perfect response together.
- **Identify edge cases:** Experts know the tricky scenarios. "What if a task is overdue but marked 'Not Started'?" or "How should we handle a question about a task that was just deleted?"
- **Audit the metrics:** Regularly review low-scoring cases with an expert. They can confirm if the metric is right or if the ground truth needs updating.



**Pro Tip:**

Treat your domain expert as a co-pilot, not a validator. Their insights should shape your evaluation criteria from day one.

## Build your test suite from real user queries

Your evaluation dataset is the foundation of your quality assurance. If it's built on artificial or hypothetical questions, it won't reflect real-world performance.

**Where to find real queries:**

- **Chat logs:** Mine conversations from your support bot, internal chat tools, or early user testing.
- **Search history:** Look at what users are searching for in your knowledge base or wiki.
- **Support tickets:** These are goldmines of user pain points and ambiguous phrasing.

Focus on high-impact questions:

**Prioritize queries that are:**

- **Frequent:** Questions that come up all the time.
- **High-stakes:** Questions about deadlines, policies, or financial information.
- **Ambiguous:** Queries with multiple interpretations (e.g., "What's due this week?" vs. "What's due today?").


This ensures your evaluation effort is focused on the areas that matter most to your users.

## Use a Mix of Evaluation Methods

No single tool gives you the complete picture. The most effective teams use a layered approach, like a diagnostic toolkit:

Method	Purpose	Strengths	Limitations
Manual Inspection	Visually verify retrieval & output quality	Fast, intuitive, reveals formatting issues	Not scalable, subjective
TruLens	Trace and debug full RAG pipeline	Shows prompt, context, and LLM call	Doesn't score quality
Ragas	Quantify performance with metrics	Objective scoring (correctness, faithfulness)	Requires ground truth

Method	Purpose	Strengths	Limitations
LLM-as-a-Judge	Automate answer quality assessment	Can evaluate tone, relevance, completeness	May hallucinate evaluations
A/B Testing	Compare prompt or retrieval changes	Measures real user impact	Needs traffic and metrics

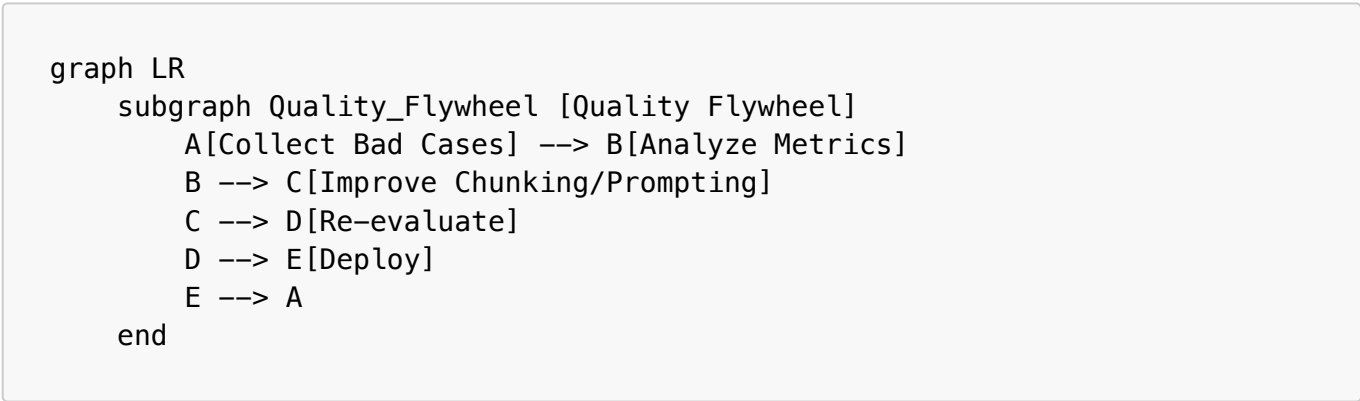
 **Best Practice:** Combine **TruLens for debugging** + **Ragas for scoring** + **manual checks for edge cases**.

Example workflow:

1. Use Ragas for a daily automated run to catch performance drops.
2. When a score is low, use TruLens to dive into the logs and see exactly what went wrong.
3. Write a DeepEval test to prevent that specific failure from happening again.
4. For a new feature (e.g., a refund policy), write a custom evaluator to ensure the answer always includes the required link.

## Close the feedback loop

Evaluation isn’t a one-off task; it’s a continuous cycle of improvement. The goal is to create a flywheel where each iteration makes your system better.



### Implementing the quality flywheel

- **Collect:** Gather failing test cases, user feedback, and low-scoring queries.
- **Analyze:** Use your metrics to diagnose the root cause (e.g., low recall = fix retrieval).
- **Improve:** Make a targeted change (e.g., adjust chunk size, rewrite a prompt).
- **Re-evaluate:** Run your test suite to see if the fix worked.
- **Deploy:** Release the improved version.
- **Repeat:** The cycle continues, building momentum toward a more reliable system.

## Treat evaluation as a product

Your evaluation framework is just as important as your application code. It should be treated with the same rigor.

- **Version control:** Store your test cases, ground truths, and evaluation scripts in Git alongside your main codebase.

- **Document everything:** Explain why each test case exists and what it's designed to catch.
- **Monitor trends:** Track your key metrics (e.g., average answer correctness) over time. A sudden drop is an early warning sign.
- **Automate in CI/CD:** Integrate your evaluation suite into your deployment pipeline. If the faithfulness score drops below 0.8, the build should fail.

## What's next?

---

### Quiz yourself!

► 1. Your RAG system returns a correct answer, but the `context_recall` score is very low. What is the most likely explanation?

- A) The LLM is distracted by irrelevant chunks
- B) The answer is incomplete
- C) The retrieved context does not contain the necessary information
- D) The LLM is hallucinating

View answer →

✅ **Correct answer:** C) The retrieved context does not contain the necessary information

📝 **Explanation :**

- Low `context_recall` means the correct information wasn't retrieved. If the answer is still correct, the model must have hallucinated the facts — a dangerous failure mode. This reveals a critical gap: your retrieval is broken, but the LLM is covering it up.

## Takeaways

- **Why evaluate RAG systems?**
  - **RAG is a pipeline, not a single step** — it involves chunking, retrieval, context assembly, and generation.
  - **When something goes wrong, it's not always clear where** — did the model hallucinate, or was the right context never retrieved?
  - **Evaluation separates symptoms from root causes:**
    - A wrong answer could be due to bad retrieval, poor prompting, or model limitations.
  - **Without evaluation, you're shipping blind** — you can't improve what you don't measure.
  - **Evaluation enables trust, scalability, and iteration** — essential for production AI.
- **Manual evaluation**
  - **Visual inspection is the first step**
  - **Look for red flags:**

- Missing critical keywords
  - Poor chunk coverage
  - Garbled text
- **Manual debugging builds intuition** — it helps you understand how retrieval and generation interact.
- **Always check the source nodes** — verify that the model had access to the right information before blaming the LLM.
- **Automated evaluation with Ragas**
  - **Ragas provides quantitative metrics** — turning subjective quality into measurable scores.
  - Key metrics:
    - **Answer Correctness**: How factually accurate is the response?
    - **Context Recall**: Does the retrieved context contain all relevant facts?
    - **Context Precision**: Are the retrieved chunks relevant, or full of noise?
    - **Faithfulness**: Does the answer derive from the context, not hallucination?
  - **Ragas turns evaluation into a CI/CD process** — test, score, and deploy with confidence.
- **Building a realistic evaluation dataset**
  - **Use real user queries** — not artificial ones:
    - Chat logs
    - Search history
    - Support tickets
  - **Include ambiguous, vague, or multi-part questions** — these are the hardest and most revealing.
  - **Define clear ground truth** — what is the *correct* answer for each query?
  - **Treat domain experts as co-pilots** — their insights should shape your evaluation criteria from day one.
  - **Start small, but start early** — even 10 high-quality test cases can reveal systemic issues.
- **Interpreting metrics**
  - **Metrics are diagnostic tools**, not report cards:
    - Low **context\_recall** → fix retrieval (better chunking, embeddings, or search)
    - Low **answer\_correctness** → improve prompting or use a stronger LLM
    - High **context\_precision**, low **answer\_correctness** → the model is ignoring good context
  - **Never look at metrics in isolation** — combine them to find root causes.
  - **Example**: Correct answer + low recall = hallucination (dangerous!)
  - **Evaluation is iterative** — measure, fix, re-measure.
- **Actionable insights from evaluation**

- **Fix upstream issues** — answer correctness is often a symptom. Use other metrics to diagnose:
  - Retrieval failure? → Improve chunking or embeddings
  - Prompt issues? → Add structure, examples, or chain-of-thought
- **Improve prompting** — use instructions like:
  - “List all tasks due today.”
  - “Answer in bullet points.”
- **Enable multi-step reasoning** — use reasoning models (e.g., Qwen-plus with thinking mode) to reduce missed details.
- **The real power of evaluation** is not in the scores — it’s in the **insights** that drive better system design.