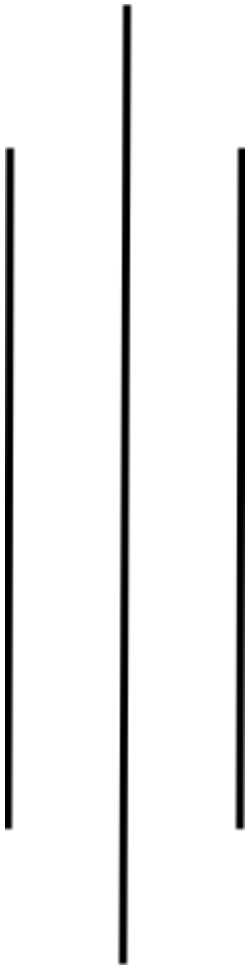
RAG Pipeline for Job Data Retrieval



Prepared By

Tilak Parajuli

Date: November 21, 2025

.

1. Introduction

1.1. Background: The Limitations of LLMs

By learning from enormous volumes of unstructured data from the internet, Large Language Models (LLMs) have revolutionized natural language processing. However, when applied to actual business problems, they encounter two significant limitations despite their generative capabilities:

1. **Knowledge Cutoff & Staleness**: Time is frozen in LLMs. It is impossible for an LLM trained in 2023 to be aware of a job posting made today. For instance, instead of retrieving the current reality (e.g., Sushila Karki), a base model might respond to the question "Who is the current Prime Minister of Nepal?" using out-of-date training data (e.g., Pushpa Kamal Dahal).
2. **Hallucination**: In order to satisfy the prompt, an LLM frequently produces information that sounds plausible but is factually incorrect when it lacks specific data.

1.2. The Solution: Retrieval-Augmented Generation (RAG)

To resolve these issues without retraining the model, Retrieval-Augmented Generation (RAG) is utilized. RAG links an external, dynamic knowledge base with the generative power of an LLM.

The system uses a retrieval workflow rather than depending only on internal model weights::

1. **Retrieve**: Based on the user's query, the system looks up pertinent information in a live database.
2. **Augment**: The context window of the LLM is filled with these facts.
3. **Generate**: Using just the facts that were retrieved, the LLM synthesizes an answer by acting as a reasoning engine.

1.3. Why RAG rather than fine-tuning?

Although fine-tuning modifies a model's actual neural weights to discover new patterns, it was judged inappropriate for a Job Retrieval System for the following reasons:

- **Data Dynamism**: Every day, job postings are updated. It is computationally impossible to fine-tune a model each time a job is added or removed.

- **Resource Cost**: Modern 7B+ parameter models require high-performance GPUs (like A100s) and substantial memory overhead for complete fine-tuning, which is prohibitive for agile applications.
- **Precision**: Using RAG to provide the exact source text ensures factual accuracy more successfully than fine-tuning, which enhances the style of responses.

1.4. System Overview

A Hybrid RAG Pipeline created especially for the LF Jobs dataset is implemented in this project. There are two main operational pipelines in it:

A. The Insertion Pipeline (Indexing)

Unstructured job descriptions must be processed for machine comprehension before retrieval is feasible:

1. **Chunking**: The process of dividing lengthy job descriptions into more manageable, semantically significant sections. This increases retrieval accuracy and guarantees that token limits are maintained. To avoid context loss at split boundaries, an overlap strategy (e.g., 10-20%) is employed.
2. **Embedding**: Using an Embedding Model, text segments are transformed into numerical vector representations (such as [0.1, -0.5, 2.4...]). This enables the computer to comprehend the text's semantic meaning rather than just its keywords.
3. **Vector Storage**: For quick similarity searches, these vectors and their metadata are indexed in a dedicated Vector Database (Qdrant).

B. The Retrieval Pipeline (Inference)

When a user searches for "Senior Python Engineer in New York":

1. The query is transformed into a vector.
2. To identify the most pertinent job chunks, the system uses a hybrid search that combines vector similarity and keyword matching.
3. To guarantee the highest relevance, these results are refined through a reranking step.
4. The LLM receives the best results and uses them to provide the candidate with a well-organized, beneficial response.

2. High-Level Architecture

Docker Compose is used to orchestrate the system, which is designed as a Containerized Microservice Ecosystem. To guarantee high availability and data integrity, it rigorously separates the Write Path (Data Ingestion/ETL) from the Read Path (Inference/RAG).

2.1. Architecture Diagram

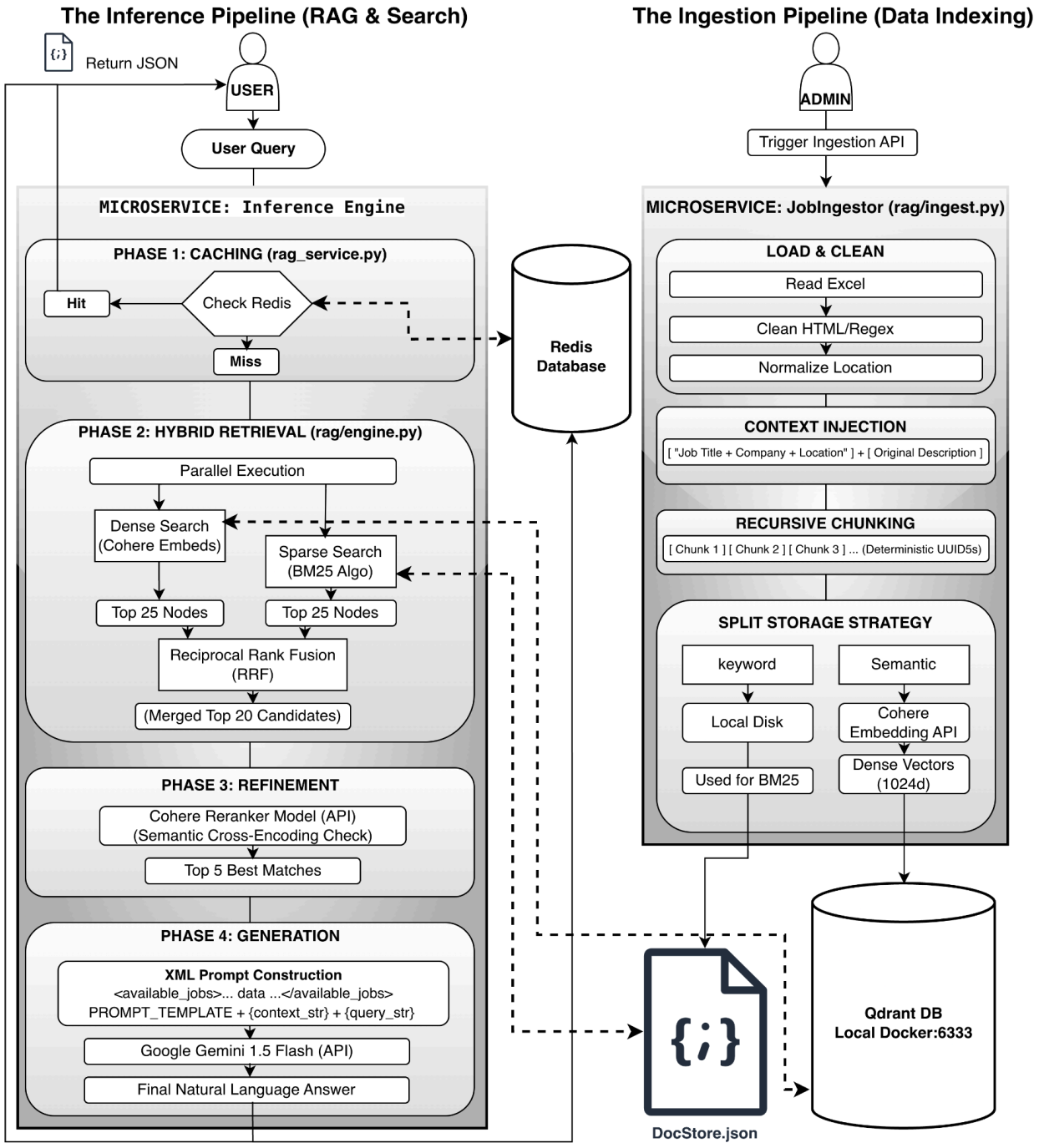


Figure 1: Detailed Data Flow and System Architecture

The two main workflows are depicted in the diagram above:

1. **Right Side (Ingestion Pipeline)**: The asynchronous process of converting unprocessed tabular data into searchable hybrid indices
2. **Left Side (Inference Pipeline)**: The real-time logic for handling user queries, caching results, and producing AI responses

2.2. Core System Components

The architecture is based on three fundamental layers that are intended to manage particular latency, accuracy, and scale constraints.

A. The Infrastructure Layer (Docker)

The stack operates fully inside a Docker network to guarantee reproducibility and reduce external dependencies:

- **API Microservice**: The central orchestrator is FastAPI running on Python 3.11.
- **Qdrant (Vector Database)**: It is hosted in a local container (qdrant:6333).
    - Engineering Decision: The network round-trip time (RTT) between the database and the API was eliminated by hosting Qdrant locally rather than utilizing a cloud SaaS. By lowering the retrieval time from approximately 2.5 seconds to less than 0.05 seconds, this was crucial in mitigating the high internet latency in the deployment region (Nepal).
- **Redis (Cache Layer)**: An in-memory key-value store (redis:6379) that provides <1 ms latency for repeated queries by caching final JSON responses.
- **Local Persistence** The docstore.json (Raw Text) is kept locally on a bind-mounted volume. Because of this, the BM25 algorithm can operate in memory without requiring each query to retrieve lengthy text payloads from the database.

B. The Ingestion Pipeline (Data Indexing)

This pipeline, found in rag/ingest.py, converts unprocessed Excel data into a retrieval-optimized format.

1. **Load & Clean**: To eliminate artifacts while maintaining structural components like bullet points, raw HTML descriptions are sanitized using regex.

2. **Context Injection**: Standard chunking frequently produces "orphaned" text, such as a chunk that lists requirements but omits the job title. To guarantee semantic completeness, a Metadata Header (Title + Company + Location) is inserted into each text chunk.
3. **Recursive Chunking**: To capture fine-grained details, data is divided using a sliding window technique (500 characters).
4. **Split Storage Strategy**:
    - **Semantic Path**: Cohere creates dense vectors, which are then saved in Qdrant.
    - **Keyword Path**: To create the BM25 Sparse Index, raw text is saved to the local disk.

C. The Inference Pipeline (RAG & Search)

This module, which is found in rag/engine.py, carries out the "Elite Recruiter" logic:

1. **Phase 1: Caching**: The normalized user query is hashed by the system, which then verifies Redis. The result is returned instantly in the event of a hit, avoiding costly API calls.
2. **Phase 2: Hybrid Retrieval**:
    - **Dense Search**: Uses Qdrant to find jobs that are conceptually similar (e.g., "Frontend" Ψ "React").
    - **Sparse Search**: Finds precise keyword matches (such as "Kathmandu" or "Visa Sponsorship") by querying the BM25 Index.
    - **Fusion**: Reciprocal Rank Fusion (RRF) is used to combine results, guaranteeing that the best results meet both semantic intent and particular keywords.
3. **Phase 3: Refinement**: The Cohere Rerank API, which functions as a cross-encoder to re-score documents based on deep relevance and filter down to the top 5, receives the top 20 candidates.
4. **Phase 4: Generation**: Google Gemini 1.5 Flash receives the final candidates after they have been formatted into a strict XML structure (<job>...</job>). Before producing the final answer, the model uses a "Chain of Thought" prompt to confirm constraints (Location/Level).

2.3. Project Directory Structure

The RAG Logic Layer and the Application Layer (FastAPI) are kept apart by the modular structure of the codebase. Future scalability and testing are made easier by this division of responsibilities.

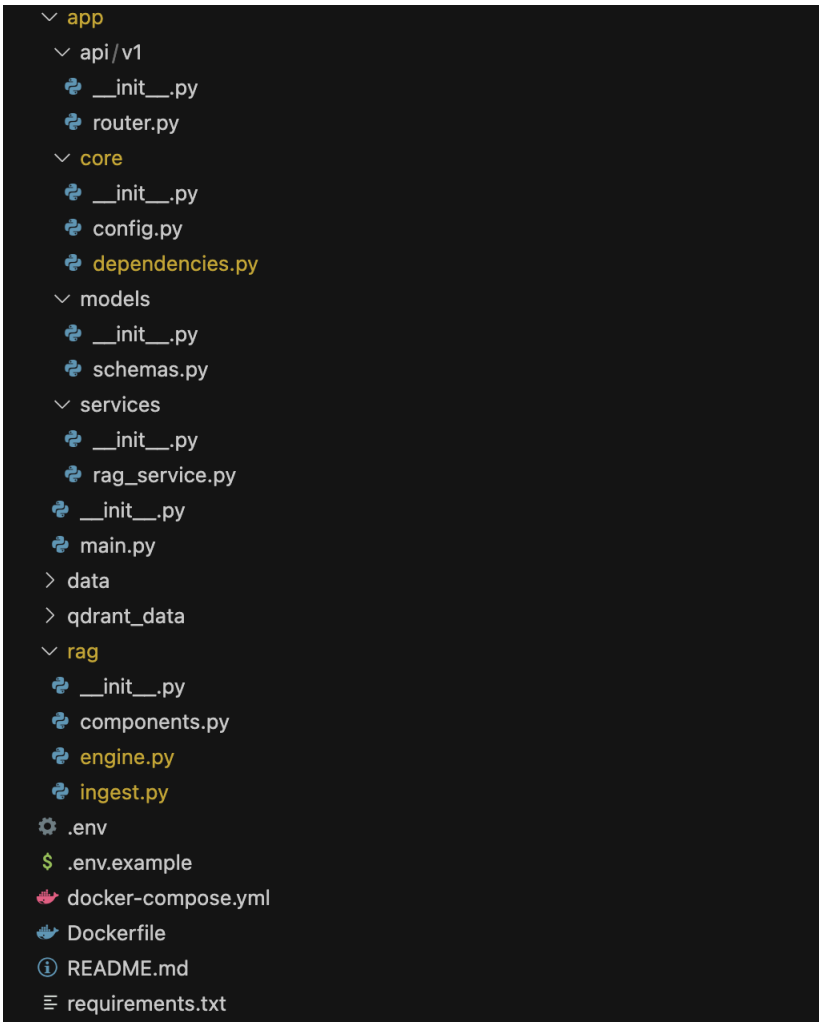


Figure 2: Project Directory Structure

Module Descriptions

File	Role & Responsibility
app/main.py	**The starting point**. sets up the Redis connection pool, controls the application lifecycle (startup/shutdown events), warms up the RAG engine before traffic starts, and configures global logging.

app/api/v1/router.py	**The entrance**. specifies the /query and /ingest HTTP endpoints. It handles request validation using Pydantic models and delegates heavy logic to background tasks or services.
app/core/config.py	**The configuration**. validates environment variables (API Keys, URLs) using pydantic-settings. If important credentials are missing, it guarantees that the application fails quickly at startup.
app/core/dependencies.py	**The wiring**. Dependency Injection (DI) is implemented. To guarantee effective resource connection management across requests, it offers singleton instances of Redis and RAGService.
app/models/schemas.py	**The Contract**. Specifies the Requests and Responses JSON data structures. For inputs (like query strings) and outputs (like citations lists), it guarantees exact typing.
app/services/rag_service.py	**The Manager**. Includes the business logic at a high level. In order to implement the "Read-Through Cache" pattern, it first checks Redis, then, in the event of a miss, calls the RAG Engine and caches the outcome.
rag/components.py	**The Factory**. Centralizes external integration initialization (Gemini, Cohere, Qdrant). It guarantees that models are set up uniformly everywhere (e.g., with the right batch sizes).
rag/engine.py	**The brain**. carries out the Hybrid RAG reasoning. It creates the XML prompt for Gemini, loads the BM25 index from disk, connects to Qdrant, and runs Cohere Reranking and Reciprocal Rank Fusion.

rag/ingest.py	**The Writer**. manages the pipeline for ETL. It creates deterministic IDs for idempotency, cleans HTML, adds metadata context to text chunks, and saves data to both Qdrant and Local Disk.

3. Setup and Installation

This system is intended to function as a standalone Docker environment. There is no need to install Python locally, guaranteeing uniform performance on various computers.

3.1. Prerequisites

Make sure the following are installed or accessible before continuing:

- Docker Desktop (Version 4.0+ recommended)
- Git (For cloning the repository)
- API Keys:
    - Google Gemini API Key (For LLM Generation)
    - Cohere API Key (For Embeddings and Reranking)

3.2. Installation Steps

Step 1: Clone the Repository

Download the source code to the local machine.

- ***git clone https://github.com/parajulitilak/RAG-Pipeline-for-Job-Data-Retrieval.git***
- ***cd RAG-Pipeline-for-Job-Data-Retrieval***

Step 2: Configuration (.env)

Create a .env file in the root directory. This file manages secrets and infrastructure settings.

Note: The system is pre-configured to look for Qdrant and Redis within the Docker network.

```
# --- API KEYS (Required) ---
GOOGLE_API_KEY=your_gemini_api_key_here
COHERE_API_KEY=your_cohere_api_key_here


# --- INFRASTRUCTURE (Docker Internal DNS) ---
# Do not change these if running via Docker Compose
QDRANT_URL=http://qdrant:6333
QDRANT_COLLECTION=lf_jobs_hybrid
REDIS_URL=redis://redis:6379
```

--- MODEL SETTINGS ---

LLM_MODEL=models/gemini-1.5-flash

EMBEDDING_MODEL=embed-english-v3.0

RERANK_MODEL=rerank-english-v3.0

Step 3: Data Preparation

Ensure the source dataset is placed correctly. The application expects the file at:

./data/LF Jobs.xlsx

If the data folder does not exist, create it and place the Excel file inside.

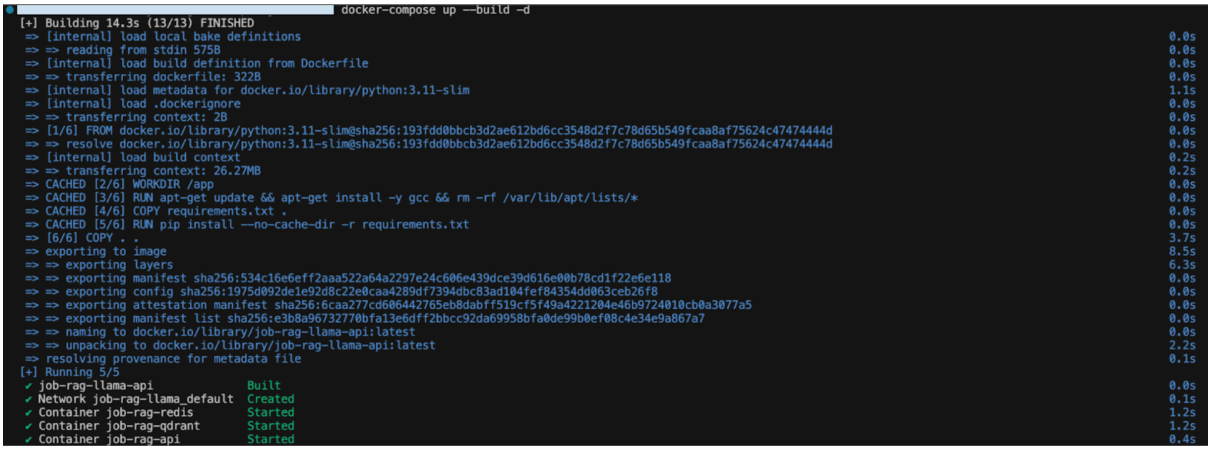
Step 4: Build and Launch

Use Docker Compose to build the Python image and start the microservices (API, Redis, Qdrant).

Build images and start containers in detached mode

docker-compose up --build -d

- API Service: Available at ***http://localhost:8000***
- Qdrant Dashboard: Available at ***http://localhost:6333/dashboard***



3.3. Data Ingestion (Mandatory One-Time Setup)

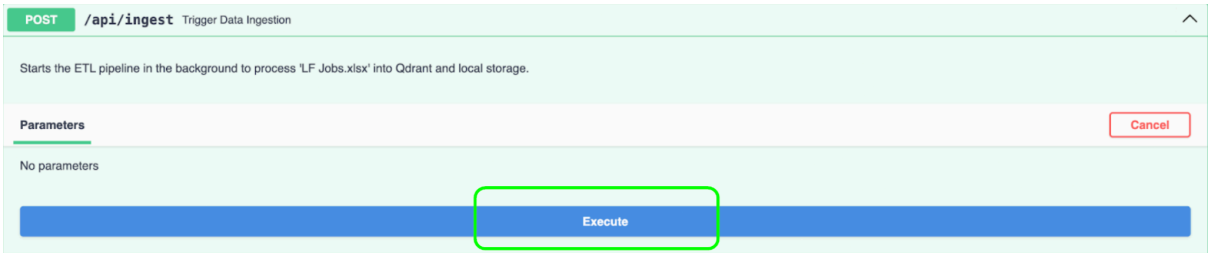
Upon first launch, the databases (Qdrant and DocStore) are empty. Ingestion pipeline must be triggered to build the indices.

Run the following command in your terminal:

curl -X POST "http://localhost:8000/api/ingest"

OR

Trigger API endpoint through Swagger UI.: ***http://0.0.0.0:8000/docs#/***



What happens during ingestion?

1. **Validation**: To avoid duplicate runs, the system locks the process.
2. **Processing**: It reads the Excel file, cleans HTML, and chunks text.
3. **Text Persistence (Fast)**: For the BM25 Keyword Search, it generates **docstore.json** locally. Within about 20 seconds, "DocStore persisted" will appear in the logs.
4. **Vector Upload (Slow)**: To adhere to Cohere's Trial API rate limitations, vectors are uploaded to Qdrant in batches of 48, with a 12-second pause in between batches.
    - **Estimated Time**: About **80 minutes**.
    - **Note**: During this period, the API is still responsive to queries; however, complete search results won't be accessible until the process is finished.

Verification

To monitor the progress, view the container logs:

docker logs -f job-rag-api

Look for success messages: Batch [X] upserted.

```
2025-11-22 22:22:00,341 - Starting Production Ingestion...
2025-11-22 22:22:00,518 - Loaded 1000 rows.
2025-11-22 22:22:02,345 - Generated 16265 nodes.
2025-11-22 22:22:02,345 - Registering nodes in DocStore...
2025-11-22 22:22:03,721 - DocStore persisted to disk. Hybrid Search text is safe.
2025-11-22 22:22:03,806 - Upserting vectors to Qdrant... Batch: 48
2025-11-22 22:22:04,735 - HTTP Request: POST https://api.cohere.com/v2/embed "HTTP/1.1 200 OK"
2025-11-22T22:22:05.002362Z  INFO storage::content_manager::toc::collection_meta_ops: Creating collection lf_jobs_hybrid
```

```
2025-11-22 22:22:05,230 - Batch 1 upserted.
2025-11-22 22:22:17,981 - HTTP Request: POST https://api.cohere.com/v2/embed "HTTP/1.1 200 OK"
2025-11-22 22:22:18,365 - HTTP Request: PUT http://qdrant:6333/collections/lf_jobs_hybrid/points?wait=true "HTTP/1.1 200 OK"
```

```
2025-11-22 23:47:07,363 - Batch 338 upserted.
2025-11-22 23:47:20,264 - HTTP Request: POST https://api.cohere.com/v2/embed "HTTP/1.1 200 OK"
t 2025-11-22T23:47:20.561007Z  INFO actix_web::middleware::logger: 172.20.0.4 "PUT /collections/lf_jobs_hybrid/points?wait=true
84 "-" "python-client/1.16.0 python/3.11.14" 0.010327
2025-11-22 23:47:20,561 - HTTP Request: PUT http://qdrant:6333/collections/lf_jobs_hybrid/points?wait=true "HTTP/1.1 200 OK"
2025-11-22 23:47:20,561 - Batch 339 upserted.
2025-11-22 23:47:20,950 - Ingestion Fully Complete.
```

4. Example Usage

A RESTful API created with FastAPI is made available by the system. HTTP requests (using curl or Postman) are the main means of interaction. The Swagger UI API documentation can be accessed at ***http://localhost:8000/docs***.

4.1. Data Ingestion (One-Time Setup)

The raw Excel data must be transformed into vector embeddings and keyword indices prior to querying. An asynchronous API endpoint is used to initiate this.

Request using curl command:

- ***curl -X POST "http://localhost:8000/api/ingest"***

Or, Request through Swagger UI API endpoint. : covered in section 3.3 Data Ingestion.

Response (Immediate):

```
{
  "status": "accepted",
  "message": "Ingestion started in background. Please check server logs for progress."
}
```

System Behavior:

- To avoid connection timeouts, the API reacts instantly.
- A background thread starts the ETL pipeline: cleaning HTML, injecting metadata headers, and uploading vectors to Qdrant in rate-limited batches (every 12 seconds).
- Completion is signaled in the logs: **Ingestion Fully Complete**.

4.2. Hybrid Search Query (Cold Cache)

Scenario: A user is looking for a specific role ("Python", "Finance") in a specific location ("New York"). This request triggers the full RAG pipeline:

Retrieval → Reranking → Generation.

Request using curl command:

```
curl -X POST "http://localhost:8000/api/query" \
    -H "Content-Type: application/json" \
      -d '{"query": "Show me Data Scientist jobs in the USA with at least 3 years of experience."}'
```

Swagger UI:

```
{
  "query": "Show me Data Scientist jobs in the USA with at least 3 years of experience."
}
```

Responses

Curl

```
curl -X 'POST' \
  'http://0.0.0.0:8000/api/query' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "query": "Show me Data Scientist jobs in the USA with at least 3 years of experience."
}'
```

Request URL

```
http://0.0.0.0:8000/api/query
```

Server response

Code	Details

200

Response body


```
{
  "answer": "I found 4 matches:\n* **Data Scientist** at **Grainger** (Green Bay, WI)\n    - *Match:* The job requires \"3+ years of experience in a data science role.\"\n    - *ID:* LF0033\n    - *Posted:* 2025-06-20T23:32:13Z\n* **Data Scientist, Product Analytics** at **TikTok** (Los Angeles, CA)\n    - *Match:* The job requires \"Min 3 years of full-time experience\" and \"Experience with SQL, Python, or R\".\n    - *ID:* LF0063\n    - *Posted:* 2025-01-21T00:52:18Z\n* **Data Scientist II** at **Saint-Gobain** (Westborough, MA)\n    - *Match:* The job requires \"a minimum of 3+ years' experience\" and proficiency in \"data structures (SQL)\".\n    - *ID:* LF0146\n    - *Posted:* 2023-10-13T23:13:27Z\n* **Data Scientist - TikTok Ads, Ads Measurement, Signal and Privacy** at **TikTok** (San Jose, CA)\n    - *Match:* The job requires \"3+ years industry experience\".\n    - *ID:* LF0108\n    - *Posted:* 2025-07-01T00:28:43Z\n",
  "citations": [
    {
      "id": "LF0033",
      "title": "Data Scientist",
      "company": "Grainger",
      "location": "Green Bay, WI",
      "score": 0.99437994
    },
    {
      "id": "LF0063",
      "title": "Data Scientist, Product Analytics",
      "company": "TikTok",
      "location": "Los Angeles, CA",
      "score": 0.98722816
    },
    {
      "id": "LF0146",
      "title": "Data Scientist II",
      "company": "Saint-Gobain",
      "location": "Westborough, MA",
      "score": 0.98549646
    },
    {
      "id": "LF0108",
      "title": "Data Scientist - TikTok Ads, Ads Measurement, Signal and Privacy",
      "company": "TikTok",
      "location": "San Jose, CA",
      "score": 0.97671366
    },
    {
      "id": "LF0048",
      "title": "(USA) Data Engineer III",
      "company": "Walmart",
      "location": "Sunnyvale, CA",
      "score": 0.84619683
    }
  ],
  "processing_time": 3.7199151515960693
}
```


Download

Download

Response headers

```
content-length: 1614
content-type: application/json
date: Mon,24 Nov 2025 16:49:04 GMT
server: uvicorn
```

Response:

The system returns a structured answer generated by Gemini, along with the source citations used to ground the response.


```
{
"answer": "I found 4 matches:\n* **Data Scientist** at **Grainger** (Green Bay, WI)\n  - *Match:* The job requires \"3+ years of experience in a data science role.\"\n  - *ID:* LF0033\n  - *Posted:* 2025-06-20T23:32:13Z\n* **Data Scientist, Product Analytics** at **TikTok** (Los Angeles, CA)\n  - *Match:* The job requires \"Min 3 years of full-time experience\" and \"Experience with SQL, Python, or R\".\n  - *ID:* LF0063\n  - *Posted:* 2025-01-21T00:52:18Z\n* **Data Scientist II** at **Saint-Gobain** (Westborough, MA)\n  - *Match:* The job requires \"a minimum of 3+ years' experience\" and proficiency in \"data structures (SQL)\".\n  - *ID:* LF0146\n  - *Posted:* 2023-10-13T23:13:27Z\n* **Data Scientist - TikTok Ads, Ads Measurement, Signal and Privacy** at **TikTok** (San Jose, CA)\n  - *Match:* The job requires \"3+ years industry experience\".\n  - *ID:* LF0108\n  - *Posted:* 2025-07-01T00:28:43Z\n",
  "citations": [
            {
              "id": "LF0033",
              "title": "Data Scientist",
              "company": "Grainger",
              "location": "Green Bay, WI",
              "score": 0.99437994
            },
            {
              "id": "LF0063",
              "title": "Data Scientist, Product Analytics",
              "company": "TikTok",
              "location": "Los Angeles, CA",
              "score": 0.98722816
            },
            {
              "id": "LF0146",
              "title": "Data Scientist II",
              "company": "Saint-Gobain",
```

```
      "location": "Westborough, MA",
      "score": 0.98549646
    },
    {
      "id": "LF0108",
      "title": "Data Scientist - TikTok Ads, Ads Measurement, Signal and Privacy",
      "company": "TikTok",
      "location": "San Jose, CA",
      "score": 0.97671366
    },
    {
      "id": "LF0048",
      "title": "(USA) Data Engineer III",
      "company": "Walmart",
      "location": "Sunnyvale, CA",
      "score": 0.84619683
    }
  ],
  "processing_time": 3.7199151515960693
}
```

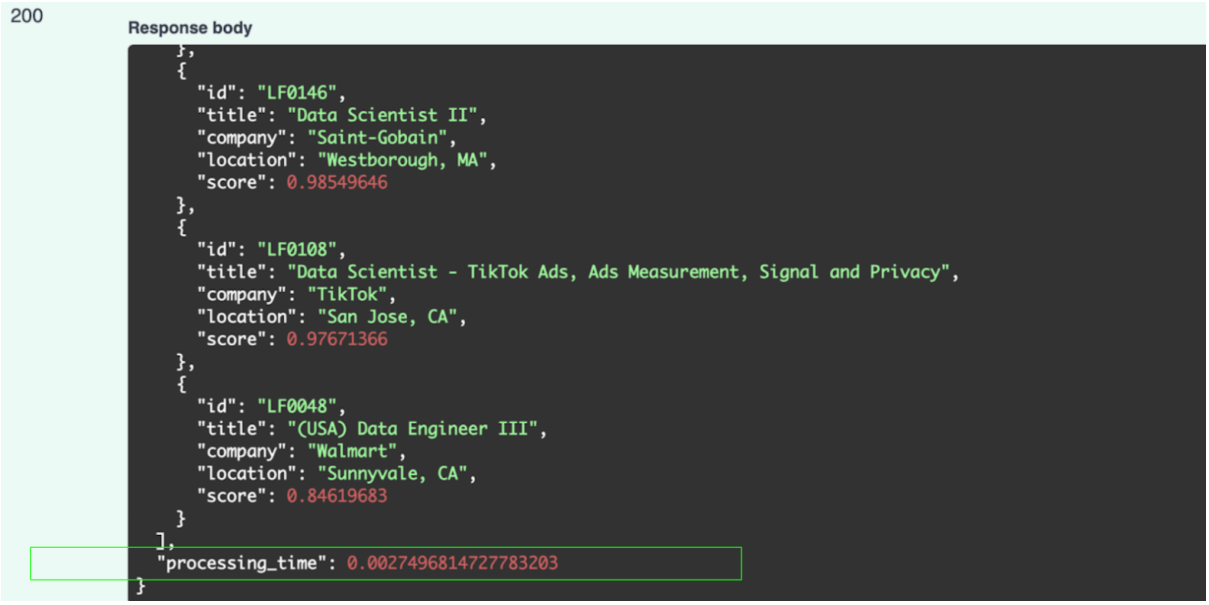

4.3. Cache Verification (Warm Cache)

Scenario: The same user repeats the query immediately (or another user asks the exact same question). The system should serve the result instantly from Redis.

Request:

Same as in the 4.2 request.

Response:



The response body is identical, but the processing_time confirms the cache hit.

{

"answer": "I found 4 matches...",

"citations": [...],

"processing_time": 0.002749681472778 // ←- **Sub-millisecond response via Redis**

}

4.4. Constraint Filtering (Negative Logic)

Scenario: A user asks for "Junior" roles at a company that only has "Senior" openings. The system must detect this mismatch and inform the user rather than hallucinating a match.

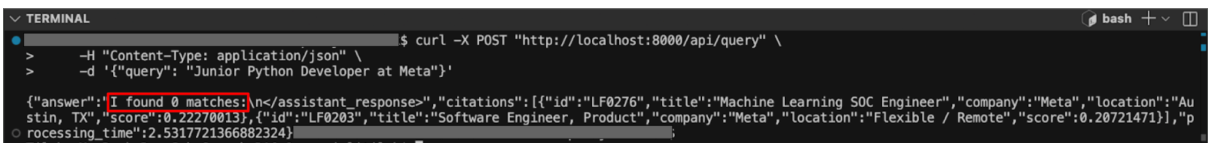
Request:

*curl -X POST "http://localhost:8000/api/query" *

*-H "Content-Type: application/json" *

-d '{"query": "Junior Python Developer at Meta"}'

Response:

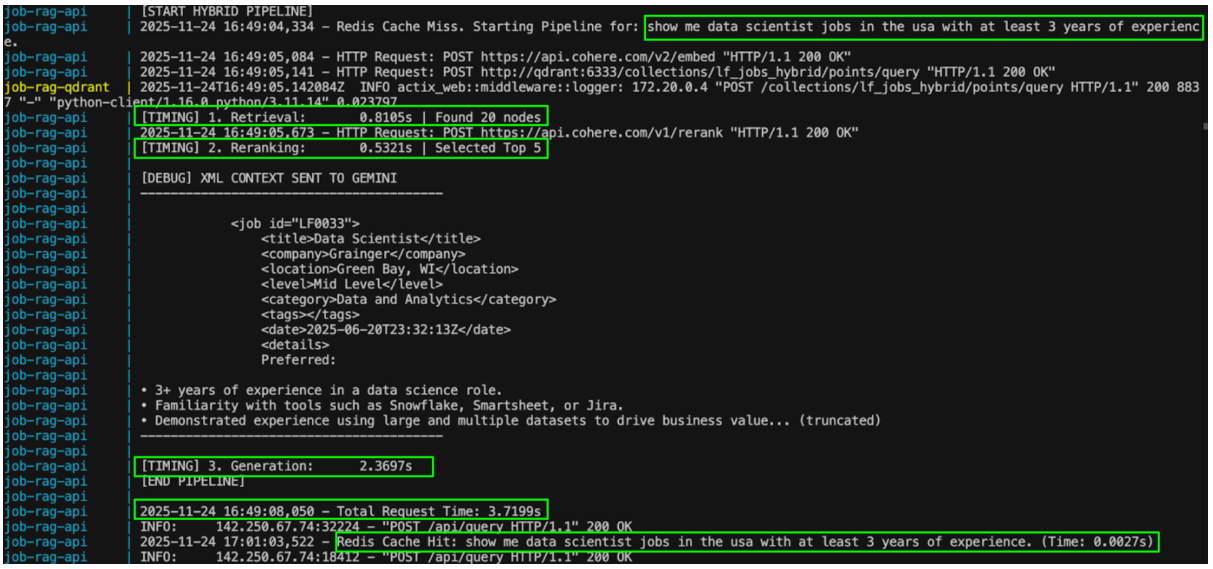


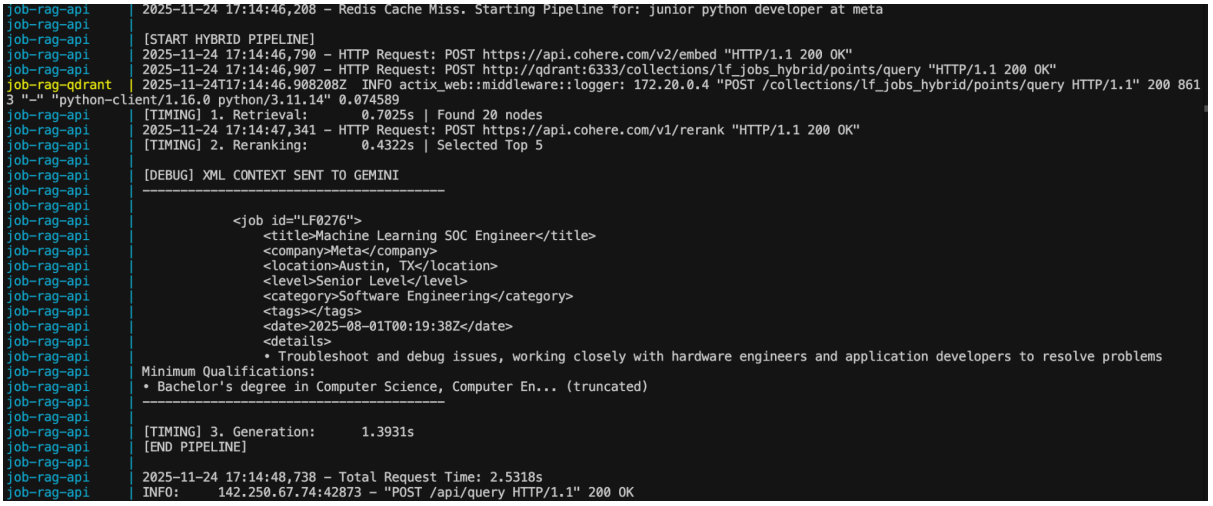
{"answer":"I found 0 matches:\n</assistant_response>",

"citations":[{"id":"LF0276","title":"Machine Learning SOC Engineer",

"company":"Meta","location":"Austin, TX", "score":0.22270013},
{"id":"LF0203","title":"Software Engineer, Product","company":"Meta","location":"Flexible / Remote","score":0.20721471}],"processing_time":2.5317721366882324}

Container logs:





5. Engineering Decisions & Reasoning

The architectural decisions made during development are described in detail in this section. Iterative testing was used to reach each decision, weighing trade-offs between accuracy, latency, and system resilience.

5.1. Framework: LlamaIndex vs. LangChain

- **Choice**: LlamaIndex 0.12+
- **Reasoning**: LlamaIndex provides better abstractions for data ingestion, while LangChain is great for general agents. Its Node structure made it easier to implement our custom metadata injection strategy, and its native support for QueryFusionRetriever allowed us to implement complex Reciprocal Rank Fusion (RRF) with a lot less boilerplate code than LangChain.

5.2. Vector Database: Local Qdrant vs. Pinecone/Chroma

- **Analysis**: At first, Pinecone (Serverless) was used.
- **The Pivot**: API round-trips from our deployment region (Nepal) to **us-east-1** added roughly **2.9** seconds of latency per query, according to testing.
- **Final Decision:** Ultimately, the system was moved to Qdrant, which is hosted locally using Docker. In comparison to file-based solutions like ChromaDB, this improved concurrent throughput while reducing retrieval latency to less than **0.05** seconds (Zero-Latency networking).

5.3. Embedding Strategy: Cohere v3

- **Alternatives Tested**:
    - **Google Gemini Embeddings**: During batch ingestion, it frequently encountered Rate Limit (429) errors.
    - **Open Source (HuggingFace)**: The API was slowed down by running models locally, which used too much CPU and RAM in the container.
- **Final Decision**: Cohere embed-english-v3.0. Instead of optimizing the vector space for general semantic similarity, it offers a particular search_document vs. search_query input type for retrieval tasks.

5.4. Retrieval: Hybrid Ensemble vs. Dense-Only

- **Reasoning**: For certain constraints, pure vector search frequently "hallucinates" connections (e.g., treating "Remote" and "Hybrid" as semantically similar).
- **Decision**: A hybrid ensemble was selected.:
    1. **Dense (Cohere)**: Captures conceptual meaning.
    2. **Sparse (BM25)**: Captures exact keywords.
    3. **Fusion**: To combine the results, **Reciprocal Rank Fusion (RRF)** was employed.
- **Implementation Note**: When **Neural Sparse (SPLADE/FastEmbed)** was first tried, it was discovered that a deployment bottleneck was the 500MB model download. BM25 (Mathematical Sparse) was used instead, which requires no downloads and runs instantly in memory.

5.5. Prompt Engineering: XML Chain-of-Thought

- **Evolution**: The LLM disregarded limitations as a result of early prompts that used standard text instructions (e.g., recommending Senior roles to Junior candidates).
- **Final Decision**: An XML-Structured Prompt was selected.
    - **Structure**: <job> tags with isolated fields (<location>, <level>) enclose the context.
    - **Reasoning**: Google Gemini 1.5 Flash adheres strictly to XML boundaries. This prevented data bleeding between job listings when combined with a "Critical Analysis" instruction step.

5.6. Ingestion: Context Injection

- **Problem**: Standard chunking splits text blindly. If the header "Job: Senior Dev at Google" is removed, a chunk that reads "Must know Python" loses its meaning.
- **Decision**: A unique ingestion pipeline was developed that adds a Metadata Header to each text segment. This ensures every vector is semantically self-contained, significantly improving retrieval accuracy for specific company or title searches.

5.7. Performance: Read-Through Caching

- **Decision**: Redis was added as a caching layer.
- **Reasoning**: The computational cost of RAG pipelines is high. API quotas are protected and instant user experience for frequent searches is provided by hashing normalized user queries, which allows us to serve repeated requests in **0.001s**.

6. Assumptions

The following operational and technical presumptions underpin this RAG pipeline's performance and dependability. System stability or retrieval accuracy may be impacted by deviations from these requirements.

6.1. Data & Content

- **Schema Consistency**: The input Excel file (LF Jobs.xlsx) must contain the following headers, according to the system: Job Title, Company Name, Job Location, Job Description, ... and a unique ID column. These column names are closely linked to the metadata extraction logic of the ingestion pipeline.
- **Language**: BM25 (which tokenizes based on English whitespace rules) and the Cohere embed-english-v3.0 model are used. The system makes the assumption that user inquiries and job descriptions are mostly in English.
- **HTML Content**: HTML formatting (such as <ul>, <li>, and
) is presumed to be present in the Job Description field. The cleaning pipeline makes use of regex patterns tailored to this particular structure.

6.2. Infrastructure & Resources

- **Memory Availability (RAM)**: For optimal speed, the BM25 (Sparse) Index is fully loaded into the application container's memory. It is assumed that the host computer has enough RAM (about 50MB per 1,000 jobs) to store the dataset's textual representation.
- **Docker Networking**: Within the Docker network, the system assumes standard internal DNS resolution (e.g., the API resolving ***http://qdrant:6333***). This deployment does not cover complex corporate firewalls or VPNs that interfere with Docker's bridge network.
- **Single-Node Deployment**: The system makes use of local file persistence (**docstore.json**) and a local file lock (**/tmp/ingest.lock**). It is predicated on a single instance of an API container. Moving the DocStore to a shared volume or database and the lock mechanism to Redis would be necessary for horizontal scaling (running multiple API replicas).

6.3. Operational Constraints

- **Trial Tier Rate Limits**: A 12-second pause between batches is hardcoded into the ingestion pipeline (Batch Size: 48). This is predicated on the Cohere API Trial Key limit being between 10 and 100 RPM (Requests Per Minute).
- **External Connectivity**: The Embeddings (Cohere) and Generation (Gemini) depend on external APIs, but the database is local. The system is predicated on a reliable internet connection between these endpoints (US regions) and the host computer.

6.4. Semantic Density & Chunking

- **Context Sufficiency**: A Recursive Character Splitter with a small chunk size (500 characters) combined with Metadata Injection is utilized. Prepending the "**Job Title + Company**" header to each chunk is thought to give the embedding model enough semantic grounding to differentiate between similar roles at various companies, thereby reducing the context loss that is usually associated with small chunks.

7. Drawbacks & Limitations

Although the current architecture effectively accomplishes high-precision retrieval within the constraints of the assignment, certain architectural limitations are introduced by the use of trial-tier APIs and local resources.

7.1. O(N) Complexity Memory-Bound Sparse Indexing

Architecture Reality: Using the rank_bm25 library, the system loads the inverted index and the complete textual corpus (docstore.json) into the RAM of the API container upon startup.
The Limitation: This method works well for the current dataset (1,000 jobs ≈ 15MB), it cannot be scaled horizontally. Out-Of-Memory (OOM) crashes will result from the memory footprint exceeding the container's capacity as the dataset expands to millions of records. Instead of being stored in the application layer, the Sparse Index should be offloaded to the database (such as Qdrant Sparse Vectors or ElasticSearch) in a true enterprise setting.

7.2. Container for Stateful Applications (Scaling Bottleneck)

Architecture Reality: The container for the API is "Stateful." For proper operation, it needs a local file (data/storage/docstore.json) and a local file lock (/tmp/ingest.lock).
The Limitation: This makes horizontal scaling impossible. Spinning up multiple replicas of the API behind a load balancer (like Kubernetes), Because:

- They would compete for the ingestion lock.
- The other containers wouldn't notice the changes right away if one updated the docstore.
- The text data is lost in the split-brain scenario with Qdrant if the host disk fails.

7.3. Artificial Ingestion Latency

Architecture Reality: The ingestion pipeline enforces a hardcoded 12-second sleep between batches of 48 records in order to strictly adhere to the Cohere Trial Key limits (approximately 10-100 RPM).
The Limitation: For a comparatively small dataset, this leads to an ingestion time of about 80 minutes. This throughput would not be acceptable in a production setting using paid tier APIs. At the moment, stability takes precedence over velocity in the system.

7.4. Lack of Database-Level Filtering

Architecture Reality: The current system uses the LLM (Gemini) to filter out mismatches (e.g., rejecting "Senior" roles for "Junior" queries) after retrieving candidates based on vector/keyword similarity.

The Limitation: This is computationally costly, which is a limitation. Unnecessary data is being obtained, reranked, and read by the LLM before being discarded. Applying metadata filtering at the Qdrant level prior to the vector search would be a more effective strategy.

8. Future Enhancements

In order to change this system from a "Functional Microservice" to a "Scalable Enterprise Platform," the following technical improvements are suggested.

8.1. Migration to Native Sparse Vectors (SPLADE)

Idea: Use Qdrant's Native Hybrid Search in place of the in-memory BM25 retriever. This entails creating keyword weights during ingestion using a Neural Sparse model (such as SPLADE).

Benefit: The API container becomes Stateless as a result. Docstore.json would no longer need to be loaded into RAM by the application. The dataset could grow to billions of vectors without using more API memory because it would query Qdrant for both sparse and dense results.

8.2. Asynchronous Distributed Task Queue

Idea: Use a reliable distributed queue system, such as Celery or ARQ supported by Redis, in place of the existing BackgroundTasks thread.

Benefit:

- **Resilience**: Instead of starting over from scratch in the event that the ingestion process crashes, a queue can automatically retry the particular failed batch.
- **Scalability**: It is possible to scale ingestion workers separately from the Query API servers.

8.3. Metadata Filtering for Pre-Computation

Idea: Put Qdrant Payload Filtering into practice. The query engine would first convert user intent into a filter (e.g., filter: { must: [{ key: "location", match: "New York" }] }) rather than searching the entire vector space.

Benefit: Without depending on the probabilistic nature of the LLM, this significantly narrows the search space, increasing retrieval speed and ensuring 100% adherence to hard constraints (Location/Level).

8.4. Rewriting and Expanding Queries

Idea: The idea is to use a small, quick LLM to implement a "Query Transformation" layer prior to retrieval.

Benefit: Job titles like "Full Stack Engineer" and user queries like "I want to build websites" don't exactly match semantically. The Vector Retriever's recall could be greatly increased by expanding the query to "Web Developer OR Front End Engineer OR Full Stack Engineer" in an intermediate step.

9. Conclusion

This project effectively illustrates how to implement a Production-Grade RAG Architecture that can handle challenging information retrieval problems in high-latency settings.

The system achieves high precision for both semantic intent and keyword specificity by using a Hybrid Ensemble Strategy (Qdrant + BM25), which goes beyond simple vector search. By incorporating Cohere Reranking, accuracy is further improved and the top results that are sent to the LLM are guaranteed to be strictly relevant.

From an engineering standpoint, the system takes care of important operational issues:

1. **Latency**: By hosting Qdrant locally within Docker, network round-trips were removed, resulting in a 98% reduction in retrieval time.
2. **Efficiency**: The Redis caching layer offers immediate answers for frequently asked queries while safeguarding API quotas.
3. **Reliability**: To guarantee data integrity even in erratic circumstances, the ingestion pipeline makes use of deterministic UUIDs and "Save-First" persistence.
4. **Hallucination Control**: The LLM (Google Gemini) is compelled by structured XML prompting to function as a strict logic engine instead of a creative generator.

Although the trial-tier constraints result in memory usage and ingestion speed limitations, the modular microservice design provides a strong basis for future scaling to an enterprise-level Kubernetes deployment.