Retrieval Evaluation Sets

Version Nov 2026 · verified 2026-11-14

Measure RAG instead of guessing — build a labelled question set, score retrieval with recall@k, MRR and nDCG, grade answers, and compare runs.

Contents

Mental model 3

What goes in the set 3

Labelling ground truth 5

Drafting questions with an LLM 7

Splits and versioning 8

Retrieval metrics 10

Ranking metrics: MRR and nDCG 12

Grading the answers 14

Comparing runs 16

Reading the failures 17

Further reading 19

Mental model

A RAG system fails in two separate places: retrieval misses the chunk that holds the answer, or generation misuses the chunks it was given. An eval set is a fixed list of questions with labels saying which documents answer each one, so you can score the two halves separately and see which one broke. A score only means something next to a baseline from the same set, so every change is a comparison, not a number.

What goes in the set

Question type	Tests
Single-fact lookup	Basic recall
Paraphrase, no shared words	Embedding quality
Exact identifier or code	Keyword or hybrid search
Needs two or more documents	Recall at larger k, fusion

Question type	Tests
Not answerable from the corpus	Refusing instead of inventing

Start with 50 to 100 real or realistic questions — from search logs, support tickets, or the people who will use the system — and cover every type above. Each item records the question, the documents that answer it and, if you will grade answers, a reference answer with the facts it must contain.

```
{
  "id": "q-0042",
  "question": "How long do refunds take?",
  "type": "lookup",
  "relevant_docs": [812],
  "reference": "Refunds post in 5-10 days.",
  "must_contain": ["5-10 business days"]
}
```

> **Tip:** Include questions the corpus cannot answer, and label them with no relevant documents. Without them you cannot measure whether the system says "I don't know" or makes something up.

Labelling ground truth

Label by	Survives re-chunking	Precision
Document ID	Yes	Coarse
Document ID + answer span	Yes	Fine
Chunk ID	No	Fine, until chunks change

The labels say which retrieved items count as relevant for a question. Label at the level you will score, and keep enough to re-derive it: a document ID plus the exact answer text or character span lets you map the label onto any chunking — a chunk is relevant if it contains

the span. Have a second person label a sample and compare; where two people disagree, the question is usually ambiguous.

```
def relevant_chunks(chunks, label):
    # chunks: dicts with doc_id, start, end
    doc, a, b = label  # answer span
    return {
        c["id"] for c in chunks
        if c["doc_id"] == doc
        and c["start"] < b and a < c["end"]
    }
```

> **Gotcha:** Chunk-ID labels are tied to one chunking. Change chunk size or overlap and the old IDs no longer exist, so the experiment you most want to run — comparing chunk sizes — scores against labels that point at nothing.

Drafting questions with an LLM

Step	Why
Generate from a sampled chunk	Known source = free label
Ask for user-style phrasing	Real users do not quote docs
Review every question by hand	Drop vague, wrong, duplicate
Mix with real user questions	Synthetic sets skew easy

An LLM can draft questions from chunks quickly, and the chunk it came from gives you the label. Have a person read every one: remove duplicates, questions only answerable by someone looking at the chunk (“what does this section say?”), and questions with more than one correct source the label misses.

```
Write one question a customer might ask
that this passage answers. Use your own
```

```
words: do not copy phrases from it.
Passage: {chunk}
```

> **Gotcha:** Questions written while looking at a chunk reuse its exact words, so keyword and vector search both find it easily and scores come out inflated. Real users phrase things differently; compare scores on synthetic and real questions separately.

Splits and versioning

Record per run	Why
Eval set version	Scores on different sets do not compare
Corpus snapshot	Documents added or removed change answers
Embedding model and version	A model change is a different system
Chunk size, overlap, k, filters, fusion weights	What you changed

Split the set: a tuning set you look at while choosing chunk size, k and weights, and a held-out set you score only to confirm the final choice. Tuning against the held-out set makes it one more tuning set, and the score stops predicting production. Store every run with its full configuration.

```
{
  "run": "2026-11-14-chunk500",
  "eval_set": "support-v3",
  "split": "tune",
  "corpus": "snapshot-2026-11-10",
  "model": "voyage-4",
  "chunk": { "size": 500, "overlap": 75 },
  "k": 10,
  "recall_at_10": 0.82
}
```

> **Warning:** Do not add a failing production question to the set and then tune until it passes without checking the rest. Version the set, and re-run the baseline whenever its contents change.

Retrieval metrics

Metric	Question it answers
Hit rate@k	Did any relevant item appear in the top k?
Recall@k	What fraction of the relevant items are in the top k?
Precision@k	What fraction of the top k is relevant?

Score each question, then average across the set. Recall@k is the headline number for RAG, because the model cannot use what was not retrieved: if recall@10 is low, no prompt change will fix the answer. Precision@k matters when context is expensive, or when irrelevant chunks distract the model.

```python
def recall_at_k(got, rel, k):
    return len(set(got[:k]) & rel) / len(rel)

def precision_at_k(got, rel, k):
    return len(set(got[:k]) & rel) / k

got = [4, 9, 2, 7, 5]   # ranked results
rel = {2, 5, 8}         # labelled relevant
recall_at_k(got, rel, 5)     # 2/3 ≈ 0.67
precision_at_k(got, rel, 5)  # 2/5 = 0.4
```

> **Gotcha:** "Recall@k" is often used to mean hit rate — at least one relevant item in the top k. With one relevant item per question the two are equal; with several they differ a lot. Say which you report, and skip questions with no relevant items in recall, since it divides by zero there.

Ranking metrics: MRR and nDCG

Metric	Rewards
Reciprocal rank	`1 / rank` of the first relevant result
MRR	Mean reciprocal rank across questions
nDCG@k	Relevant results high up, graded by how relevant

Recall ignores order inside the top k, but order matters when you truncate context or show results to people. MRR looks only at the first hit. nDCG sums each result's relevance discounted by `log2(rank + 1)` and divides by the best possible ordering, giving a score from 0 to 1 that also handles graded labels.

```
import math
def rr(got, rel):
    hits = [i for i, d in enumerate(got, 1)
            if d in rel]
    return 1 / hits[0] if hits else 0.0
```

```
def ndcg_at_k(got, rel, k):
    g = lambda i: 1 / math.log2(i + 1)
    dcg = sum(g(i) for i, d in
              enumerate(got[:k], 1) if d in rel)
    n = min(len(rel), k)
    return dcg / sum(map(g, range(1, n + 1)))

rr([4, 9, 2], {2, 5})            # 1/3
ndcg_at_k([4, 9, 2], {2, 5}, 3)  # ≈ 0.31
```

> **Note:** These definitions use binary relevance. For graded labels (0 to 3, say), use each item's grade as its gain in the DCG sum, as `sklearn.metrics.ndcg_score` does.

Grading the answers

Metric	Question it answers
Faithfulness	Is every claim in the answer supported by the retrieved chunks?
Answer correctness	Does the answer match the reference?
Context recall	Are the reference's claims all in the retrieved chunks?
Refusal rate on unanswerable	Does it say "I don't know"?

Score generation only after retrieval is acceptable, and keep the two reports separate. Faithfulness, as Ragas defines it, splits the answer into claims and divides the number supported by the retrieved context by the total. Checking `must_contain` facts with plain string or regex

matching is cheaper and more reliable than a model wherever it works; use an LLM judge for the rest.

```
def faithfulness(claims, supported):
    # supported: one bool per claim,
    # from a judge or a human
    if not claims:
        return None  # nothing to check
    return sum(supported) / len(claims)

faithfulness(["a", "b", "c"],
             [True, True, False])  # ≈ 0.67
```

> **Warning:** An LLM judge is a model with its own errors and bias. Spot-check a sample of its verdicts by hand, give it a way to say “cannot tell”, and re-check it whenever you change its prompt or model — a new judge makes old scores incomparable.

Comparing runs

Practice	Why
Fix a baseline first	Every number is a delta from it
Change one thing per run	Otherwise you cannot attribute the gain
Same set, same split, same judge	Only the system should differ
Check the noise	Small sets move a lot on one question

Knobs worth testing one at a time: chunk size, overlap, embedding model, k, filters, hybrid weights and reranking. With a small set, treat small differences as noise: at 50 questions one question is two points of hit rate, and the 95 percent margin on a score near 0.8 is about ±0.11.

```
import math
```

```
def margin(p, n, z=1.96):
    # normal approximation for a proportion
    return z * math.sqrt(p * (1 - p) / n)


margin(0.80, 50)    # ≈ 0.11
margin(0.80, 400)   # ≈ 0.04
```

> **Tip:** Compare runs per question, not just on the average. A change that fixes ten questions and breaks eight looks like a small win in the mean and hides eight regressions.

Reading the failures

Failure	Usual fix
Right document, answer split across chunks	Bigger chunks, more overlap, parent context
Exact term missed	Hybrid search with full text

Failure	Usual fix
Filter removed the answer	Fix metadata, check result counts
Retrieved but ranked below k	Reranking, larger first-stage k
Retrieved, answer still wrong	Prompt, or model — a generation problem

Averages tell you that something is wrong; failures tell you what. Read every failing question in a run, tag each with a cause, and count the tags. Most of the gain usually comes from fixing the one or two causes that recur, not from tuning everything a little.

```
q-0042  missed     answer split across chunks
q-0057  ranked 14  needs reranking
q-0081  retrieved  model ignored the chunk
q-0103  missed     doc not in corpus snapshot
```

Further reading

- Introduction to Information Retrieval — Evaluation of ranked results
- scikit-learn — ndcg_score
- Ragas — Available metrics
- Anthropic — Introducing contextual retrieval
- Anthropic — Demystifying evals for AI agents