ChatGPT

DOCS/RESEARCH/Deep_Research_MCP_RAG_Report.md

Executive Summary

This report investigates how to integrate a **Model Context Protocol (MCP)** server for low-token, fast context retrieval in code assistant workflows. It covers MCP's JSON-RPC based tool protocol and contrasts its implementation in **OpenAI's ChatGPT Developer Mode** versus **Anthropic's Claude (Code/Desktop)**, highlighting differences in tool packaging, payload limits, streaming, error handling, and sandboxing. We then examine retrieval-augmented generation (RAG) strategies for code: using span-level retrieval with stable IDs, top-**K** selection and minimum confidence (τ) thresholds, optional cross-encoder reranking for quality gains, and caching for speed. We discuss budgeting of tokens and latency, providing recommended defaults (e.g. K≈5, τ≈0.65, max ~5 spans per answer, ~10KB max content) and analyzing trade-offs: smaller K or higher τ reduces cost but risks missing relevant context, whereas rerank boosts precision ~20–35% at a ~200–500 ms latency cost [1]. We outline an **explicit** "local LLM" tool (e.g. using a local model like Ollama/DeepSeek) for offline or hybrid synthesis, noting it should only be invoked on user request (opt-in) to avoid hidden complexity. Finally, we address operational considerations: **security** (read-only file system access within the repo, secret redaction), **observability** (structured JSONL logs per query with fields like K, τ, cache hits, latency; optional Prometheus metrics), and how to handle failures (e.g. if the index is missing or stale). An evaluation plan is proposed using representative coding queries to measure precision (P@1/P@3), answer accuracy, token usage, and latency distribution, including ablation tests for reranking, chunking strategy (AST-driven vs fixed-size), threshold tuning, and caching. The report concludes with recommended default parameters and justified design choices to balance answer quality with efficiency.

Contents

A. MCP Protocol and Client Integrations

What is MCP (Model Context Protocol): MCP is an open JSON-RPC 2.0 based protocol that standardizes how external tools or data sources can interface with AI assistants [2] [3]. An MCP "server" provides a set of **tools** (exposed functions or actions) that an AI model (the client) can invoke during a conversation. Tools have defined names, parameters (often with schemas), and return values. The protocol supports **resources**

(read-only data contexts), **prompts** (predefined prompt templates), and especially **tools** that allow the AI to perform operations or fetch information [4]. Communication is typically via JSON-RPC messages over either a local process's stdio pipes or network transports like HTTP/SSE streams [5] [6]. This means the assistant sends a JSON-RPC request (specifying tool method and params) and the server returns a JSON-RPC response. MCP was pioneered by Anthropic's Claude but has evolved into a **vendor-neutral standard** now adopted by OpenAI and others [7] [3]. It's often likened to "HTTP for AI tools," enabling a rich ecosystem of third-party tool servers that multiple AI clients can use [3] [8].

ChatGPT Developer Mode (OpenAI): In September 2025, OpenAI's ChatGPT introduced a Developer Mode that allows full MCP support in chats [9]. In this mode, users can register **custom connectors** which launch or connect to an MCP server. ChatGPT's approach is to run the MCP server as a subprocess (via a command specified in a config), communicating over stdio. For example, one can configure in the ChatGPT UI a connector that runs `python my_mcp_server.py` or uses a provided CLI like `npx mcp-remote` to connect to a remote MCP endpoint [10] [11]. ChatGPT Developer Mode thus treats MCP tools similarly to its built-in web-browsing or code interpreter plugins, but with expanded capabilities (read/write actions, not just read-only fetches) [10].

Differences and constraints: ChatGPT's MCP integration initially came with some quirks. Early adopters noted that ChatGPT was **strict about return types** and tool naming. For example, complex structured JSON outputs sometimes caused errors; developers found they had to return simpler types (like strings) to avoid invocation failures [12] [13]. In contrast, Claude's client was more tolerant of structured objects. Additionally, ChatGPT currently only allows custom MCP connectors in **Developer Mode** (not in standard or team chat modes), and tools might appear disabled in existing chats – users often must start a fresh chat in Dev Mode for new tools to be active [14]. There were also reports that ChatGPT's MCP client expected certain baseline tools like `search` or `fetch` to be present [15] – possibly a requirement in the OpenAI docs to implement or declare those common actions – although this may have been a temporary limitation or documentation nuance. Another consideration is **payload and time limits**: ChatGPT's cloud service environment might cancel tool calls that take too long or return extremely large payloads. Indeed, some users experienced HTTP 424 errors and client disconnects when a tool call ran lengthy or produced a big response [16] [17]. This suggests ChatGPT imposes timeouts/backpressure on the MCP channel to ensure responsiveness. Developers worked around this by simplifying outputs or breaking up tasks. **Streaming support** in ChatGPT's MCP is limited; currently the model seems to wait for the full JSON result before resuming the conversation (no incremental partial responses are delivered to the model mid-call). Therefore, large streamable outputs may still need to be chunked or avoided for ChatGPT. Security-wise, ChatGPT's connectors can be configured with no auth or OAuth for remote servers, but not yet with custom bearer tokens (as of initial release) [18]. In summary, ChatGPT's MCP integration is powerful but still maturing, with some idiosyncrasies around tool registration and output formatting.

Claude Desktop and Claude "Code": Anthropic's Claude has supported MCP-based connectors earlier and in multiple forms. **Claude Desktop** is a local application that enables connecting to local tools (especially for file system access and IDE integration) [19] [20]. Claude's implementation of MCP tends to be **more flexible in payload** – it natively allowed complex JSON results and didn't have the same return-type restrictions that ChatGPT initially did [21] [13]. For example, an MCP server returning a structured object with fields was accepted by Claude's agent where ChatGPT would error out, per developer reports. Claude also appears to handle **streaming** outputs from tools better: in the Claude Desktop app, a long tool result (e.g. reading a file) can be streamed to the conversation incrementally (with Claude generating partial responses) in some cases, whereas ChatGPT tends to only print the final result after the call completes. Claude's connectors system (often called **Custom Tools** or **Connectors**) also allowed persistent, **multi-turn usage** – e.g. one could enable a "Filesystem" MCP server and it remains available across sessions or projects in the Desktop app [22]. Claude "Code" refers to Claude's mode or product variant optimized for coding. In Claude's web version, by mid-2025 they introduced connector support in Claude.ai as well (especially for business users), but the Desktop app is where local MCP shines. Packaging for Claude is straightforward if you have Claude Desktop: you run your MCP server locally and then in the app you add a connection (often specifying an `mcp://` URI or using a UI to point to the local server). Claude's environment might spawn the server or just connect over an IPC/localhost channel, depending on configuration [22] [20]. Unlike ChatGPT, Claude Desktop doesn't require special "developer mode" switches – local connectors are a built-in feature aimed at extending Claude's capabilities. Claude has relatively high limits on context and may allow larger tool outputs (Claude 2 models can accept ~100k token contexts in some versions), so including multiple code snippets via tools is feasible. That said, **Claude Code's sandbox** is noteworthy: while tools can perform file reads/writes, Claude's AI system itself has guardrails that prevent disallowed content output. The MCP server should ideally sanitize any sensitive data (just as a precaution), but if a tool returned something against Claude's content policy, Claude might refuse to relay it. Overall, Claude's MCP integration is considered robust – many early MCP servers were built with Claude in mind, and only later adapted to ChatGPT once OpenAI adopted the protocol [7] [3].

Other "Agentic" Clients (Cursor, Windsurf, Continue, LangGraph): Beyond ChatGPT and Claude, a number of development tools and agent frameworks use MCP or similar mechanisms to integrate retrieval and actions:

- **Cursor (IDE Assistant):** Cursor's AI assistant (the "Composer" agent in Cursor editor) can consume MCP tools seamlessly [23]. Cursor's docs note that any MCP server's tools will show up as available actions for the agent, and the agent may auto-invoke them when relevant [23]. For instance, if you run a local MCP server exposing a `searchCode` tool, Cursor's agent can call it to find code references, then use the results to inform completions. Cursor provides a CLI to connect to MCP servers (the Cursor CLI has an `--mcp` option) [24]. They also maintain a **directory of MCP servers** for common tasks (web search, database queries, etc.), showing the growing ecosystem [25].

- **Windsurf (AI-Powered IDE by Codeium):** Windsurf's agent "Cascade" was designed to orchestrate tools. Recent Windsurf releases added native MCP integration, letting developers "bring your own" MCP servers into Cascade's toolbox [26]. For example, Codeium provides MCP servers for security analysis, UI components, etc., which Cascade can call during its multi-step reasoning [27]. Windsurf emphasizes agent autonomy – Cascade can plan to use tools multiple times (e.g., search code, open file, then modify file). MCP's stateless call structure fits well, but Windsurf may have its own session management ensuring tools get the necessary context each call. Documentation for Windsurf advises limiting tool response sizes to avoid flooding the agent (since the agent's prompt is built from tool outputs plus user query) [28].

- **Continue (VS Code extension):** Continue.dev is an open-source VS Code extension for conversational coding assistance. While it doesn't explicitly call out MCP, it supports **custom tools and retrieval** via Python backends. Continue's docs on building a custom code RAG system show how to plug in an embedding-based retriever to supply context [29]. Essentially, Continue can be extended with Python functions that do searches on a codebase (similar to what an MCP `rag_query` would do) [29]. However, Continue doesn't yet use the standardized MCP JSON-RPC; instead it has its own plugin interface. We can consider it "MCP-equivalent" in that it allows an agent to call out to code search and get results, but it might require writing a Continue-specific integration (or using a LangChain agent under the hood).

- **LangGraph (Agent Framework):** LangGraph is a framework for building complex AI agents (often on top of LangChain). It has explicit support for MCP tools via an adapter library [30]. In LangGraph, you can define an agent that includes MCP server tools alongside other LangChain tools. This means our RAG MCP server could be plugged into a LangGraph-powered agent used in custom workflows. The benefit is standardization: rather than writing a new tool interface for each framework, MCP acts as a universal adapter. LangGraph's documentation confirms that agents can treat MCP server methods as first-class actions, using the `langchain-mcp-adapters` to handle JSON-RPC calls [30]. This shows MCP's growing role beyond specific products – it can glue into many AI orchestration systems.

Summary of Client Differences: In planning our MCP RAG server, we must account for these differences. ChatGPT Dev Mode usage means prioritizing simplicity and reliability: return relatively simple data structures (e.g. strings or basic dicts) to avoid known bugs [12], keep response sizes reasonable, and perhaps implement at least the expected minimal interface (like a no-op `search` tool if required, or naming conventions that ChatGPT UI expects). For Claude and others, we can expose richer info (structured JSON or multiple content parts) since they handle it. We should also incorporate **capability detection** – e.g. if our server sees the client is `openai-mcp` vs `claude-mcp` (some servers can identify the client version from handshake logs [31]), we might adjust output format accordingly (this is an advanced consideration; initially, we can stick to lowest-common-denominator output and let the client format it). We will also document how to run/connect our server in both ChatGPT (via Developer Mode connector JSON) and Claude Desktop (likely providing a local URL or just instruct to run and it auto-finds, depending on Claude's mechanism). By designing to the MCP spec and avoiding client-specific hacks as much as possible, our RAG server will be **vendor-neutral** and usable from a variety of agentic frontends.

B. RAG Integration Patterns for Codebases

Retrieval-Augmented Generation for a codebase involves extracting the most relevant code snippets (functions, classes, comments, etc.) to feed into the prompt so the LLM can answer questions accurately using the project's actual code. Our system uses a **span-level retrieval** approach with stable identifiers for each snippet. Key integration patterns and options include:

Span-Level Retrieval with Stable IDs: Instead of retrieving whole files or large chunks, we index the codebase into smaller spans. A "span" in this context is a contiguous segment of a file (e.g. a function or ~1000-character chunk). Each span is given a **stable ID** so that we can reference it unambiguously. We propose an ID format combining the file path and character or line offsets, plus a content digest for integrity. For example: `frontend/src/Auth.js:120-255:abcd1234` might identify the span covering lines 120–255 of `Auth.js`, with `abcd1234` being a hash of that span's text. This format encodes location and a checksum to detect changes (if the file changes, its hash portion won't match the index). Stable IDs allow the assistant to retrieve a specific snippet later via a `rag_get_span(id)` call, and they remain consistent across sessions as long as the code doesn't change (or if changed, the digest signals potential staleness). In the current code indexing implementation, a similar idea exists: each chunk's ID in the vector store is `file_hash_chunkIndex` [32]. The file hash acts like a digest for the entire file content, and chunk index identifies position. We will adapt this to a more human-readable span ID for external use, but it will map back to the stored chunk. The benefit of span-level granularity is **precision** – by pulling only the relevant 15–30 lines of code, we minimize irrelevant context and save token budget, compared to pulling entire files.

Top-K Retrieval and Confidence Threshold (τ): When a query is embedded and sent to the vector index, the result is a set of candidate spans with similarity scores. We will retrieve the top **K** spans (by similarity) as initial candidates. Choosing K is a trade-off: higher K (more spans) increases the chance that all relevant info is included (higher recall), but also adds noise if many are only loosely related, and consumes more tokens. Many RAG systems default to K in the 5–10 range [33]. Our current RAG UI used K=10 for context building [34], which yielded ~6K token context for a complex query. For interactive Q&A, we might start with **K=5** as a default, which often gives a good balance of capturing key points without overloading. Alongside K, we set a minimum similarity **threshold (τ)**: any span below this cosine similarity score is dropped as likely irrelevant. For example, τ=0.65 (on a 0–1 scale) could be a default – meaning we require at least 65% similarity. If only 2 spans meet that bar, we may return just those (even if K=5). The threshold prevents extremely tenuous matches from polluting the context. In effect, we're implementing **Selective Retrieval** [35]: filtering the set of top-K to only retain highly relevant chunks, thus narrowing down context to what truly matters. This improves precision (P@1, P@3) at the possible cost of missing a subtle relevant chunk if the threshold is set too high. We will likely allow tuning of τ per query (exposed as a param) but default it to a moderately high value like 0.65–0.7 based on empirical testing. Over time, observing confidence histograms of queries will guide adjustments to τ (e.g. if we see many relevant spans scoring ~0.6, we might lower τ slightly).

Optional Reranking (Cross-Encoder): Our baseline retrieval uses a bi-encoder (embedding model) for speed, but we have the option to add a second stage reranker. Reranking means taking the initial candidates (say top 10) and scoring them with a more precise model (e.g. a cross-attention model that reads the query and snippet together). This often improves the ordering and can filter out false positives that passed similarity filtering. Studies show that cross-encoder rerankers can boost RAG answer accuracy significantly – one report notes a **20–35% improvement in relevance** when using cross-encoder reranking, at the cost of 200–500 ms added latency per query [1]. The reranker essentially ensures that the top 3 spans truly contain the answer if possible, rather than just being loosely topically related. We plan to integrate reranking in a toggleable manner: by default **off** (to keep latency low and implementation simple in P0), but the framework ready to enable it. When enabled, the system would retrieve e.g. top 20 by embedding, then apply a faster cross-encoder model (like `mpnet-cross-encoder` or use a small LLM) to re-score those 20 and pick the best 5 to return to the user [33]. This can be extremely useful for ambiguous queries or ones that need specific code lines: the cross-encoder can read if a snippet actually answers the question, not just share keywords [36] [37]. The downside, as noted, is latency (hundreds of ms or more on CPU) and complexity. We'll ship with rerank disabled, but with an eye on enabling it for cases where quality is paramount (and possibly making it automatically trigger for difficult queries in future).

Chunking Strategy – AST-Aware vs Fixed Windows: How we break code into spans can influence retrieval quality. Our current indexer uses fixed-size **character windows** (~1000 chars with 200 overlap) [38] [39]. This is simple but can split functions or logical units in awkward ways (e.g. a function definition might be split mid-function if it's longer than 1000 chars, resulting in context fragmentation). An alternative is **AST-aware chunking** – use the code's syntax tree to chunk by function or class boundaries [40] [41]. AST chunking ensures each span is a coherent code block (e.g. a full function with its docstring), which improves semantic integrity. The pros are that the model sees complete logical units, making answers more accurate and reducing the chance of needing multiple spans for one function [42] [43]. The cons are variability in chunk size (some functions might be huge – possibly exceeding context limits or the embedding model's ideal length) and potential coverage gaps (if a relevant line is part of a big function, the whole function chunk is large). Our constraint is to **not change the existing .rag schema** for P0, meaning we likely stick to the current chunking (fixed ~1000 char spans) for initial implementation. But we will discuss AST-aware chunking as a future improvement. We can also incorporate hybrid approaches: e.g., keep chunks at ~1000 tokens but try to split on function boundaries when possible (maybe splitting large functions into logical sub-blocks around comments or blank lines) [44]. For P0, fixed window chunking is acceptable since it's already implemented and yields manageable chunk sizes. But we note that AST-aware chunking could boost relevance and reduce the number of spans needed per answer, at the cost of more complex indexing (e.g. using Tree-sitter [45]). This is a classic trade-off between **semantic coherence** and uniform size [46]. In evaluation, we plan an ablation comparing these strategies (if time permits to implement AST chunking in a branch) to see actual impact on P@1 or tokens saved.

Memoization and Caching: To optimize performance, especially in interactive sessions, we introduce a short-term **memo cache** for queries. The idea is to cache the results of recent queries (likely keyed by the normalized query text plus maybe session ID) so that if the user or agent repeats a question or a very similar one, we can return instantly without recomputation. The embedding computation and vector DB query are relatively fast (on the order of tens of milliseconds), but caching can still help if an agent loops over the same query or if multiple similar calls happen (imagine a user asks "Find uses of X" then "Find uses of X in module Y" – the first part might be reused). The cache would store the top results and any computed metadata (like the embedding) for a given query string. Because the codebase could update or the query context might shift, we give cache entries a **short TTL** (time-to-live), e.g. 5 minutes, or we invalidate on any file index update. We also scope it per session or client to avoid weird cross-talk (though if the repo is static, cross-session caching is fine, but for safety, we might include a session identifier in the key if available). This is a straightforward in-memory LRU cache of limited size. Memory footprint is negligible (storing a few query strings and small result lists). The benefit is shaving off ~50–100 ms on repeated queries and reducing embedding CPU usage if an agent re-asks something. It also helps in back-and-forth dialogues where the user might rephrase a question – if our system can detect the core is the same via a hash or even embedding similarity, we could reuse earlier results (that's a more advanced use: "semantic cache"). Initially, we'll stick to exact query text matching for simplicity. Another form of memoization is caching span content: when the agent later calls `rag_get_span(id)`, we could retrieve it from a cache instead of reading disk or querying the DB again. Given our use case, reading from disk is already fast (and we can memory-map the index or keep file contents in memory if needed), but caching fetched spans avoids repeated disk I/O if the same snippet is requested multiple times by the conversation. Overall, caching is a straightforward **performance booster** that doesn't change answers, so it's low-risk. We will ensure that the cache is invalidated or bypassed if the underlying index changes (e.g., if `.rag/` is rebuilt mid-session – not common during a single Q&A session, but possible if user triggers reindexing).

Server-Side Trimming & Summarization (Optional): If a query's relevant context is still too large to send in full, we implement trimming strategies. "Trimming" means limiting the number or size of spans returned. One approach is **limit the number of spans per answer** (we'll discuss more in section C): for instance, even if K=8 had 8 good hits, we might decide to return only the top 5 to constrain output size. Another approach is limiting by total bytes or tokens: e.g., don't return more than 4,000 characters of text across all spans. Our `build_context_for_task` function already demonstrates trimming to a token budget (max_tokens=8000) by breaking when adding the next chunk would overflow [47] [48]. We will mirror that logic in the interactive setting, albeit likely with a smaller default budget (since an AI chat might not use the full 8k context for just the fetched code). Summarization is a more advanced step to condense content rather than just drop it. For example, if there are 10 relevant chunks but we only have room for 5, we could attempt to summarize the other 5 into a shorter form and include that summary. This is tricky for code (loss of fidelity can be an issue), so by default we **keep summarization off**. We simply note that summarization can be an extension: using either a smaller local model or an external API to compress a chunk of code or text into a brief description. Summarization might be useful if a single relevant function is very long – rather than include 100 lines, summarize it in a sentence or two plus maybe function signature. This aligns with the "Context Summarization" strategy described in RAG literature [49] [50]. However, implementing this requires careful prompt engineering (and likely some risk of missing details), and it could confuse the assistant (which might prefer raw code). As a result, our integration will include a placeholder or flag for summarization (so the architecture is aware of it), but leave it off by default. The primary trimming mechanism remains: cut off least relevant spans when hitting limits, and ensure the total returned context stays under a safe threshold (to avoid hitting model context size or causing slowdowns on the client side). We will also consider interactive trimming: e.g., if the user's question is very broad and results in a massive context, the server might choose to return a message like "partial results (too many to show, consider refining query)" or similar, rather than overwhelming the model. This is somewhat outside typical MCP usage (we usually let the AI decide if it needs more info), but it's a potential feature to avoid pathological cases.

In summary, our RAG integration will generate a compact, focused context for each query: retrieving top-K snippet candidates (with stable IDs), filtering by confidence, optionally reranking them for better precision, caching results for reuse, and trimming to fit token budgets. The result is a list of references the AI can fetch and incorporate, rather than dumping a huge chunk of code blindly. By designing around spans, we ensure the AI can fetch only what it needs, when it needs it, achieving the "low-token, fast context" goal.

C. Token and Latency Budgeting

Designing a RAG system requires balancing how much information we provide (to answer accurately) against the cost in tokens and time. We outline recommended default limits and discuss their implications:

Top-K (Number of Chunks per Query): We recommend **K = 5** by default for `rag_query`. This means the retrieval tool will return up to 5 span IDs (subject to similarity threshold τ). In practice, 5 chunks of code ~maybe 5× (up to ~150 lines total) is often sufficient to cover the relevant parts for a single question. Many questions likely only need 1–3 chunks (e.g., "Where is the user authentication logic?" might only need the `auth.ts` file snippet). By capping at 5, we prevent overloading the model with too much context. The trade-off: some answers that span many files (e.g., "Explain the entire request flow through all layers") might benefit from a larger K like 8–10, but the downside is significantly more tokens and potential dilution of relevant info. We plan to allow overriding this (`limit` param in the tool), so the user or agent can request more if needed (like a broad search mode). But our default of 5 is chosen to fit within a typical LLM context window comfortably while capturing key points. In evaluation, we will measure how often the relevant snippet was in the top 5 vs maybe lurking at ranks 6–10. If we find that precision at 3 (P@3) is, say, 90% and P@5 is 95%, that confirms 5 is a good default – diminishing returns beyond that. If P@1 is high (the first result is usually the answer), we could even consider dropping to K=3 to be ultra-token-efficient. However, code queries can be tricky (the exact answer might be the 2nd or 3rd result if synonyms or multiple matches exist), so K=5 is a safe middle ground.

Minimum Confidence (τ): We propose a default **τ = 0.65** (65% similarity) as the cutoff for including a span. This is somewhat arbitrary and will be tuned empirically. The goal is to exclude low-score items that likely aren't relevant. If a query gets no results above 0.65, `rag_query` might return an empty list (signaling "no confident hits"). That's preferable to returning irrelevant code which could mislead the LLM. We will log the scores distribution to refine this. Many vector DBs, including our Chroma usage, provide cosine distance; we convert to a relevance score = 1 - distance [51]. Typically, identical text would be score ~1.0; decent match maybe 0.7–0.8; random unrelated ~0.4 or less. 0.65 is a moderate bar. We might also implement a **secondary condition**: ensure at least one result is returned (so if even the best is 0.5, we return that rather than nothing, possibly with a flag that confidence is low). Another approach is a dynamic threshold: e.g., if the top result is X, maybe include any results within 0.1 of X's score (so if top was 0.9, include down to 0.8). This prevents missing slightly lower but still relevant results. But to keep it simple, a fixed τ is fine. In summary, τ helps manage token usage by **cutting off tail results** that aren't strongly relevant. It can improve precision metrics – if we tune τ to achieve e.g. P@1 of 85% at the cost of slightly lower recall, that might be acceptable for a high-precision assistant. The user can always broaden search by explicitly lowering the threshold if needed (like a "–all" mode returning even weak matches).

Max Spans per Answer / Query: We plan to set **RAG_MAX_SPANS_PER_ANSWER = 5** (matching K) as the default maximum number of chunks to incorporate into a single answer context. This essentially caps how much context the LLM will receive from our system for one question. If more than 5 relevant spans exist, the remainder are dropped or would require a follow-up query. This number directly affects token usage: each span might be, say, ~100 tokens on average (if 1000 chars ~ 150 tokens). 5 spans is ~750 tokens of code, which is quite manageable for models like GPT-4 or Claude (with 8k+ context). If we allowed, say, 10 spans (~1500 tokens), it's still within many models' limits but starts to be a lot of context to juggle. Also, model performance can degrade if too many disparate context pieces are given (it might struggle to identify what's important). By limiting to 5, we enforce a focus on the top hits. Note that RAG_MAX_SPANS_PER_ANSWER might be distinct from K in cases where we allow K > 5 but still only use top 5 in final assembly. For instance, if reranking is on, we might fetch 15, rerank, then only use the best 5 of those for answer. That scenario respects the max spans per answer. This parameter can be adjusted if using a model with a huge context window (for example, if using a local LLM with 32k tokens, perhaps feeding 10 spans is fine). But for our target usage (ChatGPT and similar), 5 is a conservative choice. We will document that users should increase this if they know their model can handle it and if they face questions requiring broad context.

Max Bytes (Content Size Limit): We enforce **RAG_MAX_BYTES** (proposed default ~**10,000 bytes** or ~10 KB) as a safety limit on the amount of text returned by any single tool call (especially `rag_query` and `rag_get_span`). This prevents extremely large payloads from being sent to the AI client, which could cause timeouts or context overflow. For perspective, 10 KB of code is roughly 2,000 words, or ~2,500–3,000 tokens – about half of GPT-3.5's context or a quarter of GPT-4's 8k context. This is likely sufficient for most answers. Typically, one span is <1 KB (our chunk size ~1000 chars), so 5 spans ~5 KB. 10 KB allows some headroom (maybe a couple bigger chunks or some additional text like file names). If output would exceed this, the server can truncate or refuse. For instance, if someone tried `rag_get_span` on an ID that actually corresponds to an entire large file (imagine if we allowed multi-span retrieval), we'd cut off after 10K bytes and perhaps indicate "[truncated]". Another use of this limit is to avoid **prompt injection through file bloat** – if malicious content or extremely large data were somehow present, we ensure we don't inadvertently send a huge chunk. We can make this configurable via env variable. In internal testing, we'll confirm that typical use (like 5 code chunks) stays well under 10K bytes. We might adjust the default up or down. E.g., if using Claude 100k context, one could raise RAG_MAX_BYTES to 50k or more. For now, 10kB is a safe default for general compatibility.

Latency Considerations (CPU vs GPU, Rerank): On pure CPU, our retrieval (embedding + vector search) is quite fast – using `all-MiniLM-L6-v2` we see ~50–100 ms for embedding and query of 5-10 results. So `rag_query` should typically return in under 0.1s for uncached queries (and <0.05s if cache hit). Our target is **P50 ~60 ms, P95 ~250 ms** for rag_query on CPU (with K up to 8) – these are realistic given embedding model latency and DB lookup. Without rerank, that is. If reranking (which might add 200+ ms), we expect those numbers to shift up (P95 maybe 500+ ms). That's why rerank is off by default. If we detect a GPU is available for the embedding model or reranker, latency can drop further (embedding might be 5–10 ms). But we assume many users will run on CPU, so optimizing for that environment (possibly by using a slightly smaller embedding model if needed) is prudent. For `rag_get_span` (which just looks up a stored snippet by ID, likely a dictionary or DB fetch), latency is minimal – maybe 1–5 ms plus any disk I/O. We target P50 ~10 ms, P95 ~40 ms or better for span fetch, essentially near-memory speed. If we cache spans in memory after first load, subsequent calls are microseconds. So that is fine. The **largest latency contributor could be the LLM itself** (the actual answer generation), which is outside our scope except for local LLM usage. We ensure our part is negligible relative to model token generation time.

Trade-offs vs Quality: Reducing K, τ, or max spans all save tokens and time but can hurt answer quality if key info is omitted. We will analyze **Precision at 1 and 3 (P@1, P@3)** – these measure how often the correct answer's source is in the top results. If P@1 is high, we could even consider returning just one span. But code queries often benefit from seeing a couple of relevant spots (e.g., function definition and where it's called). P@3 is a good middle-ground metric: if we achieve, say, P@3 = 0.9, that means in 90% of questions, at least one of the top 3 spans contains the necessary info. That implies an agent would likely succeed by fetching a few. P@5 may be even higher (approaching recall). We'll tune K to maximize something like P@3 given a latency budget. If going from 5 to 8 yields a small increase in recall but large cost in tokens, we'll stick to 5. **Reranking** comes into play when initial precision is not great. For instance, if P@1 is only 50% with pure embeddings, a reranker might boost that to 70%. If we see many queries where the best answer was ranked 4th or 5th by embeddings, that might justify enabling rerank despite latency. We might also consider enabling rerank selectively: e.g., if the similarity score of the top result is moderate and many results cluster close, that indicates potential ambiguity – we could trigger rerank in those cases to ensure we pick correctly. On CPU, one must be careful: a cross-encoder like MiniLM can take ~100ms per candidate pair; for 10 candidates that's ~1s, too slow. But optimized libraries or smaller models can shorten that. A possible compromise is **hybrid retrieval**: use the LLM itself to judge which snippets are truly relevant by briefly reading them (which ironically is what the agent ultimately does when it fetches spans). This is more complex and usually not needed if our initial retrieval is strong.

Context Window and Model Types: We should also consider differences between model contexts: ChatGPT (GPT-4/3.5) typically 8k or 32k tokens; Claude up to 100k tokens; local models (like the DeepSeek 6.7B) maybe ~4k tokens. Our default limits should be safe for the smallest likely context (4k). 5 spans (~750 tokens) + user question + some answer won't overflow 4k. If someone connects a 100k context model, they can raise K or bytes if desired for more exhaustive answers. But even with huge context, retrieval is about relevance, not volume – feeding 50 chunks just because you can usually isn't optimal.

In summary, our defaults (K≈5, τ≈0.65, max spans≈5, max bytes≈10k) are intended to deliver high-precision context with minimal overhead. They aim to answer >90% of queries without omission, while keeping token usage perhaps 5–10× smaller than naively stuffing an entire file or repository context. These will be validated and adjusted as we test on real queries. The system will also monitor performance: if average rag_query latency climbs or answers seem to lack info, we know to revisit these parameters (e.g., if analysis shows many queries require >5 spans, we might up that limit for those scenarios). Logging fields like number of spans returned and total bytes per query will feed into this analysis.

D. Local-LLM via MCP (Explicit Tool)

In addition to retrieving relevant code snippets, we plan to offer a tool for **local LLM synthesis** – essentially an MCP method that takes a question, runs a local (on-prem) large language model with the retrieved context, and returns an answer. This is separate from the main `rag_query` workflow (which just fetches context). The tool might be called `local_llm_rag_query` or similar, making it clear it's doing a full query+answer using local models. Let's explore why this is useful and how to handle it:

Use Cases for a Local LLM Tool:
- *Batch or Parallel Research:* A user or agent might want to quickly get answers to multiple queries without consuming API credits or hitting rate limits. A local model can handle many queries in parallel (limited by local compute) since there's no external API cost. For example, the agent could spawn a couple of local queries on different aspects of the code and then aggregate results. This is feasible if the local model is small and runs fast on your hardware (some projects use 7B or 13B models that can answer basic questions ~10s each [52]). The local LLM tool could be invoked asynchronously (if the client supports parallel calls via threads or just sequentially, depending on MCP client).

- *Pre-filtering / Triage (Gatekeeper):* Before sending a query to a powerful (and possibly expensive) API model, one could use a local model to gauge complexity or even attempt an answer. For example, the system could first call `local_llm_rag_query` – if the local answer seems confident, perhaps that suffices, saving an API call; if not, then escalate to GPT-4. This pattern of using a "cheap" model as a gatekeeper or first pass can drastically cut costs when queries are simple. It's similar to how our orchestration uses "smart routing" [53] (deciding between local vs API based on complexity). The local LLM tool is essential to implement such logic in an AI agent context: the AI itself can decide to try the local route first by calling the tool.

- *Hybrid Synthesis for Enhanced Answers:* Sometimes combining insights from local and premium models yields the best result. For instance, a local model that has been fine-tuned on the user's codebase (or just by virtue of local context doesn't have training guardrails) might surface low-level details or internal comments that a larger, general model might miss or choose not to emphasize. On the other hand, a large API model might have superior reasoning or articulation. A workflow could be: local model generates a draft answer focusing on code specifics, then the assistant (with GPT-4) takes that and refines the wording or checks for correctness. This use case aligns with a *"co-pilot"* scenario: the local model does the heavy lifting with proprietary data, and the cloud model does higher-level guidance or polish. The MCP local tool provides the needed functionality to get that draft.

- *Offline or Privacy-sensitive Scenarios:* In some cases, users may prefer not to send any code context to an external API (for IP or privacy reasons). A local model can be used exclusively to answer questions. While the quality might be lower than state-of-the-art cloud models, it can still be useful for certain tasks like straightforward code navigation answers or summarizing internal docs. By exposing an explicit local answer tool, privacy-conscious users (or the agent configured in a privacy mode) can stick to it and avoid calling external AI at all. For example, imagine an **air-gapped**

environment where ChatGPT is not available, but a smaller open-source model is – our system could then fully operate on that via `local_llm_rag_query`.

Explicit vs Implicit Integration: We emphasize that calls to local LLM should be **explicit and opt-in**. We treat it as a separate tool rather than something `rag_query` does automatically behind the scenes. This is important for clarity and user control. If `rag_query` were to start generating answers on its own, it would blur its contract (which is supposed to just retrieve references). It could also lead to unpredictable cost or time (some local models might be slow). The anti-pattern we avoid is *nested invocation*: e.g., `rag_query` fetching spans and then internally calling a local model and returning a full answer. Instead, we expect a usage pattern like: the agent (ChatGPT or Claude) first calls `rag_query` to get relevant span IDs, then calls `rag_get_span` for content (or directly uses a combined tool to fetch content), then if needed, calls `local_llm_rag_query` with the original question (or a refined prompt) to get an answer draft, and finally, possibly, merges that into its own answer or presents it. By keeping it explicit, the AI can decide when to use the local model vs when to use its own reasoning or ask a user.

Design of the local LLM tool: It will likely accept parameters such as `query` (the question or task), maybe a `model` or `profile` selection (for instance, user could choose a faster model vs a more accurate one, similar to how the script in AGENTS.md had profiles [54]), and maybe a `max_tokens` for the answer length. It would then internally perform: `rag_query` (or directly use the internal ContextQuerier to get top spans), possibly apply summarization/trimming to fit model context, then construct a prompt with the retrieved code context and the query, run the local model inference, and return the answer. This is effectively automating what a user/assistant could do manually – but packaging it as a tool makes it convenient and ensures consistent formatting (we might include references or citations in the answer, etc., though a local 7B won't know how to cite unless we instruct it in prompt). The returned answer will likely be plain text (or markdown) content. Because ChatGPT had issues with structured returns, we'll probably just return a text string (which could include markdown for code if needed). We also need to handle errors – e.g., if the local model is not available or errors out, return an error message.

Parallelism and Performance: Running local models (like via Ollama, llama.cpp or similar backends) can be slow (several seconds) depending on model size and hardware. However, one advantage is we could theoretically generate partial output (streaming) to the client if the client supports it. Claude Desktop might stream a local tool's output as it's being produced (not 100% confirmed, but SSE transports allow chunking). ChatGPT likely doesn't stream tool outputs yet, as noted, so it will wait. So performance is a concern: a 6B model might take ~5–10 seconds to produce a multi-paragraph answer on CPU. We should document this so the user is aware (maybe naming it `local_slow_answer` would set expectations!). Use of this tool would typically be for either non-time-sensitive tasks or when offline usage trumps speed. That said, smaller models (like a 2.7B or the DeepSeek coder model they use) might be tuned to answer within a few seconds [52]. We can also allow the local model to be changed – if the user has a GPU and a 13B model, great, or if they want a 3B for speed.

Quality Considerations: A local model will likely not match GPT-4's quality, especially in complex reasoning or explaining. It might also have outdated or no knowledge beyond the provided context. This is fine as it will rely heavily on the retrieved code context for factuality. One must also consider that local models might have fewer safety filters – meaning they could output things that a cloud model might avoid (like internal secrets found in code, or more direct language). This can be both a feature (less refusal to answer internal queries) and a risk (if it says something inappropriate). Therefore, using a local model is somewhat "at your own risk", though within a dev environment this is generally acceptable.

No Automatic Local Calls in rag_query: To reiterate the anti-pattern: we do **not** want `rag_query` or the core retrieval to ever automatically call the local model. For example, we wouldn't want ChatGPT to call `rag_query` and unexpectedly get a full answer from our server (it would confuse the flow, since ChatGPT expects just references to then incorporate). Instead, `rag_query` stays true to its contract: just search. The user or agent can then decide to call the local LLM tool if they desire that answer. This separation keeps the design modular and debuggable. If the combined pipeline doesn't give good answers, one can isolate whether retrieval or the local model is the cause.

Example Workflow with local LLM: Suppose the user asks, "Explain how the permission system works in this project." The assistant (ChatGPT) might do:
1. Call `rag_query("permission system", limit=5)` -> gets back 5 span IDs (with file names perhaps).
2. The assistant decides this might need reading those files, so it calls `rag_get_span` on a couple of them (or perhaps we have a helper to fetch all top spans' content in one go). Now it has the raw code context.
3. The assistant could attempt to summarize with its own capability (GPT-4 could do it itself), or if configured to minimize tokens or if offline, it can call `local_llm_rag_query("Explain the permission system")`. Our tool would internally gather the same top spans (maybe it might do the retrieval again, or we optimize by passing it the IDs from step 1 to avoid repeat search), and then prompt the local model: "Using the following code snippets, explain the permission system: [code chunks here]". The local model generates an answer text about the permission logic.
4. The assistant receives that text. It could either directly present it (if we trust it and maybe just add "*(Answer generated by local model)*"), or the assistant could integrate it into its own answer (perhaps verifying or rephrasing). For ChatGPT, likely it will just present it as the tool's result (since it was a direct tool response). For Claude, similarly. This yields an answer without the main model having to incorporate all code into its prompt – the heavy lifting was offloaded.

Avoiding Confusion: We should clearly label in our documentation and maybe tool descriptions what `local_llm_rag_query` does, so the agent knows when to use it. For instance, the tool description might say: "Uses a local AI model to directly answer the query using retrieved code context (no OpenAI/Anthropic API used)." That clarity helps the agent decide appropriately. If the main model is extremely capable (like GPT-4), it might rarely need this – but it could still use it if instructed (for cost saving). In developer mode contexts, sometimes the human user might explicitly ask "Use the local model to draft an answer first." Then the agent knows to call this tool.

Resource Management: If using a local model, we have to ensure the model is loaded (maybe a one-time init cost). Perhaps we will lazy-load the model on first call, and cache it in memory (especially if using something like Ollama, which might manage it as a service). We might also need to handle concurrency – if the user triggers two local queries at once (rare via a single agent, but maybe sequential quickly), our single-process server might queue or run them synchronously. That is fine for now.

In conclusion, the **Local LLM tool** adds an important dimension: it empowers the user/agent with an **offline question-answering** capability, aligning with the local-first philosophy of the project (as seen in the orchestration template where local generation was primary with optional API fallback [55] [56]). It must remain a separate, explicit path to avoid hidden side-effects. We'll implement it as an optional P4 feature (once core retrieval is done), since it's not strictly necessary for retrieving context but is a value-add for certain scenarios.

E. Security, Observability, and Operations

Deploying an MCP-based RAG server in a development environment requires careful attention to security and good observability for debugging and performance tuning. Here we outline measures for ensuring the server operates safely (not exposing sensitive data or causing harmful actions) and is monitorable.

Read-Only Filesystem Access: Our RAG server will operate in read-only mode with respect to the project's code repository. The tools `rag_query` and `rag_get_span` should never modify any files – they only read from the pre-built index and the source files. We will enforce this by design: the indexer runs separately (likely a manual step or a file watcher) and not as part of the query serving process. The MCP server process will likely drop permissions to write or simply never attempt writes. Even for the local LLM tool, any code it sees is just passed to the model; we won't execute code or write out files. This is important because ChatGPT/Claude could *ask* the tool to perform things, but since our defined interface is limited (no tool for write is provided), it cannot. We essentially maintain an allow-list of operations (the defined tools) and none of them includes file write or shell execution, so the AI agent cannot instruct our server to do something outside those bounds. This is a fundamental **sandbox** approach: the AI can only do what the MCP server exposes. By limiting exposures to safe read-only queries, we ensure the AI doesn't accidentally modify the repository or leak data outside.

Path Allow-List / Project Root Isolation: We will restrict file accesses to within the allowed project root (likely the env var `RAG_REPO_ROOT` or the path used during indexing). For instance, if `rag_get_span` is asked for an ID, we will decode the file path and ensure it resides under the known repository directory. If not, we return an error. This prevents any attempt (malicious or accidental) to use our tool to read arbitrary system files (e.g., an AI might naively think it can ask `rag_get_span("../../etc/passwd")` – we must not allow stepping out of scope). The index itself can be our source of truth: if an ID's path isn't in the index metadata, we simply won't retrieve it. The ChromaDB results include the stored `file_path` in metadata [57], so we know the path is legitimate as it was indexed (and indexing skipped non-project dirs). We will also likely ignore any query attempts that look like trying to do something outside (though via MCP the only way to read something is by span ID, not by arbitrary path, so that helps).

Secret and Sensitive Data Redaction: A codebase might contain secrets (API keys, passwords) in config files or even scattered in code. If the AI directly sees these, it could potentially output them – a serious leak. To mitigate this, we plan to implement **redaction filters** on content that `rag_get_span` returns. For example, we can have regex patterns for common secret patterns (AWS keys, JWT tokens, etc.) and replace them with `[SECRET]` or a similar placeholder in the returned text. We could also decide to completely omit certain files from indexing if they look like pure secrets (like a `.env` file). Our indexer already excludes binary and obvious non-code files [58], but .env might not be in that list – we should ensure it is excluded or at least not served. Redaction on the fly is safer in case something slipped in. We will log if a redaction happened (so operators know something was hidden). Another approach is hashing secrets (e.g., show only a hash or partial). This ensures the AI can still possibly identify usage ("this value is some token"), but not the raw secret. Additionally, the logging system should avoid storing any content in plaintext that could be sensitive. We may exclude actual snippet text from logs, or if we include it for debugging, we apply the same redaction. It's worth noting that an AI could theoretically reconstruct something if it really tried (like base64 of a key?), but that's a remote risk in our context.

No Echoing of Environment or System Secrets: The server process might have environment variables (for configuration like API keys for other things). We will ensure none of our tools returns environment info. For instance, we won't provide a tool like `system_info` or anything. Even in errors, we'll be careful not to include sensitive paths or stack traces that reveal system details. Error messages will be generic or at least sanitized. The environment (like `RAG_DB_PATH`) doesn't need to be exposed to the AI or user; it's internal.

Structured Logging (JSONL): Observability is crucial. We will implement logging such that each query and important action is logged in structured JSON format, one line per event. Fields to log for each `rag_query` call might include: a timestamp, the query string (possibly truncated if very long or hashed for privacy if needed), the number of results returned, the similarity scores of top results (or min/max), whether the result was cached, how long the query took (ms), which session or client requested it (if we can identify, maybe an increment or an ID), etc. For `rag_get_span`, log the span_id requested, size of content returned, and time taken. Similarly for `local_llm_rag_query`, log model name, tokens used (if we can get from the local inference), time, etc. Using JSONL (JSON Lines) format means each log line is a JSON object – easy to parse and feed to analysis tools. This also aligns with possible ingestion to a monitoring system or simply grepping. We'll rotate logs or allow specifying a log path (e.g., default to `rag_server.log.jsonl` in the project).

Prometheus / Metrics: Optionally, we can expose a metrics endpoint or simply allow integration. For example, count of queries served, average latency, cache hit rate, etc. This is not required for functionality but good for production readiness. We might not implement a full Prometheus integration in initial version, but we'll design in a way that adding one (or pushing custom metrics) is straightforward. E.g., we could use an existing JSON-RPC to increment counters (not typical; better to do inside our code). For now, we note the option for an operator to instrument our logs or code with metrics.

Health Checks: We will provide a simple `rag_health()` tool or endpoint that returns an "OK" status and maybe versions. This is for ops – the user can call `rag_health` to see if the server is running and responsive. It might check that the index is loaded, etc. If something is wrong (like DB can't be opened), `rag_health` can return a failure code or message. Similarly, `rag_stats()` as in our design will give number of indexed chunks, last index time, etc. These help operators confirm the server has the expected data.

Failure Modes (Index Missing/Stale): If the `.rag/` index is missing (i.e., user forgot to run indexer), our ContextQuerier init currently exits with an error [59]. In server context, we shouldn't just exit; we should handle it gracefully. Possibly `rag_query` should return an error code like `E_INDEX_UNAVAILABLE` with a message "Index not found; please run setup." This way, the AI can convey to the user something like "The code index is not built yet." and maybe even instruct them how to do it (if allowed). If the index is **stale** (code changed a lot since indexing), results might be inaccurate (span content might not exactly match current files). We have a few mitigations: each span ID has a digest so if a user fetches a span but the file changed, we could detect mismatch (comparing current file hash vs ID's file hash) and perhaps warn or auto-reindex on the fly that file. However, auto-reindexing on the fly could be expensive. More likely, we'd rely on the file-watcher or user to trigger reindex periodically. We could provide a management tool like `rag_reindex()` to rebuild index (though that could be heavy to do from the assistant, better to let an external process handle). For now, our server will assume the index is up to date. We will document that if large parts of code changed, the user should re-run the indexer script (and maybe restart the server or it can auto-detect changes in the DB and reload – possible if we subscribe to Chroma updates or simpler, require restart). On stale index scenario, worst case is the tool returns slightly wrong context (e.g., line numbers off). If digest check fails, we can mention "Warning: index out of date for this file" in the returned content to let the user know.

Security Review: We will do a review of our code for potential injection or exploitation points. Because it's JSON-RPC, one might try to send a malicious JSON to trick our server. We should ensure our JSON parsing uses a safe library and we validate params (especially for tools like `rag_get_span`, ensure the ID is a string of expected pattern, not extremely long, etc., to avoid memory issues). Also ensure no command injection (not likely since we're not executing shell, but if we call external programs like a local LLM engine, ensure we pass safe inputs). Since this is internal, these risks are low, but we treat them seriously.

AI Behavior Safeguards: While outside the server's direct scope, it's worth noting: The assistant (like ChatGPT) might try to combine tools in ways not intended. For example, it could call `rag_query` then hallucinate an answer without calling `rag_get_span`. Or it might call `rag_get_span` with a made-up ID. This is generally fine – if ID not found, we return E_NOT_FOUND. Our logs will catch such misuse. Over time, prompt engineering on the AI side ensures it uses the tools properly. We cannot directly enforce that at server side beyond returning errors for invalid calls. One thing we should implement is **rate limiting or abuse detection** on tool calls: e.g., if an AI gets stuck in a loop calling `rag_query` repeatedly due to a prompt bug, it could spam the index. We could put a simple limit like "no more than 10 queries per second" or similar, or if one session made 1000 calls in a minute, maybe auto-shut or alert. This might be overkill for now, as typically usage is interactive and slow.

Privacy: If multiple projects or multi-tenant usage is envisioned, we must ensure one project's data doesn't leak to another. In our case, it's likely a single developer's environment with one codebase at a time. If multi-project indexing is done (our index supports multiple projects), the `project` filter can separate them. The `rag_query` tool could accept a `project` param or default to one. We should ensure that if such multi-project usage happens, an AI cannot retrieve project B's code while in a conversation about project A (unless explicitly asked). This ties into allow-list by path as well.

Observability – diagnosing issues: With JSONL logs, we can reconstruct what happened in a session. For example, if an answer was weird, we check logs: saw the AI called rag_query with X, got Y spans, maybe it didn't call get_span, etc. This helps refine the chain-of-thought prompting for the AI. Also, any errors thrown by the server (exceptions) will be logged. We should probably catch exceptions around the core logic and return proper error codes instead of letting the server crash or hang. E.g., if Chroma query fails, catch and return E_INTERNAL with detail.

Prometheus metrics detail (optional): If we integrate, we'd record metrics like `rag_query_duration_seconds` (histogram), `rag_query_results_count`, `rag_cache_hit_total`, etc. This could allow setting up dashboards to ensure the median latency stays low and to catch if performance degrades (maybe index size grew too large etc.). But given our scope, this might be documented as a future improvement.

In operations, we'd also define an SLO (Service Level Objective) such as: **99% of `rag_query` calls under 0.5s**, 99% of `rag_get_span` under 0.1s, error rate <1%. Logging and metrics will help ensure we meet these by showing tail latencies and error counts.

F. Evaluation and Tuning Plan

To ensure the MCP RAG server delivers real value (i.e., reducing tokens while preserving answer quality), we design an evaluation approach tailored to this repository's use cases. We need to evaluate both **retrieval**

performance (are we getting the right code snippets?) and **end-to-end QA performance** (does the assistant answer correctly with those snippets) under various settings.

Representative Dataset/Tasks: We will create a set of **benchmark questions** representative of how developers ask about their code. This includes:
- *Code navigation queries:* “Where is the function that validates user email?” or “Find all uses of `isAdmin` flag.” These test if the retrieval finds the correct file and lines (like grep but semantic).
- *Conceptual understanding queries:* “Explain how the membership upgrade process works” or “What does this project use for authentication?” These require pulling together context from multiple places (for explanation).
- *Refactoring guidance queries:* “If I want to add phone verification, where should I integrate it?” – tests if relevant modules are surfaced (not a direct answer but point to where in code).
- *Error message resolution:* Given an error or log snippet, “What could cause `NullReferenceException` in module X?” – requires linking error to code location.
- *Specific implementation details:* “Which regular expression do we use for email validation?” – expecting a snippet of code as answer, requiring precise retrieval.

We’ll draw these from real development scenarios (some can be taken from commit history or issue discussions if available, or we create them based on reading the code). We aim for at least ~20-30 queries for evaluation, covering different areas of the codebase (frontend, backend, config, etc.).

Metrics:
- *Token Usage:* For each query, we will measure how many tokens are used with RAG vs a baseline. Baseline could be a naive approach like “provide entire relevant file in prompt” or even “no context provided, just rely on model’s knowledge”. But a better baseline is giving the model the full file(s) that might contain the answer (as was possibly done before having this RAG). We anticipate our method uses far fewer tokens by only supplying snippets. We’ll compute prompt+completion token count for answers with and without RAG, showing the reduction (target: ≥40% reduction on average as stated in acceptance criteria).

- *Time-to-first-useful-answer:* This measures latency from question to when the user gets a substantial answer. If the agent has to do multiple tool calls, there’s overhead. Ideally, the agent can fetch and answer within a few tool calls (a couple of seconds at most). We can simulate the conversation or measure actual interactions to gather this. We might not automate it easily, but in tests we’ll note how long it takes to get the final answer. If using local LLM tool, that might be slower, we will note those separately.

- *Tool Call Count:* How many tool calls did the agent need to answer? We expect often exactly 2 (one `rag_query`, one or a few `rag_get_span`). If it’s often doing redundant calls, there might be room to improve prompt instructions. Ideally, the agent should call `rag_query` once, then maybe get 2-3 spans, then answer. If we find scenarios with multiple back-and-forth (like agent calls rag_query, then tries to refine query and call again, etc.), that means our initial retrieval might have been insufficient or the agent prompt made it unsure. We will aim to minimize average calls. This is important because each call adds latency and potential for error.

- *Precision @1 (P@1) and @3:* This is a retrieval metric: for each query, is the correct reference in the top 1 result? Top 3? We’ll define “correct reference” carefully – e.g., if question: “Where is X implemented?”, the “correct” reference is the file/line that actually implements X. If our top result’s span corresponds to that, that's a hit. For explanation questions, there might be multiple relevant pieces; we might consider it a hit if at least one truly relevant snippet is in results. We will calculate P@1, P@3 over our test set. Since K defaults to 5, we could also do P@5 (which should ideally be high). This tells us how effective the embedding retrieval is. If P@1 is low (say 50%), that indicates maybe needing rerank or better chunking. If P@3 or P@5 are high (>80-90%), that means the info is usually present somewhere in the returned set.

- *Confidence Score Distribution:* We will analyze the similarity scores of retrieved results, especially focusing on true positives vs false positives. Ideally, true relevant chunks have higher scores than irrelevant. If we plot a histogram of relevance scores for relevant vs irrelevant returns, we can pick a good τ to separate them. This helps justify or adjust our threshold. For example, if many irrelevant ones slip through with score 0.7, maybe we raise τ. Or if some relevant ones are around 0.6, maybe lower τ slightly. This is more of a diagnostic metric.

- *Latency Percentiles (P50, P95):* We will measure the actual time of rag_query and rag_get_span calls under typical loads. Our target (P50 $\leq$ 60 ms, P95 $\leq$ 250 ms for rag_query) will be verified. We can simulate 50 queries in a row and measure. If something is slow (maybe the first call loads model, etc.), we account for that separately. This ensures the system meets interactive speed requirements.

- *Answer Accuracy / Quality:* Ultimately, does the assistant give correct and helpful answers? We need a way to measure this. One approach: have human evaluators (or ourselves) rate the answers for correctness and completeness. Alternatively, exact-match if the question expects a specific piece of code (like an API endpoint URL) – we can check if the answer contained the right detail. Given this is more subjective, we might set up a rubric: e.g., on a scale 1-5 or pass/fail if the answer solved the user's query. We want our RAG approach to have **parity or better accuracy** compared to not using it. Likely it will be better because without RAG the model might hallucinate or be generic. We can demonstrate a few examples where "Without RAG: the answer was generic/wrong" vs "With RAG: the answer cited actual code" [60] . For instance, RAG can quote how a function is implemented instead of guessing.

- *Token Reduction vs Baseline:* We mentioned token usage; specifically, we want to quantify that at least 40% fewer tokens are used compared to a baseline scenario (like providing full file context or multi-file context manually). We can simulate a baseline where the entire file containing the answer is given as system prompt and ask the question, count tokens. Versus our approach: just the snippet. In many cases, it could be a drastic reduction (if file was 300 lines but snippet is 10 lines, that's like 90% reduction). We'll summarize the average saving.

Ablation Studies: We will run controlled variations to isolate the impact of certain features:
- *Rerank ON vs OFF:* For a subset of queries (especially ones where baseline retrieval struggled), we will enable our rerank module (if implemented) and see if relevant results move up and if the final answer accuracy improves. We'll note the added latency. This will inform whether we want to turn it on by default later.
- *Threshold τ High vs Low:* Try a lower threshold (like 0.5) which returns more spans, and a higher one (0.8) which returns very few. See how answer accuracy and token usage change. We expect higher τ to slightly reduce answer completeness (maybe missing some info) but reduce tokens. Lower τ might include irrelevant info which could confuse the model (or just be wasted tokens). Finding a sweet spot or at least confirming 0.65 is reasonable is the goal.

- *K=3 vs K=5 vs K=8:* Similar to threshold, compare a smaller number of chunks vs larger. Possibly run the test set with K=3 and see if any answers fail due to missing context that was only in chunk 4 or 5. If not, maybe we could even reduce default K further. If several queries need chunk 5, then K=5 is justified. K=8 might show marginal gains at significant token cost, validating not using such a high default.
- *AST Chunking vs Fixed:* If we implement or simulate AST-based indexing (maybe we can approximate by grouping chunks by function manually for some queries), we can see if it yields better P@1 or fewer chunks needed. For example, a question about a function: AST chunking would produce one chunk containing the whole function (so one chunk suffices), whereas fixed chunking might have split it into two chunks, requiring both to answer completely. This could reflect in answer quality or the number of get_span calls needed. If AST shows clear advantage, that might justify an index rebuild in future.
- *Cache On vs Off:* Measure throughput or latency differences when asking repeated queries. E.g., ask same query 5 times rapidly: with cache, after first time, others should be near-instant. Or ask a set of similar queries where embedding might be reused – if we had a vector cache (not implemented yet), we could check. Currently our cache is basic, so the main effect is just skipping embedding and DB lookup on exact repeats. This ablation mainly demonstrates the overhead (which is small anyway). It's more about confirming cache doesn't alter results (which it shouldn't) and helps performance in obvious scenarios.

Logging and Analysis: Using the structured logs from actual usage (maybe from a team pilot), we can compute many of these metrics. For example, logs give us query, returned spans, maybe which were fetched, etc. We can label in hindsight which spans were relevant. We might even instrument the agent prompts to output which references it used in the final answer for evaluation purposes (like have it cite span IDs in a hidden way for scoring, though that's complex). For now, manual evaluation on the test set is fine.

SLOs and Continuous Evaluation: We can set target thresholds that must be met before releasing widely. For instance: *Accuracy parity:* ensure that on our test set, answers with RAG are at least as accurate as answers from GPT-4 with no context (likely far better, but we verify). If any significant regressions, investigate. *Latency:* ensure median additional latency from using our tools is <1s, and P95 <2s, otherwise conversation may feel sluggish. *Token saving:* ensure at least 40% reduction on average as mentioned. We'll incorporate these into acceptance criteria and CI (maybe a small script to run tests with a local model to approximate or just offline metrics for retrieval).

Conclusion of Evaluation: Through these evaluations, we will refine defaults (maybe adjust K or τ) and catch issues (like if certain file types weren't indexed or the model misinterprets tool outputs). It will also give us confidence to roll out to more users (the "team pilot" phase in the integration plan will use these metrics to validate in real usage). Over time, we can expand the test questions to cover new patterns as the code evolves or as users ask new kinds of things.

Overall, the evaluation ensures our MCP RAG server truly delivers on its promise: **significantly reducing context size (tokens) and maintaining high-quality, accurate assistance for code understanding and development tasks.** The combination of retrieval metrics and end-to-end QA metrics will paint a full picture of system performance and guide any further improvements.

<br> [12] [23] [1] [34]

---

[1] [33] [36] [37] RAG Reranking Techniques For Better Search
https://customgpt.ai/rag-reranking-techniques/

[2] OpenAI Brings Full MCP Support to ChatGPT - Nerd @ Work
https://blog.enree.co/2025/09/openai-brings-full-mcp-support-to-chatgpt-what-it-means-for-developers

[3] [7] [8] OpenAI announces they are adopting MCP : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1jkjfq5/openai_announces_they_are_adopting_mcp/

[4] [5] [6] An Introduction to MCP and Authorization | Auth0
https://auth0.com/blog/an-introduction-to-mcp-and-authorization/

[9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [21] [31] MCP server tools now in ChatGPT -- developer mode - Coding with ChatGPT - OpenAI Developer Community
https://community.openai.com/t/mcp-server-tools-now-in-chatgpt-developer-mode/1357233

[19] [22] Connect to local MCP servers - Model Context Protocol
https://modelcontextprotocol.io/docs/develop/connect-local-servers

[20] Connect Claude to MCP Servers for Better AI Capabilities - MESA
https://www.getmesa.com/blog/how-to-connect-mcp-server-claude/

[23] Model Context Protocol (MCP) | Cursor Docs
https://cursor.com/docs/context/mcp

[24] MCP | Cursor Docs
https://cursor.com/docs/cli/mcp

[25] MCP Servers for Cursor - Cursor Directory
https://cursor.directory/mcp

[26] Cascade MCP Integration - Windsurf Docs
https://docs.windsurf.com/windsurf/cascade/mcp

[27] yeakub108/mcp-server: MCP Server for Windsurf - GitHub
https://github.com/yeakub108/mcp-server

[28] Windsurf agents to orchestrate my custom MCP servers - Medium
https://medium.com/@didierlacroix/building-my-youtube-subscription-assistant-part-3-windsurf-agents-to-orchestrate-my-custom-mcp-fe4b7fba2a71

[29] How to Build Custom Code RAG - Continue Docs
https://docs.continue.dev/guides/custom-code-rag

[30] Use MCP - GitHub Pages
https://langchain-ai.github.io/langgraph/agents/mcp/

[32] [38] [39] [57] [58] index_workspace.py
https://github.com/vmlinuzx/glideclubs/blob/192e899e53e6dc63cf3660b86021b03227bf03d9/scripts/rag/index_workspace.py

[34] [53] [55] [56] [60] RAG_SYSTEM_SUMMARY.md
https://github.com/vmlinuzx/glideclubs/blob/192e899e53e6dc63cf3660b86021b03227bf03d9/DOCS/NOTES/RAG_SYSTEM_SUMMARY.md

[35] [49] [50] Understanding RAG Part V: Managing Context Length - MachineLearningMastery.com
https://machinelearningmastery.com/understanding-rag-part-v-managing-context-length/

40 41 42 43 44 45 46 Mastering Code Chunking for Retrieval Augmented Generation | by Joe Shamon | Medium
https://medium.com/@joe_30979/mastering-code-chunking-for-retrieval-augmented-generation-66660397d0e0

47 48 51 59 query_context.py
https://github.com/vmlinuzx/glideclubs/blob/192e899e53e6dc63cf3660b86021b03227bf03d9/scripts/rag/query_context.py

52 54 AGENTS.md
https://github.com/vmlinuzx/glideclubs/blob/192e899e53e6dc63cf3660b86021b03227bf03d9/template/AGENTS.md