Retrieval-Augmented Generation (RAG) by Michaël Bettan

Definition: RAG pairs an LLM's parametric knowledge with an external retrieval system. At query time it finds the relevant passages, injects them into the prompt and asks the model to answer **only from that context, with citations**. The model is grounded on private, current data **without fine-tuning**, and every claim can be traced to a source.

Why RAG? (Standalone LLM limits): Solves **private data** access, **freshness** (frozen weights), **hallucinations** (adds citations), and **scale/cost** (avoids expensive long-context stuffing).

> MENTAL MODEL
> Fine-tuning is a semester of study that teaches a student *how* to think and speak like a specialist. RAG hands that student an **open-book exam** with exactly the right pages bookmarked.

Workflow: 1. Retrieval (find chunks in store) → **2. Augmentation** (hydrate prompt with chunks) → **3. Generation** (LLM writes grounded, cited answer).

1. When to use RAG (vs. fine-tuning, long context and live tools)

Approach	Best for	Update speed	Cost per query	Traceability
Fine-tuning	Tone, format, behavior, output schemas	Hours to weeks (retrain)	Low inference, high training	None: facts can't be traced to weights
Long-context stuffing	Whole-document reasoning on a small corpus	Instant	High: grows with prompt size	Coarse
Context caching	Reusing a large, fixed prefix	Minutes	Lower (cached input discount)	Coarse
RAG	Facts from large, changing, permissioned reference corpora	Seconds to minutes (index update)	Low: ~1.5k–3k relevant tokens	**Exact** (doc, chunk, page)
Live tool call (API / SQL)	Live transactional state (order status, stock levels, account balances, booking availability)	**Real-time** (system of record)	Low (1 API/DB call)	**Exact** (record ID + timestamp)

Why long context alone fails at scale: cost grows linearly with input tokens, time to first token grows with prompt length, recall degrades well before the advertised window is full ("lost in the middle"), and you can't apply per-user ACLs or exact citations to one giant prompt.

Choose long context when...	Choose RAG when...
Corpus is small (under ~50K–200K tokens)	Corpus is large or unbounded
Low query volume	High volume where unit cost and latency matter
The task needs the **whole** document at once	Questions target specific facts
Ephemeral per-session uploads	Shared corpus that changes continuously
Simplicity matters more than cost	Citations, audit and ACLs are mandatory

The production answer is often both: RAG narrows millions of documents to a few full sections (tens of thousands of tokens), then a long-context model reasons across them without fragmenting the evidence into tiny chunks.

Retrieval modes and context-window utilization

Information retrieval offers two complementary scoring families; production RAG usually runs both and fuses the rankings (§5):

Retrieval mode	How it scores relevance	Strong on	Weak on
Dense vector (semantic)	Embeds query and chunks into one vector space; ranks by cosine or dot-product distance	Paraphrase, synonyms, cross-lingual intent	Exact identifiers, rare tokens, negation
Sparse keyword (BM25)	Term overlap weighted by term frequency, inverse document frequency and length normalization	Product codes, error IDs, names, legal citations	Paraphrased or vocabulary-mismatched questions
Hybrid	Runs both in parallel, then fuses ranks (Reciprocal Rank Fusion)	Mixed natural-language plus identifier queries	Extra latency and two indexes to maintain

- **Context rot and window limits:** more context is not automatically better. Accuracy and recall degrade as token count grows (*context rot*), so curate what enters the prompt instead of filling the space available. On Claude 4.5 and newer models, a request whose input plus `max_tokens` exceeds the window is accepted and stops with `stop_reason: model_context_window_exceeded` if generation reaches the limit; earlier models return a validation error unless the `model-context-window-exceeded-2025-08-26` beta header is sent.
- **Long context vs. RAG cost model:** compare **(1)** per-query input tokens × monthly volume (after prefix caching), **(2)** the operating cost of an index pipeline, and **(3)** the migration effort if the corpus later outgrows the window.

RULE OF THUMB

RAG supplies **reference facts**, fine-tuning shapes **behavior**, long context does **deep reasoning** over a pre-filtered set, and **tool calls** fetch **live transactional state**.

COMMON TRAP: TREATING EMBEDDING DISTANCE AS VERACITY.

Nearest-neighbor scores rank how close a passage's *meaning* is to the query; they carry no information about whether the passage is *current*. An indexed copy of a mutable record (an order, a price, a balance) is a snapshot frozen at ingestion time, so it can never be authoritative. Keep reference knowledge in **RAG**; query live state through **API/SQL tools against the system of record**.

2. Architecture: offline indexing and online answering

```
OFFLINE (once per document change):
  Sources (PDF, wiki) → Parse (layout/tables) → Chunk (+ doc_id, acl)
                      → Embed → Index (dense vectors + sparse BM25)

ONLINE (every user query):
  Query → Rewrite? → Embed → Hybrid search (top-K)
        → Cross-encoder rerank (top-N) → LLM → Answer + citations
```

Phase	Steps	Key rule
Offline: ingest and index	Parse → chunk → attach metadata (`doc_id` , `page` , `date` , `acl`) → embed → store vectors, text and metadata	Expensive work happens once, so online queries stay fast

Phase	Steps	Key rule
Online: retrieve and generate	(Rewrite) → embed the query → hybrid search for top-K → rerank to top-N → build the prompt → generate with citations	Embed queries with the **same model and version** as the corpus

RAG paradigms (Gao et al. survey)

Paradigm	Shape	Adds	Weakness
Naïve	`index → retrieve → generate`	Basic chunking + vector search	Noisy context, low precision and recall
Advanced	Linear, with pre- and post-retrieval stages	Query rewriting, hybrid search, reranking, compression	Fixed one-shot path
Modular	Composable, non-linear modules	Routing, iteration, memory, **agentic loops**	More complexity, latency and evaluation cost

COMMON TRAP

Agentic RAG isn't a fourth official paradigm. It's an implementation of **modular** RAG in which an agent orchestrates the modules.

3. Indexing: parsing, chunking and enrichment

Parsing comes first

Bad parsing corrupts everything downstream, and silently: if a table is dropped or a scan is misread, the LLM won't complain; it will invent plausible facts.

- **Fast text extractors** (e.g. `pypdf`) work for clean, digital, single-column PDFs. They return nothing on scans and scramble tables.
- **Layout-aware parsers with OCR** (Docling locally; Document AI, LlamaParse or Gemini in the cloud) handle tables, multi-column layouts and scans.
- **Tiered pattern:** try the fast extractor; if it yields too little text or fails quality checks (lexical validity, character entropy, structural anchors, never raw word count alone), route the file to the layout/OCR parser, and hard-stop for human review if composite confidence remains low.
- **Parser ≠ framework:** Docling parses and chunks (stage 1 only); LlamaIndex orchestrates the whole pipeline (and defaults to `pypdf` unless configured otherwise).

Google Cloud parser (`GA`)	Target document layout	Hard page & payload limits	Chunking & dual-sink rule
Document AI Layout Parser	Multi-column PDFs, nested tables, section hierarchies (Agent Search & RAG Engine)	Online sync: **`15` pages** (**`30`** with `imageless_mode`), **`20 MB`** standalone PDF cap (`40 MB` general Document AI sync ceiling); GCS batch: **`500` pages**, **`1 GB`** ; `ImportRagFiles` : **`20 MB` / `500` pages**	Configure layout-aware chunking (`100` – `500` tokens in Agent Search `layoutBasedChunkingConfig` ; `500` – `1,024` tokens in RAG Engine) with **`include_ancestor_headings = true`** so split table rows retain H1/H2/H3 headers

Google Cloud parser (`GA`)	Target document layout	Hard page & payload limits	Chunking & dual-sink rule
Document AI Enterprise OCR Parser	Scanned PDFs, TIFFs, and faxes lacking embedded text glyphs	Online sync: **`15`** / **`30`** **pages**, **`40 MB`** ; GCS batch: **`500`** **pages**, **`1 GB`**	Extracts reading-order text reliably, **flattening multi-column tables** into linear text rather than structured cell grids
Gemini Document Understanding	Infographics, charts, blueprints, visually dense slide decks	Up to **`50 MB`** via `gs://` (~ `258` – `1,120` tokens/page by `MediaResolution`)	Interprets visual layout relationships and chart series that text/OCR parsers drop
Document AI Custom Extractor → BigQuery	Standardized forms (invoices, W-2s, claims, contract headers)	Online sync: **`15`** **pages** (`30` with `imageless_mode` ; `10` with checkboxes); GCS batch: **`200`** **pages**	**Dual-sink pattern:** sink typed JSON entities into **BigQuery** for exact corpus-wide SQL aggregations (`AVG` , `SUM` , `COUNT` across 50,000 files) alongside vector clause chunks

Chunking strategies

Strategy	How it works	Use when
Recursive character / fixed-size (baseline)	Split on paragraphs, then lines, then words	Default starting point
Overlap	Neighboring chunks share ~10–20%	Facts that span boundaries
Semantic	Split where embedding similarity between sentences drops (topic shift)	Long, mixed-topic prose
Structure-aware	Split Markdown by headings, source code by AST/function/class, and legal docs by clause; keep tables whole	Codebases, docs with markup, manuals, tables
Parent-child (hierarchical)	Embed small children (128–256 tokens), return the larger parent (1–2K tokens)	Precise matching **and** full context
Proposition	An LLM rewrites text into atomic facts	High-precision factual lookup

RULE

Match chunker to document grammar: fixed-size = uniform text; semantic = narrative prose; AST = code/markdown; hierarchical = numbered clauses & policies.

	Small chunks (100–256 tokens)	Large chunks (1–2K tokens)
Precision	**High**	Lower (one vector averages several topics)
Context	Low (pronouns lose their referents)	**High**
Tokens per answer	Low: more sources fit	High: fewer sources fit
Main risk	Severed facts	Noise, lost in the middle

HARD CONSTRAINT

A chunk must fit the embedding model's input limit (`2,048` tokens for `gemini-embedding-001` and `text-embedding-005` ; `8,192` tokens for `gemini-embedding-2`). Because `autoTruncate=true` by default, longer chunks are **silently truncated**, and their tail is never searchable.

Enrichment at index time

Technique	What it adds	Problem it fixes
Metadata (`source` , `date` , `acl` , `region` , `is_latest`)	Structured fields on every chunk	Filtering, recency, access control
Contextual headers (contextual retrieval)	Document title, section path and a short summary prepended before embedding **and** BM25	Ambiguous chunks ("Revenue grew 12%": of whom, when?)
doc2query	LLM-generated questions each chunk answers	Question vs. passage wording gap
SPLADE	Learned expansion terms in a sparse vector	BM25's blindness to synonyms

- **Match on the expansion, answer from the original:** generated text can be wrong, so always send the **original** chunk to the LLM.
- **Parent-child mechanics:** store `parent_id` on each child, **deduplicate by parent**, and ACL-check the parent too.

Contextual header example (what gets embedded/indexed): `"Revenue grew 12%"` → `"[ACME Corp 2025 annual report › EMEA results › Q3] ACME's EMEA revenue grew 12% over Q2 2025."`

The raw chunk can't match "ACME EMEA Q3 revenue" by meaning or by keyword; the contextualized one matches both. To keep ingestion cost low when generating contextual headers or `doc2query` expansions across a large document, cache the full source document in the prompt prefix and run offline backfills through asynchronous **batch APIs** (~50% discount).

RULE OF THUMB

HyDE expands the **query** at runtime (latency on every request). Document expansion expands the **corpus** once at index time (cost at ingestion only).

4. Embeddings and vector search

An **embedding** is a dense vector (typically 768–3,072 dimensions) in which texts with similar meaning sit close together, regardless of wording. Vector search is **Stage 1 semantic recall**: it finds matching passages **even when query and document share no keywords** (where BM25 returns nothing). The embedding model sets the **ceiling** on semantic recall; no prompt can rescue a poor vector space.

Concept	What to know
Same model both sides	Query and corpus vectors must come from the same model **and version**. Otherwise scores still compute but rankings are near-random
Task types (Vertex AI)	`RETRIEVAL_DOCUMENT` (+ `title`) for chunks, `RETRIEVAL_QUERY` for queries. Unset defaults to `RETRIEVAL_QUERY` , so the corpus gets query-style vectors

Concept	What to know
Distance-metric math	On unit L2-normalized vectors ($\|u\|_2 = \|v\|_2 = 1$), cosine similarity equals dot product ($u \cdot v$) and induces the exact same monotonic ranking as squared Euclidean ($\|u - v\|_2^2 = 2 - 2(u \cdot v)$), making `DOT_PRODUCT_DISTANCE` the fastest SIMD path in ScaNN. Un-normalized vectors (including raw Matryoshka slices) require `featureNormType: "UNIT_L2_NORM"` or `COSINE_DISTANCE` so large-magnitude vectors do not dominate. Watch cutoff direction: `similarity ⩾ τ` vs. `vector_distance_threshold ⩽ d_max`

Concept	What to know
Domain fit	Benchmark on **your** labeled queries; leaderboards are only a shortlist. Add BM25 or an adaptor for jargon

Google Cloud embedding model (`GA`)	Modalities & language	Max dims & input limit	Production specs & Firestore / Matryoshka rules
`text-embedding-005`	English text & code	`768` **dims** · `2,048` tokens	Supports Matryoshka `output_dimensionality` and 8 `task_type` values (`RETRIEVAL_DOCUMENT` , `RETRIEVAL_QUERY` , `QUESTION_ANSWERING` , `FACT_VERIFICATION` , `CLASSIFICATION` , `CLUSTERING` , `SEMANTIC_SIMILARITY` , `CODE_RETRIEVAL_QUERY`); unit L2-normalized at `768` dims; degrades on non-English corpora
`gemini-embedding-001`	Multilingual text & code	`3,072` **dims** · `2,048` tokens	Asymmetric `RETRIEVAL_DOCUMENT` vs. `RETRIEVAL_QUERY` (Matryoshka `768` / `1,536` / `3,072`); **to index in Firestore Vector Search, truncate `output_dimensionality ⩽ 2,048` and L2 re-normalize**
`gemini-embedding-2`	Omni-modal (text, image, audio, video, PDF)	`3,072` **dims** · `8,192` **tokens**	Unified cross-modal vector space with auto-embeddings in `global` / `us` / `eu` ; segment long audio/video into clips tagged with `start_offset` / `end_offset`

Exact vs. approximate search

Index	How it works	Strength	Trade-off
Exact k-NN	Compare with every vector	100% recall	Linear cost; unusable past ~10^5 vectors
HNSW	Multi-layer proximity graph	Best recall vs. latency	High RAM
IVF	k-means clusters; search the nearest `n_probe`	Fast partitioning	Recall drops if `n_probe` is too small
PQ	Compress sub-vectors to codebook IDs	Large memory savings	Lossy ranking
ScaNN	Google's anisotropic quantization	Highest QPS; powers **Vertex AI Vector Search** and AlloyDB AI	Tuning required

- **Tuning knobs:** HNSW `ef_search` and IVF `n_probe` . Higher means more recall and more latency.
- **Vertex AI Vector Search 1.0 (`ScaNN`) tuning:** use `treeAhConfig` (ScaNN quantization tree + asymmetric hashing for `100K` –billions of vectors at sub-10 ms latency) vs. `bruteForceConfig` (`< ~50K` vectors or 100%-recall ground-truth benchmarking). Tune `leafNodeEmbeddingCount` (default ~ `1,000` ; smaller leaves raise recall at higher RAM/build time), `leafNodesToSearchPercent` (e.g., `5` – `10` %; linear recall vs. CPU trade-off), and `approximateNeighborsCount` (candidate pool rescored at full precision; set `> k`).
- **ScaNN filtering, crowding & endpoint billing:** pre-filter with token `restricts` (`namespace` , `allow` , `deny`) and `numeric_restricts` , cap per-source dominance via `per_crowding_attribute_num_neighbors` (e.g., max 2 chunks per `document_id`), and choose `STREAM_UPDATE` (seconds freshness, higher hourly node cost, background compaction) vs. `BATCH_UPDATE` (cheaper immutable GCS rebuilds, minutes-to-hours lag). Deployed `IndexEndpoint` replicas bill **24/7 VM node hours even at 0 QPS** (`minReplicaCount ⩾ 1`), `dimensions` and `distanceMeasureType` **are immutable** after creation, Private Service Connect (`pscAutomationConfigs`) avoids VPC Peering `/26+` CIDR exhaustion, and Vector Search 1.0 stores IDs + vectors + restricts only (hydrate chunk text from GCS/DB).
- **Quantization ladder:** `float32` (1×) → `float16` (½) → `int8` (¼, the default at 100M+ vectors) → PQ (~10–30× smaller) → binary (1/32). The more you compress, the more you need **oversample + full-precision rescoring**.
- **Scale-out: shards** scale corpus size, **replicas** scale QPS and availability. Hash sharding fans out to every shard (tail latency = slowest shard). For multi-tenancy, use **namespaces** for mid-size tenants and **dedicated shards** for large or regulated ones; mixing tiny and huge tenants in one HNSW graph hurts filtered search.
- **Vector store tiers: (1) local** (Chroma) for prototypes; **(2) dedicated vector DBs** (Vertex AI Vector Search, Qdrant, Milvus, Pinecone, Weaviate) for scale and payload filtering; **(3) vector-enabled operational stores** (`pgvector` , AlloyDB AI, BigQuery, Redis, Elasticsearch) that keep vectors and rows in one ACID system, with no dual-write lag.

> COMMON TRAP
> "ANN recall" (agreement with exact k-NN, fixed with `ef_search` / `n_probe`) isn't "retrieval recall" (agreement with human relevance labels, fixed with chunking, embeddings and hybrid search). Diagnose which one is failing.

5. Retrieval: hybrid search, RRF and reranking

Hybrid search (dense + sparse, in parallel)

Decision rule: use hybrid when traffic mixes **exact identifiers/clauses** (sparse) with **conceptual questions** (dense). The trade-off is **quality vs. latency vs. upkeep** (two indexes to keep in sync); dense-only is simpler and faster but misses exact strings.

	Dense (vector)	Sparse (BM25)
Matches	**Semantic recall:** meaning, synonyms, paraphrases (when user wording ≠ document wording)	**Lexical recall:** exact terms, IDs, SKUs, error codes, clause numbers, names
Misses	Rare exact strings (`ERR-409` , `§4.2(b)`)	Synonyms ("car" vs. "automobile")
Score	Cosine in [-1, 1]	Unbounded [0, ∞)

BM25 improves on TF-IDF with term-frequency saturation, inverse document frequency (rare terms count more) and document-length normalization. Both searches run **in parallel** over the same metadata-pre-filtered corpus, and their ranked lists are merged via RRF.

Reciprocal Rank Fusion (RRF)

Raw cosine and BM25 scores can't be added (different scales), so RRF merges the candidate lists using **rank positions only**:

$$\mathrm{RRF}(d) = \sum_{m \in M} \frac{1}{k + r_m(d)}$$

With **k = 60**: chunk A ranked #1 in BM25 and #50 in dense scores 1/61 + 1/110 ≈ 0.0255. Chunk B ranked #3 and #2 scores 1/63 + 1/62 ≈ 0.0320. **B wins**, because RRF rewards **consensus** across retrievers.

- **Weighted RRF (`rrf_ranking_alpha` / `hybrid_search.alpha` in `[0.0, 1.0]`):** on Vector Search 1.0 `HybridQuery` (`rrf_ranking_alpha`) and RAG Engine `RagRetrievalConfig` (`hybrid_search.alpha`), `0.5` weights dense semantic and sparse lexical ranks equally, `0.0` uses sparse keyword ranks only, and `1.0` **silently collapses hybrid search to dense-only** (dropping exact SKU, error-code, and clause matches).

> COMMON TRAP
> RAG-Fusion and hybrid search both use RRF but fuse different lists. **Hybrid** fuses **retrievers** (BM25 + dense) for one query. **RAG-Fusion** fuses results from **several LLM-generated query variants**.

Two-stage retrieval: bi-encoder recall, cross-encoder precision

	Bi-encoder (vector search)	Cross-encoder (reranker)
Input	Query and document encoded **separately**	Query + document encoded **together** (full cross-attention)
Speed	Fast: document vectors precomputed, ANN lookup	Slow: one transformer pass per `(query, chunk)` pair
Role	**Stage 1:** recall over millions (K ≈ 50–100)	**Stage 2:** re-score RRF-fused pool (~100–150 → top N ≈ 3–10)
Examples	SBERT (symmetric, 1 encoder), DPR (asymmetric, 2 encoders), Vertex AI embeddings	Vertex AI Ranking API, Cohere Rerank, BGE-Reranker

Stage	Typical size	Tuned for
First-stage K (per retriever)	50–100	**Recall**: the reranker can fix order, not absence
Reranker input	~100–150 after RRF list fusion	Cost grows linearly
Final N (to the LLM)	3–10	**Precision**; more N means more tokens and more noise

- **Google Cloud reranking trade-offs:** Vertex AI / Agent Platform **Ranking API** (`semantic-ranker-*`) cross-encodes **~200 candidates in `< 100 ms`** (requires `discoveryengine.googleapis.com` enabled in the project or requests fail with `HTTP 403`), **Agent Search built-in ranking** applies `BoostSpec` freshness/attribute decay inline, whereas an **LLM reranker** adds `1-2+ s` **TTFT** and high input-token cost on large `top_k` pools.
- **Dynamic N & pre-generation refusal gate:** stop adding chunks when the cross-encoder relevance score falls below a calibrated threshold τ, and **abstain immediately** if even the top-1 chunk falls below τ.
- **Never cross-encode the whole corpus:** it needs one transformer pass per `(query, chunk)` pair with nothing precomputed. Pipeline: **parallel BM25 + dense → RRF fusion → cross-encoder re-scores the fused pool.**

Query transformation (before retrieval)

Technique	What it does	Best for
Query rewriting	Uses chat history to make follow-ups self-contained ("How about Emily?" → a full question)	Conversational assistants
Routing	Sends the query to the right backend: domain index, SQL, graph, web, or live API tool	Mixed sources and query types (stable reference docs vs. live transactional state)
HyDE	LLM drafts a hypothetical answer; embed the **draft** and search with it	Short or abstract queries
Multi-query	3–5 rephrasings of the **same** intent, fused with RRF	Phrasing sensitivity
Decomposition	Splits a **multi-part or multi-hop** question into sub-queries	"Compare EMEA and APAC policies"
Self-query	LLM extracts a metadata filter + semantic query	"EMEA policies updated after 2024"

Expansion adds an LLM call and multiplies candidates, so **deduplication, RRF and reranking** become mandatory.

Metadata filtering (pre-filter vs. post-filter)

Mode	How	Verdict
Pre-filter (filtered ANN)	Filter applied **during** the BM25 and ANN index scan	Returns K valid results; **mandatory for ACLs (`acl_groups`) and excluding stale/superseded data (`is_latest = true`).** Vertex AI: `restricts` / `numeric_restricts`
Post-filter	Global top-K, then drop non-matching results	Can return **zero** results (recall collapse); **never acceptable as the primary filter for ACLs or stale data**
Partition / namespace	Separate index per tenant, language or region	Fastest, hard isolation; low-cardinality keys only

WORDING TRAP

Never say *"post-filter for stale data"* or *"post-filter for ACLs"*—exclude stale revisions (`is_latest = true`) and unauthorized docs via **metadata pre-filtering** inside the index scan, then use **recency time-decay boosting** during ranking and a **live ACL re-check** on the final top-N chunks.

6. Generation: prompt hydration, refusal behavior and groundedness

- **Prompt hydration:** wrap each top-N chunk in `<document>` delimiters with its `Source ID` and `effective_date` . Instruct the model to treat those blocks as **untrusted data, never instructions** (see §11), answer **only** from them, cite every claim, and **abstain** when they don't support an answer.
- **Context ordering & prompt-cache prefix:** attention is strongest at the start and end of the prompt, so put the best chunks there ("lost in the middle"). Prefix caches evaluate **tools → system → messages**: keep static tools, instructions and pinned docs in the cached prefix, and put **retrieved chunks + the user query at the uncached tail** (per-query chunks or timestamps in `system` invalidate the whole cache).
- **Compression:** extract only the relevant sentences, or summarize each chunk, to cut tokens and noise.

Most hallucinations start in retrieval: if the evidence is missing, the model fills the gap unless stopped by refusal gates. Fix recall first, then constrain, verify and refuse unsupported answers.

Layer	Technique
Retrieval	Pre-filtered hybrid search + cross-encoder reranking, measured with `Recall@K`
Pre-generation refusal gate	If top cross-encoder score $< \tau$, short-circuit and return a deterministic refusal (*"No relevant documents found"*)
Context hygiene	Deduplicate; pre-filter stale versions (`is_latest = true`); pass `effective_date` to surface conflicts; screen chunks fail-closed (§11)
Prompt & output	Closed-world rule + explicit refusal instruction; low temperature; structured output with required `citations` checked in code
Groundedness verification	Claim-level entailment check: NLI model, LLM judge **Faithfulness**, or Vertex AI **Check Grounding API** (`support_score` 0–1)
Post-generation refusal gate	Below groundedness threshold → block answer and fall back to safe refusal (showing closest sources) or human review

Citations, weakest to strongest:

Level	Mechanism	What it guarantees
1	Free-text `[n]` brackets	Nothing: numbers can be invented
2	Cite chunk IDs from the supplied set	The source exists (IDs checked in code)
3	Exact quote spans	The quoted text exists (substring check)
4	Native grounding metadata	Claim-to-source mapping from the API: Gemini `groundingMetadata` (chunks + supports), Claude Citations API (returns source spans without billing output tokens for quoted text)

COMMON TRAP

"The answer has citations, so it's grounded." A citation proves the source **exists**, not that it **supports** the sentence. Check entailment (sampled, or always for high-stakes flows).

RULE OF THUMB

Refusal ("I couldn't find this in the available documents", plus the closest sources) is a **first-class safety feature**, enforced both before generation (reranker threshold τ) and after generation (groundedness check).

7. Agentic RAG and agent memory

A fixed pipeline fails silently when retrieval returns the wrong chunks. **Agentic RAG** places retrieval *inside* the reasoning loop so the agent decides **when** to search, **what** to query, whether the returned passages are trustworthy, and when to retry or switch tools.

Need	Pattern
Single-turn question over a knowledge base	**Standard RAG** (one-shot: retrieve → augment → generate)
Complex question needing grading, retries or several sources	**Agentic RAG** (iterative retrieval inside the reasoning loop)
Multi-step work with side effects (send, write, update)	**Agent with tools**, with RAG exposed as one tool

- **Four runtime jobs of the agentic loop:** 1. **Source validation:** inspect metadata (`effective_date` , doc type, authority) to drop superseded or low-trust hits (e.g. discarding an old blog post when a current policy doc is present). 2. **Conflict reconciliation:** detect contradictory figures across chunks (e.g. an early budget proposal vs. an audited financial report) and ground the answer in the authoritative record. 3. **Multi-hop decomposition & synthesis:** split comparative or multi-part questions (*"compare our features and pricing with Competitor X"*) into separate targeted searches before synthesizing a structured comparison. 4. **Gap detection & tool fallback:** recognize when the internal corpus lacks recent or external facts and invoke a live web search or API tool.

Pattern	Mechanism	Fixes
Corrective RAG (CRAG)	A grader labels retrieval Correct / Ambiguous / Incorrect; Incorrect triggers query rewrite + web search	Answering when the local index lacks the facts
Self-RAG	The generator emits reflection tokens: `[Retrieve]` , `[IsRel]` , `[IsSup]` , `[IsUse]`	Needless retrieval and unsupported claims
Adaptive RAG	A complexity router picks no retrieval, single-step or iterative multi-hop	Over-paying on simple queries, under-retrieving on hard ones

- **Two implementation patterns: (1) explicit state graphs** (deterministic `retrieve → grade/rewrite → generate` nodes, e.g. LangGraph) for predictable control flow, or **(2) autonomous tool-calling loops** (retriever exposed as a tool, e.g. ADK or LlamaIndex agents) where the model decides how many searches to run.

- **Retrieval as a tool:** the model decides whether to call it by reading the tool's **description**. A vague description means retrieval never fires and the model answers from memory.
- **Always-on vs. on-demand retrieval:** always-on retrieves every turn (predictable, wasteful on chit-chat); on-demand is cheaper and handles multi-hop naturally, but risks under- or over-retrieval.
- **New failure surface:** the agent loop adds latency, token cost, and reasoning failures (runaway search loops, discarding valid evidence, or task drift). Enforce a **hard cap on search iterations** and **evaluate the trajectory, not just the final answer**: required and forbidden tools, call count, latency and cost (Vertex AI `trajectory_*` metrics).

Agent memory types and the persistent scratchpad

Stateless models forget preferences and repeat mistakes; dumping raw tool outputs into the window causes goal drift. Production agents separate memory by type:

Memory type	What it holds	Scope	Typical backing store
Short-term (working)	Current turn history and active context	Single session	In-memory message list
Episodic	Past interactions and trajectories ("what was tried")	Cross-session	Key-value / session store; managed extraction (Vertex AI Memory Bank, Mem0, Zep)
Semantic	Facts, user profile, domain knowledge searched by meaning	Cross-session / global	Vector store; **separate collections** for user preferences vs. domain facts
Procedural	Rules and instructions (how to do the task)	Global	System prompt / instruction store, refined via reflection

- **Managed memory** handles fact extraction, deduplication and summarization; hand-rolled pipelines accumulate noisy, obsolete facts that saturate the prompt.
- **Time-weighted salience decay:** rank memories by similarity **and** recency / recall frequency so stale memories fade.
- **Persistent scratchpad (`Read → Refine → Repeat`):** for long tasks the agent writes its plan to an external file (`scratchpad.md`), condenses tool output into a few lines of state after each step, and **re-reads it before acting**. Humans can edit the file to correct drift, and the agent can resume from it after a crash.

> RULE OF THUMB
> If the system only needs to **know** something, use RAG. If it needs to **do** something, use an agent with RAG as a tool. Start memory simple (message list + key-value/JSON + vector store) before adding graph layers.

> COMMON TRAP
> Blaming the retriever when the agent never called it. Check the trace for a tool call first; if there isn't one, fix the tool description.

8. GraphRAG and structured data

GraphRAG stores entities (nodes) and relationships (edges) next to—or instead of—vectors. Where vector similarity only floats topically related paragraphs to the top, a **property graph** (e.g. Neo4j queried via **Text-to-Cypher** in LlamaIndex `PropertyGraphIndex` or LangChain `GraphQA`) traverses explicit relational chains (`Person -[:WORKS_ON]→ Project ←[:FUNDS]- Org`) in a single deterministic hop, and builds **corpus-wide summaries** ("main themes across all Q3 filings") via community summaries. For purely relational queries (org charts, supply chains, legal discovery, gene-disease or company-event networks), traverse the graph directly rather than relying on vector similarity.

	Vector RAG	GraphRAG	Hybrid
Strength	Local semantic match	Multi-hop entity chains (org charts, supply chains, legal/financial links), global summaries	Explicit relations + grounded passage quotes
Cost	Low to moderate	High (entity/relation extraction, rigid schema & graph upkeep, higher latency)	Highest (two stores to sync)

Structured data: embeddings are bad at `SUM`, `AVG`, sorting and joins. Route quantitative questions to **Text-to-SQL**: retrieve the relevant schema → generate **read-only** SQL → execute → put the exact result in the prompt. BigQuery and AlloyDB AI can also combine SQL filters and vector search in one query. For choosing between parameterized query tools and NL2SQL, and the guardrails for model-written SQL, see **Knowledge engineering: structured data tools**.

Multimodal RAG: multimodal embeddings put text and images in one space, so a text query can retrieve diagrams, which Gemini then reads together with the text.

9. RAG on Google Cloud

This guide stays vendor-neutral. Google Cloud's managed retrieval services (Agent Search, RAG Engine, Vector Search 1.0 and Agent Retrieval, Grounding with Google Search, the Ranking and Check grounding APIs, and in-database vectors) are covered in depth in the **Agent Platform guide: Grounding & Retrieval**.

Google Cloud vector / RAG store	Primary architecture & sweet spot	Hard limits, billing & lock-in traps
Vector Search 1.0 (`Index` + `IndexEndpoint`)	High-QPS sub-10 ms ScaNN ANN (`treeAhConfig` / `bruteForceConfig`) from `100K` to billions of vectors	Stores IDs + vectors + `restricts` only (hydrate text externally); **`IndexEndpoint` bills 24/7 VM replicas at 0 QPS**; `dimensions` & `distanceMeasureType` are **immutable**
Agent Retrieval (Vector Search 2.0)	Serverless `Collection` / `DataObject` store bundling vectors, text, and metadata with auto-embeddings (`gemini-embedding-2`), hybrid RRF, and `StreamableHTTP` MCP	Enforces custom ACLs via query-time metadata filters (no turnkey Workforce IdP sync); segment multimedia to fit the `8,192` -token embedding cap
RAG Engine (`Serverless` vs. `Spanner`)	Code-first managed corpus pipeline (`Import RagFiles` Layout Parser cap: **`20 MB`** / **`500`** **pages** per PDF)	**`Serverless`** **mode does NOT support CMEK**; corpus `encryption_spec` (supported on `RagManagedDb` in **`Spanner`** **mode**) is **immutable at corpus creation**, and switching project `RagEngineConfig` modes does not migrate existing corpora (requires a new `Spanner` -mode corpus and re-indexing)
AlloyDB AI / Cloud SQL (`pgvector`)	Operational PostgreSQL ACID transactions joining relational rows and vectors (`scann` on AlloyDB AI; HNSW / IVFFlat on Cloud SQL)	**Pin an explicit embedding model version** inside `google_ml_integration` `embedding()` calls so model alias updates never trigger silent vector drift
BigQuery **`VECTOR_SEARCH`**	Analytical warehouse SQL joins using **`IVF`** (`< 10M` rows) or **`TREE_AH`** (up to `200M` rows)	Cold/analytical latency runs hundreds of milliseconds to seconds; use for batch enrichment and SQL analytics, **not** sub-50 ms interactive chat QPS
Firestore Vector Search	Serverless operational NoSQL document + vector store for app state and user-scoped recall	Max **`2,048`** **dimensions** (truncate `gemini-embedding-001` via `output_dimensionality` ⩽ `2,048` and L2 re-normalize), max **`1,000`** **results** per query, and **no real-time snapshot listeners** on vector queries

- **Turnkey enterprise search & live web grounding:** choose **Agent Search** (formerly Vertex AI Search) when you want managed SaaS connectors (SharePoint, Drive, Confluence), hybrid retrieval, and **native Workforce / Workspace ACL enforcement** without building an ingestion pipeline; use **Grounding with Google Search** (`google_search` in ADK) for fresh public web facts.

> **NOTE**
> Older material refers to **PaLM 2**, **Matching Engine** and **Vertex AI Search**. Those are now **Gemini**, **Vector Search** and **Agent Search** on Gemini Enterprise Agent Platform.

10. Federated retrieval vs. ingested RAG

- **Ingested RAG: copy** source content into your own index (parse → chunk → embed → index). You own ranking, freshness and ACL replication.
- **Federated retrieval:** at query time, call each **live source's** search API with the user's identity and merge the results. Nothing is copied.

Dimension	Ingested RAG	Federated Retrieval
Freshness & Sync	Delayed by sync interval (requires change capture)	Live (as fresh as source index); no data copied
ACL & Privacy	ACLs replicated to chunks (revocation lag is a risk); another data copy	Native permissions per user token; immediate revocation, but query text sent to 3rd party
Latency & Ranking	Predictable latency; best ranking (hybrid/reranker over one index)	Fan-out tail latency (slowest source); scores not comparable across sources
Operations	Pipelines, chunking, re-embedding, DB migrations	API integrations, OAuth tokens, schema changes

Hybrid patterns (usually the right answer): ingest stable, high-value corpora and federate volatile or permission-complex systems; or index metadata only and fetch content live with the user's token; or ingest with ACL pre-filtering plus a live permission re-check on the final top-N. Vertex AI Search connectors support both data ingestion and federated search.

> **RULE OF THUMB**
> Asked about **quality and latency**? Ingest. Asked about **not copying data** or **live permissions/state**? Federate. Both? Propose the hybrid.

> **COMMON TRAP**
> "Federated is always more secure." It avoids a copy, but it sends every query to third parties and multiplies tokens to manage. Ingested with pre-filtering plus a live re-check can be equally safe.

11. Security and access control

Vector similarity is permission-blind. Decide authorization **before** the prompt is built: once a restricted chunk reaches the model, no instruction reliably stops it from leaking through paraphrase.

ACL strategy	How	Strength	Weakness
Early binding	ACL groups copied onto chunk metadata; pre-filter in search	Fast; always K authorized results	Only as current as the last sync
Late binding	Check each candidate against the source at query time	Always current	Slower; can return zero results
Hybrid (recommended)	Pre-filter + live re-check of the **final top-N** only	Fast and current	More moving parts

- **Store group IDs, not user lists;** resolve user → groups at query time (short cache TTL). Apply deny-overrides-allow; isolate **tenants first** (separate index or namespace).
- **Derived artifacts inherit the most restrictive source ACL:** parent chunks, summaries, graph communities, caches and memories.
- **Revoked access hides** in chunk metadata, group caches, answer caches, chat history, agent memory and summaries. Fixes: a metadata-only **ACL fast-lane** (no re-embedding), webhook-driven cache purges, and the live top-N re-check. Revocation SLA must be **minutes**, tighter than the content freshness SLA.
- **Every cache key includes the user's permission scope;** otherwise the cache becomes an ACL bypass.

Indirect prompt injection is the primary RAG threat: input guardrails screen only the user's prompt, not the retrieved passages and tool outputs that carry hidden instructions. Defend in layers, **cheap and fast first**:

1. **Deterministic filters** (~1 ms): at ingestion and query time, strip hidden text, HTML comments and zero-width characters; regex for known attack patterns.
2. **Small classifier** (e.g. Model Armor) over the user prompt **and** every retrieved chunk/tool payload before prompt assembly. **Fail closed**: halt if it errors or times out.
3. **Isolated generation:** chunks in `<document>` delimiters as **untrusted data**; cap chunks and tokens per request.
4. **Output checks:** PII/DLP filtering, citation validation and grounding check before the answer is returned.

Threat	Defense
Confused deputy / excessive agency (injected text makes the agent misuse its own credentials)	**Least privilege at the tool registry:** read-only tools and credentials for RAG agents; no send/write tools. A system-prompt instruction is not a control
Index poisoning (fake documents injected)	Ingestion approval, provenance, write-audit logs
Cascading injection in multi-agent flows	Validate at **every** handoff

> COMMON TRAP
> Post-filtering ACLs after a global top-K. A user who can see 1% of the corpus gets zero results, and response timing can reveal that restricted documents matched.

Fairness and decision logging

- **Fairness is a pipeline property:** a model that passes vendor bias benchmarks can still produce skew. Audit the **corpus** (uneven coverage), **system prompt**, **few-shot examples** and **downstream routing**.
- **Replayable decision logs:** request, exact retrieved chunks (IDs, revisions), response and downstream branch, keyed by a correlation ID. Minimize fields, cap retention (TTL) and restrict access (RBAC).
- **Slice metrics by subgroup:** good overall accuracy can hide high error rates for a small cohort.

12. Freshness, index updates and migration

Freshness SLA = time from a source change to it being searchable. Lag comes from change detection, parsing, embedding and index visibility.

Update mode	Typical lag	Use for
Full rebuild	Hours	Small corpora, embedding-model changes
Incremental batch	Minutes to hours	Most documents (policies, wikis)
Streaming upsert/delete	Seconds	Fast-changing reference content (new tickets, KB articles), **revocations** (Vertex AI `STREAM_UPDATE`, required by RAG Engine). Still not a transactional store (§1)

- **Deterministic chunk IDs** (`doc_id` + chunk index), and **delete all of a document's old chunks before inserting the new version** (preventing **orphan chunks**); propagate source deletes via **CDC tombstones** to evict **ghost vectors** and purge caches.
- **Re-embed only changed chunks** (compare content hashes); keep `is_latest` in metadata and pre-filter on it.
- **Reconciliation job:** periodically compare source counts and hashes with the index to catch missed events.

Index migration (new embedding model, chunker, index type or schema) means a **new index**, never an in-place patch:

1. Version the new index and its configuration.
2. Backfill from the **raw documents**. Vectors can't be converted between models, so always keep the source.
3. Dual-write changes to both indexes during the backfill.
4. Evaluate offline on the golden set, including ACL cases.
5. Shadow traffic, then canary.
6. Atomic alias swap, switching the **query embedding model at the same moment**.
7. Keep the old index warm for rollback, and purge the semantic cache.

> COMMON TRAP
> Cutting over the index but leaving the query encoder on the old model. Nothing crashes, but rankings become near-random.

13. Semantic caching and latency budget

Semantic cache: embed the incoming query and, if it's close enough to a past query, return the stored answer (~15 ms instead of ~1.2 s).

- **Entity masking** before embedding: "AAPL revenue in Q3 2025" → "[COMPANY] revenue in [PERIOD]", with the real values kept as exact-match keys, so a Q3 answer is never served for Q4.
- **Intent check:** a light cross-encoder confirms the candidate hit really means the same thing.
- **Thresholds by risk:** strict (≥ 0.90) for legal or financial queries, looser (≈ 0.75) for general FAQ.
- **ACL scope in the cache key**, or the cache serves one user's answer to another.

Eviction and invalidation policies (combine a size policy with a freshness policy):

Policy	Mechanism	Controls	Use for
TTL (time to live)	Entry expires a fixed time after it was written	Freshness (bounded staleness)	Content with a known update cadence
LRU (least recently used)	Evicts the coldest entry when capacity is reached	Memory size only, **not** freshness	Bounding cache cost under heavy traffic
Version-hash keys	Key includes the index version and content hashes of cited chunks; any change forces a miss	Instant invalidation on re-index or embedding-model migration	Corpora that are re-chunked or re-embedded
Event-driven purge	Document-update webhooks or CDC events delete entries citing the changed chunk	Near-real-time freshness	Policies, pricing, legal content

COMMON TRAP

Exact-string (hash) caching. "What is RAG?" and "Explain RAG" never match, so the hit rate stays near zero. Unmasked semantic caching fails the other way: confident wrong answers for a different entity or date.

COMMON TRAP: RELYING ON LRU FOR FRESHNESS.

LRU only decides *which* entry to drop when the cache is full; a popular, outdated answer stays hot indefinitely. Pair LRU with a TTL or version-hash keys. Provider prefix (KV) prompt caches are keyed on exact prefix bytes, so edited chunks simply miss; they bound cost, not correctness.

Typical online budget on a cache miss (directional; p95 time to first token under ~2 s):

Step	Typical latency	Main lever
Auth + group resolution	20–50 ms	Cache group membership (short TTL)
Cache lookup	10–30 ms	A hit skips everything below
Query rewrite	0 or 300–500 ms	**Gate it:** only for follow-ups or ambiguous queries
Query embedding	30–80 ms	Same-region endpoint
Parallel BM25 ‖ ANN	60–150 ms	`ef_search` / `n_probe`, shard by tenant
Cross-encoder rerank	100–250 ms	Fewer candidates (150 → 75), GPU batching
LLM time to first token	500–1,200 ms	Fewer chunks, prompt-cached prefix, smaller model; **stream**

The LLM dominates, so trimming the final N and caching the stable system prefix usually beat micro-optimizing search.

14. Evaluation

Measure **retrieval**, **grounded generation**, and **refusal/safety** separately: the best LLM can't answer from a chunk that was never retrieved, and must abstain when evidence is absent.

Metric Type	Metric	Focus
Reference-required (Offline Golden Set)	**Recall@K**, Precision@K, MRR, nDCG@K	**Retrieval:** Did we find the right chunks? Are they ranked well?
	Context Recall, **Answer Correctness**	**End-to-End:** Alignment with ground-truth answers in the golden set
Reference-free (Online)	**Faithfulness (Groundedness)**	**Generation:** Are claims entailed by context? (Vertex Check Grounding API)
	Context Relevance, **Answer Relevance**	**Generation:** Does context support the query? Does the answer address it?
	Refusal Accuracy	**Safety:** Cleanly refusing unanswerable queries instead of guessing

RAG Triad (TruLens): context relevance (query ↔ context), groundedness (context ↔ answer), answer relevance (query ↔ answer). **Ragas** adds context precision and **context recall** (needs reference answers).

COMMON TRAP

The RAG Triad has **no recall metric**. If retrieval returns one narrow chunk and the model summarizes it faithfully, all three scores can be perfect while most of the answer is missing. Always add Recall@K against labeled data.

- **Golden dataset:** 200–500 real questions written with subject-matter experts, labeled at document or passage level (so labels survive re-chunking). Include **unanswerable** questions (should abstain) and **permission-restricted** ones (must not leak). Add **multi-turn** cases so earlier constraints are checked after follow-ups.
- **Tune K on the Recall@K curve** (pick the knee), not by eye on a few queries.
- **Debugging funnel:** was the right chunk in the first-stage top-100? In the reranked top-N? In the final prompt? Missing early → fix parsing, chunking or hybrid search; present but answered badly → fix the prompt or model.
- **Grading hierarchy (cheapest first): (1) code assertions** (schema, valid citation IDs, exact match): free and drift-proof; **(2) LLM-as-a-judge** with a rubric, reasoning before the score, categorical labels, and a **different model family** from the generator (avoids self-preference bias); **(3) expert humans** for high-risk cases and the calibration set.
- **Per-stratum rollback gates:** set pass/fail floors **per document category** before any model or index cutover; a good blended mean can hide failing classes. Route failing classes to the stronger model instead of cutting everything over.

15. Failure modes and exam traps

Symptom / scenario	Root cause → correct fix	Tempting wrong answer
Answer misses half a table or condition	Severed context → structure-aware or parent-child chunking, overlap	Bigger LLM
"What about him?" retrieves nothing useful	Conversational query → query rewriting with history	Raising top-K
Exact codes or SKUs (`ERR-742`) not found	Dense-only search → hybrid (BM25 + dense) with RRF	Larger embedding model
High Recall@100, poor top-3 precision	Bi-encoder imprecision → cross-encoder reranking	Bigger chunks
Fact buried in a long prompt is ignored	Lost in the middle → fewer, better chunks at start and end	Longer context window
Long sessions degrade, then stop with `model_context_window_exceeded`	Context rot and window exhaustion → curate context, compaction, check `stop_reason`	Bigger window
Confident answer beyond the sources	Gap filled from memory → context-only rule + abstain + grounding check	Fine-tuning on the documents
Multi-hop / relationship question fails	Flat chunk search → decomposition, agentic loop or GraphRAG	Raising top-K
Agent loses its plan or recalls stale preferences	Cluttered context, no decay → scratchpad with `Read → Refine → Repeat` + memory decay	Bigger window
Outdated or deleted content returned	Ghost/orphan chunks → delete-before-insert, CDC tombstones, reconciliation	Post-filtering results
Wrong live order status, stock or balance	Mutable state in a vector index → live API/SQL tool on the system of record	Re-embedding with a larger model
Quality drifts with **no code change**	Corpus drift or an unpinned model alias → pin model versions, rerun golden evals, reindex	Bigger model
Rankings suddenly near-random	Query/index embedded with different models → same model and version; blue/green migration	Retuning `ef_search`
User sees a document they shouldn't	Post-filter, stale ACLs, unscoped cache → pre-filter + live top-N re-check + ACL-scoped cache keys	"Don't reveal restricted data" prompt
Instruction hidden in a retrieved doc hijacks the model	Only user input screened → fail-closed chunk screening, delimiters, read-only tools	Input guardrail only
High RAG Triad scores, answers incomplete	Triad has no recall → measure Recall@K on a golden set	Tuning the generator
Cached answer for the wrong entity or date	Unmasked semantic cache → entity masking + intent check + webhook invalidation	Exact-string cache
Popular cached answer stays outdated after a policy update	LRU-only cache (size policy, no freshness) → TTL + version-hash keys + event-driven purge	Larger cache capacity
Hallucinated total or average across 50,000 PDF contracts	Top-K retrieval sees a tiny slice and embeddings cannot compute `SUM` / `AVG` → sink typed fields via **Document AI Custom Extractor → BigQuery** alongside Layout Parser vector chunks (**dual-sink pattern**) and run read-only SQL	Raising `top_k` to 500 chunks
Firestore rejects `gemini-embedding-001` vectors, or truncated Matryoshka slices degrade `DOT_PRODUCT_DISTANCE` accuracy	Default `3,072` dims exceed Firestore's **`2,048` -dim** cap and raw slicing breaks unit length → set `output_dimensionality` ⩽ `2,048` **and L2 re-normalize** (or enforce `featureNormType: "UNIT_L2_NORM"` / `COSINE_DISTANCE` in ScaNN)	Storing vectors across Firestore subcollections
Hybrid search ignores exact SKUs despite BM25 + dense setup, or reranking returns `403` / adds `2 s` TTFT	`rrf_ranking_alpha` (Vector Search `HybridQuery`) / `hybrid_search.alpha` (RAG Engine) set to `1.0` drops sparse ranks, Ranking API (`semantic-ranker-*` , `< 100 ms` over ~200 candidates) needs `discoveryengine.googleapis.com` enabled, or an LLM reranker scored a large pool → set alpha to `0.5` and enable `discoveryengine.googleapis.com` for Ranking API	Cross-encoding the entire corpus

Symptom / scenario	Root cause → correct fix	Tempting wrong answer
Compliance audit blocks an existing RAG Engine `Serverless` corpus for missing CMEK, or online chat on BigQuery `VECTOR_SEARCH` hits multi-second latency	RAG Engine `Serverless` lacks CMEK, corpus `encryption_spec` is **immutable at creation** (switching `RagEngineConfig` modes does not migrate existing corpora), and BigQuery `VECTOR_SEARCH` (`IVF` / `TREE_AH`) has analytical latency → create a new **Spanner-mode** corpus with CMEK and re-index from raw sources; serve online chat via Vector Search 1.0, Agent Retrieval, or AlloyDB `scann`	Enabling CMEK in place on the Serverless corpus

16. Glossary

- **RAG (Retrieval-Augmented Generation):** architecture that retrieves top passages from an external store at query time and injects them into the model prompt so answers stay grounded, current, and cited without retraining weights.
- **Layout-aware & parent-child chunking:** structure-preserving splitting (`include_ancestor_headings=true`) and small-to-big retrieval where fine child chunks drive vector recall while broader parent sections hydrate the LLM prompt.
- **Dual-sink document pattern:** pairing structured field extraction (**Document AI Custom Extractor → BigQuery**, 15p sync / 200p async batch) for exact corpus-wide SQL aggregations alongside layout-chunked vector retrieval for clause-level Q&A.
- **Bi-encoder vs. Cross-encoder:** dual-tower embedding model computing independent query/chunk vectors for millisecond ANN search vs. joint query-passage transformer (**Ranking API** `semantic-ranker-*`) re-scoring top candidates (`~100-200`) for top-N precision.
- **Matryoshka representation (`output_dimensionality`) & `UNIT_L2_NORM` :** prefix-truncatable embeddings (`gemini-embedding-001` / `gemini-embedding-2`) that must be L2 re-normalized after slicing so `DOT_PRODUCT_DISTANCE` matches cosine similarity and fits store caps such as Firestore's `2,048` -dim limit.
- **ScaNN (`treeAhConfig`):** Google's quantized tree + asymmetric hashing ANN index (`leafNodeEmbeddingCount` , `fractionLeafNodesToSearch` , and `per_crowding_attribute_num_neighbors` diversity caps).
- **Hybrid search & Reciprocal Rank Fusion (RRF):** combining lexical sparse (`BM25` / SPLADE) and dense vector retrieval via rank-order fusion ($1 / (k + \text{rank})$) weighted by `rrf_ranking_alpha` (`0.5` balanced; `1.0` dense-only).
- **Pre-filtering vs. Post-filtering:** evaluating per-user ACL and tenant predicates inside the index traversal before top-K cutoff so restricted documents never leak and narrow-entitlement users never get empty results.
- **RAG Triad & `Recall@K` :** complementary evaluation metrics pairing reference-based `Recall@K` / `MRR` retrieval checks with reference-free Context Relevance, Groundedness (faithfulness), and Answer Relevance grading.
- **Semantic cache:** vector-indexed response cache partitioned by tenant/ACL scope, masked for volatile entities, and bounded by TTL plus event-driven invalidation.