# RAG for Smarter Resume Analysis: Beyond Basic LLMs

[1] Igor Zuykov
[1] Chief Software Engineer, G-71 Inc.Ashkelon, Israel

Received: 18th Nov 2025 | Received Revised Version: 28th Nov 2025 | Accepted: 19th Dec 2025 | Published: 31th Dec 2025

Volume 07 Issue 12 2025 | Crossref DOI: 10.37547/tajet/Volume07Issue12-16

**Abstract**

*The article examines an architectural approach to resume analysis based on Retrieval-Augmented Generation (RAG), designed to overcome the systemic limitations of traditional keyword-matching algorithms (like TF-IDF and BM25) and the inherent constraints of large language models (LLMs) used in isolation under conditions of an overloaded and semantically heterogeneous hiring market. The relevance of the work is driven by the growth in the volume and variability of resumes, the need to capture latent semantic correspondences between experience phrasing and vacancy requirements, and the risks of algorithmic biases, as well as the plausible yet unreliable generation of personnel decisions. The study aims to formalize a dual-loop scheme for processing a resume corpus, in which dense semantic retrieval over vector representations of document fragments is coupled with answer generation strictly constrained by the retrieved context and complex refusal rules under insufficient grounds. The scientific novelty lies in interpreting the RAG approach as a mechanism of search-based non-parametric memory for a corporate resume array, where the chunking strategy (determined at the ingestion phase) and the retrieval parameters such as topK and similarity. Threshold, directly governing the scope and quality of information passed to the retrieval act as controllable regulators of the recall–noise–cost trade-off, and where requirements for explainability, traceability, and privacy are derived from HR-specific constraints rather than declared post factum. It is demonstrated that separating retrieval and generation functions, offloading compute-intensive corpus preparation into an asynchronous loop, and locally deploying models jointly reduce LLM load, decrease the incidence of hallucinations, and enable verifiable candidate ranking based on the semantic proximity of the experience to the recruiter's query. It is concluded that the reliability of systems of this class is determined not by model strength, but by the architecture of source control and the discipline of context management. The article will be helpful for researchers and engineers developing intelligent talent selection systems, as well as for practicing recruiters and HR analysts implementing RAG solutions in corporate processes.*

Keywords: resume analysis, Retrieval-Augmented Generation, semantic search, dense retrieval, vector representations

**Cite This Article:** Zuykov, I. (2025). RAG for Smarter Resume Analysis: Beyond Basic LLMs. The American Journal of Engineering and Technology, 7(12), 152–163. https://doi.org/10.37547/tajet/Volume07Issue12-16

## 1. Introduction

The contemporary hiring market is characterized by high competition and accelerated recruitment cycles, which increases the employer-side burden of initial screening: recruiters must review hundreds of resumes to identify a limited number of candidates who truly match a role profile (Bouhsaien & Azmani, 2025). In this setting, the critical factor is not only the volume of incoming documents but also their heterogeneity in structure and style of experience description, where identical competencies are expressed via divergent terminology. Meanwhile, salient details are often embedded within descriptions of projects, responsibilities, and work execution contexts. This practical problem is explicitly recognized: the scale of resume review is significant, and standard screening procedures are insufficiently sensitive to semantic nuances and latent correspondences between the query and resume content (Ajjam & Al-Raweshidy, 2025).

Existing approaches based on keyword search provide a formally interpretable yet methodologically brittle output: they match strings rather than meaning, and thus readily miss relevant candidates when resume terminology differs from query terminology (Ajjam & Al-Raweshidy, 2025). This is illustrated by the case where a query for cloud experience fails to retrieve resumes containing specific mentions of cloud technologies, although the competence domain is substantively the same. Attempting to replace such search with a pure LLM also encounters systemic limitations: the model may exhibit strong generalization and reasoning capabilities, yet it lacks knowledge of a specific corporate resume repository until those data are explicitly provided; when the full text is directly injected into the prompt, context-window constraints and the economic inefficiency of repeatedly reprocessing the same fragments for each new question quickly become apparent.

The objective of this work is to formulate and substantiate an architectural approach to resume analysis in which an LLM responds not in general, but by relying on strictly relevant fragments retrieved from a private resume database via semantic similarity. This principle is described as providing the model with an external knowledge base, where a dedicated retrieval subsystem selects relevant evidence from the resume base, and the generation component uses only the obtained context to construct an answer.

## 2. Materials and Methodology

The research materials comprised a corpus of scientific and applied sources along three intersecting lines: (1) digital transformation and operational constraints of recruitment, manifested in the high load of initial screening and structural heterogeneity of resumes (Bouhsaien & Azmani, 2025); (2) semantic matching of resumes and job postings as an alternative to literal keyword search, where sensitivity to terminological variability is treated as a central quality criterion (Ajjam & Al-Raweshidy, 2025); (3) Retrieval-Augmented Generation architectures as a means of grounding generation in retrieved evidence, together with engineering trade-offs of recall/noise and the economics of repeated queries (Gupta et al., 2024). Additionally, the theoretical framework incorporates work on Dense Retrieval as a methodological foundation for semantic search over document fragments (Karpukhin et al., 2020) and on sentence embeddings for robust comparison of short and medium text segments within a shared vector space (Reimers & Gurevych, 2019). To substantiate reliability requirements for personnel decisions, research on algorithmic bias in hiring was considered (Raghavan et al., 2020), along with empirical demonstrations of LLM propensity toward plausible yet false claims under insufficient context, rendering source control not an option but a safety-by-design principle (Lin et al., 2022).

The methodology relies on conceptual modeling of an RAG system for resume analysis as a dual-loop computational process: an asynchronous corpus preparation (ingestion) loop and an interactive query-processing loop. In the first loop, resumes are transformed into a text stream, segmented into semantic chunks, and encoded into embeddings; the chunks are then indexed in a vector store, where search is implemented as a nearest-neighbor problem, typically requiring approximate methods to maintain latency at a practical level (Gupta et al., 2024). In the second loop, the recruiter's query is encoded into the same space, top-K chunks are retrieved by semantic similarity (Karpukhin et al., 2020), after which generation is constrained strictly to the retrieved context and augmented with a strict refusal rule when retrieval provides insufficient or low-confidence evidence, to minimize hallucination risk and improve verifiability through references to specific resume fragments (Lin et al., 2022). The parameters topK and chunk size are treated as controllable regulators of the recall–noise–cost trade-off. At the same time, explainability and robustness to linguistic variability are considered fixed requirements derived from the nature of recruiting and the risks associated with automated decision-making (Ajjam & Al-Raweshidy, 2025; Raghavan et al., 2020).

## 3. Results and Discussion

Within the considered approach, the resume analysis task is formalized as transforming an unstructured set of candidate documents into a system that, upon a recruiter query, outputs either a substantiated natural-language answer or an ordered list of candidates with stated reasons for relevance. The input comprises resumes as files, each document containing heterogeneous information about experience, skills, achievements, varying in format, detail, and overall completeness.

Additionally, a recruiter query in natural language is provided, which may be a question about a specific candidate or a profile description (who fits a role requiring a given stack and domain). The output must include, first, a natural-language response grounded in resume facts, and second, in a search scenario, a ranking of candidates by degree of match to the query, preferably with relevance scoring and references to the fragments supporting the decision. This design follows the general idea of generation augmented by retrieved context: relevant fragments are first selected from an external store, and the generator then constructs the answer by treating these fragments as non-parametric memory that can be updated independently of the model.

To implement this formulation, a dual-loop computation scheme is introduced. In the first loop, executed asynchronously relative to user requests, each document is split into coherent semantic segments. For each segment, a vector representation is constructed that enables measuring similarity by meaning rather than string overlap. The segments and their metadata are then stored in a vector database. The second loop, invoked per query, maps the query text into a vector representation and performs nearest-segment search, subsequently constraining the context to the first K retrieved fragments. The conceptual justification for dense retrieval over vector representations in context search is supported by open-domain question answering research, where dense retrieval methods substantially improve the accuracy of retrieving relevant passages compared to sparse search models (Karpukhin et al., 2020). The choice of the embedding model is critical because it determines the geometry of the semantic space. In this regard, specialized sentence-level models are particularly suitable, as they provide cosine-comparable representations and are explicitly designed for the semantic matching of short and medium-length text fragments (Reimers & Gurevych, 2019).

System requirements are conveniently divided into functional and non-functional. In terms of accuracy, candidate ranking is expected to be robust to phrasing variability in resumes and queries and not to degrade under synonyms and indirect mentions; accuracy should be treated in two senses: the quality of retrieving relevant fragments and the correctness of the final answer, which must not add information absent from the resumes. In terms of explainability, the system must expose the grounds for its responses in the form of concrete, actionable fragments; otherwise, it becomes a black box, particularly hazardous in HR processes where decisions carry significant social consequences. In terms of privacy, resumes and their derived representations must remain within the organizational perimeter, and access to the store and query logs must be controlled. This requirement is a foundational requirement for responsible deployment, as maintaining data sovereignty within the organizational perimeter is essential for conducting the internal audits, bias testing, and fairness validations required to mitigate the documented risks of automated hiring systems (Raghavan et al., 2020).

From a cost perspective, minimizing repeated computations is essential: compute-intensive preparation operations (document parsing and embedding construction) should be shifted into the ingestion loop so that interactive queries rely on a prebuilt index. For latency, a near-dialogue mode is required: the user perceives the system as an assistant, so response time should be determined by retrieval plus generation time; increasing K improves context recall but raises latency and increases the risk of context noising by including fragments that are thematically close but not factually relevant to the query. This trade-off is typical for retrieval-augmented generation systems and has been discussed in the survey literature on the subject (Gupta et al., 2024).

Typical user scenarios follow naturally from this formalization. The first scenario is a per-candidate question-answer mode: the query is framed as clarifying a fact or competence, and the system returns a concise answer grounded in specific lines of experience and projects, avoiding conjecture. The second scenario is candidate search by skills and domains: the query specifies a profile, the system retrieves semantically similar fragments, aggregates them at the candidate level, and produces a ranking with explanations of why candidates appear in top positions. The third scenario is candidate comparison: the user provides comparison criteria, and the system, preserving grounding in resume text, synthesizes differences across relevant aspects, indicating which fragments support the conclusions; here, constraining generation to retrieved context and explicitly refusing under insufficient data are significant, because otherwise the model begins to fill gaps with plausible but unverifiable assertions, contradicting the aims of reliable screening.

Semantic search in recruitment relies on vector representations of text in which proximity reflects semantic similarity rather than string overlap. This shift is significant for resumes, where competencies are often expressed indirectly through descriptions of projects and responsibilities, and therefore are poorly captured by literal matching. The practical foundation of such systems consists of models that encode sentences and paragraphs into compact vectors suitable for cosine similarity comparison; notably, sentence-level model families are optimized explicitly for semantic similarity and text clustering tasks (Reimers & Gurevych, 2019). In a related line of work on dense retrieval for question answering, it is shown that a dual-encoder architecture (encoding questions and passages separately) enables the effective retrieval of relevant excerpts without relying on sparse features, which is methodologically similar to searching over a resume corpus (Karpukhin et al., 2020). When the number of resume segments becomes large, a central engineering issue arises: performing fast nearest-neighbor searches in high-dimensional spaces. Vector stores typically implement approximate search using indexes that strike a balance between accuracy and latency. To address this, we employ a graph-based approximate nearest neighbor (ANN) index, specifically the Hierarchical Navigable Small World (HNSW) graph. In HNSW, the search begins at coarse levels of the graph and refines at lower ones, providing a practical operational compromise between query latency and retrieval recall (Ponomarenko, 2025).

At the intersection of retrieval and generation, an approach has formed in which answers are constructed not only from the parameters of a language model but also from retrieved fragments of external memory: first, relevant segments are retrieved; then these segments are added to the query context; only afterward is generation invoked, already grounded in the retrieved evidence. However, the key risk remains: language models tend to produce plausible yet incorrect claims, especially when context is incomplete or contradictory. Research on truthfulness in generation introduces target question sets that reveal systematic false answers, underscoring the need for explicit self-control mechanisms in applied systems (Lin et al., 2022). Therefore, practical implementations of retrieval-augmented generation typically enforce strict contextual constraints, requiring reliance only on provided fragments, and introduce refusal behavior when retrieval does not return sufficient grounds; this reduces the likelihood of inventing details and makes the system more controllable in an HR setting where the cost of error is particularly high.

The proposed approach relies on the idea of generation augmented by retrieval: the system does not attempt to remember all resumes within the parameters of a language model. It does not require the user to search for keyword overlaps manually, but instead constructs an intermediate layer of semantic memory. This memory is formed from resume fragments represented as vectors, enabling the retrieval of only those parts of texts that truly contain relevant signals of experience and skills. As a result, the answer becomes simultaneously more accurate and more verifiable because it is tied to concrete evidence in documents rather than probabilistic guesswork.

In applied terms, each user query is treated as an operation over a resume corpus with two possible outputs: either a concise answer to a question about a candidate or a ranked list of candidates by degree of fit to criteria. The core principle is functional separation: search is responsible for retrieving context, while the language model is responsible for coherent exposition and response normalization. This separation reduces generative load, lowers the cost of repeated queries over the same corpus, and makes system behavior more stable as the number of documents grows.

The first phase, corpus preparation, begins with extracting text from resumes, including file formats where structure is visually rather than logically defined. At this step, obtaining a maximally clean text stream is crucial: remove formatting artifacts, preserve the order of semantic blocks, and, where possible, retain section markers such as work experience, education, and skills lists so that they can be used later as metadata. Even with strong extraction, unavoidable heterogeneity remains: one candidate lists responsibilities as bullet points, another uses continuous prose, and a third emphasizes project titles; therefore, subsequent stages must be invariant to such stylistic fluctuations.

Next, the text is split into fragments, where splitting must be semantic rather than purely mechanical: overly large chunks dilute the signal and increase the risk that irrelevant details enter the context, while overly small chunks destroy causal relationships and reduce informativeness. Each chunk is transformed into a vector representation that encodes meaning and enables proximity measurement between the query and the chunk via vector-space geometry. The resulting vectors are indexed in a vector store, along with metadata that links each chunk to a specific candidate, resume section, and position in the document. This is critical for subsequent aggregation of results at the candidate level and for explainability. The algorithm of the first phase is shown in Fig. 1.

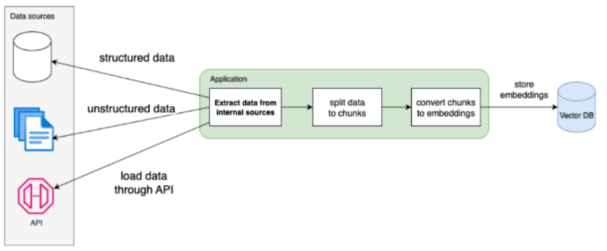


**Fig. 1. Ingestion Phase**

The second phase, query processing, begins by mapping the recruiter's query text into a vector representation in the same feature space as the resume chunks. A nearest-neighbor search by semantic similarity is then performed, selecting a limited set of the first K results to maintain a balance between recall and noise. The K constraint functions as a throttle: increasing K raises the probability of capturing all relevant evidence but simultaneously expands the context and can degrade generation accuracy if the context includes fragments that are thematically close yet factually unrelated to the query.

At the final step, retrieved fragments are aggregated into a single context and provided to the language model together with complex response rules. The rules specify that formulations must rely only on the provided context and must not add new facts; if retrieved fragments do not contain sufficient grounds, the system must explicitly report the absence of data rather than fill gaps with assumptions. Depending on the query type, generation returns either a coherent answer indicating which experience elements substantiate the conclusion or a candidate ranking, where each candidate's position is explained by the sum of discovered evidence and its relevance, thereby closing the loop from semantic retrieval to controlled, verifiable interpretation. The algorithm of the second phase is shown in Fig. 2.

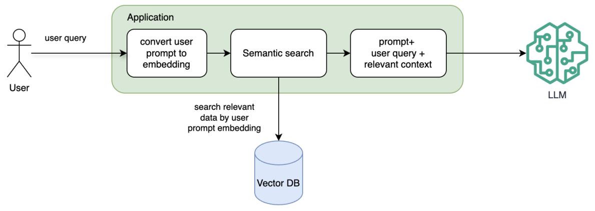

Application
user query
User
convert user prompt to embedding
Semantic search
prompt+ user query + relevant context
LLM
search relevant data by user prompt embedding
Vector DB

**Fig. 2. Intelligent Retrieval**

An essential element of the approach is the design of the model prompt template, which codifies strict interpretation rules: the model must answer exclusively within the provided context and, in the absence of context, must directly report inability to respond. This rule makes system behavior predictable and reduces the probability of fabricated assertions when the resume lacks supporting evidence or retrieval fails to find relevant fragments. Practically, this is implemented by placing retrieved pieces into a separate context block within the prompt, after which generation becomes context-conditioned and oriented toward resume facts rather than conjecture.

The solution architecture comprises a backend service, a vector store, an LLM runtime, and a client interface, all of which utilize a clear data exchange protocol. The backend ingests resumes and user queries, performs document preparation, and orchestrates the retrieval of relevant fragments. The vector store provides fast semantic similarity search, while the LLM runtime produces the final answer. The client interface supports resume upload and a dialogue-style query mode. As a concrete implementation, a server on the Spring Boot platform with Spring AI dependency to interact with the LLM is proposed, utilizing a PostgreSQL database with pgvector extension as the vector store, a local Ollama runtime for model execution, a React web interface, and container orchestration via Docker Compose, resulting in a reproducible deployment suitable for a local operational perimeter.

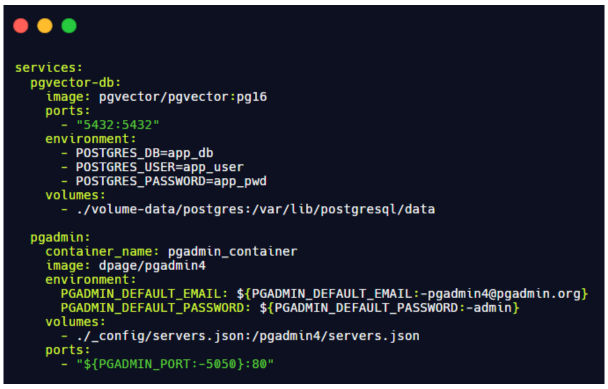
Two practical considerations drive technology selection: minimizing external dependencies and simplifying operations. A local model runtime stores resumes and derived representations within the infrastructure. In contrast, a vector store, implemented as an extension to a relational database, simplifies administration and ensures compatibility with standard corporate data storage practices. Container orchestration through Compose provides a unified launch scenario: separate services bring up the database with vector extension, an administrative interface, and the local model server, including preloading of the selected model, standardizing environments for development and testing. The server-side application interface design assumes at least two operation classes: resume upload and query execution, returning either an answer or a ranking. Typical flow: Resume file received from client, server extracts and splits the text into fixed chunks, saves chunk embeddings in a vector DB. The typical chunk size ranges from 350 bytes to 10,000 bytes, but can be split on a separator designed for splitting, typically around 800 bytes.

To keep the entire pipeline inside the organizational perimeter, the implementation can be grounded on a local LLM runtime and a relational vector store. In this configuration, Ollama is used as the local inference service, the Mistral model acts as the generative engine, and PostgreSQL with the pgvector extension functions as the vector database for semantic retrieval. This choice is operationally conservative: it reduces reliance on external APIs, makes data flows auditable, and allows model and infrastructure versions to be pinned for reproducibility across experiments and pilots.

From a deployment standpoint, there are two equivalent topologies. In the first, Ollama runs directly on the host machine, exposing a local HTTP endpoint (commonly *http://localhost:11434*) while PostgreSQL with pgvector is started in the container. In the second, both the database and Ollama are started via Docker Compose, which simplifies “one-command” startup and enforces consistent environments across machines. The practical difference is not conceptual but operational: the local Ollama option may be more convenient during iterative development, whereas the fully containerized option provides a uniform runtime boundary that is easier to replicate in staging.

A minimal prerequisite for local inference is ensuring that the Mistral model is available to Ollama.

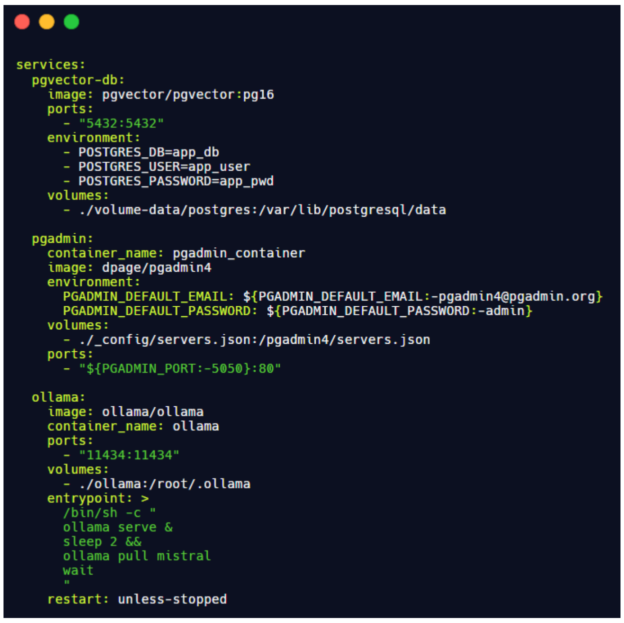
When PostgreSQL with pgvector is launched in Docker, along with an administrative UI, a Compose configuration can define a database service and a pgAdmin service, specifying explicit ports, credentials, and a persistent volume for data durability. This provides a stable vector store endpoint while keeping operational state in a mounted directory rather than inside ephemeral containers. Insert the following Docker Compose fragment as the database-only option, as shown in Figure 3.



```
services:
  pgvector-db:
    image: pgvector/pgvector:pg16
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=app_db
      - POSTGRES_USER=app_user
      - POSTGRES_PASSWORD=app_pwd
    volumes:
      - ./volume-data/postgres:/var/lib/postgresql/data

  pgadmin:
    container_name: pgadmin_container
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL:-pgadmin4@pgadmin.org}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD:-admin}
    volumes:
      - ./_config/servers.json:/pgadmin4/servers.json
    ports:
      - "${PGADMIN_PORT:-5050}:80"
```

**Fig. 3. Launch pgvector in Docker**

In a fully containerized topology, the Compose file can be extended with an Ollama service that starts the server process and preloads the Mistral model inside the container, persisting model artifacts via a mounted volume. This yields a self-contained runtime where the LLM endpoint and the vector database are co-started and can be managed uniformly. Insert the following Compose fragment as the “database + Ollama” option, as shown in Figure 4.



```
services:
  pgvector-db:
    image: pgvector/pgvector:pg16
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=app_db
      - POSTGRES_USER=app_user
      - POSTGRES_PASSWORD=app_pwd
    volumes:
      - ./volume-data/postgres:/var/lib/postgresql/data

  pgadmin:
    container_name: pgadmin_container
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL:-pgadmin4@pgadmin.org}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD:-admin}
    volumes:
      - ./_config/servers.json:/pgadmin4/servers.json
    ports:
      - "${PGADMIN_PORT:-5050}:80"

  ollama:
    image: ollama/ollama
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - ./ollama:/root/.ollama
    entrypoint: >
      /bin/sh -c "
      ollama serve &
      sleep 2 &&
      ollama pull mistral
      wait
      "
    restart: unless-stopped
```

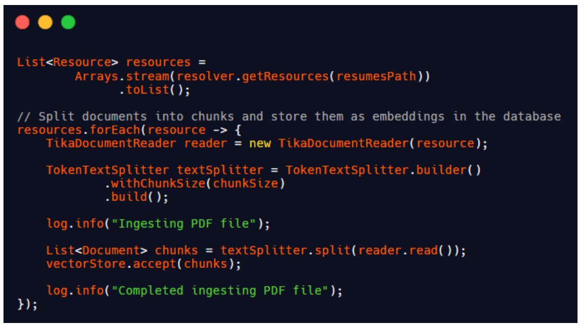
**Fig. 4. Ollama and pgvector in Docker**

Once the Compose configuration is prepared, the infrastructure is started by bringing the stack up in detached mode, which creates the networked services required for ingestion and retrieval. The exact command to start is “*docker-compose up -d*”.

Once the infrastructure is initialized, the ingestion pipeline instantiates the conceptual model described earlier: upon resume upload, the backend extracts textual content, splits the document into chunks, converts each chunk into an embedding, and writes both vectors and metadata into pgvector.

A subtle but consequential engineering decision is to separate the embedding model from the generative model. While Spring AI can reuse the same model for Q&A and embeddings, retrieval quality typically improves when embeddings are produced by a model explicitly intended for vector representations. In this case, the embedding model is selected via configuration (for example, through the parameter *spring.ai.ollama.embedding.model*), allowing the retrieval subsystem to be optimized independently of the generator's stylistic and reasoning behavior.

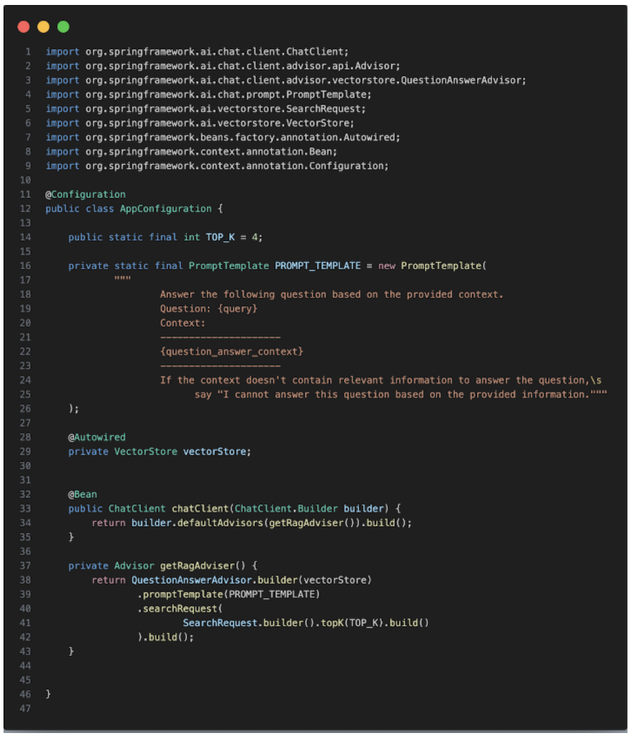
The following (Fig. 5) illustrates a representative ingestion routine, concretely implementing the steps outlined above: reading a PDF resource, splitting into chunks, generating embeddings, and persisting them to the vector store.



```java
List<Resource> resources =
        Arrays.stream(resolver.getResources(resumesPath))
              .toList();

// Split documents into chunks and store them as embeddings in the database
resources.forEach(resource -> {
    TikaDocumentReader reader = new TikaDocumentReader(resource);

    TokenTextSplitter textSplitter = TokenTextSplitter.builder()
            .withChunkSize(chunkSize)
            .build();

    log.info("Ingesting PDF file");

    List<Document> chunks = textSplitter.split(reader.read());
    vectorStore.accept(chunks);

    log.info("Completed ingesting PDF file");
});
```

**Fig. 5. Code example of Ingestion Phase**

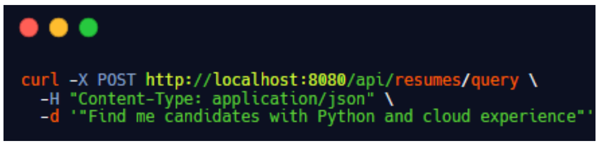
On the retrieval-and-generation side, the key mechanism that enforces grounding is an advisor that automatically performs vector search and injects retrieved fragments into a prompt template. The template explicitly binds the model's output to the provided context. It requires a refusal behavior when the context is empty, thereby turning "lack of evidence" into a controlled, observable outcome rather than an invitation to fabricate plausible details. A concrete Spring AI configuration that implements this pattern is shown in Fig. 6. It defines the prompt template, fixes the TOP_K retrieval parameter, and wires the QuestionAnswerAdvisor to the vector store.



```java
import org.springframework.ai.chat.client.ChatClient;
import org.springframework.ai.chat.client.advisor.api.Advisor;
import org.springframework.ai.chat.client.advisor.vectorstore.QuestionAnswerAdvisor;
import org.springframework.ai.chat.prompt.PromptTemplate;
import org.springframework.ai.vectorstore.SearchRequest;
import org.springframework.ai.vectorstore.VectorStore;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfiguration {

    public static final int TOP_K = 4;

    private static final PromptTemplate PROMPT_TEMPLATE = new PromptTemplate(
            """
                    Answer the following question based on the provided context.
                    Question: {query}
                    Context:
                    ---------------------
                    {question_answer_context}
                    ---------------------
                    If the context doesn't contain relevant information to answer the question,\s
                          say "I cannot answer this question based on the provided information."""
    );

    @Autowired
    private VectorStore vectorStore;


    @Bean
    public ChatClient chatClient(ChatClient.Builder builder) {
        return builder.defaultAdvisors(getRagAdviser()).build();
    }

    private Advisor getRagAdviser() {
        return QuestionAnswerAdvisor.builder(vectorStore)
                .promptTemplate(PROMPT_TEMPLATE)
                .searchRequest(
                        SearchRequest.builder().topK(TOP_K).build()
                ).build();
    }


}
```

**Fig. 6. Core RAG configuration**

Finally, an end-to-end system validation can be performed by sending a query request to the backend API, confirming that the system executes retrieval against pgvector and that the model produces an answer constrained to the retrieved resume fragments. The following request exemplifies this verification procedure.



```bash
curl -X POST http://localhost:8080/api/resumes/query \
  -H "Content-Type: application/json" \
  -d '"Find me candidates with Python and cloud experience"'
```

**Fig. 7. Example of a request**

Model choice determines the style and robustness of the entire system and is therefore appropriately treated as a managed configuration rather than a one-time decision. The embedding model defines the quality of matching queries to resume fragments, while the generative model determines clarity, coherence, and formulation discipline. With local deployment, the organization gains additional control over privacy and reproducibility: model versions can be pinned, quality changes can be tracked across updates, and compute resources can be allocated to the required load profile. This makes the system suitable for prototyping and pilot deployments, as well as for careful scaling while maintaining controllability and compliance with corporate data-perimeter requirements.

## 4. Conclusion

The architecture proposed in the article conceptualizes resume analysis as a task of semantically governed retrieval followed by generation, where the central methodological commitment is a departure from literal string matching and a rejection of a self-sufficient LLM as the sole decision mechanism. In the context of overloaded initial screening and high variability of language descriptions of experience, this stance is fundamentally justified: key relevance signals are concentrated in project and responsibility context rather than in superficially matching terms, rendering keyword approaches structurally vulnerable and prone to missing relevant candidates. Thus, RAG functions not as a cosmetic interface enhancement but as a shift of the system into a different regime of epistemic discipline: the model ceases to answer from memory and instead answers from an exposed corpus of evidence retrieved by semantic similarity.

The scientific significance of the solution is articulated through the formalization of a dual-loop computation scheme: an asynchronous ingestion phase creates non-parametric memory of the corpus via semantic segmentation and vector indexing. In contrast, the interactive query phase reduces to nearest-fragment retrieval and strict top K context limitation. This decomposition simultaneously addresses the economics of repeated queries, reduces dependence on context window size, and introduces a controlled compromise between recall and noise, which in an HR setting becomes a systemic risk rather than a minor engineering detail. It is separately emphasized that the choice of embedding model and segmentation parameters defines the geometry of the semantic space and, therefore, retrieval quality; for this reason, dense retrieval and sentence-level models are treated as a methodological foundation rather than interchangeable components.

The practical viability of the approach is determined by describing accuracy, explainability, privacy, cost, and latency as interdependent constraints rather than independent desiderata. The most important conclusion is the necessity of complex protocol rules for generation: an answer is permissible only within the boundaries of retrieved context, and under insufficient grounds, the system must refuse; otherwise, the mechanism of plausible fact completion is inevitably activated, which is incompatible with verifiable screening. In this sense, the work establishes a mature engineering-scientific position: reliability in an RAG scenario is achieved not by the strength of the generative model, but by source-control architecture, fragment traceability, and refusal discipline, making the system applicable to real recruiting processes while preserving controllability and reproducibility within the corporate perimeter.

## References

1. Ajjam, M.-H., & Al-Raweshidy, H. S. (2025). AI-driven semantic similarity-based job matching framework for recruitment systems. *Information Sciences*, *724*, 122728. https://doi.org/10.1016/j.ins.2025.122728
2. Bouhsaien, L., & Azmani, A. (2025). Challenges and Strategies in Recruitment: Insights from Digital Transformation. *Lecture Notes in Networks and Systems*, *1310*, 328–340. https://doi.org/10.1007/978-3-031-88653-9_33
3. Gupta, S., Ranjan, R., & Singh, S. N. (2024). A Comprehensive Survey of Retrieval-Augmented Generation (RAG): Evolution, Current Landscape, and Future Directions. *ArXiv*. https://doi.org/10.48550/arxiv.2410.12837
4. Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., & Yih, W. (2020). Dense Passage Retrieval for Open-Domain Question Answering. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 6769–6781. https://doi.org/10.18653/v1/2020.emnlp-main.550
5. Lin, S., Hilton, J., & Evans, O. (2022). TruthfulQA: Measuring How Models Mimic Human Falsehoods. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, *1*, 3214–3252. https://doi.org/10.18653/v1/2022.acl-long.229
6. Ponomarenko, A. (2025). Three Algorithms for Merging Hierarchical Navigable Small World Graphs. *ArXiv*. https://doi.org/10.48550/arxiv.2505.16064
7. Raghavan, M., Barocas, S., Kleinberg, J., & Levy, K. (2020). Mitigating Bias in Algorithmic Hiring. *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, 469–481. https://doi.org/10.1145/3351095.3372828
8. Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese

BERT-Networks. *ArXiv*.
https://doi.org/10.48550/arxiv.1908.10084

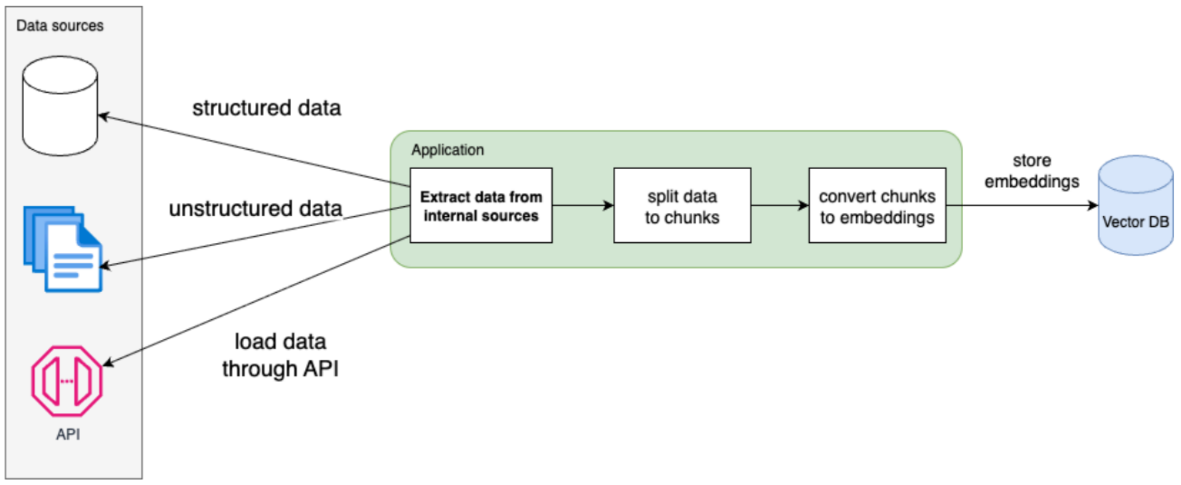


**Fig. 1. Ingestion Phase**

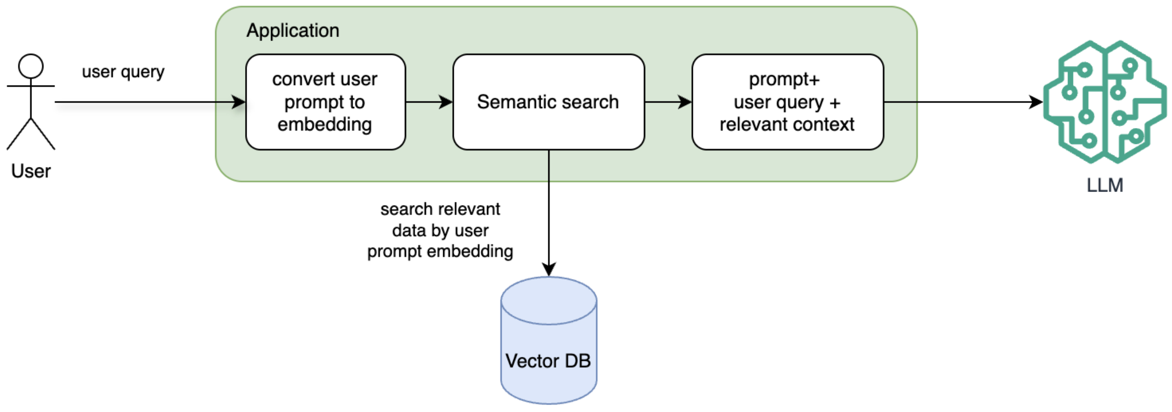
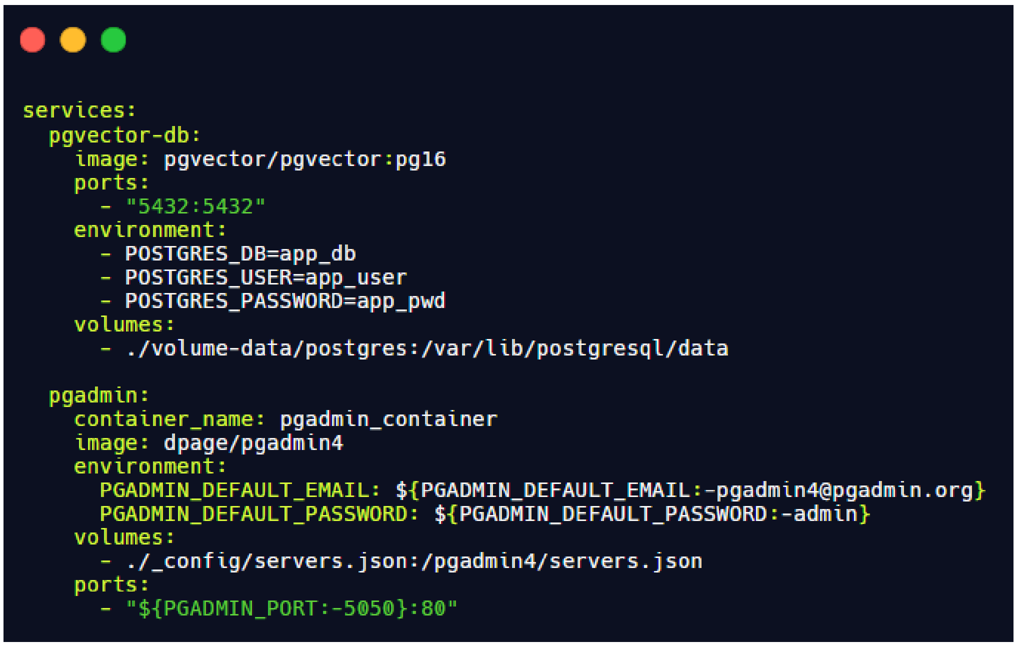
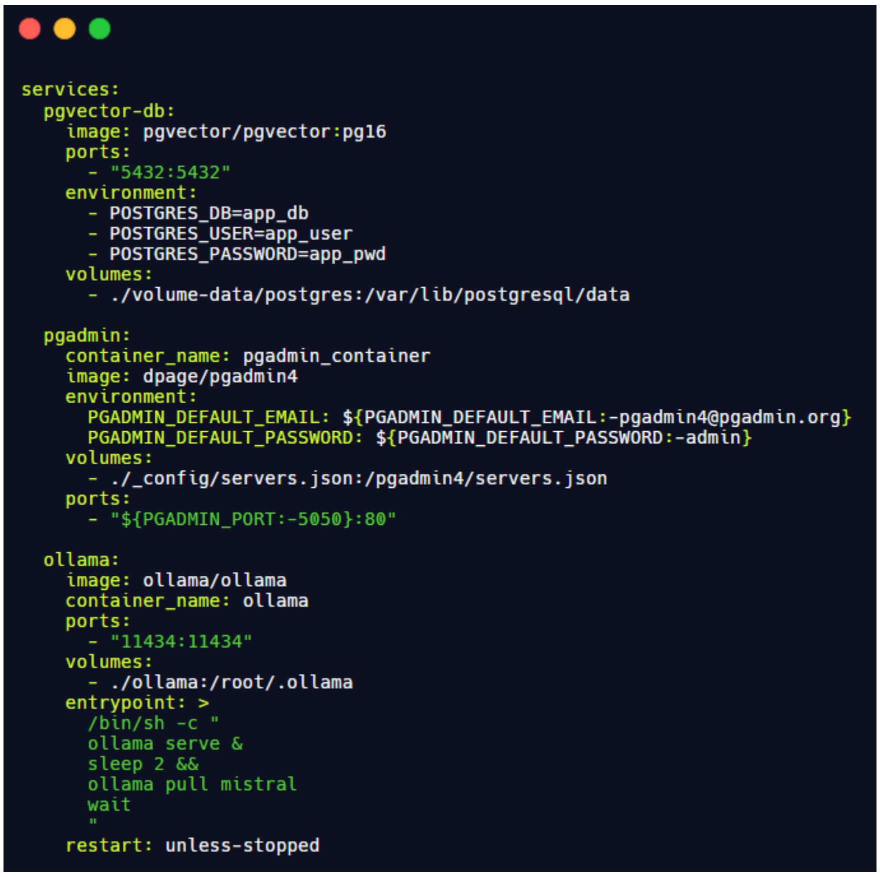

User
user query
Application
convert user prompt to embedding
Semantic search
prompt+ user query + relevant context
LLM
search relevant data by user prompt embedding
Vector DB

**Fig. 2. Intelligent Retrieval**



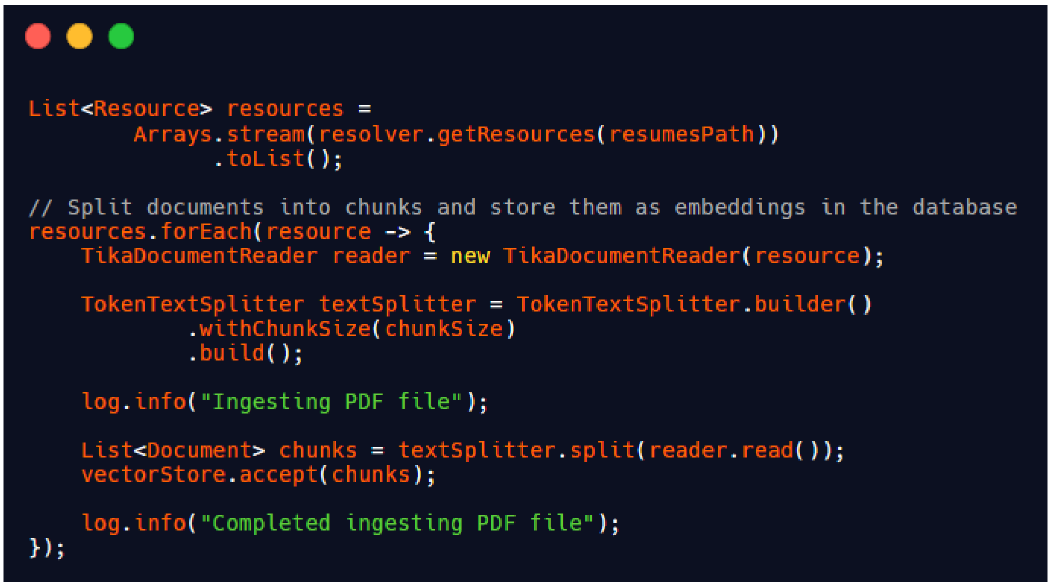
```yaml
services:
  pgvector-db:
    image: pgvector/pgvector:pg16
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=app_db
      - POSTGRES_USER=app_user
      - POSTGRES_PASSWORD=app_pwd
    volumes:
      - ./volume-data/postgres:/var/lib/postgresql/data

  pgadmin:
    container_name: pgadmin_container
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL:-pgadmin4@pgadmin.org}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD:-admin}
    volumes:
      - ./_config/servers.json:/pgadmin4/servers.json
    ports:
      - "${PGADMIN_PORT:-5050}:80"
```

**Fig. 3. Launch pgvector in Docker**



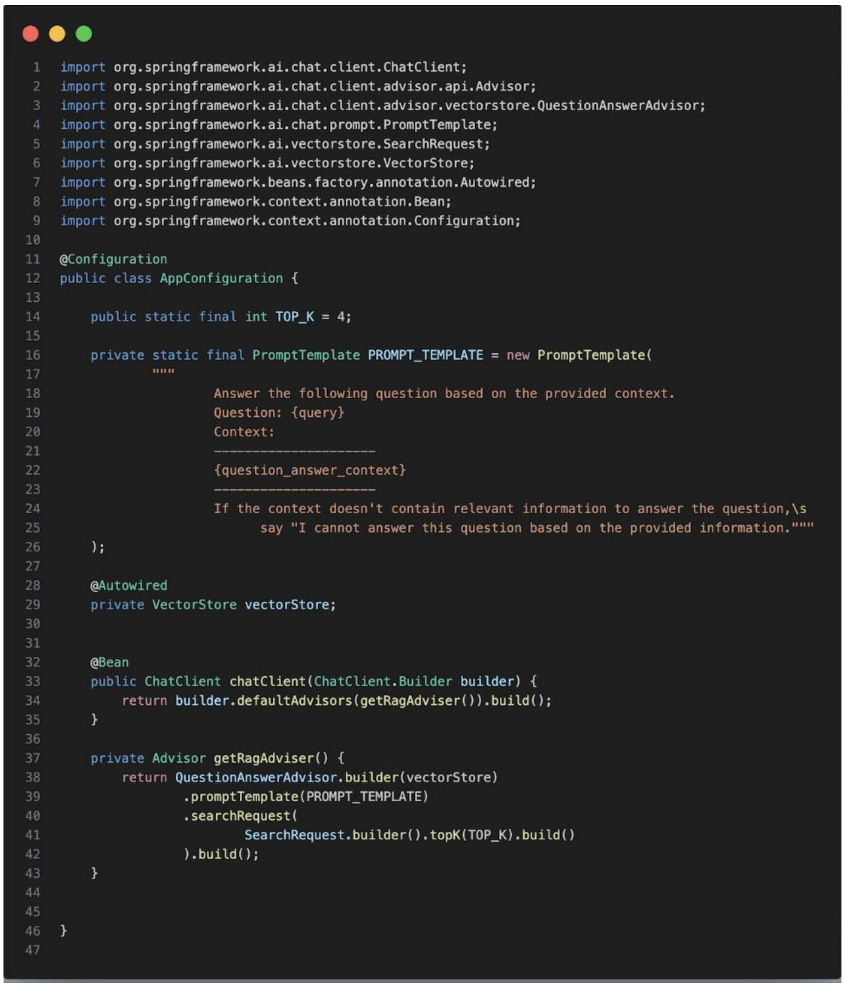
```yaml
services:
  pgvector-db:
    image: pgvector/pgvector:pg16
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=app_db
      - POSTGRES_USER=app_user
      - POSTGRES_PASSWORD=app_pwd
    volumes:
      - ./volume-data/postgres:/var/lib/postgresql/data

  pgadmin:
    container_name: pgadmin_container
    image: dpage/pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: ${PGADMIN_DEFAULT_EMAIL:-pgadmin4@pgadmin.org}
      PGADMIN_DEFAULT_PASSWORD: ${PGADMIN_DEFAULT_PASSWORD:-admin}
    volumes:
      - ./_config/servers.json:/pgadmin4/servers.json
    ports:
      - "${PGADMIN_PORT:-5050}:80"

  ollama:
    image: ollama/ollama
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - ./ollama:/root/.ollama
    entrypoint: >
      /bin/sh -c "
      ollama serve &
      sleep 2 &&
      ollama pull mistral
      wait
      "
    restart: unless-stopped
```

**Fig. 4. Ollama and pgvector in Docker**



```java
List<Resource> resources =
        Arrays.stream(resolver.getResources(resumesPath))
              .toList();

// Split documents into chunks and store them as embeddings in the database
resources.forEach(resource -> {
    TikaDocumentReader reader = new TikaDocumentReader(resource);

    TokenTextSplitter textSplitter = TokenTextSplitter.builder()
            .withChunkSize(chunkSize)
            .build();

    log.info("Ingesting PDF file");

    List<Document> chunks = textSplitter.split(reader.read());
    vectorStore.accept(chunks);

    log.info("Completed ingesting PDF file");
});
```

**Fig. 5. Code example of Ingestion Phase**



```java
import org.springframework.ai.chat.client.ChatClient;
import org.springframework.ai.chat.client.advisor.api.Advisor;
import org.springframework.ai.chat.client.advisor.vectorstore.QuestionAnswerAdvisor;
import org.springframework.ai.chat.prompt.PromptTemplate;
import org.springframework.ai.vectorstore.SearchRequest;
import org.springframework.ai.vectorstore.VectorStore;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfiguration {

    public static final int TOP_K = 4;

    private static final PromptTemplate PROMPT_TEMPLATE = new PromptTemplate(
            """
                    Answer the following question based on the provided context.
                    Question: {query}
                    Context:
                    --------------------
                    {question_answer_context}
                    --------------------
                    If the context doesn't contain relevant information to answer the question,\s
                          say "I cannot answer this question based on the provided information."""
    );

    @Autowired
    private VectorStore vectorStore;


    @Bean
    public ChatClient chatClient(ChatClient.Builder builder) {
        return builder.defaultAdvisors(getRagAdviser()).build();
    }

    private Advisor getRagAdviser() {
        return QuestionAnswerAdvisor.builder(vectorStore)
                .promptTemplate(PROMPT_TEMPLATE)
                .searchRequest(
                        SearchRequest.builder().topK(TOP_K).build()
                ).build();
    }


}
```

**Fig. 6. Core RAG configuration**

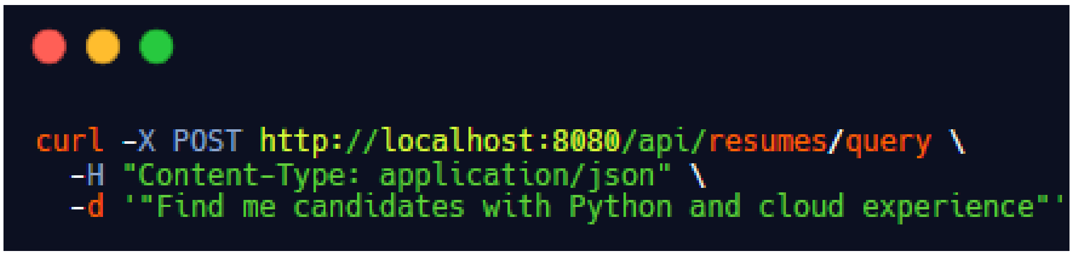

```
curl -X POST http://localhost:8080/api/resumes/query \
  -H "Content-Type: application/json" \
  -d '"Find me candidates with Python and cloud experience"'
```

**Fig. 7. Example of a request**