The Production RAG Checklist

Take Retrieval-Augmented Generation from a demo that works on your laptop to a system that survives real users — with a 0–100 production-readiness score.

A free guide from ProductionAIEngineer.com · v1.0 · Source file — deliver as the lead-magnet PDF.

Most RAG demos die in production. They answer the happy-path question in the demo, then hallucinate, cite the wrong document, blow the token budget, or fall over the first time a user asks something the corpus doesn't cover. This checklist is the gap between "cool demo" and "system I'd put my name on." Work top to bottom, tick what's true, and total your score.

How to use this checklist

- Each of the 11 sections has a **point value**. Tick every item you genuinely satisfy.
- Award a section's points **only if you can prove it** (a test, a trace, a metric).
- Total your score at the end and read the readiness band.
- Re-run it before every launch and after every major change.

Readiness bands

Score	Band	Meaning
0–49	🔴 Prototype	Fine for a demo; not for real users
50–69	🟠 Pre-production	Works, but will surprise you under load
70–84	🟢 Production-ready	Safe to ship with monitoring
85–100	🔵 Hardened	Resilient, measured, and cost-aware

---

1. Requirements & architecture — /8

Before you touch a vector DB, know what "good" means.

- [ ] **Use case is RAG-appropriate** (knowledge that changes, is large, or is private) — not something a fine-tune or a plain prompt solves better. **(2)**
- [ ] **Answer contract defined**: expected format, when to refuse, citation policy. **(2)**
- [ ] **Golden question set exists** (25–50 real questions with ideal answers). **(2)**
- [ ] **Non-functional targets set**: p95 latency, cost/query ceiling, freshness SLA. **(1)**
- [ ] **Architecture diagram** shows ingestion, retrieval, generation, and data flow. **(1)**

> **What good looks like:** you can state, in one sentence, why RAG (not fine-tuning) is the right tool here, and you have a question set to prove it works.

---

2. Ingestion & data pipeline — /10

Garbage in, garbage retrieved.

- [ ] **Source loaders** handle every real format (PDF, HTML, Markdown, DOCX) — including tables and multi-column PDFs. **(2)**
- [ ] **Cleaning step** strips boilerplate (nav, headers/footers, cookie banners). **(2)**
- [ ] **Deduplication** removes near-duplicate documents/chunks. **(1)**
- [ ] **Metadata captured** per chunk: source, title, URL, section, timestamp, permissions. **(2)**
- [ ] **Incremental re-indexing** (upserts, not full rebuilds) with a document version/hash. **(2)**
- [ ] **Deletion path** exists (a removed source leaves the index — GDPR/"right to be forgotten"). **(1)**

> **Common pitfall:** a one-shot ingestion script with no re-index story. Real corpora change daily.

---

3. Chunking strategy — /10

The single highest-leverage decision in RAG.

- [ ] **Structure-aware splitting** (on headings/sections), not fixed character counts. **(3)**
- [ ] **Chunk size tuned** to content (prose vs code vs tables) and embedding model limits. **(2)**
- [ ] **Overlap** set deliberately (enough for context, not so much you bloat the index). **(2)**
- [ ] **Small-to-big / parent-child**: retrieve small, feed larger parent context to the model. **(2)**
- [ ] **Tables & code preserved** as coherent units, not shredded mid-row. **(1)**

Chunking decision guide

Content	Strategy
Long prose / docs	Split on headings; 300–800 tokens; small overlap
Code	Split on function/class boundaries; keep signatures intact
Tables	Keep whole; attach a text summary of the table
Q&A / tickets	One record per chunk; the natural unit is the record

---

4. Embeddings & indexing — /8

- [ ] **Embedding model chosen deliberately** (dimensions, cost, domain fit) — and benchmarked on *your* data. **(2)**
- [ ] **Same model for index and query** (never mix). **(2)**
- [ ] **Vector store fits scale** (pgvector to start; Qdrant/Weaviate/managed at scale). **(1)**
- [ ] **Index params tuned** (HNSW `ef`/`M` or equivalent) for recall vs latency. **(1)**
- [ ] **Metadata filtering** supported at query time (tenant, permissions, recency). **(2)**

> **What good looks like:** you can justify your embedding model with a recall number on your own corpus, not a leaderboard.

---

5. Retrieval quality — /14

This is where most systems silently fail. The model is usually fine; retrieval is the problem.

- [ ] **Retrieval measured** with hit-rate / recall@k on the golden set. **(3)**
- [ ] **Hybrid search** (dense vectors + keyword/BM25) for names, IDs, and rare terms. **(3)**
- [ ] **Query transformation** where needed (rewriting, multi-query, HyDE). **(2)**
- [ ] **Metadata/permission filters** applied *before* similarity (never leak across tenants/users). **(3)**
- [ ] **top-k tuned** empirically (more is not better — it adds noise and cost). **(2)**
- [ ] **Empty/low-confidence retrieval handled** (a threshold → refuse, don't guess). **(1)**

Hybrid vs vector-only decision guide

- Exact terms, codes, names, acronyms in queries → **you need hybrid**.
- Purely semantic/paraphrase queries → vector-only may suffice.
- When unsure → hybrid + reranking is the safe default.

> **The #1 fix in RAG:** if answers are wrong, inspect the *retrieved chunks first*. Nine times out of ten the right context was never retrieved.

---

6. Reranking & context assembly — /8

- [ ] **Reranker** (cross-encoder or hosted rerank) reorders candidates before generation. **(3)**
- [ ] **Over-retrieve then rerank** (fetch 20–50, rerank to top 3–8). **(2)**
- [ ] **Context ordering** deliberate (most relevant near the prompt boundaries; mind "lost in the middle"). **(1)**
- [ ] **Deduplication/compression** of overlapping chunks before the prompt. **(1)**
- [ ] **Token budget enforced** for assembled context (truncation is intentional, not accidental). **(1)**

---

7. Generation & grounding — /12

- [ ] **Grounded prompt**: "answer only from context; if not present, say you don't know." **(3)**
- [ ] **Citations required** and rendered (chunk → source mapping the user can verify). **(3)**
- [ ] **Refusal path** works: no context → honest "I don't know," not a confident guess. **(2)**
- [ ] **Structured output** where the app needs it (validated JSON, not free text). **(2)**
- [ ] **Prompt is versioned** and changes go through evals. **(2)**

> **Anti-hallucination test:** ask 10 questions the corpus *cannot* answer. A production system refuses ≥ 9 of them. A demo confidently invents answers.

---

8. Evaluation — /12

You cannot improve what you do not measure.

- [ ] **Golden dataset** of Q/A pairs, versioned alongside code. **(2)**
- [ ] **Retrieval metrics**: recall@k, MRR, context precision. **(3)**
- [ ] **Generation metrics**: groundedness/faithfulness, answer relevance, citation accuracy. **(3)**
- [ ] **LLM-as-judge** calibrated against human labels (spot-check agreement). **(1)**
- [ ] **Regression suite** runs in CI on every prompt/model/retrieval change. **(2)**
- [ ] **Online feedback** (thumbs, edits) flows back into the eval set. **(1)**

Core RAG metrics glossary

Metric	Answers the question
Recall@k	Did we retrieve the right chunk at all?
Context precision	How much of the retrieved context was actually relevant?
Faithfulness / groundedness	Is the answer supported by the context (no hallucination)?
Answer relevance	Does the answer address the user's question?
Citation accuracy	Do the cited sources actually contain the claim?

---

9. Observability & monitoring — /8

- [ ] **End-to-end tracing**: query → retrieved chunks → prompt → response, per request. **(3)**
- [ ] **Logged per call**: query, chunk IDs+scores, model, tokens (in/out), latency per stage, cost. **(2)**
- [ ] **Dashboards**: latency p50/p95, cost/query, refusal rate, retrieval hit-rate trend. **(2)**
- [ ] **Alerts** on error rate, latency, cost spikes, and eval-score regressions. **(1)**

> **What good looks like:** for any bad answer a user reports, you can pull the exact trace and see whether retrieval or generation failed.

---

10. Security & privacy — /6

- [ ] **Prompt-injection defenses**: treat retrieved content as untrusted; instruction/data separation. **(2)**
- [ ] **Access control**: retrieval respects per-user/tenant permissions (no cross-tenant leakage). **(2)**
- [ ] **PII/secrets handling**: redaction in logs; no sensitive data in prompts you don't control. **(1)**
- [ ] **Output constraints**: guardrails on what the model may return (no data exfiltration via answers). **(1)**

> **Injection reality:** a poisoned document can carry "ignore your instructions." If retrieved content can steer the model, you have a vulnerability.

---

11. Cost & latency — /4

- [ ] **Caching**: embedding cache + response/semantic cache for repeat queries. **(1)**
- [ ] **Budgets**: per-request and daily cost ceilings with enforcement. **(1)**
- [ ] **Right-sized models**: small model for routing/rerank, large only where it pays. **(1)**
- [ ] **Timeouts & retries** with backoff on every external call. **(1)**

Score yourself

Total: ___ / 100

Transfer each section's earned points:

Section	Max	Score
1. Requirements & architecture	8	
2. Ingestion & pipeline	10	
3. Chunking	10	
4. Embeddings & indexing	8	
5. Retrieval quality	14	
6. Reranking & assembly	8	
7. Generation & grounding	12	
8. Evaluation	12	

Section	Max	Score
9. Observability	8	
10. Security & privacy	6	
11. Cost & latency	4	
Total	**100**	

RAG failure modes → fixes (fast reference)

Symptom	Likely cause	Fix
Confident but wrong answers	Retrieval missed the right chunk	Inspect retrieved chunks; add hybrid + reranking
Right doc, wrong section	Chunking too coarse/fine	Structure-aware chunking; parent-child retrieval
Hallucinates when corpus lacks answer	No refusal path	Grounded prompt + low-confidence threshold
Cites the wrong source	Chunk→source mapping broken	Fix metadata; verify citations in evals
Great in demo, bad on rare terms	Vector-only search	Add keyword/BM25 (hybrid)
Slow p95	Over-retrieval, big context, no cache	Tune top-k; compress context; add caching
Cost creep	Large model everywhere, no cache	Right-size models; semantic cache; budgets
Cross-user data leak	Filters applied after retrieval	Filter by permission *before* similarity

---

Bonus: the 10-minute RAG debugging flow

1. Reproduce the bad query.
2. **Print the retrieved chunks + scores.** Was the right context retrieved at all?
   - **No** → it's a *retrieval* problem (chunking, embeddings, hybrid, top-k).
   - **Yes** → it's a *generation* problem (prompt, grounding, context ordering).
3. Check the assembled prompt: is the right chunk actually in the context window?
4. Check the refusal path: should it have said "I don't know"?
5. Add the case to the golden set so it never regresses.

Your next step

You've scored your RAG system. Now close the gaps:

1. **One practical lesson a week** → subscribe to **Production AI Notes**.
2. **Instrument it properly** → grab the free **LLM Eval & Observability Cheat Sheet** to wire the metrics in Sections 8–9.
3. **Turn this into a portfolio project** → the **AI Engineer Interview & Portfolio Kit** ships RAG project specs, diagrams, and the interview answers that prove you can build this.

Built by Gururaj · ProductionAIEngineer.com · Ship AI that survives real users. Public/synthetic examples only. Opinions my own.
