ML Engineer Take-Home

Build a RAG System Over a Messy Knowledge Base

Take-Home (1-2 days) · ~10-14 working hours

The Pitch

Your company's internal knowledge base is a disaster. It's a mix of Markdown docs, PDF policy manuals, exported Slack threads, meeting transcripts, and code READMEs—some current, some outdated, some contradictory. Employees keep asking the same questions and getting wrong answers because nobody can find the right document.

You've been asked to build a **Retrieval-Augmented Generation (RAG) system** that employees can query in natural language. The VP of Engineering wants a working prototype in two days, a clear evaluation showing it actually works, and an honest assessment of where it fails.

What Makes This Task Hard

This is not a clean tutorial problem. The data is messy *by design*. There is no single correct architecture, chunking strategy, or embedding model. Every decision has tradeoffs, and we want to see how you navigate them.

We will evaluate your judgment and engineering rigor, not whether your system achieves a specific accuracy number. A well-reasoned system that scores 60% with clear error analysis beats a black-box system that scores 75%.

Deliverables

Submit a **Git repo** (or zip) containing:

- **Working code** — we will `docker compose up` (or `pip install + python run.py`) and query your system. If it doesn't run, it's an automatic reject.
- **Engineering notebook** (Markdown or PDF, 3-5 pages) — a chronological log of what you tried, what worked, what didn't, and why you made each major decision. Think lab notebook, not polished report.
- **Evaluation results** — metrics, outputs, and error analysis artifacts.

Use any LLM provider or local model. If using paid APIs, we'll reimburse up to $15. Open-source/local models are equally valid. Document your choice and reasoning.

THE DATA

What You're Working With

You'll receive a `knowledge_base/` directory containing ~150 documents from a fictional company called **Meridian Technologies**. The data is intentionally messy. Here is what's in it:

`docs/policies/*.md`

HR and engineering policies (PTO, oncall, deploy process, incident response). **Problem:** some are outdated—e.g., the PTO policy was updated in Q3 2024 but the old version is still in the folder with no deletion.

`docs/engineering/*.md`

Architecture docs, ADRs (Architecture Decision Records), runbooks. **Problem:** two ADRs contradict each other on the database migration strategy. One was superseded but never marked as such.

`docs/product/*.pdf`

Product specs and PRDs as PDFs. **Problem:** some are scanned images (not text-selectable), others are native PDF. Mixed quality.

`slack_exports/*.json`

Exported Slack threads from #engineering and #incidents channels. **Problem:** noisy—contains banter, emojis, half-conversations, and critical context buried in reply threads. Timestamps and user IDs included.

`meetings/*.txt`

Raw meeting transcripts (auto-generated, not hand-corrected). **Problem:** speaker diarization errors, filler words, partial sentences. But some contain critical decisions not documented anywhere else.

`code/README*.md`

READMEs from 8 internal repos. **Problem:** some reference deprecated endpoints or deleted services.

`meta/document_manifest.csv`

A CSV listing every document with metadata: title, author, last_modified date, department, status. **Problem:** ~15% of entries have incorrect or missing metadata. Some documents in the folder are not in the manifest. Some manifest entries point to files that don't exist.

> **Intentional data problems you must handle:** Contradictory documents (old vs new policy), missing metadata, documents not in manifest, near-duplicate files (same content with minor edits), OCR-quality PDFs, noisy Slack/meeting transcripts, and at least one document that is completely irrelevant (a personal recipe someone accidentally committed).

PART 1 — BUILD

Build the RAG Pipeline

Build a working RAG system that takes a natural-language question and returns an answer grounded in the knowledge base. The system must:

1A. Ingestion & Preprocessing

- Process all document types in the knowledge base (Markdown, PDF, JSON, TXT, CSV).
- Handle the messy data — **you decide** how. Strip noise from Slack? Ignore meeting transcripts? Merge near-duplicates? De-prioritize outdated docs? Every choice is valid if you justify it.
- Produce a chunked, embedded index ready for retrieval. Document your chunking strategy (size, overlap, whether you chunk by section/paragraph/semantic boundary, etc.) and why you chose it.

1B. Retrieval & Generation

- Implement retrieval (vector search, BM25, hybrid — your call).
- When the answer comes from a document, **cite the source** (filename + section/chunk). Employees need to verify answers.
- When the knowledge base contains **contradictory information**, the system must surface the contradiction rather than silently picking one version. This is a hard requirement.
- When the system doesn't know the answer, it must say so. **No hallucination is better than a confident wrong answer.**

1C. Interface

A simple CLI or web interface. Nothing fancy — we care about the pipeline, not the UI. But it should be easy for us to type a question and see an answer with sources.

> **Budget Constraint**
>
> You have a **$5 budget for your entire evaluation run** (Part 2). This means you need to think about embedding costs, LLM inference costs, and how many queries you can afford. If you use local models, state the hardware you ran on. This constraint is real and intentional — production ML always has a cost ceiling.

PART 2 — EVALUATE

Evaluate It Honestly

We provide a file `eval/questions.jsonl` containing **40 questions** that employees might actually ask. Each question has:

- **question** — the natural language query

- **gold_answer** — a human-written reference answer
- **gold_sources** — list of document filenames that contain the answer
- **category** — one of: `factual`, `procedural`, `contradictory`, `unanswerable`, `multi-doc`
- **difficulty** — easy / medium / hard

2A. Metrics — Design Your Own

We intentionally do **not** prescribe metrics. Part of this task is deciding how to measure whether your system works. You should implement **at least 3 metrics**, which might include (but are not limited to):

- **Retrieval quality** — did the right chunks get retrieved? (Recall@K, MRR, etc.)
- **Answer quality** — is the generated answer correct? (LLM-as-judge, ROUGE, semantic similarity, exact-match for factual questions, etc.)
- **Faithfulness** — does the answer actually follow from the retrieved chunks, or did the LLM hallucinate beyond them?
- **Contradiction handling** — for the `contradictory` category, did the system flag the conflict?
- **Abstention quality** — for `unanswerable` questions, did the system correctly say "I don't know"?

> **What we're really evaluating here**
>
> We care less about which metrics you pick and more about **why** you picked them, whether they actually measure what matters, and whether you understand their limitations. Saying "I used ROUGE but it doesn't capture semantic correctness well for our use case" is better than reporting a ROUGE score without comment.

2B. Error Analysis — The Most Important Part

For every question your system gets wrong (or partially wrong), write a **brief root-cause analysis**. We want to see:

- **Was it a retrieval failure?** (right doc not retrieved) — why? Wrong chunk size? Poor embedding for that query type?
- **Was it a generation failure?** (right doc retrieved but wrong answer) — why? Context window overflow? Instruction not followed?
- **Was it a data problem?** (the answer is simply not in the knowledge base, or the source doc is too noisy to extract from)

Group the failures into a **taxonomy** (e.g., "retrieval miss due to vocabulary mismatch" or "answer hallucinated beyond retrieved context"). Estimate what % of errors fall into each category. Propose one concrete fix for the top failure mode.

PART 3 — ITERATE

Make One Meaningful Improvement

Based on your error analysis in Part 2, identify the **single highest-impact failure mode** and fix it. This is not about trying everything — it's about making a **targeted, evidence-based improvement** and measuring whether it actually helped.

Examples of what this might look like (pick one, or come up with your own):

If your top failure is...	You might try...
Vocabulary mismatch in retrieval	Add a query expansion step (HyDE, LLM-rewrite), switch to hybrid search (BM25 + vector), or fine-tune your embedding model on domain terms
Noisy Slack/transcript chunks polluting results	Implement a two-stage retrieval with a reranker, or improve preprocessing to extract only decision-bearing sentences from conversations
Model hallucinating beyond retrieved context	Add a faithfulness check (NLI-based or LLM-based), restructure your prompt to be more restrictive, or implement citation verification
Contradictions not being detected	Retrieve top-K from multiple time periods and compare, or embed document dates into retrieval metadata and build a conflict detection step
System doesn't know when it doesn't know	Implement confidence calibration (e.g., self-consistency across multiple generations, or a separate classifier trained on your eval failures)

Required: Before/After Comparison

Run your eval harness (Part 2) on **both** the original and improved system. Report:

- Which metrics improved, by how much, and on which question categories
- Whether anything **regressed** (this is common and expected — own it)
- The cost difference (if any) of the improvement

> **We know you only have a few hours for this part**
>
> A small, well-measured improvement is far better than a sprawling refactor you can't evaluate. If your fix only improves 4 out of 40 questions but you can clearly explain *which* 4 and *why*, that's excellent work.

PART 4 — THINK AHEAD

Production Readiness (Write-Up Only)

In 1-2 pages of your engineering notebook, address this scenario:

Scenario

The VP liked your prototype. She wants to roll it out to all 800 employees next month. The knowledge base will grow from 150 to ~2,000 documents. New docs are added weekly. Some docs get updated in-place (same filename, new content). The system needs to handle ~500 queries/day with <5 second response time.

Cover these questions (in whatever structure makes sense to you):

Incremental indexing

A new policy doc is uploaded on Thursday. How does it get into the system? Do you re-embed everything or only the new doc? How do you handle a doc that was updated (same filename, different content)?

Staleness & trust

The knowledge base has outdated docs *today* at 150 docs. At 2,000, this problem will be worse. How do you handle document freshness? How does the user know if the answer they got is based on current policy or a 2-year-old draft?

Monitoring & failure detection

How do you know if the system is giving bad answers in production? You can't have a human check every response. What signals would you monitor? How would you build a feedback loop?

Cost at scale

You spent $X on 40 eval queries. Extrapolate: what does 500 queries/day cost over a month? At what point does it make sense to fine-tune a smaller model or self-host? Show rough numbers.

How We Evaluate Your Submission

Criterion	Weight	Unacceptable	Strong
It runs (table stakes)	Pass / Fail	We can't get it running in 10 minutes of setup	docker compose up (or equivalent), query works on first try
Data handling (Part 1)	25%	Ignored messy data, treated all docs the same, no preprocessing rationale	Thoughtful decisions about each data type, handled contradictions / staleness, justified tradeoffs
Evaluation rigor (Part 2)	25%	One metric, no error analysis, numbers without interpretation	Multiple metrics chosen with rationale, thorough error taxonomy, honest about limitations
Iteration quality (Part 3)	20%	Tried random things, no before/after comparison, no connection to error analysis	Targeted fix for top failure mode, clean before/after data, discussed regressions honestly
Engineering notebook	15%	Polished marketing-speak, no decisions explained, reads like it was generated wholesale	Genuine reasoning log, shows dead ends and pivots, explains *why* not just *what*
Production thinking (Part 4)	15%	Vague hand-waving, no concrete numbers, ignores practical constraints	Specific architecture choices, back-of-envelope cost math, realistic failure mode discussion

A Note on AI Coding Tools

Use whatever tools you normally use — Copilot, Claude Code, ChatGPT, Cursor, all fair game. We use them too.

However: this task is designed so that **the hard parts are decisions, not code**. An AI can write you a vector search function in 30 seconds, but it can't tell you whether your Slack preprocessing is throwing away critical context, or whether your chunk size is causing retrieval misses on multi-paragraph procedures, or whether your "improvement" actually regressed on a specific question category.

Your engineering notebook is the signal. We will read it carefully, and we will ask you about specific decisions in the follow-up interview. **If you can't explain why you made a choice, it will count against you regardless of the code quality.**

Suggested Time Allocation

Phase	Time	What to Focus On

Data exploration & preprocessing	2-3 hrs	Understand what's in the data. Make and document your preprocessing decisions.
Build RAG pipeline (Part 1)	3-4 hrs	Get a working end-to-end system. Start simple, then improve.
Evaluate (Part 2)	2-3 hrs	Run eval, compute metrics, do error analysis. This is where you differentiate yourself.
Iterate (Part 3)	1-2 hrs	One targeted improvement with before/after measurement.
Write-up (Part 4 + notebook)	1-2 hrs	Production thinking + polish your engineering notebook.

A crisp 10-hour submission where you clearly explain what you built and why beats a scattered 16-hour submission that tries everything. **Depth over breadth.**

FAQ

Where is the starter data?

The `knowledge_base/` directory and `eval/questions.jsonl` are provided alongside this PDF. If you haven't received them, contact your recruiter.

Can I modify the eval questions or add my own?

You must evaluate on the provided 40 questions for comparability. You *may* add additional test cases if you want, but clearly separate them from the core eval.

What if I can't run a local model and don't want to use paid APIs?

Free tiers of most API providers (OpenAI, Anthropic, Google, Groq) should be sufficient for this task. Alternatively, Ollama with a 7B-parameter model runs on most modern laptops.

Do I need to handle real-time document updates?

No. For the prototype (Parts 1-3), a static index is fine. Part 4 is where you discuss how you'd handle updates.

What if my system fails badly on an entire question category?

That's fine and expected. What matters is that you *know* it fails, you can explain *why*, and you have a plausible plan to fix it.

> *Good luck. We are looking for engineers who can navigate ambiguity, not people who can produce clean code for well-defined problems. Show us how you think.*