Federated Retrieval-Augmented Generation: Architecture, Privacy, and Security

1. Federated RAG Architectural Patterns (The Architectural Gap): Traditional federated search (meta-search) techniques query multiple siloed sources and merge results [ar5iv.labs.arxiv.org] [aclanthology.org]. Early IR work (e.g. Shokouhi and Si 2011) proposed federated search without centralized indexes; RAG now extends this by adding a generative LLM step [aclanthology.org]. Recent surveys note that federated RAG systems vary along two axes: *retrieval* (naive "query all" vs. selective or encrypted search) and *generation* (centralized vs. per-silo vs. hybrid) [aclanthology.org] [aclanthology.org]. For example, **RAGRoute** (Guerraoui et al. 2025) learns a lightweight router (neural classifier) to skip 75% of irrelevant silos and focus on high-value sources [aclanthology.org]. Other designs introduce a central "gateway" or index for routing: some architectures use a **global summary index** or directory to predict which silos hold relevant data (e.g. Weber 2025's "global summary vector DB" for cross-node retrieval [billweber.io]), while others use dynamic in-enclave search or secure aggregation (C-FedRAG, FRAG). Industry guidelines also describe hierarchical retrieval: first query a global index to find relevant sources, then dispatch queries only to those sources [medium.com] [billweber.io]. In practice, federated RAG pipelines often employ a centralized "data federation layer" that enforces access controls and merges results [medium.com]. However, none of these classical approaches originally addressed generation, so recent work emphasizes new "micro-architectures" that tightly couple retrieval with LLM synthesis. For instance, Chakraborty et al.'s mapping (2025) taxonomy categorizes systems from *naive aggregation* to *encrypted retrieval* and highlights hybrid generation patterns [aclanthology.org]. In summary, dominant patterns include (a) **flat federated search** (query all silos and merge), (b) **hierarchical gateway** (global index or router selects silos) and (c) **secure enclave-based retrieval**. A central gateway can greatly reduce overhead: by filtering out irrelevant silos before full retrieval (via metadata classifiers or light embeddings), it cuts latency and privacy risk [aclanthology.org] [medium.com]. Yet there remains little empirical evaluation of "gateway" designs in RAG settings, exposing a gap between old federated search theory and modern LLM-augmented IR.

2. Differential Privacy for Code Embeddings (Privacy–Utility Trade-off): Differential Privacy (DP) methods like DP-SGD are well-known in federated learning for bounding information leakage from model updates, but their impact

on *code* embedding models is under-explored. General DP studies (e.g. Abadi et al. 2016) show that adding noise to gradients can protect privacy but tends to degrade model accuracy, especially in high-dimensional settings [arxiv.org]. In domains like vision or simple NLP, DP-SGD often forces a trade-off: more noise → stronger privacy but weaker performance. Code embeddings (e.g. CodeBERT, GraphCodeBERT) are **structured** and **very high-dimensional**; intuitively, DP noise may scramble subtle syntactic/semantic cues. Indeed, recent work on privacy-preserving retrieval (e.g. STEER, 2025) finds that injecting DP-like noise into embeddings creates *"semantic-agnostic perturbations"* that can sharply degrade recall [arxiv.org]. In practice, existing DP-for-ML research rarely addresses code: almost all Opacus or Google DP-SGD case studies involve image classification or text classification, not program source. We found no prior study applying DP-SGD (via Opacus or similar) to a CodeBERT-like retriever. Thus it remains an open question how DP noise would affect **alignment** between code queries and documents. The expectation is that DP-SGD will worsen nearest-neighbor accuracy: similar code snippets might no longer map to nearby vectors under heavy noise. In short, while we can draw analogies from general DP work [arxiv.org] [arxiv.org], there is a **novel gap**: no one has yet measured the privacy–utility curve of DP-trained code embeddings in a federated retrieval scenario.

3. Structural vs. Sequence-Based Retrieval: Traditional IR "treats code as text," splitting files into fixed-size chunks by lines or tokens [arxiv.org]. However, code has rich structure: functions, classes, and control flow. Modern **code-intelligence** models exploit this. For example, CodeBERT (Feng et al. 2020) and GraphCodeBERT (Guo et al. 2021) learn joint embeddings of code and natural language and incorporate code's syntax/flow [microsoft.com] [arxiv.org]. GraphCodeBERT even uses data-flow edges (where values come from) to improve code search performance [arxiv.org]. In experiments, these models outperform generic text-only models on code search benchmarks. More recently, chunking strategies leveraging ASTs have been proposed. *cAST* (Zhang et al. 2025) uses the abstract syntax tree to form chunks that respect function and class boundaries, rather than arbitrarily splitting by character count [arxiv.org]. This yields **semantically coherent** chunks. Empirically, cAST boosts retrieval metrics across tasks: e.g. it increases Precision/Recall by up to ~4% and lowers nDCG scores compared to fixed-size chunks [arxiv.org]. On one benchmark (*RepoEval*), recall improvements of 1.8–4.3 points were reported when using AST-based chunks [arxiv.org], and downstream code-completion accuracy (Pass@k) similarly rose by a few points. The gains arise because AST chunking avoids splitting a function or loop across chunks, keeping retrieval context intact [arxiv.org] [arxiv.org]. These results suggest structure-aware retrieval outperforms naive chunking.

Nonetheless, all such code-retrieval work has been centralized: no study has evaluated CodeBERT/GraphCodeBERT or AST-based chunking in a **federated or zero-trust** context. Code search benchmarks like CodeSearchNet and RepoBench exist, but no federated extension of them has been applied. Thus, while AST-chunking and graph-based embeddings clearly help retrieval in general [arxiv.org] [arxiv.org], their behavior under siloed privacy constraints remains untested. We identify this as a gap: promising code-specific retrievers (CodeBERT, UniXcoder, CodeRetriever, etc.) and smart chunking have never been evaluated in a multi-organization RAG pipeline.

4. Mitigating Data Leakage in RAG: A critical but under-studied problem is preventing **sensitive content leakage** during generation. Most RAG literature talks about hallucination errors, but in our context the danger is that the LLM might regurgitate private IPs, keys, or secrets present in retrieved chunks. Few formal studies address this. Industry sources warn that RAG pipelines often bypass original access controls: when documents are embedded, metadata like ACLs can be stripped out, so the LLM may output information the user shouldn't see [crowdstrike.com]. The OWASP Top 10 GenAI list even flags *sensitive data exposure* (including code and keys) as a major risk [crowdstrike.com]. In practice, the only defenses currently proposed are **engineering filters**. For example, Vijay Poudel (2025) recommends a "post-retrieval sanitization" layer: every retrieved document should be scanned through content filters or regexes to catch adversarial prompts and secret tokens, and tools like Guardrails.ai or LangChain validators should vet the LLM input [medium.com]. Elastic's search labs demonstrate masking techniques: they use NLP pipelines (e.g. SpaCy named-entity recognition) or even a local LLM to replace PII or credential patterns with placeholders before sending context to a public model [elastic.co] [elastic.co]. Similarly, red-team advice emphasizes **removing or redacting** any API keys, IP addresses, or sensitive fields upfront [we45.com] [medium.com]. In short, one can strip or replace sensitive tokens (e.g. with "[MASKED]") before or after retrieval to avoid leakage [elastic.co] [medium.com]. However, such solutions are ad hoc: we found no published evaluation of their effectiveness or impact on code correctness. There is a tension – aggressive filtering could remove needed context and break the code logic – so designing precise **security filters** is hard. To our knowledge, **no academic work** has systematically evaluated post-retrieval sanitization for code RAG. Existing best practices come from blogs or industry (AWS, Palo Alto Networks, etc.), and they underscore the gap. The literature on LLM security (prompt injection, data extraction) suggests general guardrails, but specific techniques for *sanitizing code contexts* remain largely unexplored [medium.com] [elastic.co]. This lack of research on intermediate "safety filters" is

a notable gap: future work should address how to strip secrets without crippling the generated code.

Sources: This review draws on recent surveys and papers in federated learning, retrieval, and code intelligence. Federated search foundations are summarized in Shokouhi & Si (2011) and recent RAG mappings [ar5iv.labs.arxiv.org] [aclanthology.org]. Notable system designs include RAGRoute (2025) [aclanthology.org], FedE4RAG (2025) [arxiv.org], and others. Code retrieval baselines and models come from CodeBERT (Feng et al. 2020) [microsoft.com], GraphCodeBERT (Guo et al. 2021) [arxiv.org], and cAST (Zhang et al. 2025) [arxiv.org] [arxiv.org]. Privacy/DP insights draw on DP-SGD analyses [arxiv.org] [arxiv.org]. Security recommendations are from recent practitioner reports and blogs [medium.com] [elastic.co]. Together, they highlight both the state of the art and the gaps above.