ChatGPT

Modular Document Retrieval Pipeline – Design & Scaffold

Architecture Overview

The pipeline follows a **modular, hybrid retrieval architecture** that integrates multiple strategies (lexical, sparse, and dense) for superior relevance [1]. Documents are first **chunked** into semantically coherent blocks, then **embedded** using both dense and sparse models, and finally **indexed** into appropriate stores for retrieval. At query time, the system executes **multi-strategy search** (BM25, SPLADE sparse, dense) in parallel and fuses results via weighted or Reciprocal Rank Fusion, with an optional reranking step [2]. This design emphasizes open-source tools and recent research: spaCy for NLP, vLLM for serving large models (Qwen-3), HuggingFace for embeddings, OpenSearch for BM25 & SPLADE, and Qdrant for vector search. Each component is **adapter-based**, allowing interchangeable implementations through configuration.

Key benefits of this design include: **domain-aware chunking** (e.g. respecting IMRaD sections in papers [3]), **GPU-accelerated embeddings** (large models served via vLLM [4]), **hybrid indexing** (dense vectors in Qdrant, sparse signals in OpenSearch), and **fusion ranking** for robust retrieval [5]. All modules are loosely coupled following Ports & Adapters, ensuring easy extensibility and compatibility with RAG pipelines (retrieved chunks include source metadata for LLM consumption).

Project Structure (Ports & Adapters)

The project is organized into a **clean architecture** layout separating domain logic from infrastructure. A typical structure might be:

```
project_root/
├── src/
│   ├── core/                    # Domain models and interfaces (ports)
│   │   ├── models.py            # Pydantic models: Document, Chunk, Embedding,
etc.
│   │   ├── interfaces/          # Abstract base classes for adapters
│   │   │   ├── chunker.py       # BaseChunker interface
│   │   │   ├── embedder.py      # BaseEmbedder (for dense & sparse)
│   │   │   ├── indexer.py       # BaseIndex (for vector DB)
│   │   │   ├── searcher.py      # BaseSearch (for OpenSearch queries)
│   │   │   └── reranker.py      # BaseReranker interface
│   │   └── services/            # Use-case orchestration (application layer)
│   │       ├── ingestion_service.py   # Coordinates chunk -> embed -> index
│   │       └── retrieval_service.py   # Executes multi-strategy search & fusion
│   ├── adapters/               # Infrastructure implementations (adapters)
```

```
│   │   ├── chunkers/            # Various Chunker implementations
│   │   │   ├── imrad_chunker.py      # IMRaD/semantic chunking logic
│   │   │   ├── section_chunker.py    # Section-aware chunker
│   │   │   └── sliding_window_chunker.py  # Sliding window chunker
│   │   ├── embedders/
│   │   │   ├── vllm_embedder.py       # Qwen-3 8B via vLLM (dense)
│   │   │   ├── hf_embedder.py         # HuggingFace embedder (e.g. E5, BGE)
│   │   │   └── splade_embedder.py     # SPLADE-v3 expansion (sparse)
│   │   ├── indexers/
│   │   │   ├── qdrant_index.py        # Qdrant client for vector storage
│   │   │   ├── opensearch_index.py    # OpenSearch client for indexing chunks
│   │   │   └── pgvector_index.py      # (Optional) Postgres/pgvector adapter
│   │   ├── searchers/
│   │   │   ├── opensearch_searcher.py # Executes BM25 & SPLADE queries via
OpenSearch
│   │   │   └── qdrant_searcher.py     # (Optional) if using Qdrant for KNN
queries
│   │   └── rerankers/
│   │       ├── bge_reranker.py        # Cross-encoder reranker using BGE or
similar
│   │       └── qwen_reranker.py       # Reranker using Qwen (if available)
│   ├── api/                     # FastAPI entrypoints (controllers)
│   │   ├── main.py                  # FastAPI app initialization
│   │   └── routes/
│   │       ├── ingest.py            # POST /documents ingestion route
│   │       └── query.py             # GET/POST /search query route
│   └── config/
│       └── retrieval.yaml           # Pipeline configuration (chunker, models,
etc.)
└── tests/
    ├── test_chunking.py         # Unit tests for chunkers
    ├── test_embedding.py        # Unit tests for embedders (using small inputs)
    ├── test_indexing.py         # Tests for indexing (e.g., ensuring data is
stored/retrievable)
    ├── test_retrieval.py        # Tests for multi-strategy search & fusion
    └── test_api.py              # Tests for FastAPI routes responses
```

Ports & Adapters: The core defines abstract interfaces (ports) for each component (chunking, embedding, indexing, searching, reranking). Implementations in the `adapters/` layer conform to these interfaces. This decoupling allows swapping components via config (e.g. use `SlidingWindowChunker` instead of `IMRadSemanticChunker` by changing one setting). The service layer orchestrates use-cases without depending on concrete classes – it uses dependency injection to bind the chosen adapters at runtime. This aligns with the plug-in adapter architecture described in the system design [6] [7], ensuring new implementations can be added without modifying core logic.

Core Data Models (Pydantic)

All data entities are defined as **Pydantic v2 models with strict validation** [8]. Key models include:

- **Document**: Represents a source document (e.g. an article or trial report). Fields: `id` (str), `title` (optional str), `content` (full text), `metadata` (dict for any source-specific info, e.g. publication, authors, etc.), and possibly a list of `chunks`. This corresponds to the IR Document in the system, preserving high-level info before chunking.

- **Chunk** (or **Block**): A semantic chunk of a document, designed for indexing. Fields: `id` (str, unique chunk ID), `doc_id` (reference to parent Document), `text` (chunk content), `section` (optional section title or type, e.g. "Introduction", "Methods"), `order` (int position in document) and any additional `meta` (e.g. facet tags, offsets) [9]. Each Chunk is a self-contained unit suitable for retrieval. For example, a Block model might look like:

```
class Chunk(BaseModel):
    id: str
    doc_id: str
    section: Optional[str]        # e.g. "Results" or "Eligibility Criteria"
    text: str
    order: int
    metadata: Dict[str, Any] = {}
```

This mirrors the **Block** concept (semantic chunk) used in the design docs [9]. Chunks are created such that they respect logical boundaries (paragraphs, sections) and preserve references (like section titles or figure captions).

- **Embedding**: Represents an embedding vector or features for a chunk. We define separate types for dense and sparse embeddings:
- `DenseEmbedding`: fields for `model` (which model used, e.g. "Qwen-3-Embedding-8B"), `vector` (List[float]), and `dim`. This may be embedded in the Chunk model or handled separately when indexing.
- `SparseEmbedding`: for SPLADE expansion, fields like `features: Dict[str, float]` mapping terms to weights, and `model` (e.g. "SPLADE-v3").
- Optionally, a combined `EmbeddingResult` model can hold both dense and sparse representations for a chunk (or multiple vectors if multi-vector models). For example, an `EmbedResult` could contain a chunk_id plus a sparse vector and dense vector [10]. In practice, we often handle dense and sparse embeddings separately since they go to different backends.

Pydantic is used to validate lengths (e.g., dense vector length matches expected dimension) and types. These models also facilitate conversion to JSON (for OpenSearch indexing or API responses). The **federated data model** approach allows extension for domain-specific fields (e.g. medical metadata) while keeping core fields consistent [11] [12].

Document Chunking Module

Chunking is handled by interchangeable **Chunker** adapters implementing a common interface (e.g. `BaseChunker.chunk(document) -> List[Chunk]`). The system supports multiple strategies to adapt to different document types:

- **IMRaD Semantic Chunker:** Intended for research papers and similar content with well-defined sections. It uses document structure (headings like Introduction, Methods, Results, Discussion) to split the text. Using spaCy or regex, it can detect section headings and create chunks per section or subsection. Within each section, further splitting may occur based on paragraph boundaries or semantic units. It ensures chunks respect clinical or logical boundaries like those in IMRaD, trial outcome measures, or regulatory documents [3]. This chunker can also leverage **LLM-based coherence**: for example, it may embed sentences using the Qwen-3 model to measure similarity, ensuring each chunk contains topically cohesive sentences (splitting where similarity drops). The result is that each chunk is a semantically self-contained unit (e.g. one outcome description or one subsection of results).

- **Section-Aware Rule-based Chunker:** A simpler chunker that uses rule-based heuristics and NLP. It might split by explicit sections (if the document has structured sections like headings or XML/HTML tags), or by paragraphs and bullet points. For instance, it can use spaCy to segment text into sentences and paragraphs, then group them under section labels present in the metadata (e.g. if `document.metadata['section_titles']` exists). This ensures chunks align with natural document sections (e.g. splitting a drug label by sections like Indications, Dosage, etc., using known markers). The rules can be configured per source type. This approach is fast and doesn't require a model, making it useful for structured texts.

- **Sliding Window Chunker:** A fallback strategy that chops text into fixed-size windows (e.g. 300 tokens) with overlap (e.g. 50 tokens) to preserve context. This is used for very large or unstructured texts or as a safety net when logical segmentation is hard. The window size is chosen based on the embedding model's max input length and desired chunk granularity (e.g. ensuring < 512 tokens per chunk for Qwen embeddings). Overlap prevents information near boundaries from being lost. This chunker is domain-agnostic and ensures no chunk exceeds token limits. It may be triggered for extremely long sections (splitting them) or as a default for unknown document types.

All chunkers output a list of `Chunk` Pydantic objects. Each chunk knows its source document ID and (if applicable) section context. The chunking process may also produce **facet summaries per chunk** (e.g. a brief summary of the chunk's content such as PICO elements for a clinical trial paragraph) which can be stored in chunk metadata [3]. These summaries, possibly generated via an LLM, can later assist in retrieval filtering or in providing quick previews.

By configuring which chunker to use in `retrieval.yaml` (e.g. `chunker: "imrad_semantic"`), the system can easily switch strategies. New chunkers (e.g. a PDF-specific layout chunker) can be added by implementing the BaseChunker interface and referenced in config, without changing the core pipeline.

Embedding Module (Dense & Sparse)

After chunking, each chunk's text is passed to the **Embedding module**. This module produces both dense and sparse embeddings using pluggable models:

- **Dense Embeddings:** We default to using **Qwen-3-Embedding-8B** (a large text embedding model) served via **vLLM** for performance [4]. The embedder adapter (`vllm_embedder.py`) communicates with the vLLM server (e.g., via an HTTP or gRPC endpoint) to get embeddings. Batches of chunk texts can be sent for efficiency. The Qwen model returns a 768-dimensional vector (or 1536, depending on the model configuration) per chunk [10]. These vectors are typically L2-normalized or suitable for cosine similarity. The embedder wraps the service call and outputs a `DenseEmbedding` object for each chunk (model name, vector, etc.).

Alternative dense embedders can be configured: e.g. a HuggingFace-based embedder that loads a model like **E5-large** or **BGE (Beijing Embeddings)** locally via Transformers. The `hf_embedder.py` can use SentenceTransformer or `transformers` pipeline to generate embeddings on CPU/GPU. This is useful for offline or smaller-scale setups. The system could even support multi-vector models or the **FlagEmbedding** library if needed, by swapping in an adapter.

All dense embedding generation is assumed to be **GPU-accelerated**. In alignment with the project's fail-fast GPU philosophy, the embedder will check for GPU availability and throw an error if none is present (no silent CPU fallback) [13]. This ensures that we either have the hardware to produce high-quality embeddings or we fail early with a clear message.

- **Sparse Embeddings (Expansion):** For sparse document-side expansion, we integrate **SPLADE-v3** (from NAVER) [4]. SPLADE generates a sparse vector where each dimension corresponds to a vocabulary term, and the values are term importance weights. We use a SPLADE model (e.g. via HuggingFace Transformers) in the `splade_embedder.py` adapter. Given a chunk's text, it outputs a dictionary of term->weight (only for terms present, so it's a sparse map). For example, "patients" might get weight 2.3 if mentioned prominently. The dimensionality of the full vector space is large (tens of thousands of terms), but each chunk's representation is **very sparse** (only significant terms carry weight) [10]. We do **thresholding** (e.g., ignore very low weights to compress the representation) [14]. The result is stored as a `SparseEmbedding` (e.g. `{"splade_features": {"term1": 1.2, "term2": 3.4, ...}}`).

At query time, we also embed the **query text** with SPLADE (using the same model) to get a sparse query vector [15]. This is used for retrieval against documents expanded features.

- **Integration:** The embedding service can be run in parallel for multiple chunks. In a production deployment, a **GPU microservice** might handle embedding requests in bulk (taking chunk texts and returning both dense and sparse vectors) [16]. Our design mirrors this by possibly batching calls: e.g., `Embedder.embed_chunks(list_of_chunks) -> {chunk_id: (dense_vec, sparse_features)}`. This corresponds to an `EmbedChunks` RPC where each result contains a sparse and dense vector [17]. We keep the dense and sparse embeddings separate for indexing: dense vectors go to the vector DB, sparse features go to OpenSearch as additional index fields.

The embedding module is fully configurable: to use a different dense model, one changes the `embedder.model` in config; to disable SPLADE, one could set `use_splade: false` so that only dense embedding is done (or vice versa for a lexical-only pipeline).

Indexing Module (Vector Store & OpenSearch)

Once chunks are embedded, the pipeline **indexes** them into two systems for hybrid search:

- **OpenSearch Index (Lexical + SPLADE):** We create an OpenSearch index (e.g. index name `chunks`) to store chunk texts for traditional BM25 and their sparse features for learned sparse retrieval. The index uses a JSON mapping that includes both normal text fields and a `rank_features` field for SPLADE weights. For example, the mapping (in `config/chunks_index.json`) might define:

```
{
  "settings": {
    "index": {
      "analysis": {...},                // appropriate analyzer, e.g. standard
or English
      "similarity": {"default": {"type": "BM25"}}
    }
  },
  "mappings": {
    "properties": {
      "doc_id":    { "type": "keyword" },
      "chunk_id":  { "type": "keyword" },
      "tenant_id": { "type": "keyword" },        # to support multi-tenant
filtering
      "section":   { "type": "keyword" },        # e.g., "Methods", for
optional filtering or boosting
      "content":   { "type": "text", "analyzer": "english" },
      "title":     { "type": "text", "analyzer": "english" },  # if chunk
carries doc title or section title
      "splade_features": { "type": "rank_features" }           # stores sparse
weights per term
    }
  }
}
```

Here, **BM25F** functionality can be achieved by indexing multiple text fields (e.g. if we treat `title` or `section` with higher weight). At query time, a multi-field query can boost certain fields. For instance, the search API might use a `multi_match` on `title^2` and `content^1` for BM25, simulating BM25F (field-aware scoring). In our design, each chunk mainly has `content`, but if needed we can incorporate document-level fields (like title) to weight them more [18].

The `splade_features` field (type `rank_features`) holds a sparse map of terms to weights for each chunk. This allows OpenSearch's `rank_feature` **query** or a custom script score to match a SPLADE-

embedded query to documents. Concretely, when a query comes in, we generate its SPLADE term-weight map, then use the OpenSearch `knn` or `neural_sparse` query (if supported) to retrieve by dot-product between query sparse vector and the doc's `rank_features` [19]. Alternatively, we can convert the query's sparse vector to a boolean query with weights (e.g. using the `rank_feature` query type for each term). The OpenSearch adapter will handle constructing this query JSON.

We also use OpenSearch for pure lexical BM25. That simply relies on the `content` field (with standard analyzer). The index is configured to use BM25 similarity, and we use normal `match` or `multi_match` queries [20]. The combination of BM25 and SPLADE in one index enables **hybrid scoring**. (In some setups, we might use separate indices, but here a single index can suffice, or we query the same index with different query clauses.)

BM25F: If we want to leverage BM25F (field weighting) more explicitly, we could store chunks under multiple fields (e.g., if a chunk has structured subfields, like `heading` text vs `body` text). The mapping above treats `title` separately which could be used as a minor signal. Alternatively, if we index whole documents rather than chunks, BM25F would weight different document fields. However, since we are chunk-centric, BM25F usage is limited to perhaps boosting certain chunk types or metadata fields. In any case, the OpenSearch mapping is designed to support both **BM25 and SPLADE** retrieval.

- **Qdrant Vector Collection:** For dense vectors, we use **Qdrant** as the vector store. We create a collection (e.g. named `"chunks_vectors"`) using the Qdrant client. The schema definition (either via client code or a YAML) specifies:
- **Vector size**: e.g. 768 (matching Qwen-3 embedding dim).
- **Distance metric**: cosine (for normalized embeddings) or dot product.
- **HNSW parameters**: M (graph fanout, e.g. 16 or 32) and ef (search ef_search and build ef_construct) for approximate NN. This is to enable efficient HNSW indexing.
- **Quantization**: optional. Qdrant supports product or scalar quantization to compress vectors [21]. We can include a `quantization_config` (e.g. using int8 scalar quantization with 0.8 quantile to preserve high-magnitude components [22]) to reduce memory usage. This can be toggled via config (e.g. `qdrant.quantize: true/false`). For an 8B model producing high-dim embeddings, quantization can be beneficial.
- **Payload (metadata)**: We store identifying and filtering information as Qdrant payload. Each vector point will carry at least `doc_id` and `chunk_id` (so we know which document/chunk it came from), and possibly `tenant_id` for multi-tenant isolation, plus any other metadata (section, etc.). Qdrant supports **payload filtering**, so at query time we can add filters (like `tenant_id == X`) to restrict search results, aligning with multi-tenant support [23].

Example using the Python client:

```
qdrant_client.create_collection(
    collection_name="chunks_vectors",
    vectors_config=VectorParams(size=768, distance=Distance.COSINE,
                                hnsw_config=HnswConfig(m=16, ef_construct=100)),
    optimizers_config=OptimizersConfigDiff(indexing_threshold=20000),
    quantization_config=ScalarQuantizationConfig(type="int8", quantile=0.8,
always_ram=True),
```

```
    payload_schema={ "doc_id": "keyword", "tenant_id": "keyword", "section":
"keyword" }
)
```

This would create a collection with the desired config. (In Qdrant, payload schema is often inferred, but here we illustrate for clarity.)

When indexing a chunk, the pipeline will do two things: index the chunk's text & sparse features to OpenSearch, and upsert the chunk's dense vector to Qdrant (with the same IDs). We ensure consistency by using the same `chunk_id` as the primary key in both systems, so data can be correlated.

Both OpenSearch and Qdrant instances would likely run in Docker (OpenSearch requires Java and Qdrant is Rust-based) – our design expects these to be available services. We can manage them via docker-compose for local dev, and use the Python clients (`opensearch-py` and `qdrant-client`) in our adapters.

To support alternatives: The design can switch to **OpenSearch KNN** for vectors if Qdrant is not desired. In that case, the OpenSearch mapping would include a `dense_vector` field (with `dims: 768`) and we'd use OS's k-NN query for vectors. Similarly, an adapter for **Postgres pgvector** could be used (storing vectors in a PG table). These swaps are configured in `retrieval.yaml` (e.g. `vector_backend: "qdrant"` or `"opensearch"` or `"pgvector"`). The indexing service would route the vector either to Qdrant or to the chosen backend's adapter accordingly.

Finally, we enable **snapshotting** and backup strategies for the indexes. Qdrant supports snapshot creation via API [24], which we could automate (e.g. periodic snapshot of the collection stored to disk for backup or for moving to production). OpenSearch indices can similarly be snapshotted via snapshot API. Our design would include maintenance scripts or API calls for these as needed (possibly an admin route or a background job for snapshotting).

Retrieval & Fusion Mechanism

For querying, we implement a **RetrievalService** (in `core/services/retrieval_service.py`) that orchestrates the hybrid search. The retrieval flow is:

1. **Query Processing:** When a query comes in (via API), the service first decides which strategies to use based on config or query type. For example, default might be all three: BM25, SPLADE, and dense. The query text is optionally preprocessed (e.g., lowercased or passed through the same analyzer as indexing if needed).

2. **Parallel Search:** The service triggers searches in parallel for each selected strategy [25]:

3. **BM25 search:** Uses the OpenSearch adapter to run a full-text search on the `content` field. This uses a query DSL with match or multi_match. We include a filter on tenant_id if multi-tenancy is in use [18]. The OpenSearch client returns the top N hits with scores. We convert those to our internal `ScoredDocument` objects (with doc_id, score, content snippet, etc.).
4. **SPLADE search:** We embed the query with the SPLADE model to get sparse features [15]. Then the OpenSearch adapter executes a **neural sparse query** using these features. This could be done via a

script score that multiplies query term weights with document `splade_features` [19]. OpenSearch (particularly if running on AWS's flavor) also has a `knn` query for sparse vectors. The result is a list of hits (doc scores based on sparse match). Those are converted to `ScoredDocument` objects (with a field `strategy="splade"`).
5. **Dense vector search:** The query is also embedded by the dense model (e.g., Qwen, via the same embedder but possibly using a smaller query model or same embedding model). The vector is sent to Qdrant (or chosen vector DB) for ANN search. The Qdrant adapter performs a `search` on the "chunks_vectors" collection with the query vector and any filter (e.g., tenant filter). This returns the closest chunk vectors with similarity scores. These are also wrapped as `ScoredDocument` (we retrieve the chunk content or reference via stored payload). If using Qdrant, we might store the chunk text as part of payload for convenience; otherwise, we can retrieve it from OpenSearch by ID if needed to present results.

These searches happen concurrently (e.g., using `asyncio.gather` for async calls) [25], so the latency is bounded by the slowest of the three, not sum of all.

1. **Fusion of Results:** We then combine the results from all strategies into a single ranked list. We implement two fusion modes:
2. **Reciprocal Rank Fusion (RRF):** This is the default, as it's simple and effective [26]. For each strategy's result list, we assign a score like `1/(k + 60)` (if using k=60 as in code) for a result at rank k [27]. We sum these scores for each document across lists. This produces a fused ranking that gives weight to items that appear in multiple lists. The top fused results (by summed score) are taken.
3. **Weighted Fusion:** Alternatively, the config can specify weights for each strategy, e.g. 0.3 for BM25, 0.3 for SPLADE, 0.4 for dense (weights summing to 1) [28]. In this mode, we normalize scores within each list (e.g., divide by max score or use some scaling), then compute a weighted sum: `score = w_bm25 * bm25_score + w_splade * splade_score + w_dense * dense_score`. This requires scores to be comparable; we might simply use rank-based scores or raw scores if they're on similar scales after normalization. Weighted fusion allows tuning (e.g., more weight to dense if we trust semantic matching more).

The fusion yields a single list of documents/chunks with combined scores. The service will typically take the top *K* fused results for the next step.

1. **Reranking (optional):** If enabled, we take the top *K* fused results (or top *3K* as candidates, to allow reshuffling) and rerank them using a stronger model. For example, we can use a **BGE reranker** (a cross-encoder that takes query and each result text and produces a relevance score) or a smaller Qwen model fine-tuned for reranking. The `reranker` adapter will likely call a model like `BAAI/bge-large-en` in a cross-attention fashion (this could be done via Transformers or an inference service). Reranking considers the query and full chunk text to produce a new score, which often improves the final ordering by fine-grained relevance (at the cost of extra latency). Our service then selects the top K results after reranking [29].

If reranking is off (for faster responses), we just use the fused order.

1. **Result Packaging:** The final results are packaged into a `RetrievalResult` model which includes the query, a list of result chunks with their content, scores, source metadata (like the original document ID, title, etc.), and info about which strategies were used and whether rerank was applied [30]. Each result item might look like: `{chunk_id, doc_id, content, score, strategy:`

`"fusion", source_scores: {bm25: ..., dense: ...}, metadata: {...}}`. We ensure that each result has the necessary provenance (e.g., document title or at least an ID that can be used to fetch the full document or reference), as **provenance-first design** is a key goal [31] [32].

This retrieval pipeline is designed to be **fully hybrid and configurable**. One could enable only dense search (for pure semantic search) or only BM25, by adjusting the `strategies` list in config or query parameters. The default uses all, achieving the multi-strategy hybrid retrieval noted in the project specs [31]. The approach ensures high recall (by combining complementary methods) and improved precision (via fusion and reranking). It also readily supports RAG pipelines: since results are returned as discrete, source-attributed chunks, they can be fed into an LLM for answer generation with proper citations.

FastAPI API Stubs

We expose a simple REST API (and potentially GraphQL/gRPC, though REST is primary) to interact with this pipeline. Using FastAPI, we define routes (controllers) that call into the service layer. Key endpoints include:

- **POST** `/documents` **(Ingest Document):** Accepts a document payload (e.g. JSON with `id`, `title`, `text`, and optional metadata). This route handler will construct a `Document` model and call the ingestion service:
- Chunk the document (`chunker.chunk(doc)`).
- Embed the chunks (`embedder.embed_chunks(chunks)` to get dense and sparse embeddings).
- Index the chunks (`indexer.index_chunks(chunks, embeddings)` which internally calls both OpenSearch and Qdrant adapters).

It returns an acknowledgment with the document ID or number of chunks indexed. This could be an async background task if needed (for large docs, we might want to not block the request). In our stub, we simply illustrate the synchronous pipeline call.

- **POST** `/embed` **(optional utility):** This could allow a client to get embeddings for an arbitrary text or list of texts. It's mainly for testing or for embedding queries on the client side. It would call the embedder and return the vector(s). This is not essential for core functionality (since we embed internally), but can be useful.

- **POST** `/index` **(optional separate step):** If we separate embedding and indexing steps (perhaps for pre-computed embeddings), we could have an endpoint to directly index given embeddings for a chunk. In practice, we integrate embedding in the ingest flow, so this may not be needed.

- **GET or POST** `/search` **(Query):** Allows clients to retrieve information. The request may include a query string and optional parameters like `strategies` (to override default search strategies), `top_k`, and filters (e.g. `{ "section": "Results", "tenant_id": "abc" }`). The handler calls `RetrievalService.retrieve(query, strategies, top_k, filters, tenant_id)` and returns the results. The response contains the list of top results with their text and metadata. For example, a JSON response might look like:

```
{
  "query": "What were the primary outcomes in the trial?",
```

```
  "results": [
    {
      "doc_id": "NCT12345",
      "chunk_id": "NCT12345:3",
      "text": "The primary outcome was a significant reduction in HbA1c at
6 months ...",
      "score": 12.34,
      "highlights": null,
      "source": {"title": "Trial XYZ Results", "section": "Results"},
      "strategies_used": ["bm25","dense","splade"]
    },
    ...
  ],
  "fusion_method": "rrf",
  "reranked": true,
  "total_results": 57
}
```

(The exact format can be refined, possibly aligning to JSON:API or OpenAPI specs as per project standards.)

- **Misc endpoints:** We might stub out health checks (e.g. GET `/health` returns OK if dependent services are up), or debugging endpoints (e.g. GET `/documents/{id}` to fetch a document or its chunks from the index for verification). Also, if GraphQL or gRPC are planned, analogous operations would be defined (the architecture document mentions multi-protocol support, but for brevity we focus on REST here).

The FastAPI controllers themselves contain minimal logic – they validate input (FastAPI + Pydantic schemas), call the appropriate service method, and return the result. This ensures the business logic resides in the service layer and adapters. Authentication (OAuth2/JWT) and multi-tenant context extraction (from claims) can be integrated in a middleware if required, ensuring each request is handled in the context of the tenant (e.g. we might extract `tenant_id` from token and pass it into the service calls to apply filtering).

Configuration System

A central configuration (e.g. `retrieval.yaml`) is used to make the pipeline **extensible and tunable without code changes**. This YAML/JSON config can define:

- **Active Components:** Which implementations to use for each interface. E.g.:

```
chunker: imrad_semantic    # or section_aware, sliding_window
dense_embedder: vllm_qwen3  # or hf_e5large, etc.
sparse_embedder: splade_v3  # could toggle off by setting none
vector_index: qdrant        # or opensearch, pgvector
lexical_index: opensearch
```

```
reranker: bge_large          # or qwen_rerank, or none
fusion_method: rrf           # or weighted
```

The code will load this config and use a factory or registry to instantiate the specified classes (e.g. a mapping from `"imrad_semantic"` to the `ImradSemanticChunker` class). We may use entry points or a simple if/elif registry in a factory module.

- **Model and API Parameters:** Under each component, we include specific settings. For example:

```
models:
  qwen3:
    host: "http://vllm-server:8000"    # endpoint for embedding service
    embedding_dim: 768
  splade:
    model_name: "naver/splade_v3"
    min_weight: 0.01                   # weight threshold for features
  bge:
    model: "BAAI/bge-large-en"         # reranker model name
opensearch:
  index_name: "chunks"
  host: "http://localhost:9200"
  user: "admin"
  password: "admin"
qdrant:
  host: "http://localhost:6333"
  collection: "chunks_vectors"
  search_params:
    top: 100
    ef_search: 50
retrieval:
  default_strategies: ["bm25","dense","splade"]
  top_k: 10
  fusion_method: "rrf"
  rerank: true
  weights: [0.3, 0.3, 0.4]   # if fusion_method = weighted
```

This structure is just illustrative. The idea is to externalize details like index names, model endpoints, and tweakable knobs (like number of candidates to consider for reranking, or SPLADE weight cutoff). The application reads this config on startup and configures the services accordingly.

- **Thresholds and Misc:** We could include things like the RRF `k` value (e.g., 60 as in code), chunk size limits, etc., in config as well.

By having a config-driven setup, non-developers or ops engineers can adjust the retrieval behavior (e.g. disable SPLADE if it's too slow, or switch embedding model) without code changes. This also makes the

system adaptable to future models or indexes: e.g., integrating a new vector DB would involve writing an adapter and pointing the config to it.

The config can be a single YAML or split into sections (like separate config for Qdrant, OpenSearch, etc., included in a main config). Pydantic can be used to load and validate config schemas, ensuring required fields are present.

Testing Strategy (Unit Test Stubs)

We will create comprehensive tests to ensure each module functions correctly and the pipeline meets quality targets (the project aims for 80%+ coverage and strict CI tests [33]). Some **unit test stubs** to include:

- **Chunker Tests:** Provide sample texts (e.g., a mock research abstract with known sections, a long paragraph for sliding window) and verify the Chunker outputs. For IMRaD chunker, ensure that sections are correctly split and labeled (e.g., the text "Introduction … Methods …" yields two chunks with section "Introduction" and "Methods"). For sliding window, ensure the chunks are of correct size and overlapping as expected. Edge cases like very short documents (should produce one chunk) and very long ones (should chunk into multiple) are tested.

- **Embedding Tests:** Using a small dummy model or monkeypatch the embedder to avoid actual large model calls (e.g., stub the vLLM response with a fixed vector). Ensure that the embedder returns vectors of expected length and that the sparse embedder returns a reasonable feature dict for a known input (could use a tiny vocabulary SPLADE variant or just stub). Also test that GPU-required embedder raises an error or skip if GPU not available (simulate `torch.cuda.is_available()` false to see it fails fast, as design specifies [13]).

- **Indexing Tests:** Here we can spin up test instances or use Docker containers for OpenSearch and Qdrant in a CI environment. Alternatively, use mocks for the OpenSearch client. For OpenSearch: test that indexing a chunk calls the client with the correct index name and document body (verify that `content` and `splade_features` are present). For Qdrant: test that the `upsert` is called with the correct vector and payload. If using actual Qdrant (maybe via the `qdrant-client` in-memory mode), insert a vector and query it to ensure it's stored. We should also verify that our mappings are properly created (perhaps by retrieving the mapping via OpenSearch API in tests).

- **Retrieval Logic Tests:** We can simulate the retrieval without external services by mocking the search adapters. For instance, stub the OpenSearch searcher to return predetermined results for BM25 and SPLADE (e.g., if query = "X", return chunk1, chunk2 with scores). Similarly stub the Qdrant searcher. Then test that `RetrievalService.retrieve` correctly fuses them. For RRF, we can craft scenarios where each strategy returns different items and verify the fused result contains all with appropriate ordering. For weighted fusion, test with known scores and weights. Also test the reranker integration by stubbing a reranker to shuffle results (e.g., reverse order) and see that the final output is reranked. Ensuring that `RetrievalResult` is populated correctly (strategies_used, etc.) is also important.

- **API Tests:** Using FastAPI's test client, simulate requests. E.g., POST `/documents` with a small JSON and verify a 200 response and that it triggers the pipeline (here we might stub out internals to avoid

heavy computations). For `/search`, simulate a query and ensure the response contains expected fields. We should test error conditions too (like invalid input or missing fields leading to 422 responses from FastAPI/Pydantic validation).

Given the importance of performance (target P95 < 500ms for retrieval) [34], we might also include performance tests or at least notations for them (though those are more integration tests). In our unit tests, we can ensure that fusion and reranking algorithms perform within expected time for, say, dozens of results.

By scaffolding these tests early (even as simple asserts or using pytest with `@pytest.mark.asyncio` for async methods), we embed quality checks into the development. Continuous integration will run these tests, and we can add more as we implement details. This approach aligns with the project's testing strategy (unit tests for 80% coverage, plus integration tests in Docker Compose, etc.) [33].

By following this design, we achieve a **modular, extensible retrieval pipeline**. Each piece (chunker, embedder, index, search) can evolve independently – e.g., adopting a new embedding model or adding a new sparse encoder like TILDE can be done by writing a new adapter and updating config. The architecture balances **open-source practicality** (leveraging robust libraries like spaCy, Transformers, Qdrant, OpenSearch) with state-of-the-art techniques (hybrid retrieval, SPLADE expansion, LLM-based chunking). All components are built with inter-operability in mind, ensuring the system meets the advanced retrieval goals outlined in the project scope (high recall hybrid search, clinical relevance, provenance tracking) [3] [31]. This scaffold sets the stage for implementation and future extension while adhering to the system's design rationale.

Sources:

- System Architecture & Design – *Hybrid Retrieval Approach (BM25 + SPLADE + dense)* [31] [5]; *Adapter-based design* [6]
- Overall Project Scope – *Chunking (IMRaD, medical-aware) and Indexing/Retrieval (Qwen embeddings, BM25F, SPLADE, fusion)* [3]; *GPU enforcement for embeddings* [13]
- Project Comprehensive – *RetrievalService logic for multi-strategy search and RRF fusion with rerank* [2] [29]; *Pydantic models and validation* [8]; *OpenSearch BM25 query with tenant filter* [20]; *Embedding result structure (dense & sparse vectors)* [10]

---

[1] [5] [6] [7] [9] [10] [11] [12] [16] [17] [26] [31] [32] System Architecture & Design Rationale.md
file://file-2C2o79VDkNLLK4VfJ8omq8

[2] [8] [14] [15] [18] [20] [23] [25] [27] [28] [29] [30] [33] [34] project_comprehensive.md
file://file-R7qdkhAemxSLT3y1tiHHxE

[3] [4] [13] Overall_Project_Scope.md
file://file-PoGUR1BAARTsgKYizTu87y

[19] Neural sparse search using raw vectors - OpenSearch Documentation
https://docs.opensearch.org/latest/vector-search/ai-search/neural-sparse-with-raw-vectors/

21 22 24 Collections - Qdrant
https://qdrant.tech/documentation/concepts/collections/