Tech Brief:

Retrieval-Augmented Generation (RAG) Reference Architectures

Overview: Retrieval-Augmented Generation (RAG) systems augment large language models (LLMs) with domain-specific knowledge by first retrieving relevant document segments ("chunks") and then prompting an LLM to generate answers grounded in those chunks [1] [2]. This two-phase architecture (indexing vs. query) enables up-to-date, factual responses while mitigating hallucinations [2] [3]. The figure below illustrates a typical RAG pipeline: raw enterprise data are ingested and preprocessed (chunked and embedded), stored in a vector database, and at query time a user's query is embedded and used to retrieve top-*k* chunks. Those chunks are optionally re-ranked and passed, along with the query, to an LLM, which produces a final grounded answer [3] [4].

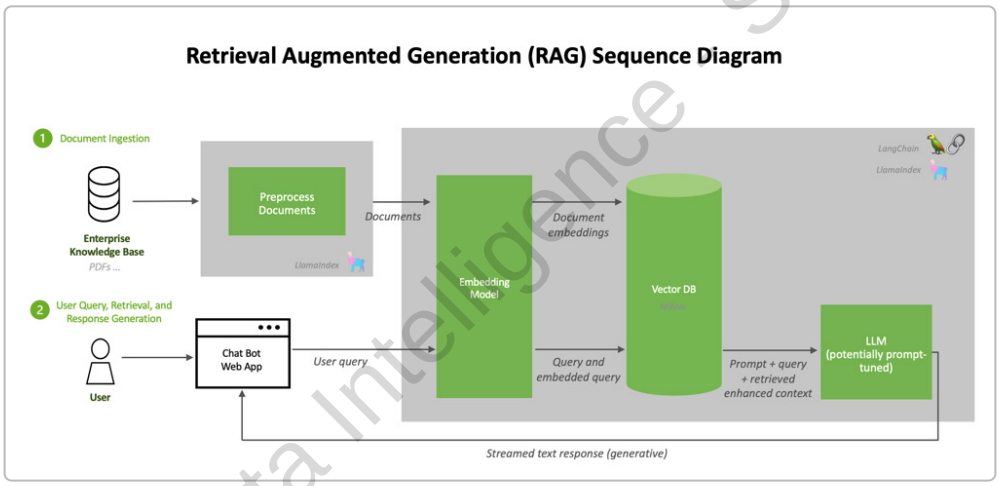

Figure: High-level RAG pipeline architecture (data ingestion, vector indexing, retrieval, and LLM response generation). Retrieved chunks are used to ground the LLM's answer, improving factuality [3] [4].

Chunking Strategies

Effective chunking (breaking documents into indexable segments) is critical. Three common strategies are:

- **Fixed-size chunking:** Split text into uniform blocks (e.g. every N tokens) [5] [6]. It's simple and fast to implement, but may sever semantic boundaries (cutting sentences or paragraphs) and lose context [5] [6]. Use fixed-size chunks when processing speed is paramount and documents have relatively uniform structure.
- **Semantic (adaptive) chunking:** Divide by natural language units (sentences, paragraphs, sections) [7] [8]. This preserves contextual integrity – each chunk remains a coherent semantic unit – which improves retrieval relevance [8]. However, it requires linguistic analysis (e.g. NLP tools) and can produce variable chunk sizes. Semantic chunking is preferred for domains like legal or medical text, where context must not be arbitrarily cut [7] [8].
- **Hybrid chunking:** Combine methods to balance speed and context. For example, use a coarse fixed-size

pass to create initial chunks, then refine boundaries semantically [9] [10]. One approach is to allow overlapping chunks (sliding windows) so that no sentence is split; for instance, including the last 50 tokens of one chunk at the start of the next improves continuity [11] [12]. Hybrid strategies (e.g. fixed initial split then semantic refinements) can yield both efficient indexing and high retrieval accuracy [9] [10].

In all cases, the **chunk size** trades off context versus cost: smaller chunks fit more precisely to queries but increase the number of items to embed and search [13] [14]. Overlapping or larger chunks preserve narrative flow but raise computational load [15] [16]. An enterprise RAG should tune chunk size by measuring retrieval recall and LLM response quality; domain-specific needs (e.g. detailed legal clauses vs. brief tech specs) will dictate the optimal segmentation.

Re-ranking Methods

Most RAG systems use a two-stage retrieval pipeline: a fast first-stage retriever (bi-encoder or sparse index) to fetch many candidates, followed by a slower but more accurate second-stage reranker [17] [4]. Key methods include:

- **Bi-encoder (dense retriever):** A bi-encoder (or dual-encoder) model independently encodes the query and each document chunk into vectors, then retrieves by nearest-neighbor search (e.g. cosine similarity) [4] [18]. Bi-encoders (or sparse retrievers like BM25) are very fast at query time, since chunk embeddings can be precomputed. However, they compress semantics and miss fine-grained relevance cues (word order, negation) [19] [20]. Bi-encoders excel at high recall – catching many broadly relevant chunks – but their top results may include noise.

- **Cross-encoder (neural re-ranker):** A cross-encoder (or reranker) takes a query and one document chunk as joint input to a transformer (e.g. BERT, T5) and outputs a refined relevance score [4] [17]. Because it can attend to query-document interactions, a cross-encoder usually yields much higher precision than a bi-encoder [19] [20]. The tradeoff is latency: each reranking scoring is a full model inference. Thus RAG pipelines typically apply the cross-encoder only to the top-N results from the first stage (e.g. rerank the top 20–100) [4] [21]. As Pinecone notes, “we retrieve plenty of documents… then reorder and keep just the most relevant for our LLM” using rerankers [4].

- **Ensembles and hybrid retrievers:** Enterprises often combine multiple retrievers/rerankers to boost accuracy. For example, a system might union the results of dense (bi-encoder) and sparse (BM25) search, then apply a neural re-ranker [22] [23]. Academic work also shows top systems often ensemble multiple rerankers (e.g. several cross-encoders) to eke out gains [23]. In practice, an ensemble trade-off is cost versus benefit: single strong rerankers (such as a fine-tuned T5 or Electra model) often suffice [23], but advanced applications may layer multiple models for critical precision. Another middle ground is **late-interaction models** like ColBERT, which allow some context interaction without full cross-attention, enabling faster re-ranking at scale.

Taken together, a reference RAG architecture typically looks like: chunk and embed all documents (offline), then at query time run the query through the embedding model, use vector search (and/or keyword search) to get an initial set, then apply cross-encoder scoring or other filter to return the final top-*k* chunks [4] [20]. This two-stage design maximizes retrieval recall while keeping LLM context manageable.

Guardrails and Safety Measures

Enterprise RAG systems must incorporate **guardrails** to ensure outputs are safe, correct, and policy-compliant. Key measures include:

- **Input moderation and sanitization:** Before feeding user queries to the LLM or retrieval system, filter or flag malicious or disallowed content. This can include domain-specific privacy or compliance checks (e.g. block requests that attempt to retrieve confidential info). Systems may apply pretrained content-safety models or rule-based filters to each query [24]. For example, NVIDIA's NeMo Guardrails provides "content safety check" modules that classify input text against policies (hate speech, sensitive data, etc.) and block unsafe requests [24].

- **Prompt injection defenses:** Malicious users may embed commands in their input to hijack the system ("jailbreak" prompts). Defenses include wrapping the system's instructions in cryptographically "salted" tags or unique tokens so that user input cannot override them [25]. AWS recommends templates where all system instructions are in a single salted section, and the model is told to ignore any instructions outside it [26]. Another strategy is **attack-pattern detection**: include explicit instructions in the prompt telling the LLM to recognize common injection patterns and refuse them [27]. (For example, the LLM can be instructed to answer "Prompt Attack Detected" if it sees suspicious embedded commands.) These techniques raise the bar for adversarial input.

- **Content moderation:** Even after generation, the output should be checked for violations (e.g. privacy leaks, hate speech). Guardrail frameworks like NeMo can route the LLM's response through an "output flow" using content-safety models [24]. Alternatively, one can use external APIs (e.g. OpenAI Moderation) or custom classifiers to scan the answer and block or redact any disallowed content. This ensures that even if the LLM "hallucinates" unsafe content, it won't be served to users.

- **Factuality and hallucination checks:** RAG reduces hallucinations by design, but errors can still occur. A best practice is to verify or filter the LLM's answer against the provided context. Techniques include: (a) **Self-consistency checks** – sample multiple candidate answers from the LLM and check if they agree. NVIDIA Guardrails' "self-check hallucination" rail, for instance, compares the chosen answer to additional generated responses and flags contradictions [28] [29]. (b) **LLM-as-a-judge** – use another model or the same model in evaluation mode to verify statements. For example, prompt an LLM to label each statement in the answer as "supported by the context" or not [30]. (c) **Citation auditing** – ensure that every factual claim in the answer can be traced to a retrieved chunk (often enforced in the prompt with instructions like "only use the provided sources"). Metrics like *groundedness* measure the fraction of answer tokens that match the context [30]. If an answer is flagged as hallucinated or ungrounded, the system can either retry the query or warn the user (NVIDIA allows both "blocking" and "warning" modes [31]).

- **Access controls and privacy:** On the architectural level, guard against data leakage by isolating knowledge bases. Employ document-level or chunk-level access controls so that only authorized users see sensitive info. Enterprise RAG platforms often provide role-based filters during retrieval and generation. (For instance, Squirro emphasizes "granular access controls" to ensure each user only retrieves permitted data [32].)