RAG Implementation Approach for GitHub Repo Summarizer

Date: 2025-10-18 18:52 UTC

# 1. Context & Goals

This document proposes a Retrieval-Augmented Generation (RAG) design tailored to this repository’s current architecture (browser extension + Python backend using Groq).

* Ground LLM outputs with actual repository content to reduce hallucinations.
* Keep footprint light; avoid external managed vector DBs for MVP.
* Be resilient to large repos and GitHub API rate limits.
* Return provenance (which files/chunks influenced the answer).

# 2. Current Architecture Snapshot

* Extension (MV3): extracts repo/owner/description from GitHub pages.
* Local FastAPI (`src/backend.py`) proxies to deployed summarizer API.
* Deployed API (`api/index.py`) calls Groq `llama-3.3-70b-versatile` for summary/paper.
* No vector DB; no retrieval of arbitrary files beyond optional tree fetch.

# 3. Proposed RAG (MVP: RAG‑Lite)

## 3.1 Retrieval Sources

* Detect default branch via GET /repos/{owner}/{repo}.
* List files via GET /git/trees/{branch}?recursive=1.
* Fetch README raw and select small-to-medium text files via GET /contents/{path}?ref=branch (Accept: raw).

## 3.2 File Filters & Limits

* Include: md, txt, rst, py, js/ts/tsx, json, yml/yaml, toml, go, rs, java, kt, swift, c/cpp/h, cs, php, rb.
* Exclude binaries and oversized files; per-file cap ~64 KB; total fetch cap ~1–2 MB.
* Skip vendor, node\_modules, .git, dist, build, \_\_pycache\_\_.

## 3.3 Chunking

* Target chunk size 800–1200 characters with ~100 character overlap.
* Preserve metadata: file path, start/end offsets, language.

## 3.4 Ranking

* Use BM25 (rank-bm25) to score chunks against the query.
* Query = repo description + short README summary + task (summary/paper).
* Select top-k (8–12) within context budget (~16k chars).
* Bias to README and top-level docs when scores are similar.

## 3.5 Prompting

* Prepend a “Context” section containing selected chunks with clear delimiters and file path citations.
* Explicitly instruct the model to only use provided context; say “Not in context” if missing.
* Two prompts: Summary (concise), Project Paper (structured, one page).
* Maintain a max context length; truncate least-relevant chunks first.

# 4. API Changes

* Inputs (optional): top\_k (int), max\_context\_chars (int), include\_tree (bool).
* Outputs: summary, project\_paper, used\_context (list of {path, char\_range, preview}).
* Graceful fallbacks when rate-limited: degrade to README-only.

# 5. Configuration & Env

* API\_KEY (Groq) – required.
* GITHUB\_TOKEN – optional, boosts rate limits.
* Timeouts: 10–15s per GitHub call; total request budget ~45–60s.

# 6. Operational Concerns

* Caching: short-lived per-repo content cache (in-memory or disk) keyed by commit SHA.
* Rate limits: Respect X-RateLimit-\* headers; backoff and partial retrieval.
* Error handling: Skip unreadable files; continue best-effort.
* Observability: Log retrieval counts, chunk stats, ranking times.

# 7. Security & Safety

* Prompt injection hardening: delimit context; reinforce instruction hierarchy.
* PII: Do not extract secrets; ignore .env and credential files.
* Output discipline: Cite source files; avoid speculative details.

# 8. Phased Rollout

* Phase 1 (MVP): BM25 retrieval over README + small docs/code; provenance returned.
* Phase 2: Local embeddings (sentence-transformers) + FAISS; reranking.
* Phase 3: Hosted vector DB (pgvector/Pinecone/Weaviate) and caching by commit.

# 9. Acceptance Criteria

* Summaries cite source file paths for at least 70% of content.
* Quality beats baseline on internal eval set (≥+20% win rate).
* Latency p95 ≤ 6s for medium repos; graceful degradation on large ones.

# 10. Implementation Notes (Code Pointers)

* Modify api/index.py: add retrieval, chunking, ranking, prompt building, provenance.
* Add dependency: rank-bm25 to requirements.txt.
* Keep src/backend.py unchanged; it proxies to the deployed API.