Ghost Knowledge: When Your RAG System Cites Documents That No Longer Exist

February 25, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Ghost knowledge is what happens when your RAG system retrieves embeddings from documents that have been updated, deleted, or corrected. The old embeddings persist in the vector database with no expiration date and no link to the current version of the source. The system confidently cites information that is no longer true. No vector database (ChromaDB, Pinecone, Qdrant, FAISS) handles this natively. Up to 70% of RAG systems fail in production, with stale data as a leading cause. Fix it by storing content hashes and timestamps with every embedding, running nightly sync scripts that compare source docs to stored embeddings, and implementing deletion cascades so removing a source document also removes its vectors. If you do nothing else: add a last_modified timestamp to your embedding metadata today.

📚 **Related:** Local RAG Search · Context Rot and the Forgetting Fix · Agent Trust Decay · Context Length Explained · Planning Tool

A Mastercard data scientist shared this one: their RAG system was built when interest rates were 4%. Six months later, rates had jumped to 5.5%. The system was still confidently telling users the rate was 4%. No error message. No uncertainty qualifier. Just a wrong answer delivered with full confidence, retrieved from an embedding that hadn't been updated since the day it was created.

This is ghost knowledge. Your RAG system retrieves information from documents that have been updated, deleted, or corrected. The embeddings persist in the vector database long after the source material changed. The system doesn't know the difference between current facts and historical artifacts. It treats everything in the vector store as equally valid, because nothing tells it otherwise.

Every RAG system has ghost knowledge. The question is how much, and whether you've noticed yet.

What ghost knowledge looks like

It shows up in four ways, and none of them produce an error message.

The stale answer. A user asks about company policy. Your RAG retrieves chunks from a policy document that was updated three months ago. The model answers confidently with the old policy. The user follows the old rules. Nobody notices until an audit, a complaint, or a lawsuit.

The orphan citation. A user asks about a product feature. The source document was deleted when the feature was retired. But the embedding still sits in your vector database. The model cites a feature that no longer exists. The user tries to use it and gets confused.

The coin flip. A user asks a technical question. The original document was corrected after someone found an error. Both the old (wrong) and new (correct) versions are in the database. Which one does the retriever pick? Whichever embedding happens to be closer in vector space to the query. Sometimes you get the right answer. Sometimes you get the wrong one. Same question, different luck.

The confident synthesis. Two source documents disagree because one is outdated. The retriever returns chunks from both. The model, doing what language models do, synthesizes a coherent answer from contradictory sources. The result sounds authoritative and is wrong in a way that's hard to detect because it contains elements of both the old and new information.

The pattern across all four: the failure is silent. The retrieval score is fine. The model's confidence is high. The answer reads well. You have to independently verify against the current source to catch the problem, and nobody does that at scale.

Why every RAG tutorial skips this

Open any RAG tutorial. The pipeline looks like this:

1. Load documents
2. Chunk them
3. Embed the chunks
4. Store in vector database
5. Query and retrieve
6. Feed to LLM
7. Done!

Step 7 is where every tutorial ends and where every real-world problem begins. What happens when the documents in step 1 change? What happens when someone deletes a source file? What happens when your embedding model gets a version bump?

The tutorial doesn't say, because the tutorial was written against a static document set in an afternoon. The author loaded 10 PDFs, asked five questions, got good answers, and published the post. The scenario where those PDFs get updated six weeks later and the vector database quietly goes stale was never tested.

This is the pattern: RAG tutorials teach you how to build a snapshot. They don't teach you how to maintain a living system. A team builds a proof-of-concept in a week, demos it successfully, deploys it to production, and it quietly rots. Multiple surveys confirm this. Up to 70% of RAG systems fail in production, with stale data as a leading cause. 80% of enterprise RAG projects experience critical failures; only 20% achieve sustained success.

The "set it and forget it" RAG tutorial is a time bomb with a variable fuse.

The taxonomy of ghost knowledge

Not all ghosts are the same. Understanding the type tells you how to fix it.

Stale embeddings

The source document was updated. The embedding was not. This is the most common form and the hardest to detect because the embedding still points to a real document. It's just the wrong version of that document.

The Mastercard interest rate example is a stale embedding. The document existed and was current when it was embedded. The world changed. The embedding didn't.

Detection: compare content hashes. Store a SHA-256 hash of the source text alongside each embedding. On a schedule, re-hash the current source and compare. Any mismatch means the embedding is stale.

Orphan embeddings

The source document was deleted. The embedding persists. This is the RAG equivalent of a broken link, except the system doesn't return a 404. It returns a confident answer based on a document that no longer exists.

Orphans also create a legal problem. Under GDPR and CCPA, deleting a document doesn't satisfy the "right to be forgotten" if the embedding (which encodes the document's content) remains in the vector store. You need to delete both the text and the vector.

Detection: maintain a registry of all source document IDs. Periodically scan the vector database for embeddings whose source ID doesn't match any current document. These are orphans.

Duplicate versions

Old and new versions of the same document both exist in the database. Retrieval becomes a coin flip. Sometimes the old version is closer in vector space to the query, sometimes the new one.

This happens when you re-embed an updated document without first deleting the old embeddings. You end up with two (or more) sets of chunks from different versions of the same source.

Detection: query the vector database for all embeddings with the same source document ID. If you get chunks from multiple versions, you have duplicates.

Drift embeddings

A document changes gradually. Small edits, incremental updates, minor corrections over weeks or months. Each individual change is small enough that nobody triggers a re-embedding. But the cumulative drift means the embedding no longer represents what the document actually says.

The subtlety makes this one dangerous. The embedding is "close enough" to the current content that it doesn't flag as stale in a hash comparison (because you'd need to re-hash after every minor edit), but the meaning has shifted enough that retrieval quality degrades.

Detection: periodic full re-embedding on a schedule, regardless of whether individual changes seem significant.

Contradictory embeddings

Two source documents disagree, and the retriever returns chunks from both. The model synthesizes an answer that draws from contradictory information. In a safety manual scenario, this returned conflicting procedures because the same manual existed in four versions across three document stores.

Detection: score retrieved chunks for internal consistency before passing them to the LLM. If chunks contradict each other, flag the answer as uncertain rather than synthesizing a confident response.

Why no vector database solves this for you

I checked. None of them do.

Feature	ChromaDB	Pinecone	Qdrant	FAISS
Native TTL	No	No	No	No
Staleness detection	No	No	No	No
Delete by metadata	Yes	Yes	Yes	No
Upsert	Yes	Yes	Yes	No (remove + add)
Zero-downtime index swap	No	Namespace versioning	Collection aliases	Manual
Conditional updates	No	No	Yes	No

ChromaDB supports `upsert`, `update`, and `delete` by ID or filter. But it has no mechanism to tell you an embedding is stale. It stores what you give it and returns it when asked. If you never update or delete, it never notices.

Pinecone supports deletion by ID and by metadata filter. It has namespaces you can use for versioning. But it has no TTL, no freshness scoring, no automatic invalidation.

Qdrant has the most useful primitives: datetime range filtering, collection aliases for zero-downtime reindexing, and conditional updates with version fields that prevent stale overwrites. But detection and scheduling are still your problem.

FAISS is the worst case. It's a vector indexing library, not a database. Deletion is extremely slow (documented at 400ms per vector removal). Updates require remove-then-add. Building an IVF or HNSW index on a large corpus can take days. During rebuilding, you serve from a stale state unless you run a parallel pipeline.

The bottom line: staleness detection, TTL, and automatic re-embedding are application-level concerns. Your vector database is a storage layer. Keeping it fresh is your job.

How to detect it

Five methods, from cheap to thorough.

Content hash audit

Store a SHA-256 hash of the source text with every embedding. On a schedule (nightly is a good default), re-hash each source document and compare against the stored hash. Mismatches mean the embedding is stale.

```
import hashlib

def check_freshness(doc_id, current_text, metadata_registry):
    current_hash = hashlib.sha256(current_text.encode()).hexdigest()
    stored_hash = metadata_registry.get_hash(doc_id)
    return current_hash == stored_hash
```

This catches stale and drift embeddings. It won't catch orphans (the source is gone, so there's nothing to hash). A daily audit of 100K+ documents takes 2-3 minutes.

Orphan scan

Query your vector database for all unique source document IDs. Compare against your current document inventory. Any embedding whose source ID has no matching current document is an orphan. Delete it.

Freshness scoring

Add `ingested_at` timestamps to embedding metadata. At query time, filter or down-weight embeddings older than a threshold. A research paper (arXiv:2509.19376) showed that a half-life recency prior achieves perfect accuracy on "latest document" retrieval:

```
score = 0.7 * cosine_similarity + 0.3 * 0.5^(age_days / 14)
```

With a 14-day half-life, a two-week-old embedding gets 50% of its recency weight. A month-old embedding gets 25%. This doesn't fix the underlying staleness, but it limits the damage while you build a proper sync pipeline.

Canary queries

Maintain a set of 10-20 questions with known-correct answers. Run them on a schedule. If the system's answers diverge from the known-correct set, something in the pipeline has drifted. This catches problems you didn't anticipate in your automated checks.

Contradiction detection

Before passing retrieved chunks to the LLM, run a lightweight check for internal consistency. If two chunks make contradictory claims about the same topic, flag the retrieval as uncertain. This is harder to automate than the other methods, but it catches the most dangerous failure mode: confident answers built from conflicting sources.

How to prevent it

Store metadata with every embedding

At minimum, every embedding should carry:

```
{
  "document_id": "policy_handbook_v3",
  "content_hash": "sha256:a3f2b8c...",
  "source_path": "/docs/policies/handbook.pdf",
  "ingested_at": "2026-02-25T14:30:00Z",
  "embedding_model": "nomic-embed-text-v1.5"
}
```

The `document_id` lets you find all chunks from a given source. The `content_hash` lets you detect staleness. The `source_path` lets you verify the source still exists. The `ingested_at` lets you apply freshness scoring. The `embedding_model` version matters because different model versions produce incompatible vector spaces. You can't mix embeddings from different models in the same index without degrading retrieval quality.

Build a sync pipeline

The gap between "RAG works" and "RAG stays accurate" is a sync pipeline that keeps embeddings aligned with sources. Three levels of sophistication:

Simple: nightly cron job. Scan source documents, hash their contents, compare against stored hashes, re-embed anything that changed, delete orphans. A financial services team cut their knowledge base rebuild from 14 hours to 8 minutes by switching from full re-embedding to incremental updates that only process changed documents.

Medium: scheduled re-embedding with validation. Same as above, plus a golden query set that runs after each sync to verify retrieval quality didn't degrade. Log the results. Alert if accuracy drops below a threshold.

Advanced: event-driven re-embedding. File changes trigger re-embedding automatically. Python's `watchdog` library monitors directories. S3 event notifications trigger Lambda pipelines. CMS webhooks fire on content updates. Typical latency: 3-4 minutes per document from change to updated embedding. With parallel workers, a 50-document batch processes in about 90 seconds.

Implement deletion cascades

When a source document is deleted, its embeddings must be deleted too. This isn't optional. It's a data integrity requirement, and in regulated industries, it's a legal one (GDPR, CCPA).

Store the source document ID with every embedding. When a document is removed from your system, query the vector database for all embeddings with that document's ID and delete them. Soft deletion (marking as "deleted" without removal) is not compliant with right-to-be-forgotten regulations. You need to actually remove the vector.

Set TTLs by content type

Not all documents go stale at the same rate. Match your freshness requirements to the content:

Content type	Suggested TTL	Re-embed trigger
Financial data (rates, prices)	1-4 hours	Price feed update
Policy documents	7 days	Document version change
Technical documentation	30 days	Release notes publication
Meeting notes, decisions	14 days	Content hash mismatch
Historical reference, evergreen	90+ days	Periodic scheduled check

Pin your embedding model version

When the embedding model changes (even a minor version bump), the new model produces vectors in a different space than the old one. Mixing embeddings from different model versions in the same index degrades retrieval without any visible error.

Track which embedding model version produced each vector. When you upgrade models, re-embed the entire corpus. Don't mix old and new vectors. LlamaIndex handles this through its

`IngestionPipeline` with transformation caching: it hashes each node-plus-transformation combination and skips recomputation for unchanged documents. LangChain's Indexing API uses a `RecordManager` that hashes document content and supports three cleanup modes (none, incremental, full), though only `full` mode handles deletions correctly.

If a full re-embed is too expensive, the dual-index approach works: maintain the old index alongside a new one, route queries to both during migration, validate quality, then cut over. Qdrant's collection aliases make this a zero-downtime operation.

The practical fix, step by step

Level 1: add timestamps (30 minutes)

If your RAG system has zero freshness tracking, start here. Add `ingested_at` and `content_hash` fields to your embedding metadata. This costs nothing, changes no behavior, and gives you the foundation for everything else.

```
import hashlib
from datetime import datetime

def embed_with_metadata(text, doc_id, source_path):
    content_hash = hashlib.sha256(text.encode()).hexdigest()
    metadata = {
        "document_id": doc_id,
        "source_path": source_path,
        "content_hash": content_hash,
        "ingested_at": datetime.utcnow().isoformat(),
        "embedding_model": "nomic-embed-text-v1.5",
    }
    # Your existing embedding code here
    embedding = embed_model.encode(text)
    collection.add(
        ids=[f"{doc_id}_chunk_{i}"],
        embeddings=[embedding],
        documents=[text],
        metadatas=[metadata],
    )
```

If you already have embeddings without metadata, you can backfill. Query all embeddings, add the metadata fields with current timestamps (not ideal, but better than nothing), and use the current source hash. From this point forward, all new embeddings carry freshness data.

Level 2: nightly sync script (2-4 hours to build)

A script that runs on a cron schedule, compares source documents against stored embeddings, and flags or fixes discrepancies.

```
import os
import hashlib

def sync_embeddings(source_dir, collection, metadata_registry):
    current_docs = {}
    stale = []
    orphans = []

    # Scan current source documents
    for filepath in scan_documents(source_dir):
        doc_id = path_to_doc_id(filepath)
        with open(filepath, "r") as f:
            content = f.read()
        current_docs[doc_id] = hashlib.sha256(content.encode()).hexdigest()

    # Check each embedding against current sources
    all_embeddings = metadata_registry.get_all()
    for entry in all_embeddings:
        doc_id = entry["document_id"]
        if doc_id not in current_docs:
            orphans.append(entry)
        elif entry["content_hash"] != current_docs[doc_id]:
            stale.append(entry)

    # Delete orphans
    for orphan in orphans:
        collection.delete(where={"document_id": orphan["document_id"]})

    # Re-embed stale documents
    for entry in stale:
        old_chunks = collection.get(where={"document_id": entry["document_id"]})
        collection.delete(ids=old_chunks["ids"])
        re_embed_document(entry["document_id"], source_dir, collection)

    return {"orphans_removed": len(orphans), "stale_updated": len(stale)}
```

Run this nightly via cron: `0 2 * * * python sync_embeddings.py`. The 2 AM slot gives you a clean sync before the workday starts.

Level 3: event-driven pipeline (1-2 days to build)

Use `watchdog` to monitor your document directory and trigger re-embedding on file changes:

```
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

class DocChangeHandler(FileSystemEventHandler):
    def __init__(self, collection, embed_fn):
        self.collection = collection
        self.embed_fn = embed_fn

    def on_modified(self, event):
        if event.src_path.endswith((".md", ".pdf", ".txt")):
            doc_id = path_to_doc_id(event.src_path)
            # Delete old embeddings
            self.collection.delete(where={"document_id": doc_id})
            # Re-embed with current content
            self.embed_fn(doc_id, event.src_path, self.collection)

    def on_deleted(self, event):
        if event.src_path.endswith((".md", ".pdf", ".txt")):
            doc_id = path_to_doc_id(event.src_path)
            self.collection.delete(where={"document_id": doc_id})

observer = Observer()
observer.schedule(DocChangeHandler(collection, re_embed_document),
                  path="/docs/", recursive=True)
observer.start()
```

This gives you near-real-time sync: document changes propagate to the vector database within minutes. Add the nightly cron as a safety net for anything the file watcher misses (network file systems, batch imports, edge cases).

What this means for local AI builders

If you're running RAG on your own hardware, ghost knowledge is your problem in ways it isn't for someone using a managed enterprise RAG product.

Enterprise tools like LlamaIndex and LangChain have built-in document management. LlamaIndex's `IngestionPipeline` caches transformation hashes and only re-embeds changed documents. LangChain's Indexing API tracks content hashes through a `RecordManager`. These aren't perfect (LangChain's incremental mode can't detect deletions), but they're something.

If you built your RAG pipeline from a tutorial using raw ChromaDB and a Python script, you almost certainly have no freshness tracking. Your embeddings were created once and never validated against their sources. If any of those sources have changed since you embedded them, you have ghost knowledge right now.

This connects to a pattern we've covered before. Context rot happens when an agent's context window fills with outdated information. Agent trust decay happens when long-running agents drift from their original purpose. Ghost knowledge is the RAG-specific version: your retrieval layer silently filling the model's context with outdated or deleted information.

The complexity cliff hits again. Getting RAG working is stage 4 on that cliff. Keeping RAG accurate over time is a stage most people never reach. The proof-of-concept works. The demo impresses. Then the system runs for three months, source documents change, and nobody rebuilds the index.

The fix doesn't require sophisticated infrastructure. A nightly cron job that hashes your source documents and compares against your embedding metadata catches 90% of ghost knowledge. The other 10% (drift embeddings, contradictory sources, embedding model changes) requires more thought, but the basics are a few hours of work.

If you do one thing after reading this: add `content_hash` and `ingested_at` to your embedding metadata. Today. Everything else builds on that foundation, and without it, you're flying blind.

> Already running RAG? Check your setup against the embedding models guide and the session-as-RAG memory system. Use the VRAM Calculator to make sure your hardware can handle re-embedding.

Related guides

- Local RAG: Search Your Documents with Private AI
- OpenClaw Memory: Context Rot and the Forgetting Fix
- Agent Trust Decay: Why Long-Running AI Agents Get Worse Over Time
- Session-as-RAG: Teaching Your Local AI to Actually Remember

- Context Length Explained
- Embedding Models for RAG

Source: https://insiderllm.com/guides/ghost-knowledge-rag-stale-embeddings/