Production RAG-MCP Server

(indexer.py)

Document Type	Technical Reference
Subject	`indexer.py` — RAG ingestion pipeline
Author	Brian Vowell
Revision	1.02
Vector Store	ChromaDB (HNSW cosine, hnsw:space='cosine')
Embedding Model	`Qwen3-Embedding-0.6B` (1024-dim, ONNX Runtime)
Hardware Target	Dual RTX 3060 12 GB / 60 GB DDR3 RAM
Throughput	~103 chunks/sec (sustained, dual-GPU)

Table of Contents

Table of Contents 2
Executive Summary 5
1 System Architecture Overview 6
1.1 Key Configuration Parameters 6
2 Startup Sequence 8
2.1 Step 1: ONNX Model Export (One-Time) 8
2.2 Step 2: Tokenizer Load 8
2.3 Step 3: Persistent Shared Memory Pool Allocation 8
2.4 Step 4: GPU Worker Process Spawning 9
2.5 Phase Output 9
3 PDF Discovery and Deduplication 10
3.1 File Discovery 10
3.2 Hash-Based Deduplication 10
3.3 Prefetch Architecture 10
3.4 Phase Output 10
4 Text Extraction and Chunking 11
4.1 PDF Text Extraction `extract_pdf_text()` 11
4.2 Text Cleaning `clean_text()` 11
4.3 Chunking Strategy `chunk_text()` 11
4.4 Phase Output 12
5 Batch Accumulation and Flush Trigger 13
5.1 Queue Architecture 13
6 The Flush Pipeline (Steps 1 through 7) 14
6.1 Step 1: Parallel Tokenization 14
6.2 Step 2: Write to Persistent Shared Memory 14
6.3 Step 3: Round-Robin GPU Submission 14
6.4 Step 4: Concurrent Result Drain 14
6.5 Step 5: Read Results from Shared Memory 15
6.6 Step 6: L2 Normalization 15

6.7 Step 7: Convert to Python List 15
6.8 Phase Output 15
7 GPU Worker Internals 16
7.1 Worker Initialization 16
7.2 Shared Memory Attach (One-Time) 16
7.3 Event Loop: Batch Processing 16
8 Asynchronous ChromaDB Upsert Pipeline 17
8.1 Upsert Worker `_upsert_worker()` 17
9 OCR Deferral, Checkpointing, and Error Handling 18
9.1 OCR Deferral Log 18
9.2 Breadcrumb Files 18
9.3 Checkpoint Logging 18
9.4 Graceful Shutdown 18
10 Design Decisions and Rationale 19
10.1 ONNX Runtime over Direct PyTorch Inference 19
10.2 Persistent Shared Memory over Per-Flush Allocation 19
10.3 Module-Level Worker Functions for Windows Pickling 19
10.4 `Qwen3-Embedding-0.6B` Model Selection 19
10.5 SQLite Performance Tuning 20
11 End-to-End Data Flow Summary 21
11.1 Thread and Process Map 22
12 Performance Tuning Guide 23
12.1 `BATCH_SIZE` 23
12.2 `ENCODE_BATCH_THRESHOLD` 23
12.3 SQLite Cache Size 23
12.4 OMP / MKL Thread Count Asymmetry 23
13 Annotated Function Reference 24
13.1 Functions 24
13.2 Classes 24
14 Failure Modes and Troubleshooting 26
14.1 Diagnostic Commands 26

Appendix A: Complete Configuration Reference 28
Appendix B: Library Dependencies 29
Appendix C: Memory Sizing Reference 30
C.1 Persistent SHM Pool 30
C.2 Required `MAX_FLUSH_BATCHES` 30
C.3 Peak RAM During Flush 30
C.4 ChromaDB HNSW Index RAM 30
Appendix D: Query-Time Usage via MCP Server 31
D.1 Query Pipeline 31
D.2 Retrieval Parameters 31
D.3 Returned Metadata Fields 31

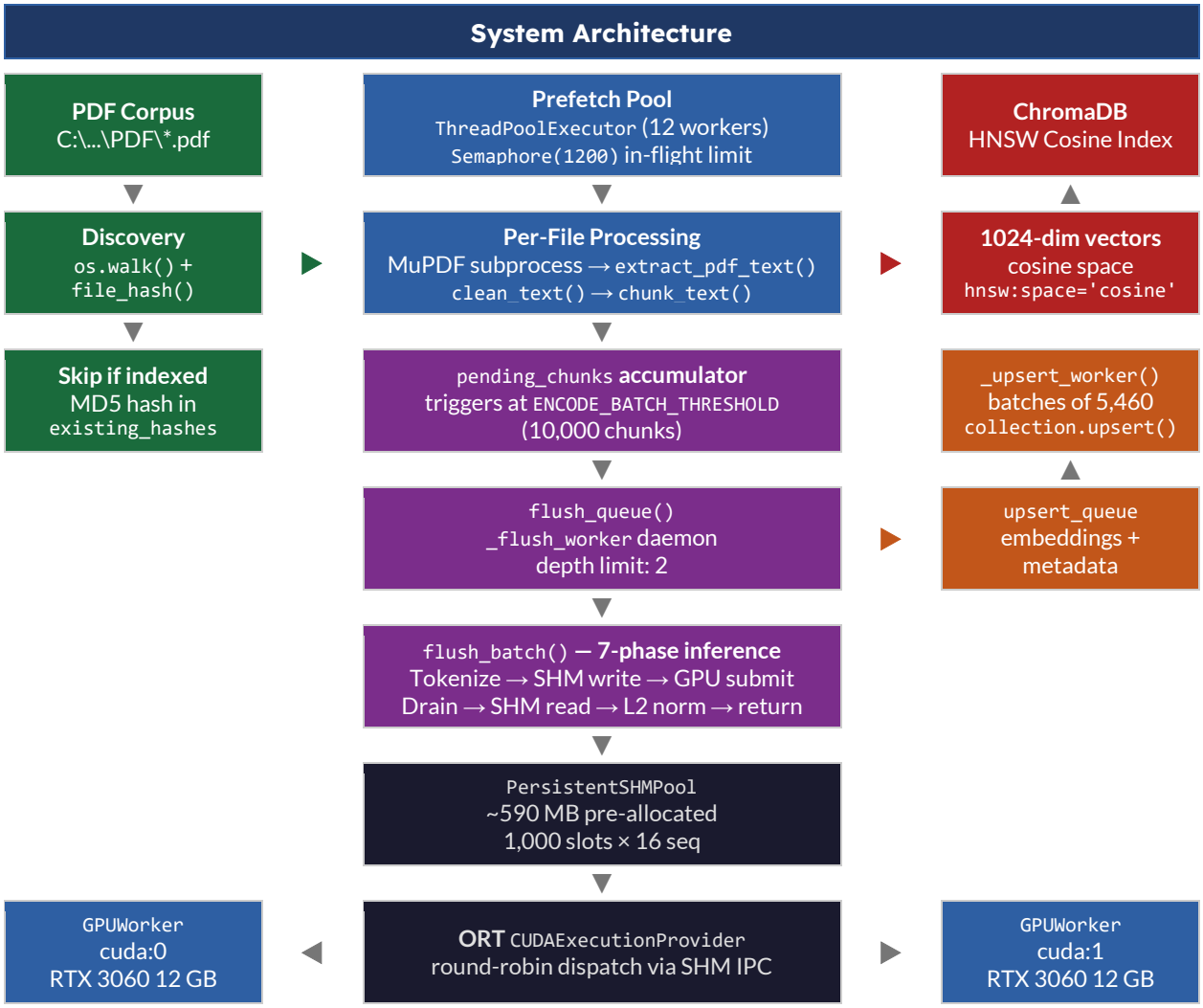
Executive Summary

`indexer.py` is a high-throughput RAG (Retrieval-Augmented Generation) ingestion pipeline designed to process a large corpus of PDF technical books and produce a queryable ChromaDB vector store. The pipeline is fully automated: it discovers PDFs, deduplicates them against the existing database, extracts and cleans text, chunks it into semantically coherent units, embeds those chunks using a dual-GPU ONNX inference stack, and upserts the resulting vectors into ChromaDB — all in a single invocation of main().

The design prioritizes sustained GPU utilization through three-stage asynchronous pipelining (prefetch → flush → upsert), zero-copy shared-memory IPC between the main process and GPU worker processes, and persistent ONNX Runtime sessions that eliminate per-batch CUDA context overhead. At sustained throughput, the pipeline processes approximately 103 chunks/sec on dual RTX 3060 12 GB hardware.

1 System Architecture Overview

The following diagram shows all major subsystems and their data flow relationships:



1.1 Key Configuration Parameters

The following table documents every tunable constant that controls pipeline behavior:

Parameter	Value / Description
MODEL_NAME	`Qwen3-Embedding-0.6B`
EMBEDDING_DIM	1,024 — full hidden-state dimension
MAX_SEQ_LEN	1,024 tokens — max tokenizer window
CHUNK_SIZE	1,000 words per chunk
OVERLAP_FRAC	5% word overlap between chunks
BATCH_SIZE	16 sequences per ORT inference call

ENCODE_BATCH_THRESHOLD	10,000 chunks — flush trigger
UPSERT_BATCH_LIMIT	5,460 — ChromaDB max per upsert call
PREFETCH_WORKERS	12 CPU threads for parallel extraction
PREFETCH_BUFFER	1,200 futures in-flight at once
FLUSH_QUEUE_DEPTH	2 — max pending flush batches
MAX_FLUSH_BATCHES	1,000 SHM slots (~19,200 max chunk capacity)
TEXT_LAYER_THRESHOLD	50 chars/page — minimum before OCR deferral
GPU_DEVICES	cuda:0 and cuda:1 (dual RTX 3060 12 GB)
DB_PATH	C:\RAG-MCP\chroma_db
PDF_ROOT	C:\Users\Brian\Desktop\PDF
ONNX_CACHE_DIR	C:\RAG-MCP\onnx_cache

2 Startup Sequence

When `main()` executes, it performs a fixed sequence of initialization steps before any PDF processing begins. Each step must succeed for the pipeline to proceed.

Phase 1 — **Startup Sequence**
Four sequential initialization steps before any PDF work begins

2.1 Step 1: ONNX Model Export (One-Time)

The first operation is `export_onnx()`. The function checks whether a cached `.onnx` file already exists at `ONNX_CACHE_DIR`. If it does, the export is skipped (cache hit). If not, the `Qwen3-Embedding-0.6B` model is loaded from `LOCAL_MODEL_PATH` or the Hugging Face hub as a PyTorch `AutoModel`, wrapped in `_QwenONNXWrapper`, and exported to ONNX format using `torch.onnx.export()`. The resulting `.onnx` file is reused on all subsequent runs.

Why ONNX Runtime instead of PyTorch?

PyTorch inference allocates and deallocates CUDA tensors on every forward pass, introducing high per-call overhead and memory fragmentation. ORT with `CUDAExecutionProvider` maintains a persistent CUDA execution context, uses operator fusion and graph-level optimization (`ORT_ENABLE_ALL`), and supports `enable_mem_reuse` — eliminating the `VirtualAlloc/VirtualFree` overhead that dominated earlier versions of this pipeline. The ONNX format is also framework-agnostic, separating the export phase (CPU, run once) from the inference phase (GPU, run many).

2.2 Step 2: Tokenizer Load

`AutoTokenizer` is loaded from `LOCAL_MODEL_PATH` (or the Hugging Face hub as a fallback). `padding_side` is explicitly set to 'left' per the `Qwen3-Embedding` documentation, which improves embedding quality for variable-length sequences by anchoring the `[CLS]` representation at a consistent position in the attention window.

2.3 Step 3: Persistent Shared Memory Pool Allocation

`PersistentSHMPool` allocates four named `SharedMemory` blocks at startup, sized for `MAX_FLUSH_BATCHES` slots. The pool is intentionally pre-allocated as a single large block rather than allocated per flush, eliminating Windows `VirtualAlloc/VirtualFree` calls that were previously the dominant bottleneck at high flush rates.

SHM Block	Size Formula & Purpose
shm_ids (input_ids)	`MAX_FLUSH_BATCHES` × `BATCH_SIZE` × `MAX_SEQ_LEN` × 8 bytes (int64)
shm_mask (attention_mask)	Same shape and size as shm_ids
shm_out (embeddings)	`MAX_FLUSH_BATCHES` × `BATCH_SIZE` × `EMBEDDING_DIM` × 4 bytes (float32)
shm_dims (batch dims)	`MAX_FLUSH_BATCHES` × 2 × 4 bytes (int32: actual_bs, actual_sl)

At 1,000 SHM slots, `BATCH_SIZE=16`, `MAX_SEQ_LEN=1024`, `EMBEDDING_DIM=1024`, the total allocation is approximately: (1000 × 16 × 1024 × 8) × 2 + (1000 × 16 × 1024 × 4) + (1000 × 2 × 4) = 262 MB + 262 MB + 65.5 MB + ~0 MB ≈ 590 MB total.

2.4 Step 4: GPU Worker Process Spawning

One `GPUWorker` is created per entry in `GPU_DEVICES` (`['cuda:0', 'cuda:1']`). Each `GPUWorker` spawns a dedicated child process using multiprocessing.get_context('spawn'). The spawn context is critical on Windows: the '`fork`' context is not available, and even if it were, forking a process with live CUDA state produces undefined behavior.

Why spawn context for GPU workers?

Windows does not support `fork()`. Additionally, CUDA contexts cannot be reliably inherited across process boundaries even on Linux — each CUDA context is owned by the process that created it. Using `spawn` ensures that each GPU worker starts with a clean CUDA context, loads its own ORT session, and communicates with the main process only through `multiprocessing.Queue` (IPC) and the pre-allocated shared memory pool. Module-level worker functions (`_gpu_worker_process`) are required for picklability on Windows.

2.5 Phase Output

onnx_path	Absolute path to cached `.onnx` file (`C:\RAG-MCP\onnx_cache\*.onnx`)
tokenizer	`AutoTokenizer` instance with `padding_side='left'`
shm_pool	`PersistentSHMPool` with four named SHM blocks (~590 MB)
workers	List of two `GPUWorker` instances (`cuda:0, cuda:1`), ready for attach

3 PDF Discovery and Deduplication

Phase 1 — **Discovery & Deduplication**
Traverse, hash, and filter before any extraction begins

3.1 File Discovery

`os.walk()` recursively traverses `PDF_ROOT` and collects all files with a `.pdf` extension into the `pdf_files` list. This single-pass discovery is performed before any worker threads are active, ensuring the list is stable for the duration of the run.

3.2 Hash-Based Deduplication

Deduplication is the first filter applied to every candidate PDF. Rather than storing filenames or paths (which change as the corpus is reorganized), the pipeline uses MD5 file hashes as stable identifiers. The function `file_hash()` reads the file in 64 KB blocks and accumulates the hash, avoiding loading large PDFs into memory.

At startup, `load_existing_hashes()` queries ChromaDB's underlying SQLite database directly for the `DISTINCT` set of `file_hash` metadata values stored across all existing embedding chunks. This is faster than querying ChromaDB's Python API when the database contains millions of chunks.

SQLite Index Strategy

The hash index (`idx_meta_file_hash`) on `embedding_metadata(key, string_value)` is `DROPPED` before bulk insertion and `RECREATED` afterward. This mirrors the standard bulk-load pattern used in relational database ETL: maintaining an index during millions of individual row inserts is far more expensive than rebuilding it once at the end. The `DROP` occurs in `main()` before the file loop; the `CREATE` occurs in `main()` after all flushes complete and both worker threads have exited.

3.3 Prefetch Architecture

Prefetching decouples I/O-bound PDF extraction from the GPU-bound embedding pipeline. A `ThreadPoolExecutor` with `PREFETCH_WORKERS=12` threads submits each PDF file as a `Future`. A `Semaphore(PREFETCH_BUFFER=1200)` limits how many futures may be in-flight simultaneously, preventing unbounded memory consumption.

Acquire Semaphore	Write Breadcrumb	MD5 Hash Check	MuPDF Subprocess	Release Semaphore
Block if 1,200 in-flight	Crash recovery trail	O(1) set lookup	Isolated crash boundary	On exit (success/fail)

3.4 Phase Output

pdf_files	Stable list of all .pdf paths under PDF_ROOT
existing_hashes	Set[str] of MD5 hashes already in ChromaDB
futures	ThreadPoolExecutor futures, semaphore-gated at 1,200

4 Text Extraction and Chunking

Phase 2 — **Text Extraction & Chunking**
PDF → clean text → semantically-bounded chunks

4.1 PDF Text Extraction `extract_pdf_text()`

`extract_pdf_text()` uses `PyMuPDF` (`fitz`) to iterate over every page and call `page.get_text()`. Pages that return fewer than `TEXT_LAYER_THRESHOLD` (50) characters are counted as OCR-required pages (`ocr_page_count`). These are not processed and the count is returned to the caller.

The 50-character threshold was chosen based on observation: legitimate text-layer pages in technical PDFs reliably produce hundreds to thousands of characters. Pages with < 50 characters are either blank, image-only, or have corrupted text layers requiring OCR.

Why isolate MuPDF in a subprocess?

`PyMuPDF` (`fitz`) uses the MuPDF C library, which has historically produced fatal segfaults on certain malformed or encrypted PDFs. A segfault in the main process would terminate the entire pipeline and corrupt the in-progress database state. By running each MuPDF extraction in a separate one-shot subprocess (Pool(1) with spawn context), any crash kills only that child process. The parent catches the exception, records the error, and continues to the next PDF. The 600-second timeout catches hangs on corrupt files.

4.2 Text Cleaning `clean_text()`

Before chunking, extracted text passes through `clean_text()`:

- Non-ASCII characters (codepoints > 0x7F) are replaced with a space. This removes ligatures, em dashes, and encoding artifacts common in PDFs that confuse word-boundary detection.
- Sequences of multiple blank lines are normalized to exactly two newlines (paragraph break).
- Each paragraph is stripped and re-joined with normalized whitespace.
- Empty paragraphs are discarded.

4.3 Chunking Strategy `chunk_text()`

`chunk_text()` implements a paragraph-aware sliding-window algorithm. `CHUNK_SIZE=1,000` words and `OVERLAP_FRAC=0.05` (50-word overlap at maximum chunk size) govern its behavior.

- Paragraphs are reconstructed from the double-newline-delimited text.
- For each paragraph, the total accumulated word count is checked against `CHUNK_SIZE`.
- Paragraphs that themselves exceed `CHUNK_SIZE` are split at sentence boundaries (regex on [.!?] followed by whitespace) to avoid splitting in mid-sentence.
- When a new paragraph would push the current chunk over `CHUNK_SIZE`, the chunk is emitted and the overlap window (last N words) is carried into the next chunk.
- Chunks shorter than 50 characters are discarded as noise.

Why 1,000-word chunks with 5% overlap?

Chunk size encodes the 'semantic pixel size' of the knowledge base permanently. Chunks that are too small lose context; chunks that are too large dilute relevance. For dense technical reference material (engineering manuals, physics textbooks), 1,000 words captures enough context for an embedding model to represent a coherent technical concept. The 5% overlap ensures that concepts spanning two chunks are represented in at least one chunk in their entirety, preventing hard retrieval failures at chunk boundaries.

4.4 Phase Output

result['chunks']	List[str] — normalized 1,000-word text chunks
result['ocr_pages']	int — count of pages below `TEXT_LAYER_THRESHOLD`
result['status']	'ok' | 'already_indexed' | 'error' | 'timeout'

5 Batch Accumulation and Flush Trigger

Phase 2 **Batch Accumulation**
Buffer chunks until `ENCODE_BATCH_THRESHOLD` is reached, then flush

The main loop iterates over futures from the `ThreadPoolExecutor` using `as_completed()`. As each prefetch result arrives, its chunks are appended to `pending_chunks` and its file metadata to `pending_files`.

When `len(pending_chunks)` reaches or exceeds `ENCODE_BATCH_THRESHOLD` (10,000), the accumulated batch is pushed to `flush_queue` as a tuple of `(pending_files, pending_chunks, flush_id)`. The accumulators are then reset to empty lists.

A final flush is issued after the file loop completes to handle any remaining chunks that did not reach the threshold.

Why 10,000 chunks per flush?

GPU throughput scales with batch count. At `BATCH_SIZE=16`, a 10,000-chunk flush produces 625 inference batches, which gives both GPUs enough work to amortize kernel launch overhead and maintain high utilization. Flushing too frequently (small threshold) underutilizes the GPUs; flushing infrequently (large threshold) increases peak RAM consumption and latency to first ChromaDB write. 10,000 was empirically tuned to achieve ~103 chunks/sec at stable throughput without exceeding available system RAM.

5.1 Queue Architecture

Three queues coordinate the three-stage async pipeline:

Queue	Producer → Consumer / Purpose
`flush_queue (maxsize=2)`	`main()` → _flush_worker thread. Holds `(files, chunks, id)` batches pending GPU inference.
`upsert_queue (maxsize=2)`	`_flush_worker` → `_upsert_worker` thread. Holds `(files, chunks, embeddings, id, timing)` batches.
`result_queue (unbounded)`	`_upsert_worker` → `main()`. Holds `('ok'|'err', flush_id, counts)` completion signals.

`maxsize=2` on `flush_queue` and `upsert_queue` provides back-pressure: if the GPU flush or ChromaDB upsert falls behind, the main loop blocks on `queue.put()` rather than accumulating unbounded memory.

6 The Flush Pipeline (Steps 1 through 7)

Phase 3 — **Flush Pipeline**
Seven sequential steps from raw text to normalized embeddings

`flush_batch()` is the core inference function. It accepts a batch of text chunks and returns a list of 1024-dimensional normalized embedding vectors. The function executes in seven sequential steps:

Tokenize	SHM Write	GPU Submit	Drain	SHM Read	Normalize	Return
12-thread parallel	Zero-copy fill	Round-robin dispatch	Concurrent collect	Parallel vstack	L2 per embedding	List of vectors

6.1 Step 1: Parallel Tokenization

All `n_batches` tokenization calls run concurrently via `cpu_tok_executor` (`ThreadPoolExecutor`, 12 workers). The tokenizer converts each `BATCH_SIZE` slice of text chunks into `input_ids` and `attention_mask` numpy arrays (int64), with `padding=True`, `truncation=True`, and `max_length=MAX_SEQ_LEN=1024`. Using a `ThreadPoolExecutor` for tokenization releases the Python GIL during numpy operations and tokenizer C-extension calls, achieving real parallelism across 12 CPU threads.

6.2 Step 2: Write to Persistent Shared Memory

After tokenization, all batches are written into the pre-allocated SHM pool using `cpu_shm_executor` (a separate 12-worker `ThreadPoolExecutor`). Each `write_batch()` call copies `input_ids` and `attention_mask` into the 3D numpy arrays `(arr_ids, arr_mask)` at slot index `i`, and records the actual batch size and sequence length in `arr_dims`. A separate executor for SHM writes avoids blocking the tokenizer threads while memory copy is in progress.

6.3 Step 3: Round-Robin GPU Submission

Batch indices are distributed across workers using a simple round-robin modulo: `w_idx = batch_idx % n_workers`. For two GPUs, batches 0, 2, 4... go to cuda:0 and batches 1, 3, 5... go to cuda:1. Each `workers[w_idx].submit(batch_idx)` places a `('batch', batch_idx)` message into the worker's in-queue. Round-robin distribution provides uniform load balancing when both GPUs have identical capability (both RTX 3060 12 GB), avoids any scheduling overhead, and keeps both CUDA devices continuously occupied.

6.4 Step 4: Concurrent Result Drain

Two `drain_worker` threads (one per GPU) run concurrently, each blocking on `worker.collect()` for its share of submitted batches. As results arrive, they are recorded in the `completed[]` boolean array under `collect_lock`. Separating drain threads from the submit loop allows both GPUs to run and return results in parallel. If either GPU reports an error or fails to respond within 240 seconds, the entire flush is aborted.

6.5 Step 5: Read Results from Shared Memory

Once all batches are confirmed complete (`completed[]` all True), `read_batch()` reads `arr_out[i, :actual_bs, :]` for each batch index via `cpu_shm_executor`. The results are vertically stacked via `np.vstack()` into a single `(n_chunks, 1024)` float32 array. The `.copy()` call is required: numpy views into shared memory reference the SHM buffer directly, and without `.copy()`, vstack would collect views that could be overwritten by the next flush before L2-normalization completes.

6.6 Step 6: L2 Normalization

Each `embedding` vector is divided by its L2 norm: `embeddings /= norm(embeddings, axis=1, keepdims=True)`. Norms that are exactly zero (degenerate zero vectors) are clamped to 1.0 to avoid division by zero. ChromaDB's HNSW index is configured with `hnsw:space='cosine'`. Cosine similarity between two vectors equals their dot product when both vectors are unit-normalized. Pre-normalizing before storage means ChromaDB's inner product computation at query time is equivalent to cosine similarity without the additional normalization overhead.

6.7 Step 7: Convert to Python List

`embeddings.tolist()` converts the numpy array to a Python list of lists, which is the format required by the ChromaDB Python client's `upsert()` method. The numpy array is immediately deleted after conversion to release the ~40 MB block.

6.8 Phase Output

embeddings_list	List[List[float]] — shape (n_chunks, 1024), L2-normalized
timing_dict	Dict[str, float] — per-phase wall-clock timings in seconds

7 GPU Worker Internals

Phase 3 **GPU Worker Lifecycle**
Initialization → SHM attach → persistent event loop

Each `GPUWorker` manages a persistent child process running `_gpu_worker_process()`. The worker lifecycle consists of three stages: initialization, attach, and event loop.

7.1 Worker Initialization

On startup, the worker process:

- Sets `OMP_NUM_THREADS` and `MKL_NUM_THREADS` to `1` to prevent nested thread pool contention inside ORT
- Adds the CUDA DLL directory to the Windows DLL search path
- Creates an ORT `SessionOptions` with full graph optimization enabled, memory reuse enabled, and memory pattern optimization enabled
- Creates an ORT `InferenceSession` from the cached ONNX file, with `CUDAExecutionProvider` as primary and `CPUExecutionProvider` as fallback
- Verifies that `CUDAExecutionProvider` was actually activated (not silently downgraded to `CPU`)
- Runs a warmup inference pass (`batch_size=2`, `seq_len=128`) to force CUDA kernel compilation before timing-sensitive work begins
- Posts 'ready' to the output queue, signaling the parent process to proceed

Why persistent workers rather than per-flush subprocess pools?

Spawning a new process for each flush incurs 2-5 seconds of overhead on Windows: Python interpreter startup, module imports, ONNX model loading, and CUDA context initialization. With hundreds of flushes per run, this would dominate wall-clock time. Persistent workers pay this cost exactly once at startup, then process all subsequent batches with sub-millisecond IPC overhead via shared memory.

7.2 Shared Memory Attach (One-Time)

After startup, the parent sends an `('attach', ids_name, mask_name, out_name, dims_name)` message. The worker attaches to all four SHM blocks by name and constructs numpy views with the same dimensional layout as the parent's `PersistentSHMPool`. This attach is performed once and the handles are held for the lifetime of the worker process.

7.3 Event Loop: Batch Processing

In its event loop, the worker blocks on `in_q.get()`. When a `('batch', batch_idx)` message arrives, it reads the actual batch size and sequence length from `arr_dims`, slices the corresponding `input_ids` and `attention_mask` from SHM, runs `session.run()`, writes the output embeddings back into `arr_out`, and posts 'ok' to the output queue. When a None sentinel arrives, the worker closes its SHM handles and exits cleanly.

8 Asynchronous ChromaDB Upsert Pipeline

Phase 4 — **ChromaDB Upsert Pipeline**
Parallel flush + upsert ensures GPU is never stalled on I/O

`_flush_worker` runs in a dedicated daemon thread. It pulls `(pending_files, pending_chunks, flush_id)` from `flush_queue`, calls `flush_batch()` to run the full 7-steps inference, then immediately posts the resulting embeddings to `upsert_queue` and returns. This design ensures that the GPU is never stalled waiting for ChromaDB I/O. While `_upsert_worker` is writing the previous batch to SQLite, the GPUs are already running inference on the next batch.

8.1 Upsert Worker `_upsert_worker()`

`_upsert_worker` runs in a second dedicated daemon thread. For each batch, it iterates over `pending_files` and slices the corresponding chunk text and embedding vectors, then calls `collection.upsert()` in sub-batches of `UPSERT_BATCH_LIMIT` (5,460 — ChromaDB's per-call maximum).

Each chunk is stored with four metadata fields:

Metadata Field	Value / Description
source	Relative path of the source PDF from `PDF_ROOT`
filename	Basename of the PDF file
chunk_index	Zero-based index of this chunk within the file
file_hash	MD5 hash of the source PDF (used for deduplication)

ChromaDB HNSW Configuration

The collection is created with `hnsw:space='cosine', hnsw:M=16, hnsw:construction_ef=64`, and `hnsw:search_ef=100`. `M=16` controls the number of bidirectional links per node in the HNSW graph — higher M improves recall at the cost of memory and construction time. `construction_ef=64` sets the size of the dynamic candidate list during graph construction. `search_ef=100` is the query-time candidate pool size. These values were chosen for high recall on a technical corpus where false negatives (missed relevant chunks) are more costly than slightly increased query latency.

Chunk IDs are generated as `'{fhash}_{chunk_index}'`, making each ID globally unique across the corpus and deterministic: re-indexing the same file produces identical IDs, so `upsert()` is idempotent. `existing_hashes` is updated in-memory after each file completes so that the deduplication check remains current within a run.

9 OCR Deferral, Checkpointing, and Error Handling

Phase 5 — **Error Handling & Recovery**
OCR deferral, breadcrumbs, checkpoints, graceful shutdown

9.1 OCR Deferral Log

Pages with fewer than `TEXT_LAYER_THRESHOLD` (50) characters are counted as requiring OCR. When a PDF has any OCR-required pages, its hash, page count, and path are appended to `OCR_DEFER_LOG` (`C:\RAG-MCP\OCR_deferred.txt`). This log persists across runs and serves as the input manifest for a future OCR pass.

Importantly, the input PDF file is still indexed from its available text-layer pages. The OCR deferral only records the pages that could not be extracted — it does not discard the file. This ensures that partially text-layer PDFs contribute their extractable content immediately.

9.2 Breadcrumb Files

Each prefetch thread writes a breadcrumb file at `C:\RAG-MCP\prefetch_{tid}.last` containing the current PDF path. The upsert worker writes to `last_upsert.txt`. These files are overwritten on every step and provide a post-mortem trail if the process crashes, enabling identification of the last successfully processed file.

9.3 Checkpoint Logging

Every `CHECKPOINT_EVERY` (10) files processed triggers a summary log to the `tqdm` console showing: chunks this run, total in DB, skip/fail counts, OCR pages, flush count, RAM usage, and average throughput rate. This provides real-time visibility into pipeline health during long-running indexing sessions.

9.4 Graceful Shutdown

The main loop runs inside a `try/finally` block. On `KeyboardInterrupt` or any exception, the `finally` block sends None sentinels to `flush_queue` and `upsert_queue` to signal both worker threads to exit, calls `shutdown(wait=False)` on all three `ThreadPoolExecutors`, calls `shutdown()` on each `GPUWorker` (which sends a None sentinel to each worker process and joins it), and calls `shm_pool.shutdown()` to release all SHM blocks and prevent Windows shared memory leaks.

10 Design Decisions and Rationale

This section documents the key architectural choices made during development, including the problems they solved and the alternatives that were considered and rejected.

10.1 ONNX Runtime over Direct PyTorch Inference

Dimension	PyTorch (rejected)	ONNX Runtime (selected)
Per-batch overhead	High: tensor allocation, CUDA memcpy, Python GIL	Low: memory reuse, persistent session, no per-batch alloc
Graph optimization	None at inference time	Full graph fusion (`ORT_ENABLE_ALL`)
Portability	Requires PyTorch, CUDA Toolkit, matching versions	Single `.onnx` file, any ORT-compatible platform
Memory reuse	`enable_mem_reuse` not available	`enable_mem_reuse` + `enable_mem_pattern` active

10.2 Persistent Shared Memory over Per-Flush Allocation

Earlier versions of the pipeline allocated and deallocated `SharedMemory` objects on every flush. On Windows, this translates directly to `VirtualAlloc`/`VirtualFree` system calls, which are relatively expensive (~1-5 ms per call). At hundreds of flushes per large corpus, this overhead was measurable. The persistent pool eliminates this entirely.

10.3 Module-Level Worker Functions for Windows Pickling

`_gpu_worker_process` is defined at module level, not as a lambda or nested function. Windows multiprocessing with spawn context uses `pickle` to serialize the target function for transmission to the child process. Only module-level functions are picklable. Lambda functions and closures are not, and would raise a `PicklingError` at worker spawn time.

10.4 `Qwen3-Embedding-0.6B` Model Selection

`Qwen3-Embedding-0.6B` was selected for the following properties:

- 1024-dimensional embeddings (vs. 384 for `all-MiniLM-L6-v2`) — denser representation space for technical vocabulary
- 32,767-token context window (vs. 512 for BERT-based models) — capable of embedding long paragraphs without truncation
- 600,000,000 parameter scale — fast enough for the dual RTX 3060 hardware without requiring model parallelism
- Strong MTEBv1 (Massive Text Embedding Benchmark) retrieval performance on technical domains
- Local model weights stored at `C:\RAG-MCP\qwen3_embedding_0.6b` — no internet required after initial download
- Supported Languages: 100+

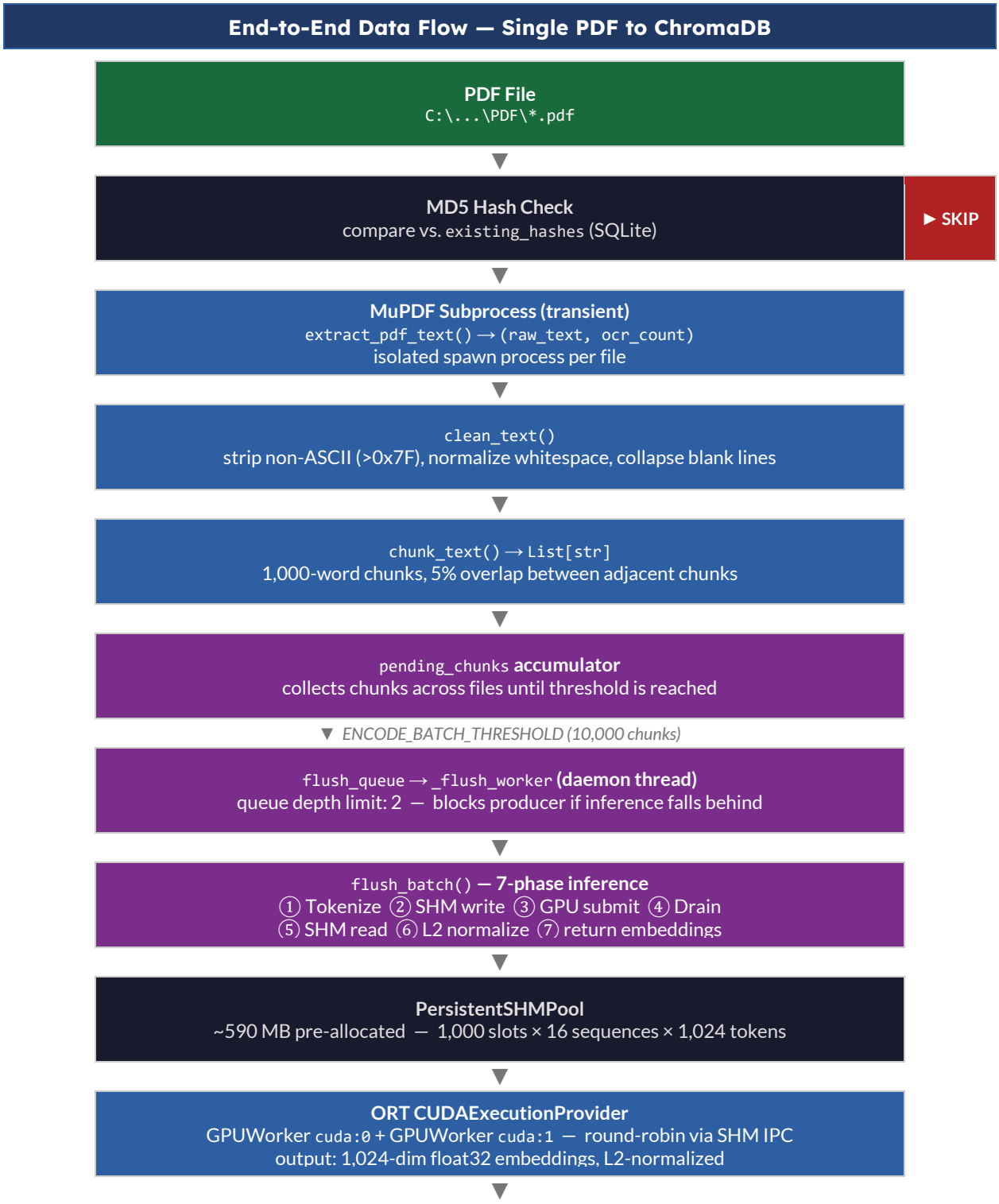
10.5 SQLite Performance Tuning

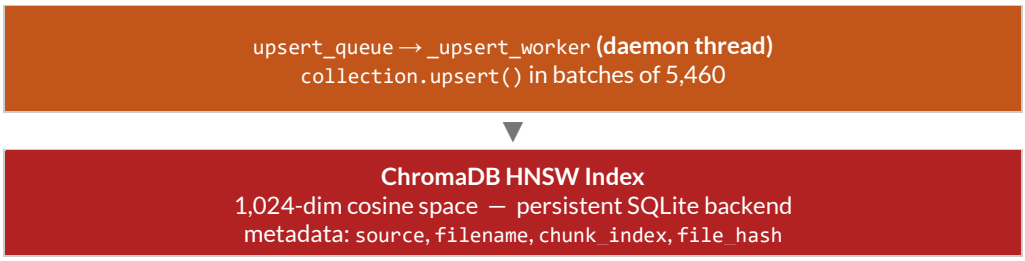
ChromaDB uses SQLite as its persistence layer. Three `PRAGMA` settings are applied at startup to optimize throughput for bulk writes:

PRAGMA	Value	Effect
synchronous	NORMAL	Reduces `fsync` calls; safe for crash recovery after transaction commit
cache_size	-10485760	10 GB page cache — keeps B-tree working set in RAM during bulk writes
temp_store	MEMORY	Stores temp tables in RAM instead of the temp file on NVMe

11 End-to-End Data Flow Summary

The following diagram traces a single PDF from disk to ChromaDB:





11.1 Thread and Process Map

The following table documents every concurrent execution unit in the pipeline:

Execution Unit	Type	Role
main()	Thread (main)	File loop, queue management, result aggregation
_flush_worker	Daemon Thread	Pulls flush_queue → runs flush_batch() → posts upsert_queue
_upsert_worker	Daemon Thread	Pulls upsert_queue → collection.upsert() → updates existing_hashes
cpu_tok_executor	ThreadPoolExecutor (12)	Parallel tokenization of BATCH_SIZE chunks
cpu_shm_executor	ThreadPoolExecutor (12)	Parallel SHM read and write
executor (prefetch)	ThreadPoolExecutor (12)	Prefetch futures for PDF files
GPUWorker cuda:0	Spawn Process	ORT inference on GPU 0 via SHM IPC
GPUWorker cuda:1	Spawn Process	ORT inference on GPU 1 via SHM IPC
MuPDF subprocess	Spawn Process (transient)	Isolated PDF extraction per file

12 Performance Tuning Guide

The following subsections explain how each tunable parameter affects throughput, memory consumption, and GPU utilization, calibrated to the RTX 3060 12 GB / 60 GB DDR3 configuration.

12.1 BATCH_SIZE

`BATCH_SIZE` controls how many text chunks are embedded per ORT inference call. VRAM per pass ≈ `BATCH_SIZE` × `MAX_SEQ_LEN` × model_hidden_size × 4 bytes.

BATCH_SIZE	VRAM per Pass
8	~33 MB
16 (current)	~67 MB
32	~134 MB
64	~268 MB

12.2 ENCODE_BATCH_THRESHOLD

Controls how many chunks accumulate before a GPU flush. Current: 10,000 / 16 per batch = 625 ORT batches per flush, split across 2 GPUs = ~313 batches per GPU. This provides 3-5 seconds of continuous GPU work per flush at observed throughput.

12.3 SQLite Cache Size

`PRAGMA cache_size = -10485760` requests a 10 GB page cache. With 60 GB RAM this keeps the entire B-tree working set for `embedding_metadata` in memory during bulk writes, eliminating the random-I/O wall that would otherwise form as the database exceeds the NVMe random-write IOPS ceiling. Reduce to `-2097152` (2 GB) on machines with less RAM.

12.4 OMP / MKL Thread Count Asymmetry

`OMP_NUM_THREADS` and `MKL_NUM_THREADS` are 12 in the main process and 1 in each GPU worker. The main process uses numpy for L2 normalization and array vstack, which benefit from parallelism. GPU worker processes must not spawn additional threads that contend with CUDA kernel execution and the shared CPU resources.

`MAX_FLUSH_BATCHES` sizing constraint

`MAX_FLUSH_BATCHES` must always be >= `ceil(ENCODE_BATCH_THRESHOLD / BATCH_SIZE)`. At current settings: `ceil(10000 / 16) = 625`. Current setting of 1000 provides 60% headroom. Each 1,000-slot increase in `MAX_FLUSH_BATCHES` adds approximately 312 MB to the SHM pool allocation. Violating this constraint raises a `RuntimeError` immediately at flush time, before any GPU work begins.

13 Annotated Function Reference

Complete signature and behavior reference for every function and class in `indexer.py`.

13.1 Functions

Function / Class	Signature and Description
clean_text(text)	str → str. Strips non-ASCII (>0x7F), normalizes whitespace, collapses blank lines. No side effects.
chunk_text(text, max_words, overlap_frac)	str, int, float → List[str]. Paragraph-aware sliding-window chunker. Chunks < 50 chars discarded.
file_hash(path)	str → str. MD5 hex digest via 64 KB streaming blocks. Returns 32-char hex string.
extract_pdf_text(path)	str → Tuple[str, int]. PyMuPDF text extraction. Returns (cleaned_text, ocr_page_count).
configure_sqlite_db(db_file)	str → None. Applies synchronous=NORMAL, 10 GB cache, temp_store=MEMORY PRAGMAs.
load_existing_hashes(collection, total_in_db, db_file)	Collection, int, str → Set[str]. Creates hash index if absent, returns set of known hashes.
export_onnx(model_name, cache_dir)	str, str → str. One-time ONNX export or cache hit. Returns absolute .onnx path.
_extract_in_subprocess(pdf_path, pdf_root)	str, str → Optional[dict]. Spawn target. Hash + extract + chunk. Returns status dict.
prefetch(pdf_path, pdf_root, semaphore, existing_hashes)	str, str, Semaphore, Set[str] → Optional[dict]. Semaphore-gated hash check + subprocess extraction.
_gpu_worker_process(dev, onnx_path, in_q, out_q)	str, str, Queue, Queue → None. Spawn target. ORT session lifecycle + SHM event loop.
flush_batch(...)	Core inference function. Phases 1–7. Returns (n_chunks, files, chunks, embeddings_list, timing_dict).
_flush_worker(...)	Daemon thread. Pulls flush_queue → runs flush_batch → posts to upsert_queue.
_upsert_worker(...)	Daemon thread. Pulls upsert_queue → collection.upsert() → updates existing_hashes → posts result_queue.

13.2 Classes

Class Method	Description
PersistentSHMPool.__init__()	Allocates four named SHM blocks. Exposes arr_ids, arr_mask, arr_out, arr_dims numpy views.
PersistentSHMPool.names	Property. Returns (ids_name, mask_name, out_name, dims_name) for worker attach.
PersistentSHMPool.shutdown()	.close() + .unlink() on all four blocks. Must be called before exit to prevent SHM leaks on Windows.

GPUWorker.__init__(dev, onnx_path)	Spawns child process. Waits up to 240s for 'ready' signal.
GPUWorker.attach_shm(pool)	Sends attach message; waits for 'attach_ok'. One-time call per worker lifetime.
GPUWorker.submit(batch_idx)	Non-blocking send of ('batch', batch_idx) to worker in_q.
GPUWorker.collect(timeout)	Blocking receive from worker out_q. Raises RuntimeError on dead process or timeout.
GPUWorker.shutdown()	Sends None sentinel, joins process (30s timeout), terminates if still alive.

14 Failure Modes and Troubleshooting

Common failure patterns, root causes, and remediation steps.

Symptom	Root Cause	Remediation
GPU Worker RuntimeError: 'failed to start'	ORT cannot load ONNX, CUDA DLL missing, or bad device_id	Verify onnxruntime-gpu installed; check ONNX_CACHE_DIR path; confirm GPU_DEVICES list
CUDA Execution Provider absent from active providers	onnxruntime-gpu not installed; CPU-only package present	pip install onnxruntime-gpu; uninstall onnxruntime
MuPDF subprocess Timeout Error (600s)	Corrupt or encrypted PDF hanging in MuPDF C loop	Remove file from PDF_ROOT or decrypt it before indexing
RuntimeError: Flush requires N batches but pool holds M	ENCODE_BATCH_THRESHOLD / BATCH_SIZE > MAX_FLUSH_BATCHES	Increase MAX_FLUSH_BATCHES to >= ceil(ENCODE_BATCH_THRESHOLD / BATCH_SIZE)
ChromaDB upsert ValueError on batch size	Batch exceeds 5,461-item ChromaDB limit	Lower UPSERT_BATCH_LIMIT below 5,461
Throughput < 20 chunks/sec	ORT falling back to CPU, or ENCODE_BATCH_THRESHOLD too low	Verify CUDAExecutionProvider active; increase ENCODE_BATCH_THRESHOLD
Shared Memory leak on restart	PersistentSHMPool.shutdown() not reached before kill	Run cleanup script: shm.SharedMemory(name).unlink() for each leaked block name
All chunks produce identical embeddings	ONNX export used wrong model path (untrained weights)	Delete ONNX cache and re-export with correct LOCAL_MODEL_PATH
Tokenizer padding produces degraded embeddings	padding_side not set to 'left'	Ensure tokenizer.padding_side = 'left' is set before any tokenization

14.1 Diagnostic Commands

Check GPU VRAM:

nvidia-smi

```
nvidia-smi --query-gpu=name,memory.total,memory.free --format=csv,noheader
```

Verify ORT CUDA provider:

python

```
python -c "import onnxruntime as ort; print(ort.get_available_providers())"
```

Check ChromaDB chunk count:

python

```
python -c \"import chromadb; c=chromadb.PersistentClient('C:\\RAG-MCP\\chroma_db');
print(c.get_collection('technical_books').count())\"
```

Count indexed PDFs via SQLite:

```python
python -c \"import sqlite3; c=sqlite3.connect('C:\\RAG-MCP\\chroma_db\\chroma.sqlite3');
print(c.execute('SELECT COUNT(DISTINCT string_value) FROM embedding_metadata WHERE
key=\"file_hash\"').fetchone())\"
```

Appendix A: Complete Configuration Reference

All configurable constants are defined in the Configuration section at the top of `indexer.py`.

PDF_ROOT	C:\Users\Brian\Desktop\PDF
DB_PATH	C:\RAG-MCP\chroma_db
ONNX_CACHE_DIR	C:\RAG-MCP\onnx_cache
OCR_DEFER_LOG	C:\RAG-MCP\OCR_deferred.txt
COLLECTION_NAME	technical_books
MODEL_NAME	Qwen/Qwen3-Embedding-0.6B
LOCAL_MODEL_PATH	C:\RAG-MCP\qwen3_embedding_0.6b
EMBEDDING_DIM	1024
MAX_SEQ_LEN	1024
CHUNK_SIZE	1000
BATCH_SIZE	16
OVERLAP_FRAC	0.05
CHECKPOINT_EVERY	10
TEXT_LAYER_THRESHOLD	50
PREFETCH_WORKERS	12
GPU_DEVICES	['cuda:0','cuda:1']
ENCODE_BATCH_THRESHOLD	10000
UPSERT_BATCH_LIMIT	5460
PREFETCH_BUFFER	1200
FLUSH_QUEUE_DEPTH	2
MAX_FLUSH_BATCHES	1000

Appendix B: Library Dependencies

Library	Role in Pipeline
PyMuPDF (fitz)	PDF text extraction — page.get_text() via MuPDF C library
transformers (AutoTokenizer, AutoModel)	Qwen3-Embedding tokenizer load and ONNX export wrapper
onnxruntime (ort)	GPU inference engine — CUDAExecutionProvider on RTX 3060
chromadb	Vector database — HNSW cosine index, persistent SQLite backend
numpy	Tokenized tensor construction, SHM array views, L2 normalization
torch	ONNX export only — not used during inference
sqlite3	Direct SQLite queries for hash loading and index management
psutil	RAM utilization reporting in checkpoint logs
tqdm	Progress bar and live console output
multiprocessing.shared_memory	Zero-copy IPC between main process and GPU workers
concurrent.futures.ThreadPoolExecutor	Parallel tokenization, SHM I/O, and PDF prefetching
hashlib (MD5)	File identity hashing for deduplication

Appendix C: Memory Sizing Reference

Formulas for recalculating memory requirements when tuning `MAX_FLUSH_BATCHES`, `BATCH_SIZE`, `MAX_SEQ_LEN`, or `EMBEDDING_DIM`.

C.1 Persistent SHM Pool

SHM Pool Sizing Formulas

```
shm_ids  = MAX_FLUSH_BATCHES * BATCH_SIZE * MAX_SEQ_LEN * 8   (int64 bytes)
shm_mask = MAX_FLUSH_BATCHES * BATCH_SIZE * MAX_SEQ_LEN * 8   (int64 bytes)
shm_out  = MAX_FLUSH_BATCHES * BATCH_SIZE * EMBEDDING_DIM * 4 (float32 bytes)
shm_dims = MAX_FLUSH_BATCHES * 2 * 4                          (int32 bytes)
TOTAL    = shm_ids + shm_mask + shm_out + shm_dims

At defaults (MAX_FLUSH_BATCHES=1000, BATCH_SIZE=16, MAX_SEQ_LEN=1024, EMBEDDING_DIM=1024):
shm_ids  = 1000 * 16 * 1024 * 8 = 131,072,000 bytes  (125 MB)
shm_mask = 131,072,000 bytes  (125 MB)
shm_out  =  65,536,000 bytes  (62.5 MB)
shm_dims =       8,000 bytes
TOTAL    =   ~312.5 MB
```

C.2 Required `MAX_FLUSH_BATCHES`

Constraint

```
MAX_FLUSH_BATCHES >= ceil(ENCODE_BATCH_THRESHOLD / BATCH_SIZE)
Current: ceil(10000 / 16) = 625.  Setting of 1000 provides 60% headroom.
```

C.3 Peak RAM During Flush

Transient RAM During flush_batch()

```
embeddings_np   = n_chunks * EMBEDDING_DIM * 4     (~40 MB at 10,000 chunks)
embeddings_list = n_chunks * EMBEDDING_DIM * 8     (~80 MB at 10,000 chunks)
Both are transient; np array is deleted before list is created, so peak is ~80 MB above SHM
pool.
```

C.4 ChromaDB HNSW Index RAM

HNSW RAM Estimate

```
HNSW RAM (bytes) ≈ n_vectors * (EMBEDDING_DIM * 4 + M * 2 * 8)
At 500,000 vectors: 500,000 * (4096 + 256) = ~2.1 GB
With 60 GB RAM and 16 GB chroma_memory_limit_bytes, the index can grow to
approximately 12 million vectors before RAM constrains growth at these settings.
```

Appendix D: Query-Time Usage via MCP Server

The index built by `indexer.py` is consumed at query time by the RAG-MCP server ('technical_books' MCP tool). This appendix documents how indexed data is retrieved and used to answer queries from Claude.

D.1 Query Pipeline

When Claude calls the technical_books MCP tool, the following sequence executes:

User Query String	Qwen3 Tokenizer	ORT CUDA Inference	L2 Normalize	ChromaDB HNSW Query
Natural language input	Same settings as indexing	1024-dim vector	Unit vector for cosine	Top-K nearest chunks

The query embedding must use the same `Qwen3-Embedding-0.6B` model, the same `padding_side='left'` tokenizer setting, and the same L2 normalization. Any deviation from indexing-time embedding produces vector space misalignment and degrades retrieval quality.

D.2 Retrieval Parameters

n_results (K)	10-20 chunks
hnsw:search_ef	100 (set at collection creation)
where (filter)	Optional: {source: 'book.pdf'}
Distance metric	Cosine (hnsw:space='cosine')

D.3 Returned Metadata Fields

source	Engineering\Shigley\shigley.pdf
filename	shigley.pdf
chunk_index	47
file_hash	a3f7c91d...

Cosine Distance Interpretation

ChromaDB returns cosine distances (lower = more similar). Because all stored embeddings are L2-normalized, cosine distance = 1 - dot_product. A distance of 0.0 = identical vectors; 2.0 = maximally dissimilar. Typical relevant chunk distances for technical queries against this corpus fall in the range 0.05-0.35. Chunks above 0.6 are usually off-topic and should be filtered by the MCP server before passing to Claude.