Evaluating RAG Systems

A Product Manager's Practical Guide to Measuring Retrieval Quality

Brad Hinkel · April 2026

Why This Guide Exists

Most RAG teams test before they ship. They run queries, read the outputs, confirm the results look reasonable. Some compute faithfulness or relevancy scores using tools like RAGAS. What most teams don't do is systematically evaluate retrieval configuration — testing one variable at a time against a ground-truth dataset to understand which design decisions actually drive quality, cost, and latency. That gap is where the largest optimization opportunities hide.

This guide is written for product managers who own or influence RAG-powered features and want a practical framework for evaluating retrieval quality. It covers when to invest in evaluation, how to build a ground-truth test set, which metrics matter and why, and how to structure a configuration sweep that isolates the impact of each design decision. The framework is general-purpose; examples throughout are drawn from a D&D content generation system I built and deployed to production, where this evaluation approach improved retrieval precision by 85% and reduced token cost by 65%.

When to Invest in RAG Evaluation

Not every RAG project needs a formal evaluation framework. If you are building a quick prototype or internal tool where quality is "good enough" and cost is negligible, a vibe check may genuinely suffice. But there are clear signals that you've crossed the threshold where systematic evaluation pays for itself:

- **You are making configuration decisions with cost or latency implications.** Chunk size, embedding model, top-k, reranking — each of these affects your per-query cost and response time. Without measurement, you're guessing at the tradeoff.
- **You are going to production.** Once real users depend on your system, you need a baseline you can regression-test against. A numerical baseline lets you answer "did this change make things better or worse?" without re-reading hundreds of outputs.
- **You're choosing between models, chunk strategies, or retrieval methods.** A/B comparisons require a shared yardstick. Without one, the team will debate subjective impressions of output quality instead of looking at numbers.
- **Your corpus is large or heterogeneous enough that spot-checking doesn't cover it.** If your corpus has multiple categories, document types, or domains, a few manual tests will miss failure modes. A structured test set covers the surface area systematically.

If none of these apply, skip the framework and ship. If any of them do, the investment is modest — typically 15–30 hours of setup work — and the return compounds across every tuning decision you make afterward.

Step 1: Build the Ground-Truth Test Set

The foundation of any evaluation framework is a set of questions with known correct answers. This is the most labor-intensive step and also the most valuable. Without it, every metric you compute is measuring against nothing.

What Makes a Good Test Set

Each question should be tied to a specific document (or set of documents) in your corpus so that retrieval correctness can be scored objectively. Aim for 80–100 questions distributed across your content categories and across six question types that probe different retrieval failure modes:

Question Type	What It Tests	Example
Direct Lookup	Can the system find a specific item by name?	"What are the properties of Widget X?"
Attribute Filter	Can it retrieve items matching specific attributes?	"List all premium-tier items in category Y."
Semantic Similarity	Can it find thematically similar items?	"Find items with fire-related features."
Cross-Category	Can it connect information across categories?	"Which people are associated with location Z?"
Boundary / Negative	Does it handle queries outside its corpus?	"What are the specs for an item not in the corpus?"
Generation Quality	Is generated content grounded in retrieved context?	"Generate a new item in category X with trait Y."

The mix matters. If you only test direct lookups, you'll optimize for exact-match retrieval and miss semantic or cross-category failures. If you only test generation quality, you won't know whether quality problems originate in retrieval or in the LLM.

Ground-Truth Format

Each entry should record the question, the expected answer, the expected source documents, the category, the question type, and a difficulty rating. Store these as structured JSON. The investment in this structure pays off immediately: your evaluation harness can score retrieval by comparing retrieved document IDs against expected source documents, with no manual review required.

Tips for Writing Good Questions

- **Be specific enough to have one right answer.** "What curse does the Blade of Shadows carry?" is testable. "Tell me about dark weapons" is not.
- **Include hard negatives.** If your corpus has fire swords and fire staves, ask about fire weapons. Does the system retrieve the right type?

- **Test boundary conditions.** Ask about items that span chunk boundaries. Can retrieval reconstruct the full item?
- **Include unanswerable questions.** Ask about content that doesn't exist in your corpus. A good system should acknowledge this gracefully rather than hallucinate.

Step 2: Choose Your Metrics

RAG evaluation metrics fall into three categories: retrieval metrics that measure whether you found the right context, generation metrics that measure whether the LLM used that context well, and operational metrics that measure cost and speed. You need all three. Retrieval metrics are the most discriminative for tuning decisions; generation metrics are better suited to regression detection; operational metrics keep you honest about what you're spending.

Metric	What It Tells You	PM Decision It Informs
Precision@k	Of the k chunks retrieved, how many were actually relevant?	Is the context window clean or noisy?
Recall@k	Of all relevant chunks in the corpus, how many did retrieval find?	Is the system missing important information?
MRR	How high does the first relevant result appear in the ranking?	Does the top result alone usually suffice?
Faithfulness	Is the generated answer grounded in the retrieved context? (LLM judge)	Hallucination risk for your domain
Answer Relevancy	Does the generated answer actually address the question? (LLM judge)	Are you answering the right question?
Tokens / query	How many tokens are consumed per generation call?	Cost per query and context window budget
E2E Latency	Wall-clock time from query to finished answer	User experience and timeout risk

A common mistake is to rely solely on LLM-judge scores like faithfulness and relevancy. In my experience, these scores are broadly stable across configurations — they tell you when something is badly broken, but they don't help you distinguish between a good config and a slightly better one. Retrieval metrics, especially Precision@k, are where the actionable signal lives.

Metric Definitions in Plain English

If you are setting up evaluation for the first time, you need to understand not just what each metric measures but how it is calculated. Here is each metric with an intuitive example and the formula behind it.

Precision@k. You ask a question and retrieve 5 chunks. A human (or your ground-truth dataset) says 3 of those 5 were actually relevant. Precision = 3 / 5 = 0.60. The formula is: relevant chunks retrieved

divided by k (the total number of chunks retrieved). High precision means the context window is clean; low precision means the LLM is wading through noise.

Recall@k. For the same question, suppose there are 4 relevant chunks in the entire corpus. Your retrieval found 3 of them. Recall = 3 / 4 = 0.75. The formula is: relevant chunks retrieved divided by total relevant chunks in the corpus. High recall means you are not missing important information; low recall means critical context is being left out. Note: count distinct relevant documents covered, not duplicate chunks from the same document.

MRR (Mean Reciprocal Rank). For each question, find the position of the first relevant result. If the first relevant chunk is at position 1, the reciprocal rank is 1/1 = 1.0. If it is at position 3, the reciprocal rank is 1/3 = 0.33. MRR is the average of these reciprocal ranks across all questions. High MRR means the system consistently puts relevant results at the top; low MRR means users (or the LLM) have to dig deeper to find useful content.

NDCG@k (Normalized Discounted Cumulative Gain). This metric cares about rank order, not just presence. A relevant chunk at position 1 gets more credit than the same chunk at position 5. The "discount" is logarithmic: each lower position contributes less. The score is then normalized against the ideal ranking (where all relevant chunks appear at the top). NDCG of 1.0 means perfect ranking; lower scores mean relevant results are scattered across positions rather than concentrated at the top.

Faithfulness (LLM judge, 0–1). A second LLM reads the generated answer and the retrieved context, then scores whether every claim in the answer is supported by the context. A score of 1.0 means the answer is fully grounded; lower scores indicate hallucination — the LLM invented information not present in the retrieved chunks. This is the primary guardrail against making things up.

Answer Relevancy (LLM judge, 0–1). A second LLM evaluates whether the generated answer actually addresses the question that was asked. A perfectly grounded answer can still miss the point. For example, if you ask about a weapon's curse and the system returns a detailed description of the weapon's physical appearance, faithfulness may be high (the appearance is in the context) but answer relevancy will be low (you asked about the curse).

Tokens per query. The total number of input and output tokens consumed by the LLM for a single generation call. This is a direct cost proxy: more tokens = higher API cost. It also correlates with latency (larger context windows take longer to process) and inversely correlates with precision (noisy, irrelevant chunks inflate the token count without improving quality).

End-to-end latency. Wall-clock time from when the user submits a query to when the final answer is returned. This includes embedding the query, searching the vector store, sending context to the LLM, and receiving the response. It is what the user actually experiences. Useful to break down into sub-components (embedding latency, retrieval latency, generation latency) when diagnosing bottlenecks.

Step 3: Structure the Configuration Sweep

The most important principle is to change one variable at a time. This sounds obvious, but in practice most teams change multiple things between runs because it feels more efficient. It isn't. When you

change two variables simultaneously, you cannot attribute the result to either one. If quality goes up, you don't know which change helped. If quality goes down, you don't know which change hurt. You've spent the compute without learning anything.

Understanding the Variables

Before running the sweep, it helps to understand what each variable actually controls in the pipeline and why changing it matters.

Metadata filtering. Your corpus likely has categories, types, or other structured labels. Metadata filtering restricts retrieval to only documents matching a given label (for example, only searching "weapons" when the user asks for a weapon). Without it, the vector store searches the entire corpus and may return semantically similar but categorically wrong results — a monster description that happens to mention swords when you wanted a sword entry. This is typically the cheapest variable to test and often the highest-leverage.

Chunk size. Before documents are embedded and stored in the vector database, they are split into chunks — fixed-length segments measured in tokens. Chunk size controls the granularity of what gets retrieved. Larger chunks (1024 tokens) carry more context per retrieval but include more noise; smaller chunks (256 tokens) are more targeted but may split important information across chunk boundaries. Overlap (the number of tokens shared between adjacent chunks) mitigates boundary-splitting: an overlap of 50 tokens means the end of one chunk and the beginning of the next share 50 tokens of text.

Embedding model. The embedding model converts text into numerical vectors that capture semantic meaning. Different models produce different vector spaces, which means the same query may retrieve different chunks depending on which model you use. Newer models (like OpenAI's text-embedding-3-small) often outperform older ones (like ada-002) on benchmarks, but the improvement varies by domain. Open-source models (like BGE) avoid API costs but may require GPU acceleration to run at acceptable speed. The tradeoff is typically quality versus cost and latency.

Search method. Vector search finds chunks whose embeddings are closest to the query embedding in vector space — it excels at semantic similarity. BM25 is a keyword-matching algorithm that scores documents based on term frequency — it excels when the query contains specific, rare terms. Hybrid search combines both scores, typically with a tunable weight. The hypothesis is usually that hybrid captures both semantic and keyword matches, but in practice the benefit depends on whether your queries contain distinctive keywords that embeddings miss.

Top-k. The number of chunks retrieved per query. Top-3 means you send 3 chunks to the LLM as context; top-10 means you send 10. More chunks increase the chance of including relevant information (higher recall) but also increase noise, token cost, and latency. The right value depends on how much relevant information is scattered across your corpus: if most questions are answerable from 1–2 chunks, top-3 suffices; if the answer requires synthesizing multiple sources, you may need top-10.

Reranking. A reranker is a second model that re-scores the retrieved chunks before they are sent to the LLM. The initial vector search is fast but approximate; a cross-encoder reranker reads each chunk alongside the query and produces a more accurate relevance score. The tradeoff is latency: reranking

adds a second model inference step. The benefit depends on domain fit — most off-the-shelf rerankers are trained on web search data and may not generalize to specialized corpora.

A sequential sweep carries the winner forward from each phase into the next. The recommended order starts with the cheapest, highest-leverage variables and works toward diminishing returns:

Phase	Variable	Hold Everything Else Constant	What You Learn
0	Baseline	Current production config	Your starting point
1	Metadata filtering	Baseline chunk, embedding, top-k	Value of scoping retrieval
2	Chunk size	Best filter + baseline rest	Optimal context granularity
3	Embedding model	Best filter + chunk + baseline rest	Is a better model worth the cost?
4	Search method	Best filter + chunk + embedding	Does hybrid or BM25 help?
5	Top-k	All previous winners	Context budget tradeoff
6	Reranking	All previous winners	Is latency cost justified?

This order is deliberate. Metadata filtering and chunk size are typically the highest-leverage and cheapest-to-test variables. Embedding model and search method are moderate. Top-k and reranking are refinements that matter less in most domains. Testing in this order means you spend most of your compute on the decisions that matter most.

A Practical Caveat

Sequential sweeps create a dependency chain: each phase uses the previous phase's winner as its held-constant config. If you make a mistake in an early phase (for example, using a placeholder chunk size before the chunk-size phase has run), that error propagates through every subsequent phase. The fix is simple: finalize all phase configs before running anything, and lock the winner from each phase before moving to the next. In my D&D project, I learned this the hard way when Phases 4–6 ran with a placeholder chunk size of 512 instead of the Phase 2 winner of 256. The results were still valid for within-phase comparisons, but couldn't be directly compared to the Phase 2 winner.

Step 4: Interpret the Results

Once you have a completed sweep, the results often contain surprises. Here are patterns I've seen and what they mean for product decisions.

Precision and cost move together. In the D&D project, switching from 1024-token to 256-token chunks improved Precision@k by 78% and cut token usage by 61%. Smaller chunks are more targeted, so the context window contains less noise. This is counterintuitive — PMs often assume more context is better — but in structured-data domains, less context is often both cheaper and higher-quality.

Pre-trained components carry domain assumptions. A cross-encoder reranker trained on web search queries added 79% latency to the D&D pipeline with zero quality gain. The model didn't understand fantasy-lore queries. Before adding any pre-trained component, ask: was this trained on data that looks like mine?

Metadata filtering has outsized impact on generation quality. In the D&D project, filtering retrieval by content category improved faithfulness and answer relevancy more than any change to the embedding model or search method. Scoping retrieval to the correct category eliminates cross-category noise from the context window, which helps the LLM stay grounded.

"Hybrid search didn't help" is a narrower finding than it sounds. BM25-style keyword matching didn't improve results in a corpus where queries are conceptual and vocabulary is homogeneous. But a different kind of hybrid — using strongly-typed attributes (rarity, category, type) as structured pre-filters before vector search — would likely be more effective and is architecturally simpler. Evaluation tells you exactly what you tested, not the more general thing it sounds like.

LLM-judge scores plateau quickly. Faithfulness and answer relevancy were broadly stable across all configurations in the D&D sweep. They're valuable as regression detectors — if faithfulness drops below 0.85, something is wrong — but they're not sensitive enough to guide optimization. Use retrieval metrics for tuning; use judge scores for guardrails.

Getting Started: A Minimal Checklist

If you are a PM looking to introduce RAG evaluation into your team's workflow, here is the minimum viable version:

1. **Write 80–100 ground-truth questions** tied to specific source documents in your corpus. Distribute them across categories and question types. This is the hardest step and the most important one.
2. **Instrument your pipeline to report Precision@k, Recall@k, tokens per query, and end-to-end latency.** These four metrics cover retrieval quality, cost, and speed. Add LLM-judge scores (faithfulness, relevancy) if your team has the tooling.
3. **Run a baseline evaluation** against your current production config. This is your anchor. Every future change is measured as a delta from this baseline.
4. **Sweep one variable at a time,** starting with metadata filtering and chunk size. Carry the winner forward into each subsequent phase.
5. **Document your findings** as a comparison table with deltas from baseline. This becomes your team's shared reference for why the system is configured the way it is.

The total effort for a first evaluation pass is typically 15–30 hours, spread across building the test set, writing the harness, and running the sweep. The test set and harness are reusable; future sweeps take a fraction of the initial investment.

Closing Thought

RAG evaluation is not a research exercise. It is the mechanism by which a PM turns subjective output quality into measurable product decisions. The framework described here is deliberately simple — ground-truth questions, a handful of metrics, one variable at a time — because the leverage comes from the discipline, not from the sophistication of the tooling. Build the test set, run the sweep, and let the numbers tell you things your eyes can't.

This guide is drawn from a production RAG system for D&D content generation (dnd.bradhinkel.com). The full case study, including architecture details, results tables, and lessons learned, is available as a companion document.