

 RETRIEVAL PIPELINE

Agentic RAG

How Corporate GPT answers questions strictly from an organization's own documents, with **groundedness, authorization, and observability** throughout

AUTHOR

Neurologic AI

PUBLISHED

July 2026

PLATFORM

Corporate GPT

ABSTRACT

Corporate GPT is an enterprise knowledge assistant that answers questions strictly from an organization's own documents rather than from a language model's parametric memory. This paper describes its *agentic* Retrieval-Augmented Generation (RAG) architecture: a top-level orchestrator that plans, delegates to specialized sub-agents, and enforces a grounding contract; a two-stage *preview* → *select* → *answer* retrieval flow that keeps a human in the loop; a hybrid (dense + sparse) retrieval engine backed by the Qdrant vector database with LLM-based reranking under a “deep search” mode; and defense-in-depth access control that enforces per-user, per-group, and per-project document authorization on every query. We also cover the asynchronous ingestion pipeline, PII/PHI redaction, and the provider-agnostic model layer that lets the same architecture run against self-hosted or commercial models. The design goals throughout are *groundedness* (never fabricate), *authorization* (never leak), and *observability* (every retrieval is traced).

Groundedness

Never fabricate.

Authorization

Never leak.

Observability

Every retrieval is traced.

01

Introduction

Large language models (LLMs) are fluent but unreliable narrators of private enterprise knowledge: they hallucinate facts, cannot cite sources, and have no notion of who is authorized to see what. Retrieval-Augmented Generation (RAG) mitigates the first two problems by grounding generation in retrieved passages, but a naive “embed, search, stuff-the-prompt” pipeline is insufficient in a corporate setting. Real deployments must additionally decide *whether* to retrieve, *where* to retrieve from, *how many times*, and *for whom*, and they must do so while respecting document-level access control and regulatory constraints on personal data.

Whether

to retrieve

Where

to retrieve from

How many times

bounded multi-hop

For whom

document-level
access control

Corporate GPT addresses these requirements with an *agentic* RAG design. Rather than a fixed retrieval chain, an orchestrating LLM agent reasons about each request, routes it to the appropriate specialized sub-agent (document RAG, web search, SQL, Jira, ticketing), and controls a bounded multi-hop retrieval loop. Retrieval itself is hybrid, combining dense semantic vectors with sparse lexical (BM25) matching, and can escalate to an LLM-reranked “deep search” when precision matters. Critically, every retrieval is filtered through an access-control layer derived from the requesting user's identity, group membership, role, and project scope, and the corpus is scrubbed of sensitive personal data at ingestion time.

This paper documents that architecture as implemented, describing the concrete components, data flow, and design decisions that make retrieval grounded, authorized, and observable.

02

System overview

Corporate GPT is a multi-agent system in which the underlying model provider (self-hosted open models or a commercial API such as Claude) is a deployment-time choice rather than a fixed dependency. The major subsystems are:

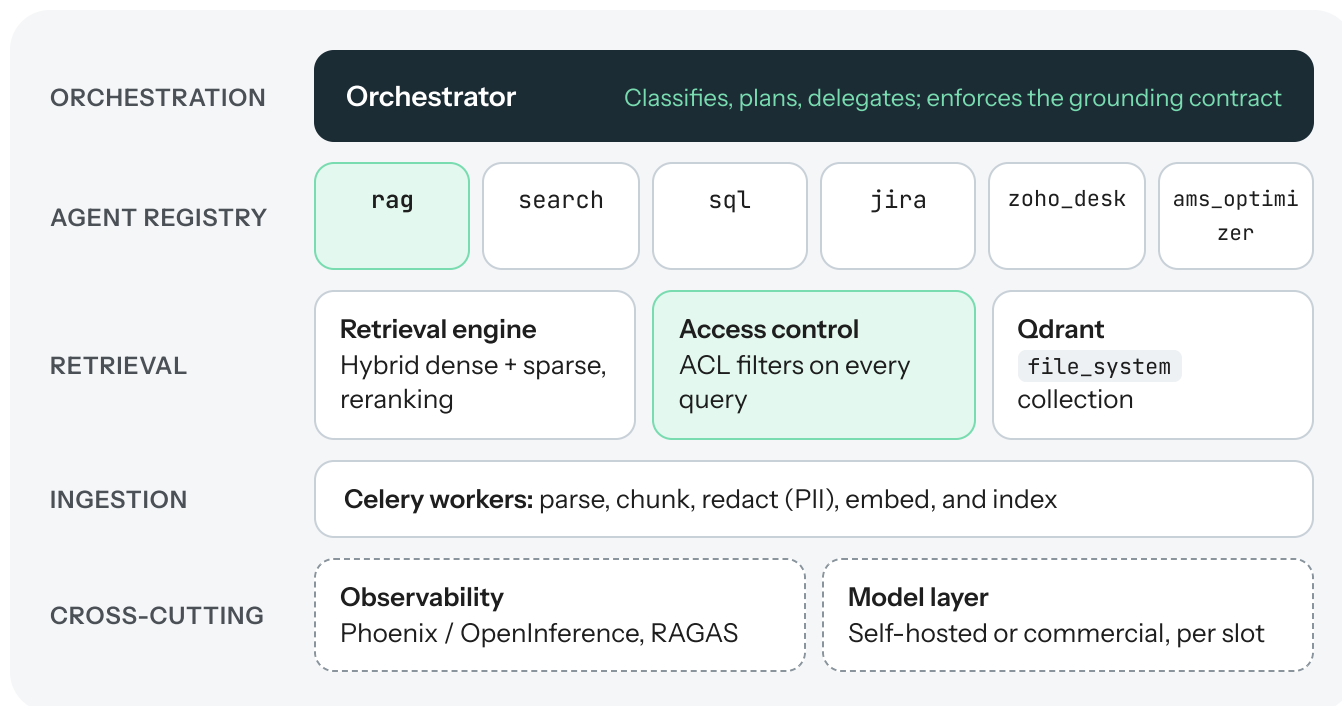
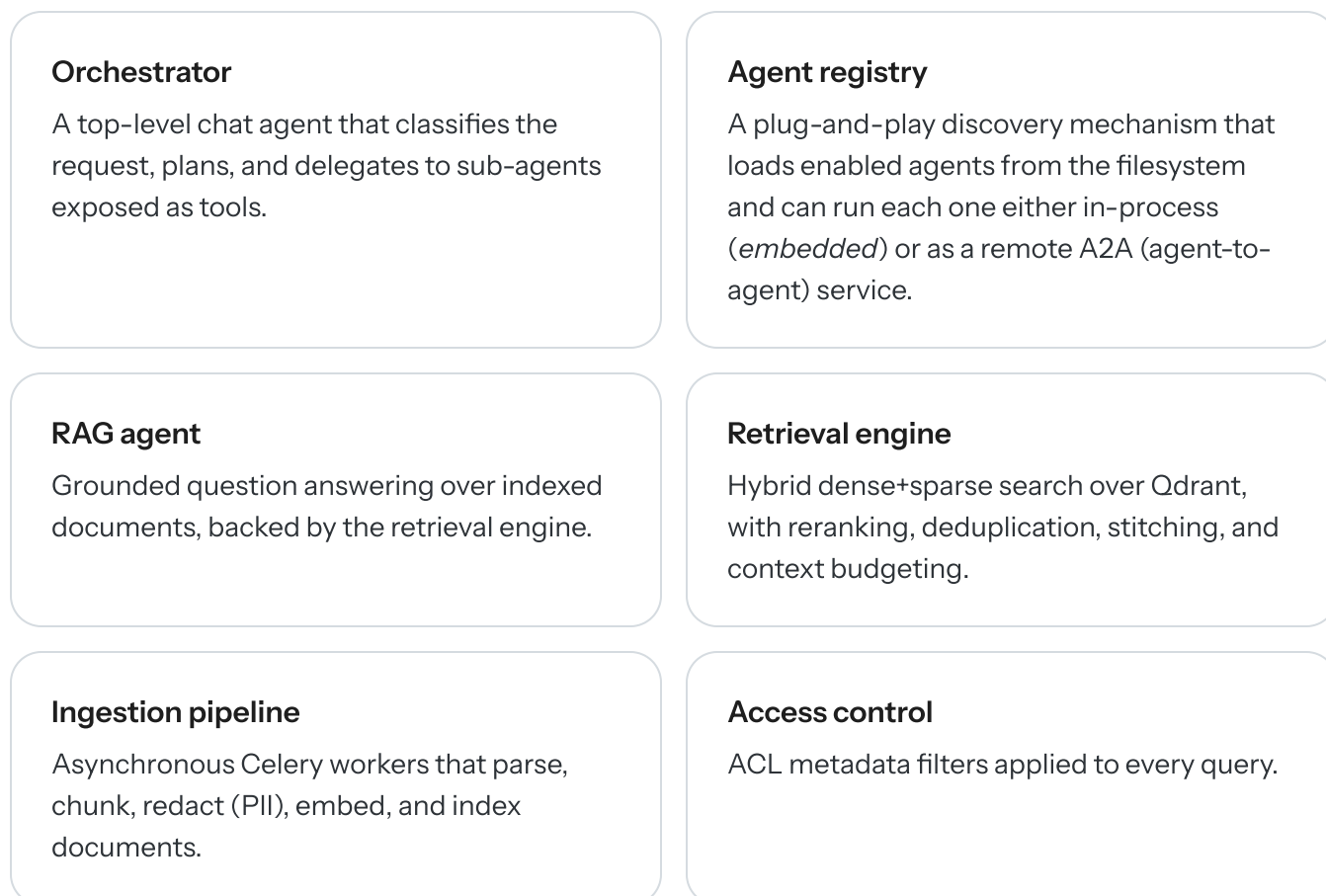


Figure 1. The major subsystems of Corporate GPT's agentic RAG architecture.



Observability

Phoenix / OpenInference distributed tracing and RAGAS-based evaluation.

2.1 The agent registry

The registry discovers every agent package on startup. The set of active agents is driven purely by environment configuration (`ENABLED_AGENTS`) and the filesystem, so the system topology can be reconfigured without code changes. For each enabled agent, a per-agent `<NAME>_MODE` selects an *embedded* in-process instance or a *remote* `RemoteA2aAgent` reached over the network. The currently shipped agents include:

Agent	Responsibility
<code>rag</code>	Grounded QA over indexed corporate documents (Qdrant)
<code>search</code>	Web-search fallback (Brave) for public/current info
<code>sql</code>	Natural-language querying over relational databases
<code>jira</code>	Retrieval over Jira tickets, issues, and sprints
<code>zoho_desk</code>	Retrieval / actions over Zoho Desk support tickets
<code>ams_optimizer</code>	Domain-specific optimization agent

Table 1. Specialized sub-agents discovered by the registry.

This decoupling is what makes the system *agentic* at the architectural level: the orchestrator does not hard-code a pipeline; it composes whatever tools the registry currently exposes.

03

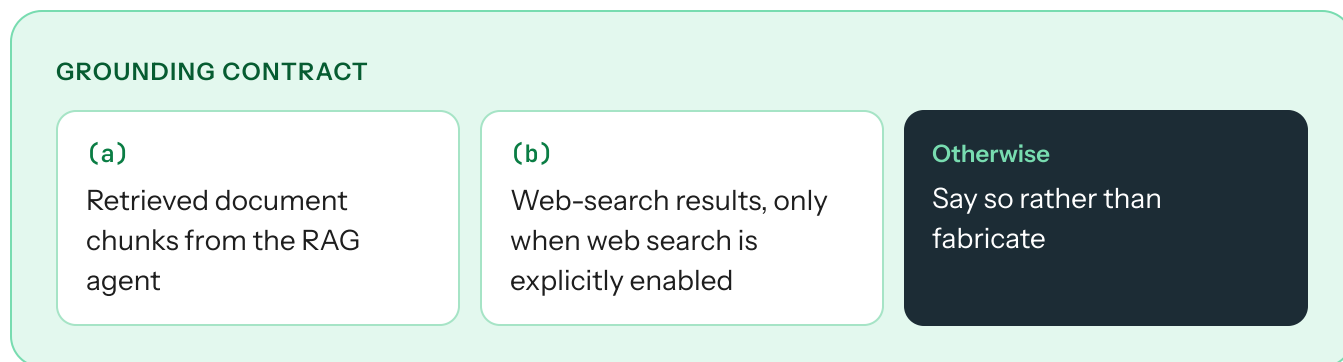
The agentic orchestration layer

The chat orchestrator is itself an agent. It wraps each enabled sub-agent as a callable tool (the RAG agent, and, when enabled, the Brave web searcher, the Jira agent, the AMS optimizer, and the Zoho Desk agent) and loads a natural-language operating instruction

(a “skill”) that governs how it reasons. The orchestrator's job is *planning and delegation*: it decides which tool answers the current turn and enforces the system-wide grounding contract.

3.1 The grounding contract

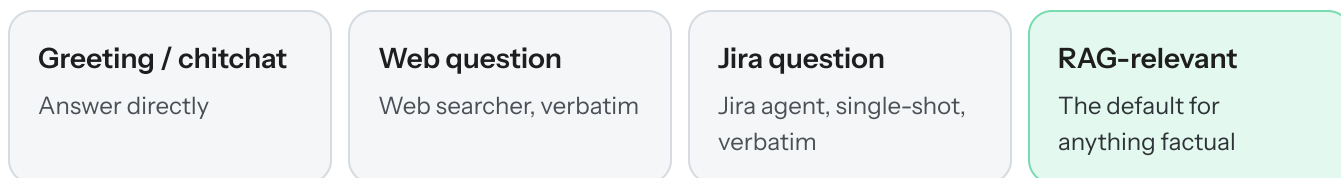
The orchestrator's instruction encodes an absolute rule: it is *not* a general-knowledge assistant and must never answer a substantive question from its own training data. Every non-trivial answer must be grounded in either (a) retrieved document chunks from the RAG agent, or (b) web-search results (and only when web search is explicitly enabled). The only messages it may answer directly are pure greetings and chitchat that carry no information request. When the system cannot ground an answer, it is required to say so rather than fabricate: surfacing the document preview or an explicit “could not find relevant information” result instead of a hallucination.



3.2 Two-stage retrieval: preview → select → answer

A distinctive feature of Corporate GPT is that document retrieval is a *two-stage, human-in-the-loop* flow rather than a single hidden call. This keeps the user in control of provenance and prevents the model from silently answering from the wrong document.

Stage 1: Initial (preview mode). On the first turn the orchestrator classifies the message into exactly one of: greeting/chitchat (answer directly); a web question (delegate to the web searcher, verbatim); a Jira question (single-shot delegate to the Jira agent, verbatim); or (the default for anything factual, informational, or company-specific) a RAG-relevant question. For RAG-relevant questions, the orchestrator calls the RAG agent to retrieve candidate sources, but *does not yet generate an answer*. Instead it returns a `preview` object listing the unique files and the specific chunks that matched, and asks the user to select which documents to answer from.



Stage 2: Continue (answer mode). Once the user selects files, those selections are passed back as `selected_document_ids`. The orchestrator calls the RAG agent *exactly once* more; the agent retrieves strictly from the selected, access-controlled documents and produces the final JSON answer with inline citations. The orchestrator returns this response *verbatim*: it must not re-summarize, re-order, deduplicate, or wrap the sub-agent's grounded output, so the answer the user sees is byte-for-byte what the retrieval agent produced.

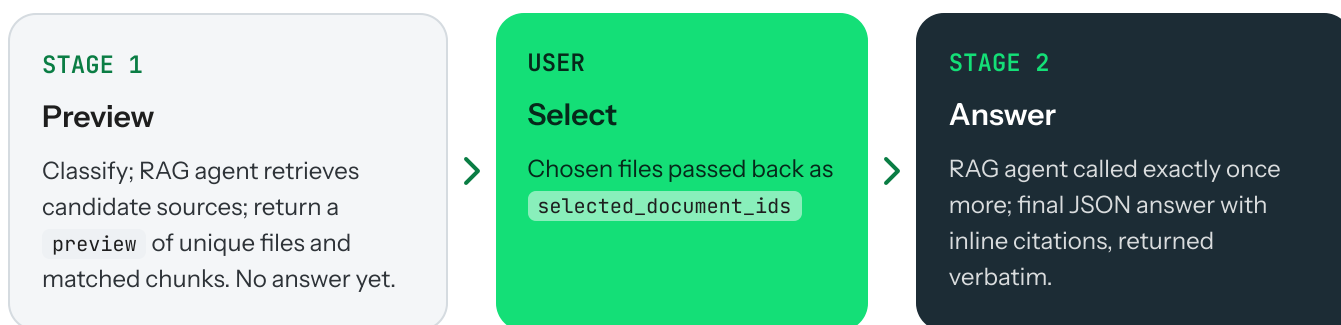


Figure 2. The two-stage, human-in-the-loop retrieval flow: preview → select → answer.

The answer the user sees is **byte-for-byte what the retrieval agent produced.**

3.3 Bounded multi-hop retrieval and tool-choice control

Within the RAG agent, retrieval is a controlled loop rather than a single shot. A *tool-choice policy* decides, on each model step, whether tool use is forced, forbidden, or optional. The decision is a pure function of the conversation state and resolves in the following precedence:

1. NONE

For a greeting turn, or once the retrieval budget is spent, or a file-export cap is hit. A looping model is cut off and must answer with what it already has.

2. ANY(retrieve)

The first hop of a non-thinking turn is *forced* to call the retrieval tool, so the agent can never answer before grounding.

3. ANY(file creators)

When the user asked for a downloadable artifact, sources were found, and no creator has yet run.

4. AUTO

Otherwise, the model chooses freely.

This policy is enforced at the model-invocation layer, binding the model's native tool-choice controls rather than merely suggesting behavior in the prompt, which makes grounding a hard invariant. A configurable budget (default 10 retrievals per user query) bounds the loop; both successful and failed retrieval attempts increment the counter so that a persistently failing query terminates gracefully with a “retrieval failed” answer instead of looping to the model-call limit. When the deployment enables an extended reasoning (“thinking”) mode, the agent is trusted to retrieve on its own initiative and the forced-first-hop rule is relaxed.

— 04

The retrieval engine

The retrieval engine owns the Qdrant connection, the embedding model, the reranker, and the document and summary indexes, and mediates every query against the corpus.

4.1 Hybrid dense + sparse search

Documents are indexed into a single unified Qdrant collection (default name `file_system`) with *native hybrid search* enabled. Each query is answered by combining two retrieval channels:

Dense (semantic)

A neural embedding of the query is matched against chunk embeddings by vector similarity. The embedding model is resolved from configuration (e.g. a Qwen3-Embedding model served via vLLM/Ollama), so the semantic space is a deployment choice.

Sparse (lexical, BM25)

A sparse text embedding (`fastembed` 's `SparseTextEmbedding`) provides exact-term matching, which recovers rare identifiers, codes, and acronyms that dense vectors blur together.

The sparse channel uses a BM25 sparse encoder (default `Qdrant/bm25` , loaded from a local cache first to avoid a network fetch on startup). The two ranked lists are combined by a configurable **fusion strategy**. The system implements three, selectable via `QDRANT_HYBRID_FUSION` :

Reciprocal Rank Fusion (RRF) DEFAULT

The default; each document's fused score is computed as below, with $k = 60$ by default. Rank-based fusion is robust to incomparable score scales.

$$\text{fused score} = \sum_i \alpha_i / (k + \text{rank}_i) \quad k = 60$$

Distribution-Based Score Fusion (DBSF)

Per-channel $\mu \pm 3\sigma$ normalization then a weighted sum, mirroring Qdrant's server-side DBSF.

Relative Score Fusion

Min-max style normalization, wrapped to handle the degenerate both-channels-empty case safely.

The engine requests roughly twice the requested top- k from each channel before fusion, giving the fusion and downstream reranking stages a larger candidate pool to work from. The default answer size is `DEFAULT_TOP_K = 5` .

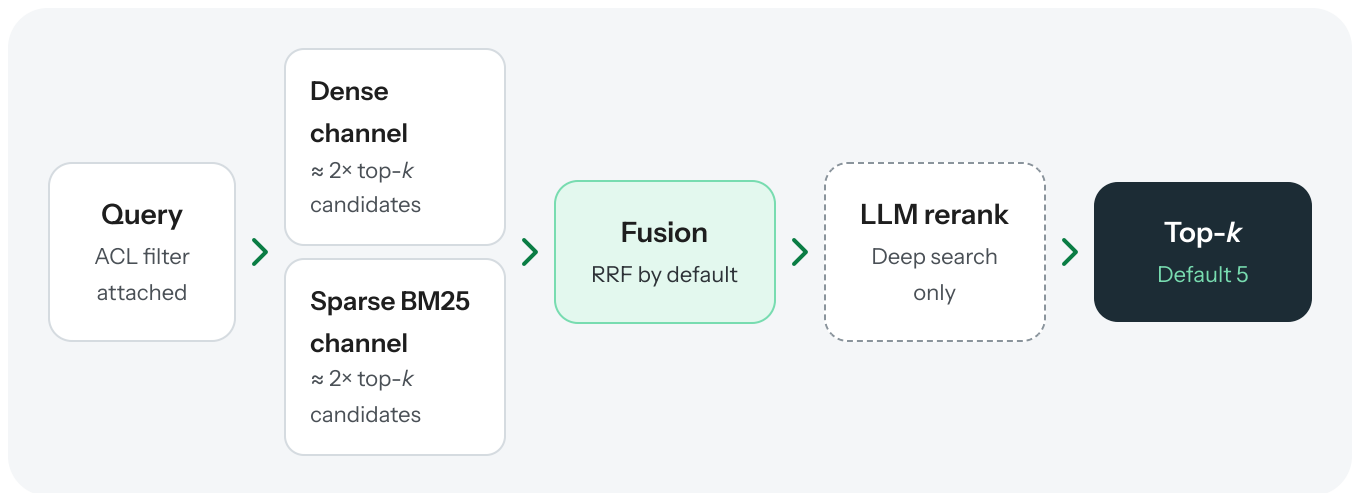
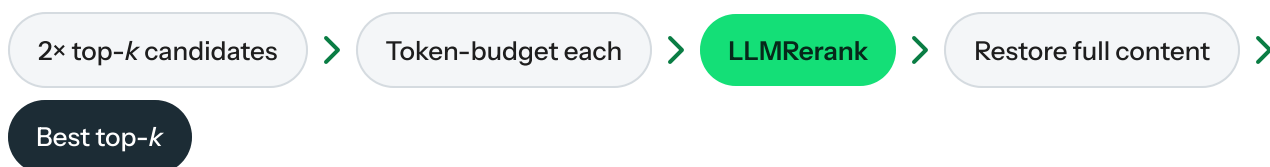


Figure 3. Hybrid retrieval: two channels, each over-fetched, fused into one ranked list and optionally reranked under deep search.

4.2 Deep search: LLM reranking

When a query needs higher precision, the caller enables *deep search*. In this mode the engine doubles the candidate pool ($2\times \text{top-}k$) and then applies an LLM-based reranker (LlamaIndex `LLMRerank`) driven by a no-thinking “parser” LLM to re-score and select the best $\text{top-}k$ passages. Because reranking feeds full chunk text to the LLM, the engine token-budgets each candidate before the rerank call (computing a per-node budget from the parser’s context window divided by the batch size) and *restores* the full untruncated content of the surviving nodes afterward, so the downstream answer generator still sees complete passages. Reranking runs on a dedicated background event loop with a timeout; if it times out, the engine gracefully falls back to the pre-rerank ordering.



On timeout, the engine falls back to the pre-rerank ordering.

4.3 Post-retrieval processing

Beyond fusion and reranking, several passes improve answer quality and robustness:

Same-page stitching

Adjacent nodes from the same page are stitched so a passage split across chunk boundaries is presented whole.

Content deduplication

Duplicate chunks are collapsed, keeping the highest-scoring instance, before the top-*k* cut.

Context-window budgeting **90%**

Final sources are accumulated top-down until they would exceed 90% of the answer model's (VLM) context window; if even the single top source overflows, it is truncated to fit rather than dropping all sources. This guarantees the answer generator always receives an ingestible, non-empty context.

Figure inventory

For the primary matched document, the engine performs a Qdrant metadata *scroll* (no vector search) to retrieve the document's full inventory of figures, so the model can reference relevant figures even when vector retrieval did not surface them. Figure nodes carry bounding boxes for precise cropping and inline rendering.

When the model chooses to display a figure, the engine extracts the actual image from the source PDF using a three-strategy fallback, in priority order: (1) a precise bounding-box crop using the `bbox` stored in Qdrant metadata; (2) direct extraction of the sole embedded image when a page contains exactly one; and (3) a full-page render as a last resort for vector drawings that have no extractable raster.

1

Bounding-box crop

`bbox` from Qdrant metadata



2

Sole embedded image

When a page contains exactly one



3

Full-page render

Last resort for vector drawings

4.4 A multimodal, cited corpus

Sources returned to the model are richly typed. Each carries a `source_type` (`website` , `image` , `audio` , or `file`), page labels, document IDs, similarity scores, and (for figures) bounding boxes and object indices. This lets the RAG agent produce citations, place inline images, and distinguish web-ingested pages from uploaded files. Document summaries are maintained in a parallel Qdrant summary collection to support summary-level retrieval.

4.5 Domain-adapted embeddings

The quality of dense retrieval is bounded by how well the embedding model separates the organization's domain vocabulary. Corporate GPT therefore includes a pipeline for producing a *domain-adapted* embedding model via parameter-efficient (LoRA) fine-tuning of the base Qwen3-Embedding-8B. The pipeline is fully synthetic-supervision driven: PDF → token-aware chunks → synthetic queries generated by a hosted instruction LLM → hard-negative mining (FAISS-based, with a margin filter and a same-document exclusion to avoid false negatives) → InfoNCE contrastive training with in-batch and mined negatives → retrieval and RAGAS evaluation.



Figure 4. The synthetic-supervision pipeline for LoRA fine-tuning of Qwen3-Embedding-8B.

Because the fine-tuned adapter is served behind the same OpenAI-compatible embedding endpoint, it is swapped in via a single configuration variable with no changes to the retrieval code: the domain model and the base model are interchangeable at the slot level. On a 32-paper evaluation corpus, the adapter measurably improved retrieval:

Metric	Base	+ LoRA	Δ
Recall@1	0.5705	0.6682	+0.098
Recall@5	0.8909	0.9386	+0.048
Recall@10	0.9614	0.9818	+0.020
NDCG@10	0.7736	0.8402	+0.067

Table 2. Retrieval quality, base vs. domain-adapted (LoRA) embeddings.

An end-to-end RAGAS A/B test (8 held-out papers, 31 questions) showed corresponding downstream gains in context utilization (+0.108), context recall (+0.048), context precision (+0.035), and answer faithfulness (+0.043). Note that because base and adapted vectors occupy different embedding spaces, the corpus must be re-indexed when switching models: querying adapted vectors against a base-built index would be meaningless.

+0.108

Context utilization

+0.048

Context recall

+0.035

Context precision

+0.043

Answer faithfulness

RAGAS A/B gains: 8 held-out papers, 31 questions.

05

Access control: authorization on every query

In an enterprise corpus, retrieval without authorization is a data-leak vector. Corporate GPT enforces document-level access control *inside* the retrieval call itself, so no code path can retrieve a document the requesting user is not entitled to see.

For each query, the system looks up the user's group membership and role and constructs a Qdrant payload filter with OR semantics covering:

- documents marked public (`acL_is_public`);
- documents whose `acL_user_ids` include the requesting user;
- documents in any of the user's groups (`acL_user_group_ids`);
- admin-public documents (for `ADMIN` / `GLOBAL_CURATOR`);
- curator-public documents (for `CURATOR` / `GLOBAL_CURATOR`).

The design is **fail-closed**: even if the database lookup fails, the function still returns a non-empty baseline filter (public documents plus the user's own documents), so an error *narrows* what is visible rather than opening the corpus. On top of the ACL filter, two further filters are composed with AND semantics:

Selected-document scoping

In Stage 2, the user's file selection (`selected_document_ids`) is intersected with the ACL, so the answer is drawn only from documents that are both authorized and chosen.

Project isolation

When enabled, the chat session's project scope is AND-ed on so that project chats see only that project's documents and project documents never leak into personal or group chats. This too is fail-closed: an unresolved scope defaults to the personal scope.

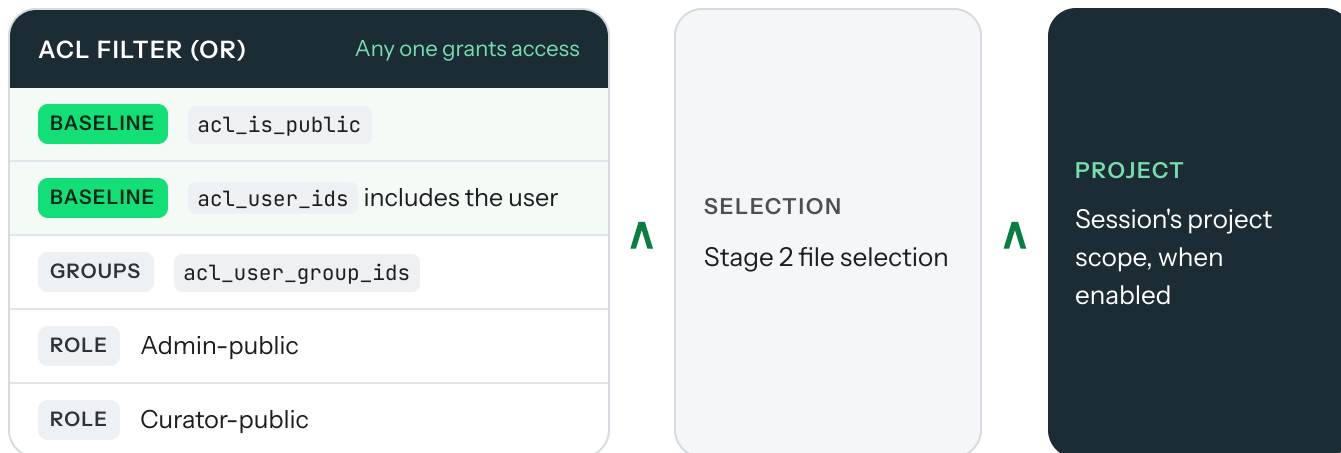


Figure 5. The composed filter pushed down into Qdrant. The two baseline arms survive a failed database lookup, so an error narrows visibility.

Because these filters are pushed down into Qdrant as payload conditions, access control is applied at the vector-search layer: unauthorized documents are never even scored, let alone returned to the model.

06

Ingestion, chunking, and PII redaction

Retrieval quality is set at ingestion time. Corporate GPT ingests documents asynchronously through Celery workers backed by Redis, organized into dedicated queues (*parsing*, *embedding*, and *permissions*) so that CPU-bound parsing, GPU-bound embedding, and ACL synchronization scale independently. Files are first registered, then parsed and chunked, then embedded and written to Qdrant. The pipeline is GPU-aware (it defers work when free device memory is below a threshold), uses distributed locks to prevent duplicate processing across workers, and carries a bounded retry budget with a state machine that supports pause, resume, and stop of an in-flight source.

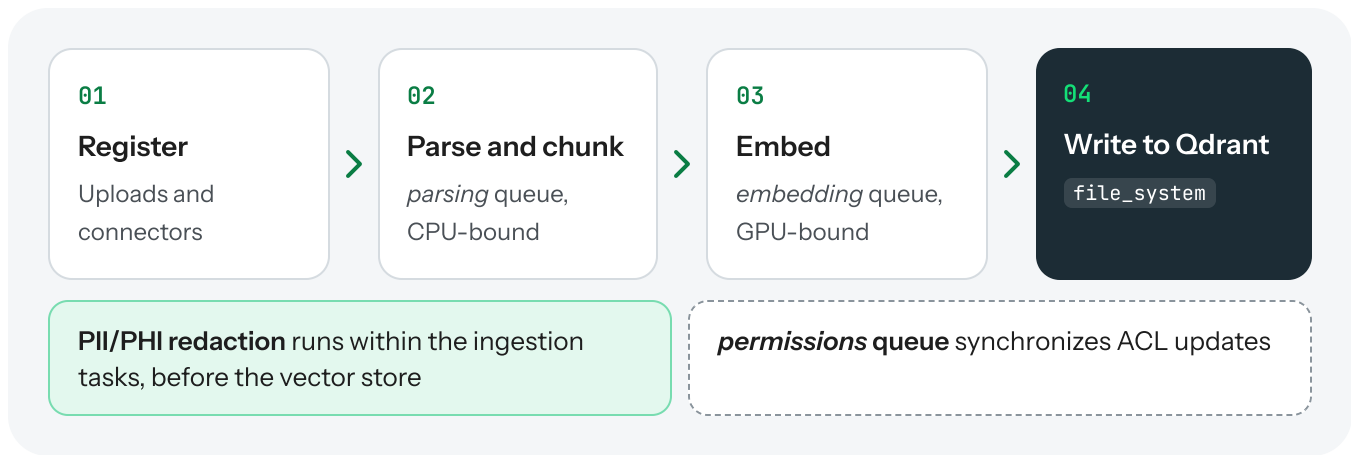


Figure 6. The asynchronous ingestion pipeline and its dedicated queues.

Documents enter this pipeline either as direct uploads or through a *connector platform* that ingests from external systems of record (for example issue trackers, CRMs, and databases), normalizing each source into the same chunk-and-embed path. This is what makes the single unified `file_system` collection a genuinely cross-source corpus while keeping one retrieval and access-control code path.

6.1 Structure-aware chunking with Docling

Parsing and chunking use Docling's `HybridChunker` with a HuggingFace tokenizer (default chunk size 2048 tokens; connectors use a smaller default). The chunker is structure-aware: the engine preserves nested list indentation, re-wraps code blocks as complete fenced blocks (since token-splitting a code item would otherwise leave dangling fences), and splits large tables into token-bounded pieces while keeping them well-formed. This structural fidelity is what lets the downstream model reason over tables, code, and lists rather than over shredded text.

6.2 PII / PHI detection and redaction

Before content is indexed, it passes through a PII/PHI pipeline built on Microsoft Presidio with a spaCy NLP engine, exposing 50+ recognizers (SSN and credit card with Luhn validation, IBAN, bank and passport numbers, driver license, ITIN, crypto addresses, and country-specific identifiers such as Indian PAN and Aadhaar, among others). A policy separates entity types into an *always-redact* set (financial and government identifiers, redacted even when nested inside an otherwise-safe span such as a URL parameter) and a *never-redact* pass-through set (e.g. dates, locations, organizations, URLs, and email addresses that are safe to retain for retrieval context). Redacted spans are fingerprinted with

an HMAC-SHA256 hash of the raw value, allowing consistent handling and audit without persisting the sensitive plaintext. This pipeline runs within the ingestion tasks so that sensitive data is scrubbed *before* it ever reaches the vector store.

50+

recognizers

ALWAYS REDACT

Financial and government identifiers, even when nested inside an otherwise-safe span

NEVER REDACT

Dates, locations, organizations, URLs, and email addresses safe to retain

6.3 Permission synchronization

Access-control changes propagate through the dedicated *permissions* queue, which batch-synchronizes ACL updates between the relational database and the Qdrant payloads and purges deleted users and groups from the index. This keeps the authorization metadata that the retrieval filters depend on consistent with the system of record.

07

Model orchestration and observability

7.1 Provider-agnostic model layer

All model calls flow through a resolution layer that maps a logical *slot* (chat, parsing, embedding, speech-to-text) to a concrete, centrally-configured model for the requesting user. Slots decouple *role* from *implementation*: the chat/answer model, the parser/reranker model, the embedding model, and the speech-to-text model can each be swapped independently, and the resolver enforces class constraints (e.g. multimodal answer generation requires a vision-capable VLM; embedding requires an embedding model). An optional extended-reasoning (“thinking”) mode is negotiated per request and downgraded gracefully when the resolved model does not support it. The net effect is that the same agentic RAG architecture runs unchanged against self-hosted open models (via vLLM or Ollama) or a commercial API such as Claude.

SLOT

Chat / answer

Vision-capable VLM for multimodal

SLOT

Parser / reranker

SLOT

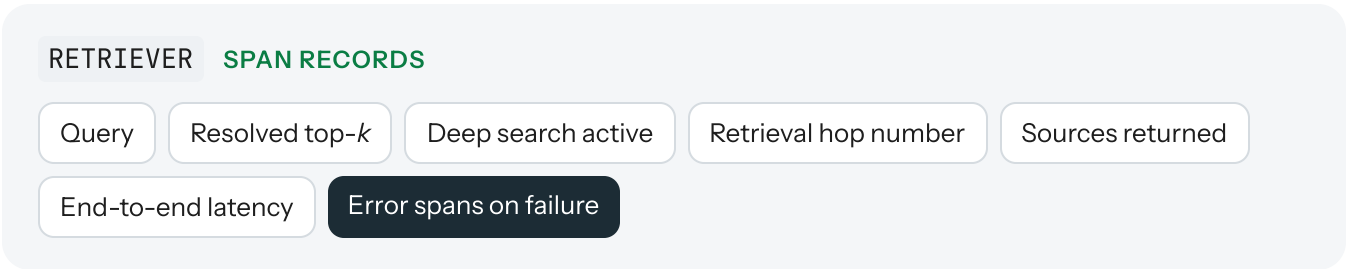
Embedding

SLOT

Speech-to-text

7.2 Distributed tracing and evaluation

Every retrieval is instrumented. The RAG tool opens an OpenInference `RETRIEVER` span (exported to Phoenix) recording the query, the resolved top- k , whether deep search was active, the retrieval hop number, the number of sources returned, and end-to-end latency; failures are captured as error spans with exception detail. This gives operators a per-query, per-hop view of the agentic loop. Retrieval and answer quality are measured offline with a RAGAS-based evaluation harness, closing the loop between production traces and measured groundedness.



7.3 RAGAS results

The RAGAS harness scores the end-to-end pipeline on two complementary families of metrics, all on a $[0, 1]$ scale where higher is better:

RETRIEVAL QUALITY

Context precision (are the retrieved passages relevant?), *context utilization* (does the answer draw on the retrieved context?), *context recall* (was the information needed to answer actually retrieved?), and *context entity recall* (were the key entities retrieved?).

ANSWER QUALITY

Answer relevancy (does the answer address the question?) and *faithfulness* (is every claim in the answer grounded in the retrieved context, i.e. free of hallucination?).

Table 3 reports three evaluation runs: two model versions (Qwen3.5 and Qwen3.6) on the same 40-question tuning set, and a larger 221-question validation set that confirms the results hold at scale. Retrieval is consistently strong (context recall ranges from 0.75 to 0.97, so the pipeline reliably surfaces the information needed to answer) and faithfulness of 0.70 to 0.86 indicates answers stay grounded in that retrieved context rather than drifting into model priors. The tuned Qwen3.5 configuration is the strongest overall, leading on every retrieval metric and on answer relevancy, while the larger validation set posts the highest faithfulness (0.859), confirming groundedness is stable as question diversity grows.

RAGAS metric	Qwen3.5 (40 Q)	Qwen3.6 (40 Q)	Qwen3.5 (221 Q)
Context precision	0.750	0.713	0.625
Context utilization	0.789	0.695	0.668
Context recall	0.967	0.923	0.750
Context entity recall	0.356	0.266	0.200
Answer relevancy	0.825	0.646	0.668
Faithfulness	0.841	0.698	0.859

Table 3. End-to-end RAGAS scores across model versions (Qwen3.5, Qwen3.6) on a 40-question tuning set, plus a Qwen3.5 run on a larger 221-question validation set.

08

End-to-end request flow

Putting the pieces together, a typical document question proceeds as follows:

- 1 The chat endpoint invokes the orchestrator agent with the user's message and identity.
- 2 The orchestrator classifies the message. A greeting is answered directly; a web or Jira question is delegated to the corresponding agent; anything factual is treated as RAG-relevant.
- 3 For a RAG-relevant question, the orchestrator calls the RAG agent, whose first hop is forced to retrieve. The retrieval tool builds the user's ACL (and project) filters and invokes the retrieval engine.
- 4 The retrieval engine runs hybrid dense+sparse search over Qdrant with the ACL filter pushed down, fuses the channels (RRF by default), stitches and deduplicates results, optionally LLM-reranks under deep search, and budgets the final context.
- 5 The orchestrator returns a preview of the matched files and chunks; the user selects which

documents to use.

6

On the follow-up turn, the RAG agent retrieves once more (now intersected with the selected document IDs) and generates a grounded, cited answer, which the orchestrator returns verbatim.

7

Throughout, each retrieval is traced to Phoenix, and the retrieval budget bounds the number of hops.

09

Design principles

Three principles recur throughout the architecture:

01 Groundedness over fluency

The system is engineered to refuse rather than fabricate. Forced first-hop retrieval, the absolute grounding rule, and verbatim pass-through of the retrieval agent's answer all serve the same end: the answer the user reads is traceable to a retrieved source.

02 Authorization by construction

Access control lives inside the retrieval call and is pushed down into the vector store, is fail-closed on error, and is composed (ACL × selection × project) so that no single bug widens visibility. Sensitive personal data is removed before indexing.

03 Observability and configurability

Every retrieval is traced; every model is a resolvable slot; every agent is a registry entry. The system's behavior can be reconfigured (different models, different fusion strategy, different set of agents) without changing its code.

Conclusion

Corporate GPT demonstrates that agentic RAG in an enterprise setting is less about a single clever retrieval trick and more about *control*: control over when and how often to retrieve, over which specialized agent handles a request, over which documents a user may see, and over what data ever enters the index in the first place. By combining a planning orchestrator, a human-in-the-loop preview/select/answer flow, a hybrid and rerankable retrieval engine, pushed-down fail-closed access control, and full tracing, the system delivers answers that are grounded, authorized, and auditable: the properties that distinguish a trustworthy corporate knowledge assistant from a fluent guesser.

THE RESULT

Agentic RAG in an enterprise setting is less about a single clever retrieval trick and more about *control*.