Production RAG-MCP Server

(rag-server.py)

Document Type	Technical Reference
Subject	`rag-server.py` — two-stage RAG retrieval server
Author	Brian Vowell
Revision	1.01
MCP Transport	stdio JSON-RPC (FastMCP)
Stage 1 Model	`Qwen3-Embedding-0.6B` (1024-dim, SentenceTransformer, `cuda:0`)
Stage 2 Model	`model_sliced.onnx` (Qwen3-Reranker-0.6B, ORT `cuda:1`)
Vector Store	ChromaDB (HNSW cosine, `technical_books` collection)
Hardware Target	Dual RTX 3060 12 GB / 60 GB DDR3 RAM
Result Count	`K_RETRIEVE=50` candidates → `TOP_K=10` final results

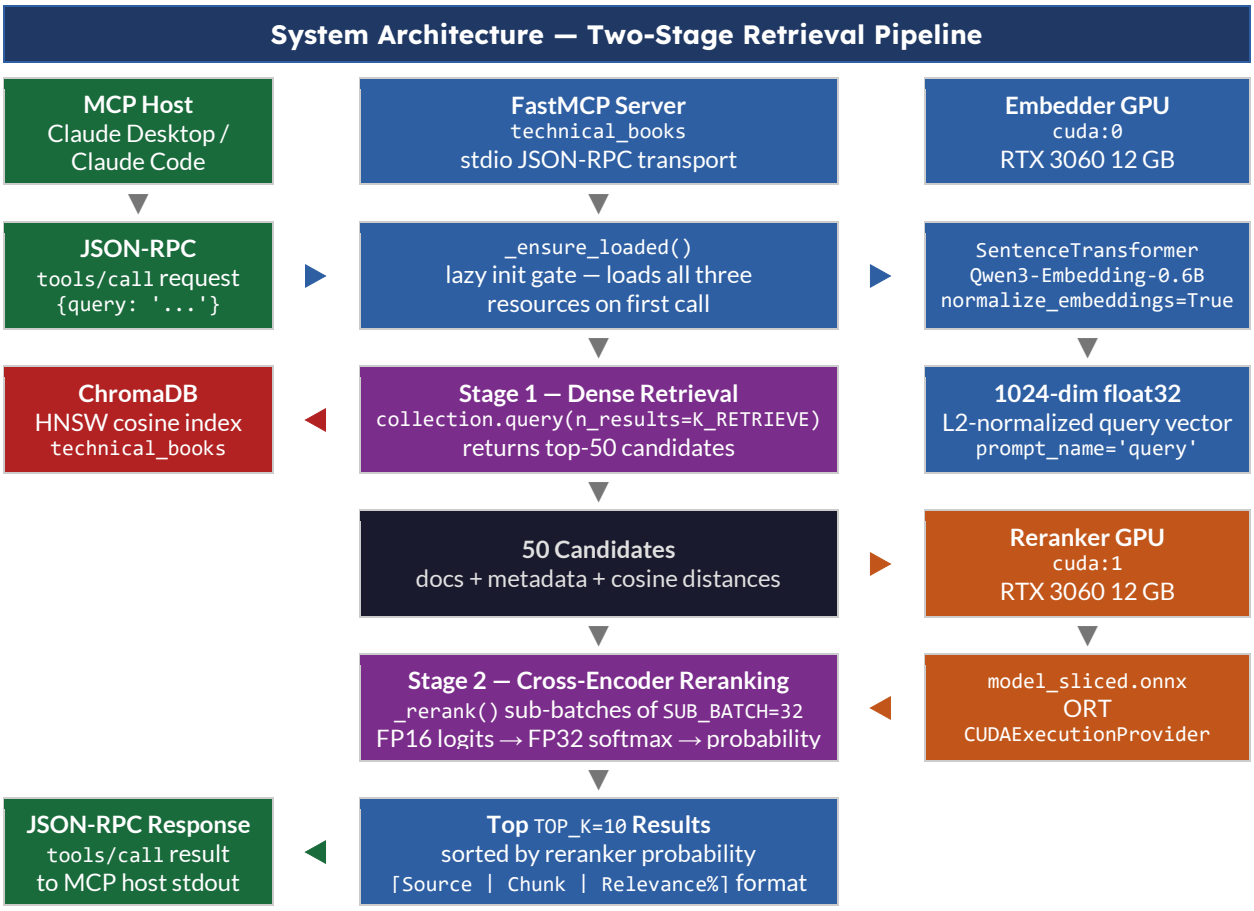
Table of Contents

1. System Overview

`rag-server.py` implements a two-stage semantic search pipeline exposed as a local MCP (Model Context Protocol) tool server. Its sole exported tool — `search_technical_books` — accepts a natural language query and returns the ten most relevant passages from a ChromaDB vector collection of indexed technical literature. The pipeline uses a Qwen3 bi-encoder for fast approximate nearest-neighbor retrieval in Stage 1, followed by a Qwen3 cross-encoder reranker running under ONNX Runtime for high-precision re-scoring in Stage 2.

Core design principle: the retrieval problem is split into two sub-problems. Stage 1 optimizes for recall (retrieve broadly and cheaply from the entire indexed corpus). Stage 2 optimizes for precision (score each candidate accurately against the query using joint attention over both texts). This is the standard two-stage retrieval architecture described in the information retrieval literature as 'retrieve-then-rerank.'

The diagram below shows the complete end-to-end pipeline from MCP tool invocation to final result delivery:



2. Configuration Constants

All runtime parameters are defined as module-level constants near the top of the script. They serve as the single source of truth for the pipeline's behavior and are intentionally not passed as function arguments, simplifying the call signatures of internal functions.

`DB_PATH`	C:\RAG-MCP\chroma_db — filesystem path to the ChromaDB persistent store
`COLLECTION_NAME`	`technical_books` — name of the ChromaDB collection to query
`EMBED_MODEL`	`Qwen/Qwen3-Embedding-0.6B` — HuggingFace model ID for the bi-encoder. Must match the model used at index time
`RERANKER_MODEL`	`zhiqing/Qwen3-Reranker-0.6B-ONNX` — HuggingFace model ID for the ONNX cross-encoder reranker
`EMBED_DEVICE`	`cuda:0` — PyTorch device string for the `SentenceTransformer` embedder. Shares the display GPU to leave the second GPU free for ONNX
`RERANK_GPU_ID`	1 — ORT `CUDAExecutionProvider` device index. Bound to the non-display RTX 3060 to isolate VRAM pressure
`K_RETRIEVE`	50 — candidate count pulled from ChromaDB in Stage 1. Over-retrieves to give Stage 2 a rich pool
`TOP_K`	10 — final result count returned after reranking. Balances context richness against prompt token budget
`MAX_RERANK_LEN`	1536 — maximum token length for each (query, doc) pair fed to the reranker. Accounts for prefix+suffix overhead on top of Qwen3's 1024 `MAX_SEQ_LEN`
`SUB_BATCH`	32 — number of (query, doc) pairs processed per ORT forward pass. Limits peak VRAM consumption during reranking

3. Startup, DLL Injection, and Environment Configuration

3.1 DLL Path Registration

Before any CUDA-dependent library is imported, the script calls `os.add_dll_directory()` twice to register the CUDA runtime and cuDNN binary directories with the Windows DLL search path. This is necessary because the NVIDIA CUDA toolkit places its shared libraries in non-standard paths that are not automatically discoverable by Python's import machinery on Windows. Without this, importing `onnxruntime` or `sentence_transformers` would fail with a DLL load error.

DLL Registration

```
os.add_dll_directory('C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.9\bin')
os.add_dll_directory('C:\RAG-MCP\venv\Lib\site-packages\nvidia\cudnn\bin')
```

3.2 stdout Isolation

The MCP stdio transport uses stdout exclusively for JSON-RPC message framing. Any extraneous output on stdout — including Python's built-in `print()`, `tqdm` progress bars, or transformer model loading logs — would corrupt the JSON-RPC stream and cause the MCP host to fail parsing. The server addresses this at the module level with two mechanisms:

- `print()` is monkeypatched to always write to `sys.stderr`, not `sys.stdout`.
- The `TRANSFORMERS_NO_TQDM` and `HF_HUB_DISABLE_PROGRESS_BARS` environment variables are set to suppress tqdm output from the HuggingFace ecosystem.

stdout Corruption Risk

Any diagnostic output written to stdout after the module-level patch — including output from third-party libraries loaded after the patch — will break the JSON-RPC channel. Always verify new library imports with stderr logging before deploying.

3.3 Lazy Global State

Eight module-level globals are declared as `None` and populated on the first call to `_ensure_loaded()`. This defers all GPU memory allocation and disk I/O to the first tool invocation rather than server startup.

Global	Type	Description
`_embed_model`	SentenceTransformer	`Qwen3` bi-encoder loaded on `cuda:0`. `None` until first call.
`_collection`	ChromaDB Collection	Handle to the persistent ChromaDB collection. `None` until first call.
`_reranker_session`	ort.InferenceSession	ONNX Runtime session for the sliced `Qwen3-Reranker` graph. `None` until first call.

`_reranker_tokenizer`	AutoTokenizer	Tokenizer for the reranker. Configured with `padding_side='left'`.
`_yes_id` / `_no_id`	int	Token IDs for the strings 'yes' and 'no'. Used to extract binary classification logits.
`_prefix_ids` / `_suffix_ids`	list[int]	Pre-tokenized chat-template wrapper. Encoded once at load time to avoid per-query overhead.

4. `_ensure_loaded()` — Lazy Initialization

`_ensure_loaded()` is the central initialization gate. It checks each of the three resource globals in sequence and initializes any that are still `None`. Because Python's Global Interpreter Lock (GIL) serializes execution, this check is safe without additional locking under the assumption that the MCP server handles one request at a time (which is the standard FastMCP behavior with stdio transport).

The flowchart below illustrates the three-stage initialization logic:

4.1 Embedding Model Load

`SentenceTransformer` loads `Qwen3-Embedding-0.6B` from the HuggingFace cache (populated by a prior `hf` CLI download) and moves all weights to `cuda:0`. A single warmup inference call is issued immediately after load to trigger CUDA kernel compilation and eliminate first-query latency spikes.

4.2 ONNX Reranker Load

`snapshot_download()` resolves the local HuggingFace cache path for the reranker repository. The ONNX file loaded is `model_sliced.onnx` — a graph that has been pre-sliced to expose only the final-token logits rather than the full vocabulary projection. The ORT session is configured with:

- `ORT_ENABLE_ALL` graph optimization — allows ORT to apply all available kernel fusions and memory layout optimizations.
- `kSameAsRequested` arena strategy — the GPU memory allocator allocates exactly the amount requested rather than rounding up to the next power-of-two.
- `HEURISTIC` cudnn_conv_algo_search — avoids the long benchmark phase of `EXHAUSTIVE` while still selecting a performant convolution algorithm.

After session creation, the script verifies that `CUDAExecutionProvider` is actually the active provider (ORT silently falls back to CPU if CUDA is unavailable) and logs a warning to stderr if it is not.

The reranker tokenizer is loaded with `padding_side='left'`. This is a contract imposed by the Qwen3-Reranker architecture: the model reads its yes/no classification signal from the final (rightmost) token. Left-padding preserves this alignment across variable-length sequences in a batch.

> **`model_sliced.onnx` Input Signature**
>
> The sliced graph accepts three int64 inputs: `input_ids`, `attention_mask`, and `position_ids`, each shaped [batch, seq_len]. Output: `sliced_logits` of shape [batch, 2] in FP16. `position_ids` must be supplied explicitly — ORT cannot infer them from the attention mask for this architecture.

4.3 ChromaDB Collection Load

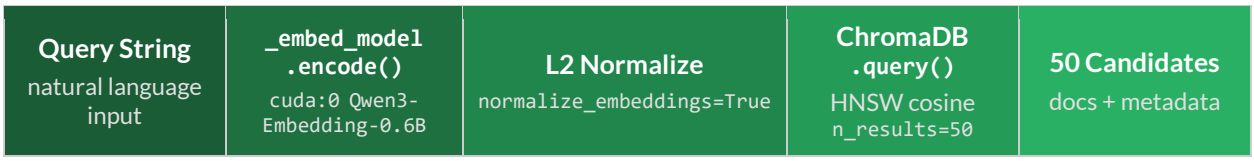
`chromadb.PersistentClient` opens the on-disk database at `DB_PATH` and retrieves a reference to the `'technical_books'` collection. The collection was built by a separate indexer script using the same `Qwen3-Embedding-0.6B` model, ensuring that the embedding space of stored vectors and query vectors is identical.

5. Stage 1 — Dense Retrieval (Bi-Encoder + ChromaDB)

Stage 1 — Dense Retrieval
Bi-encoder query encoding + ChromaDB HNSW approximate nearest-neighbor search

Stage 1 is a bi-encoder retrieval step. The query is independently encoded into a dense vector and compared against all stored chunk vectors using cosine similarity. Because the query and document embeddings are computed independently (hence 'bi-encoder'), this stage scales to millions of chunks without computational explosion — the similarity comparison is a single matrix multiply against precomputed vectors.

Stage 1 pipeline:



5.1 Query Encoding

The query string is passed to `_embed_model.encode()` with two critical arguments:

- `normalize_embeddings=True` — L2-normalizes the output vector so that cosine similarity reduces to a simple dot product. ChromaDB's HNSW index is built with cosine distance, so this normalization is required for consistent scoring.
- `prompt_name='query'` — the Qwen3 embedding model supports instruction-following prefixes. The 'query' prompt name activates a prefix that steers the encoder to produce query-optimized representations. Without this, the model applies the 'passage' prefix, which produces vectors tuned for document-to-document similarity rather than query-to-document similarity.

5.2 ChromaDB Vector Query

The normalized query vector is passed to `collection.query()` with `n_results=K_RETRIEVE` (50). ChromaDB's HNSW (Hierarchical Navigable Small World) index performs approximate nearest-neighbor search, returning the 50 chunks with the smallest cosine distance to the query. The results include the raw chunk text (`documents`), chunk-level metadata (`metadatas`), and cosine distances (`distances`). The distances from Stage 1 are not used in the final output — they are superseded by the more accurate Stage 2 scores.

Why K=50?

The bi-encoder produces approximate similarity scores because it compresses each text into a fixed-size vector, necessarily losing some semantic detail. Over-retrieving (K=50 >> TOP_K=10) compensates for this approximation by providing the Stage 2 cross-encoder with enough candidates to find the truly relevant ones even when the bi-encoder rankings are imperfect. The 5:1 ratio is a commonly used heuristic in two-stage retrieval systems.

6. Stage 2 — Cross-Encoder Reranking `_rerank()`

Stage 2 — Cross-Encoder Reranking
ONNX Qwen3-Reranker scoring all 50 Stage 1 candidates on cuda:1

Stage 2 re-scores all 50 Stage 1 candidates using a cross-encoder model. Unlike the bi-encoder, a cross-encoder processes the query and document jointly in a single forward pass, allowing full self-attention between the two texts. This produces much more accurate relevance scores but scales linearly with the number of candidates — which is why it is applied only to the small Stage 1 shortlist rather than the entire corpus.

Stage 2 pipeline:

50 Candidates from Stage 1	Chat Template Construction	Left-Pad Sub-Batches	ORT Forward Pass	Softmax Scoring
docs + metadata	prefix + instruct/query/doc + suffix	SUB_BATCH=32 attention_mask position_ids	model_sliced.onnx cuda:1 [batch, 2] FP16	FP16→FP32 cast max-subtraction probability

6.1 Chat Template Construction

The Qwen3-Reranker is an instruction-tuned model and expects its input in a specific chat template format. For each (query, document) pair, the input token sequence is constructed as:

Token Layout per (query, doc) Pair

```
[prefix_ids]  +  [instruct_ids + query_ids + doc_ids]  +  [suffix_ids]
    ↑                        ↑                              ↑
pre-tokenized at          tokenized per call             pre-tokenized at
  load time               (truncated to budget)            load time
```

The prefix and suffix are pre-tokenized at model load time and stored as `_prefix_ids` and `_suffix_ids`. Only the middle section — the instruct/query/document content — is tokenized per call. The middle section is truncated using `'longest_first'` strategy to a budget of `MAX_RERANK_LEN - len(_prefix_ids) - len(_suffix_ids)`. This ensures the total sequence length never exceeds `MAX_RERANK_LEN` regardless of document length.

6.2 Left-Padding and Sub-Batch Processing

Because each (query, document) pair produces a different-length token sequence, batch processing requires padding. The tokenizer's `pad()` method is called per sub-batch with `padding=True` and `return_tensors='np'`. The batch is also accompanied by:

- `attention_mask` — a binary mask that tells the model which tokens are real (1) and which are padding (0).
- `position_ids` — a row of sequential integers [0, 1, ..., seq_len-1] for each sequence in the batch. The ORT sliced graph requires explicit position IDs; they cannot be inferred from the attention mask.

The 50 candidates are processed in sub-batches of `SUB_BATCH=32`. This bounds peak VRAM usage: at 32 sequences × 1536 tokens × FP16, the input tensor is approximately 3 MB, well within the available VRAM headroom on the reranker's GPU.

Left-Padding Contract

The Qwen3-Reranker architecture reads its yes/no classification signal from the final (rightmost) token position. Right-padding would shift that token leftward and silently break all scoring. The tokenizer must always be initialized with `padding_side='left'`. Never change this setting.

6.3 ORT Forward Pass and Softmax Scoring

The ORT session runs the sliced ONNX graph which returns a `[batch, 2]` tensor of FP16 logits. The two columns correspond to the model's logit for the tokens 'no' (index 0) and 'yes' (index 1) at the final token position. These represent the model's raw confidence that the document is not relevant vs. is relevant to the query.

The logits are cast from FP16 to FP32 before softmax to avoid numerical overflow. The max-subtraction trick (subtracting the row maximum before exponentiation) is applied for numerical stability. The final relevance probability for each candidate is:

Softmax Relevance Probability

```
p = exp(yes_logit) / ( exp(no_logit) + exp(yes_logit) )

Implementation:
  logits_f32 = logits_fp16.astype(np.float32)          # FP16 → FP32
  logits_f32 -= logits_f32.max(axis=1, keepdims=True)  # max-subtraction
  exp_l = np.exp(logits_f32)
  probs = exp_l / exp_l.sum(axis=1, keepdims=True)      # normalize
  score = float(probs[0, 1])                            # yes-column probability
```

6.4 Sorting and Result Formatting

The 50 (doc, meta, score) tuples are sorted in descending order by the reranker probability and the top `TOP_K=10` are kept. Each result is formatted as a plain text block containing the source filename, chunk index, relevance percentage, and the raw chunk text. The ten blocks are joined with a `'---'` separator and returned as the MCP tool response.

Result Block Format

```
[Source: shigley.pdf | Chunk: 47 | Relevance: 94%]
<chunk text content here>
---
[Source: hibbeler_dynamics.pdf | Chunk: 112 | Relevance: 87%]
<chunk text content here>
```

7. MCP Interface — FastMCP Server

The server is instantiated as a FastMCP application named `'technical_books'`. FastMCP is a Python library built on top of the MCP specification that handles all JSON-RPC message framing, tool discovery, and argument deserialization automatically. The server exposes a single tool decorated with `@mcp.tool()`:

Tool Declaration

```
@mcp.tool()
def search_technical_books(query: str) -> str:
    """Search the technical books vector database for passages relevant to query."""
    _ensure_loaded()
    q_vec = _embed_model.encode(query, normalize_embeddings=True, prompt_name='query')
    results = _collection.query(query_embeddings=[q_vec.tolist()], n_results=K_RETRIEVE)
    return _rerank(query, results)
```

The tool signature is type-annotated, allowing FastMCP to automatically generate the JSON Schema tool definition that MCP hosts (such as Claude Desktop or Claude Code) use to understand the tool's interface. The docstring becomes the tool description visible to the language model.

7.1 Transport: stdio

`mcp.run()` starts the server on the stdio transport: it reads JSON-RPC requests from stdin and writes responses to stdout. This is the standard transport for MCP tools launched as child processes by a host application. The host spawns the server process, communicates over the process's stdio pipes, and kills it when the session ends.

7.2 Preload Mode

When run directly (`python rag-server.py`), the `__main__` block calls `_ensure_loaded()` before `mcp.run()`. This eagerly loads all three resources before the first MCP request arrives, eliminating the cold-start latency of the first query. When imported as a module by another script, the lazy initialization path still applies.

The server intentionally uses synchronous (non-async) tool handlers. The GPU forward passes (embedding and reranking) are not async-compatible without thread pool wrapping, and the single-client stdio transport does not require concurrent request handling. Using synchronous handlers keeps the code simple and avoids potential CUDA context sharing issues across asyncio event loop threads.

__main__ Block

```
if __name__ == '__main__':
    print('Preloading models...', file=sys.stderr)
    _ensure_loaded()
    print('Models loaded. Starting MCP server.', file=sys.stderr)
    mcp.run()
```

8. Design Decisions and Rationale

The table below summarizes the key architectural and implementation decisions made in rag-server.py, along with the reasoning behind each choice.

Decision	Rationale
Lazy loading (`_ensure_loaded`)	Models are not loaded at import time. This keeps the MCP server startup instant and avoids loading large GPU models unless the tool is actually invoked. All three resources — embedder, reranker, collection — are initialized only on the first call to `search_technical_books()`.
Dual-GPU isolation	The bi-encoder runs on `cuda:0` (the display-connected card) because it is a lightweight 0.6B model that coexists well with the desktop compositor. The ONNX reranker is isolated to `cuda:1` to prevent VRAM contention during batch forward passes.
ONNX Runtime over PyTorch for reranker	ORT `CUDAExecutionProvider` provides deterministic memory allocation via `kSameAsRequested`, eliminating the power-of-two over-allocation behavior of PyTorch's caching allocator. This is critical when operating at the edge of available VRAM on 12 GB cards.
Sliced ONNX graph (`model_sliced.onnx`)	The model graph is pre-sliced to expose only the final-token logits for the 'no'/'yes' output nodes. This avoids materializing the full vocabulary projection (32K+ tokens) on every forward pass, reducing memory bandwidth and latency.
Pre-tokenized prefix/suffix	The chat-template wrapper (prefix and suffix token IDs) is tokenized once at model load time and stored in `_prefix_ids`/`_suffix_ids`. This eliminates repeated tokenizer calls for the fixed portions of every (query, doc) pair.
Left-padding for the reranker	The Qwen3-Reranker architecture reads its classification signal from the final (rightmost) token position. Left-padding ensures that the final token position is consistent across variable-length sequences in a batch. Right-padding would misalign the signal token.
`K_RETRIEVE=50`, `TOP_K=10` (5:1 ratio)	Over-retrieving by 5x from ChromaDB before reranking is a standard two-stage retrieval trade-off: the fast bi-encoder provides recall, the slower cross-encoder provides precision. Fifty candidates is sufficient to cover most relevant content while keeping reranker latency under ~2 seconds on an RTX 3060.
FP16→FP32 cast before softmax	The sliced ONNX graph outputs FP16 logits. Softmax is numerically unstable in FP16, particularly when logit magnitudes are large. Casting to FP32 before the max-subtraction trick ensures stable probability computation.

FastMCP / stdio transport	The server communicates via stdout JSON-RPC (stdio transport), the standard MCP transport for local subprocess tools. All diagnostic output is explicitly redirected to stderr using a monkeypatched `print()` to prevent any log output from polluting the JSON-RPC channel.

9. Complete Data Flow Summary

The following is a step-by-step trace of a single call to `search_technical_books(query)`:

- Step 1 — MCP host sends a JSON-RPC `'tools/call'` request with argument `{query: '...'}`. FastMCP deserializes the argument and calls the Python function.
- Step 2 — `_ensure_loaded()` is called. All three globals are already populated after the first call; all three checks are fast no-ops.
- Step 3 — The query string is encoded by `_embed_model.encode()` on `cuda:0` into a 1024-dimensional float32 vector.
- Step 4 — `_collection.query()` performs HNSW approximate nearest-neighbor search and returns 50 candidate chunks with their metadata.
- Step 5 — `_rerank()` wraps each of the 50 (query, doc) pairs in the `Qwen3-Reranker` chat template, left-pads into sub-batches of 32, and runs the sliced ONNX graph on `cuda:1`.
- Step 6 — FP16 logits are cast to FP32. Softmax is applied over the (no, yes) logit pair per candidate to produce 50 relevance probabilities.
- Step 7 — The 50 candidates are sorted by probability in descending order. The top 10 are kept.
- Step 8 — Each of the 10 results is formatted as `'[Source: file | Chunk: N | Relevance: X%]\nchunk text'`. The 10 blocks are joined with `'---'`.
- Step 9 — The formatted string is returned as the tool response. FastMCP serializes it into a JSON-RPC response and writes it to stdout.

Latency Breakdown (typical)

Stage 1 encoding + HNSW query: ~50–150 ms. Stage 2 reranking (50 pairs, 2 sub-batches): ~500–1500 ms. Total wall-clock time per call: ~0.6–1.7 seconds on dual RTX 3060 hardware. First-call latency is higher (~15–30 s) due to model loading and CUDA kernel compilation; preload mode eliminates this spike.

Appendix A: Complete Configuration Reference

All configurable constants are defined in the Configuration section at the top of `rag-server.py`.

DB_PATH	C:\RAG-MCP\chroma_db
COLLECTION_NAME	technical_books
EMBED_MODEL	`Qwen/Qwen3-Embedding-0.6B`
RERANKER_MODEL	`zhiqing/Qwen3-Reranker-0.6B-ONNX`
EMBED_DEVICE	`cuda:0`
RERANK_GPU_ID	1
K_RETRIEVE	50
TOP_K	10
MAX_RERANK_LEN	1536
SUB_BATCH	32

Appendix B: Library Dependencies

Library	Role in Pipeline
sentence_transformers (SentenceTransformer)	Qwen3 bi-encoder load and query encoding on cuda:0
onnxruntime (ort)	GPU inference engine for the sliced reranker graph — CUDAExecutionProvider on RTX 3060
transformers (AutoTokenizer, snapshot_download)	Reranker tokenizer load and HuggingFace cache resolution
chromadb	Vector database client — HNSW cosine query against the technical_books collection
numpy	Sub-batch array construction, FP16→FP32 cast, softmax computation
mcp (FastMCP)	MCP server framework — JSON-RPC framing, tool registration, stdio transport
os	DLL path registration (os.add_dll_directory) and environment variable injection
sys	stderr redirect target for monkeypatched print() and diagnostic logging

Appendix C: VRAM Budget

Approximate VRAM usage on each GPU at steady-state with default configuration constants.

C.1 cuda:0 — Embedder

cuda:0 VRAM Estimate

```
Qwen3-Embedding-0.6B weights (FP32):  ~2.4 GB
CUDA kernel overhead + activations:   ~0.3 GB
Desktop compositor (display GPU):     ~0.2–0.5 GB
Estimated total at steady-state:      ~3.0–3.2 GB of 12 GB available
```

C.2 cuda:1 — Reranker

cuda:1 VRAM Estimate

```
model_sliced.onnx weights (FP16):     ~1.2 GB
ORT arena pre-allocation:             ~0.5–1.0 GB
Peak input tensor (SUB_BATCH=32,
 MAX_RERANK_LEN=1536, FP16):            ~0.003 GB  (transient per sub-batch)
Estimated total at steady-state:      ~1.8–2.5 GB of 12 GB available
```

VRAM Headroom

Both GPUs operate well within their 12 GB limits at the default configuration. The primary VRAM pressure on `cuda:1` is the ORT arena allocation strategy. `kSameAsRequested` is used precisely to prevent ORT from pre-allocating power-of-two chunks that could balloon usage above available headroom.

Appendix D: MCP Host Integration

The following snippet shows the entry in `claude_desktop_config.json` that registers `rag-server.py` as an MCP server with Claude Desktop.

claude_desktop_config.json entry

```
"technical_books": {
  "command": "C:\\RAG-MCP\\venv\\Scripts\\python.exe",
  "args": ["C:\\RAG-MCP\\rag-server.py"],
  "env": {}
}
```

Claude Desktop will launch `rag-server.py` as a subprocess and communicate with it via the process's stdin/stdout pipes. Because preload mode is active (the `__main__` block calls `_ensure_loaded()` before `mcp.run()`), all models will be loaded before the first `tools/call` request arrives.

Querying Effectively

Descriptive multi-term phrases significantly outperform keyword-only queries against this corpus. For example: 'methods for calculating gear tooth bending stress using Lewis equation' retrieves far more precisely than 'gear stress'. The bi-encoder's query-optimized embedding prefix is most effective when the query resembles a complete informational question rather than a search keyword.