# RAG InPratica

Lorenzo Leoni
lorenzo.leoni5@studio.unibo.it

*November 2025*

## Abstract

AI RAG InPratica is an intelligent search platform designed to transform how legal professionals interact with complex road circulation regulations. By moving beyond traditional keyword matching, the system utilizes Retrieval-Augmented Generation to allow users to ask natural language questions and receive comprehensive, synthesized answers grounded in specific legal texts.

The solution functions as a non-invasive integration layer, seamlessly synchronizing with legacy document repositories to ensure up-to-date accuracy without disrupting existing infrastructure. It supports two primary operational workflows: a dedicated web interface for editorial staff to refine AI-generated drafts, and an automated background service that integrates directly with the corporate ticketing platform. This dual approach significantly reduces research time and standardizes response quality, enabling legal authors to focus on high-value verification rather than manual information retrieval.

## Disclaimer

*During the preparation of this work, the author used AI tools to assist with code commenting, documentation, and frontend development. After using these tools, the author(s) reviewed and edited the content as needed and takes full responsibility for the content of the final report/artifact.*

## Concept

### Product Type

AI RAG InPratica is an intelligent search system that transforms how legal professionals access and understand the InPratica corpus. Rather than requiring users to manually search through hundreds of documents or rely on exact keyword matching, the system enables natural language questions like "What happens if I drive without a proper driver license for a truck?" and returns comprehensive, synthesized answers with proper citations to source materials. The system provides two complementary access modes: a web application for internal editorial staff that allows reviewing and editing draft responses before publication, and a REST API that integrates with the existing corporate ticketing platform, enabling automatic response proposals directly within the workflow legal authors already use.

A key architectural decision was to design the system as a non-invasive integration layer. The

backend services automatically keep a MongoDB database and Qdrant vector index synchronized with the legacy InPratica system through a polling mechanism and document download service. This approach was necessary because the legacy system does not maintain a centralized document database but rather composes documents on-the-fly from various sources. By mirroring this data into modern storage systems, the architecture enables a gradual migration path away from legacy technologies without disrupting current operations. The same pattern applies to the classification tree monitoring, which synchronizes a hierarchical taxonomy from the legacy system into MongoDB with a 24-hour cache refresh cycle.

The system implements a distributed microservices architecture with three primary layers. The backend services layer comprises two specialized microservices: a .NET processing service that monitors the legacy InPratica system for document changes and processes them into structured paragraphs, and a Python-based RAG (Retrieval-Augmented Generation) API service that orchestrates semantic search using embeddings and large language models. The web application layer provides the user interface through a Vue.js single-page application where users submit queries and receive AI-generated responses. The infrastructure layer relies on a distributed data storage architecture with three distinct database technologies: MySQL as the source of truth for the legacy InPratica system, MongoDB storing processed documents and hierarchical classification metadata in a three-node replica set configuration, and Qdrant providing vector similarity search capabilities using n-dimensional embeddings.

The architecture follows an event-driven design pattern, with RabbitMQ serving as the message broker for asynchronous communication between services.

## Use Cases and User Interaction

The system serves three primary user groups within the organization, each with distinct workflows and interaction patterns in the context of Italian road circulation law documentation.

**Primary Use Case: Editorial Staff Response Workflow** The editorial team at Egaf Edizioni uses the web interface to handle legal queries submitted by our customers. With a typical workload of 2-5 queries per day, editors access the internal web application to submit questions, receive AI-generated draft responses, review and refine the proposed answers, and attach additional supporting documents before final publication. This workflow maintains editorial control and quality assurance while significantly reducing the research time required to formulate comprehensive responses. The web interface is accessible only within the corporate network, protected by firewall rules and proxy.

**Secondary Use Case: Legal Author Ticketing Integration** Legal professionals and contributing authors interact with the system indirectly through the existing corporate ticketing platform. When submitting a Quesito through the ticketing interface, a minimal modification to the platform's workflow redirects the request to the RAG system's API endpoint before forwarding to the ticketing system. The response pipeline processes the query and automatically populates a draft answer directly within the ticket, visible to the author without requiring them to learn a new tool or change their established workflow.

**Tertiary Use Case: Automated Document Synchronization** The system continuously monitors the legacy InPratica system for document changes, detecting additions, updates, or

removals every 5 minutes. This background workflow enables the gradual transition to modern technologies while maintaining operational continuity with legacy systems.

## User Location and Access Patterns

Users access the system from office workstations within the organization's internal network. The web interface is designed exclusively for desktop browser access, with no mobile-specific optimization. User interactions follow a typical request-response pattern: a user submits a question, waits 10-15 seconds for the system to process the query (including semantic search, document retrieval, and LLM generation), and receives a formatted response with citations.
Response times may be slightly longer than those of a typical search, as the system prioritizes higher answer quality and leverages an LLM that, while somewhat slower in inference, provides noticeably greater accuracy.
The system handles a moderate workload of 2-5 legal queries per day through the editorial interface. Legal authors submitting queries through the ticketing platform represent additional usage that flows through the API integration rather than direct web interface access.

The web interface relies on network-level protection through firewall rules, proxy and a JWT-based authentication shared with REST API endpoints through Keycloak integration.

## User Roles and Responsibilities

**End Users:** Primary system users who submit natural language queries to search the legal document corpus. They interact exclusively through the web interface, receiving AI-generated responses with citations to source documents.

**System Administrators:** Technical staff responsible for deploying, monitoring, and maintaining the distributed system. They interact with the system through Docker Compose commands, log files, and monitoring endpoints.

**Developers and Integrators** Technical users who might extend the system or integrate it with other applications. They interact with the system through its REST API endpoints, which accept JSON requests and return structured responses. The API documentation provides OpenAPI specifications accessible through Swagger UI.

# Requirements

## Functional Requirements

### FR1: Natural Language Query Processing

- The system shall accept legal queries in Italian natural language and return synthesized responses with citations to source document paragraphs
- *Acceptance Criteria*: Query "Quali sono gli obblighi per un possessore della CQC?" returns relevant paragraphs with InPratica document references

### FR2: Automatic Document Synchronization

- The system shall detect document additions, updates, and deletions in the MySql DB
- *Acceptance Criteria*: all document changes (additions, updates, deletions) are detected with 99.9% accuracy

**FR3: RTF Document Processing**

- The system shall convert RTF documents to structured and nested paragraphs and preserve hierarchical classification
- *Acceptance Criteria*: RTF document with embedded classification codes produces paragraphs with correct classification

**FR4: Semantic Search**

- The system shall retrieve the top-k most relevant document paragraphs for a given query and support filter by classification taxonomy levels
- *Acceptance Criteria*: Query returns paragraphs ranked by semantic relevance score

**FR5: Draft Response Management (Web Interface)**

- The system shall allow users to review, edit, and approve AI-generated draft responses
- *Acceptance Criteria*: Editor can modify draft text and add document links before generating the final response

**FR6: Ticketing System Integration (API)**

- The system shall expose REST API endpoint accepting query text and populate draft responses in ticketing platform tickets via an email
- *Acceptance Criteria*: the ticketing platform receive an email with all the information of the Quesito and the AI generated response

## Non-Functional Requirements

**NFR1: Availability**

- The system shall maintain 99% uptime during business hours and continue serving queries when legacy DB is unavailable
- *Acceptance Criteria*: the RAG pipeline must work even when the Mysql DB is not available.

**NFR2: Performance**

- The system shall respond to queries within 20 seconds for most of the queries.
- *Acceptance Criteria*: All queries submitted during business hours receive response within 20 seconds

**NFR3: Data Consistency**

- The system shall ensure eventual consistency between InPratica source and search index
- *Acceptance Criteria*: Document modified in InPratica reflects in search results within minutes; MongoDB document count matches InPratica active document count

**NFR4: Data Integrity**

- The system shall prevent data loss during service failures or restarts and maintain idempotent processing to avoid duplicate document indexing
- *Acceptance Criteria*: RabbitMQ message redelivery does not create duplicate MongoDB documents; service restart does not lose in-flight processing tasks

## Implementation Requirements

**IR1: C# .NET Platform**

- Document processing service shall be implemented using C# .NET 9
- *Why*: Organization standardized on .NET platform, more fast and safe in work context

**IR2: Python AI Stack**

- RAG service shall be implemented using Python 3.11+ with FastAPI framework
- *Why*: Python ecosystem provides mature libraries for embeddings, LLM integration, and vector search

**IR3: Docker Containerization**

- All services shall be deployable as Docker containers orchestrated via Docker Compose
- *Why*: On-premises deployment requirement; need for reproducible environments across development and production

**IR4: MongoDB Document Storage**

- Processed documents shall be stored in MongoDB with three-node replica set
- *Why*: Document-oriented data model matches unstructured legal text; replica set provides automatic failover for high availability requirement (NFR1)

**IR5: Keycloak Authentication**

- API endpoints shall use JWT tokens issued by existing corporate Keycloak instance
- *Why*: Organization already operates Keycloak for SSO across applications

## Glossary

- **InPratica**: proprietary document type from Egaf; it is the interpretation and commentary on laws and other regulations by our author.
- **Classification Tree**: Four-level hierarchical taxonomy organizing legal documents by topic area

- **Quesito**: Legal query submitted by users requiring comprehensive documented response

# Design

## Architecture

### Architectural Style: Event-Driven Microservices

The system adopts an event-driven microservices architecture where loosely-coupled services communicate asynchronously through RabbitMq.

### Why Event-Driven Microservices?

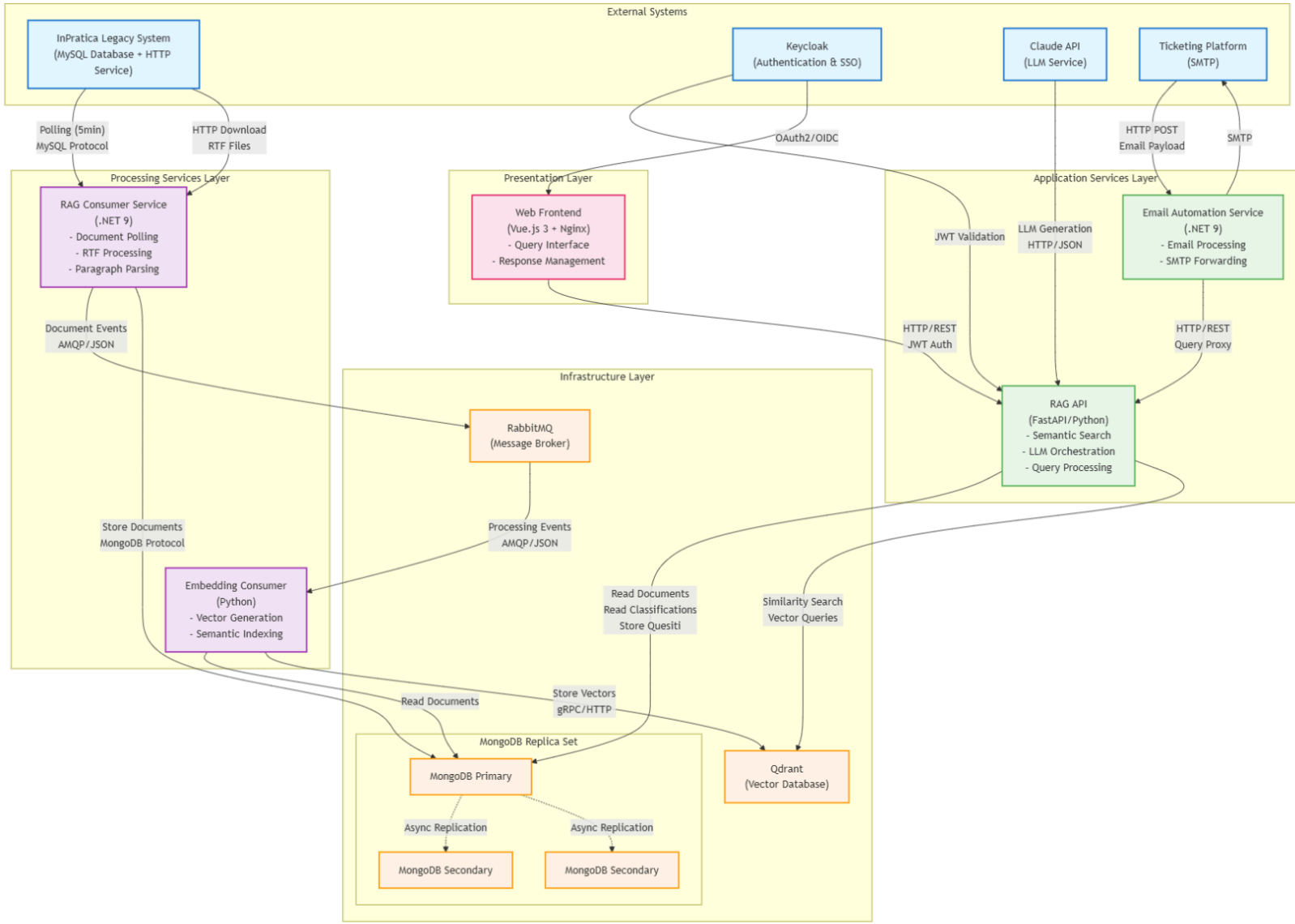
1. **Technology Heterogeneity**: The system integrates two distinct technology stacks - .NET services for document processing (requirement IR1) interfacing with InPratica MySQL database, and Python services for AI operations (requirement IR2) leveraging Hugging Face and LangChain ecosystems. Event-driven communication via JSON messages over AMQP enables seamless interoperability without tight coupling.

2. **Temporal Decoupling**: Document processing and embedding generation operate at different speeds. Document processing handles approximately 8 documents/second during migration operations, while embedding generation processes 1 paragraph/second with transformer model inference. Asynchronous message queues absorb this speed differential, preventing fast producers from overwhelming slow consumers.

3. **Independent Scalability**: Different services have distinct resource requirements. InPraticaPollingService is I/O-bound with low CPU usage, DocumentProcessingService requires high CPU for LibreOffice RTF conversion, EmbeddingConsumer needs high memory for the transformer model, and the RAG API handles variable user-facing query load. Microservices enable horizontal scaling of individual components without scaling the entire system.

4. **Fault Isolation**: Service failures remain localized. If embedding generation fails, document processing continues unaffected. The RabbitMQ Dead-Letter Queue (DLQ) captures failed messages for manual inspection and retry. MongoDB replica set provides automatic failover without service downtime, satisfying NFR1 (99% availability) by tolerating partial system failures.

5. **CAP Theorem Trade-offs**: The system chooses Availability + Partition Tolerance (AP) over immediate consistency. Document changes propagate asynchronously through the pipeline (MySQL → MongoDB → Qdrant) with eventual consistency acceptable for legal document indexing. The InPratica source system itself synchronizes with the publishing database via nightly batch procedures, so minute-level lag in the RAG pipeline is negligible. Daily document modifications average 3-5 changes with major updates occurring approximately once every two days. The system continues operating during network partitions between services, aligning with NFR1's emphasis on availability during disruptions.




*Component diagram*

**Why Choreography Instead of Orchestration?**

The system uses choreography rather than orchestration for coordinating the document processing pipeline. In an orchestrated design, a central orchestrator service would explicitly command each step: "download this document," "process it," "generate embeddings," "index it." Instead, we chose choreography where services publish events about what happened, and other services react independently.

This decision stems from several practical considerations for this specific system:

1. **Linear Pipeline Simplicity**: The document processing flow follows a straightforward chain (detect changes → process document → generate embeddings → index vectors).

2. **Independent Service Evolution**: Each service can evolve its internal logic without updating a central orchestrator.

3. **Fault Tolerance Through Decentralization**: If any service crashes, RabbitMQ retains the unacknowledged messages, and processing resumes when the service restarts.

4. **Event-Driven Semantics**: Publishing events and letting services react aligns with the business reality that we're synchronizing state across systems, not executing a predetermined workflow.

The trade-off is reduced visibility into the overall pipeline state but for our use case with moderate document volume and reliable message processing, this trade-off is acceptable.

## Infrastructure

**Deployed Components:**

The entire system runs as 10 Docker containers managed through Docker Compose.

**1. Data Persistence Layer (4 containers):**

*MongoDB Replica Set*

- *Configuration*: 1 PRIMARY + 2 SECONDARY nodes with `replicaSet`
- *Storage*: Persistent volumes
- *Data*: Processed documents with parsed paragraph, classification tree, polling state, Quesiti response and cited documents

*Qdrant Vector Database*

- *Storage*: In-memory index with disk persistence
- *Data*: 768-dimensional embeddings in legal_docs collection
- *Purpose*: Semantic similarity search for RAG queries (requirement FR4)

**2. Message Broker (1 container):**

*RabbitMQ (rabbitmq_broker)*

- *Exchanges*: document-events, processing-events (type: topic, durable: true)
- *Queues*: document-changes, text-processing + 2 Dead Letter Queues (7-day TTL)
- *Purpose*: Asynchronous event-driven communication between services

**3. Processing Services (2 containers):**

*RAG Consumer Service*: Document change detection and processing.

*Embedding Consumer*: Semantic embedding generation and vector indexing.

**4. Application Services (3 containers):**

*RAG API (rag_api)*

- REST endpoints
- *Authentication*: JWT validation via Keycloak (requirement IR5)
- *Purpose*: RAG query interface and LLM orchestration (requirements FR5, FR6)

*Email Automation Service (email_automation_service)*

- Purpose: Ticketing platform integration via email (requirement FR6)

*Frontend (rag_frontend)*

- Deployment: Nginx static file server
- Purpose: End-user web interface for legal professionals

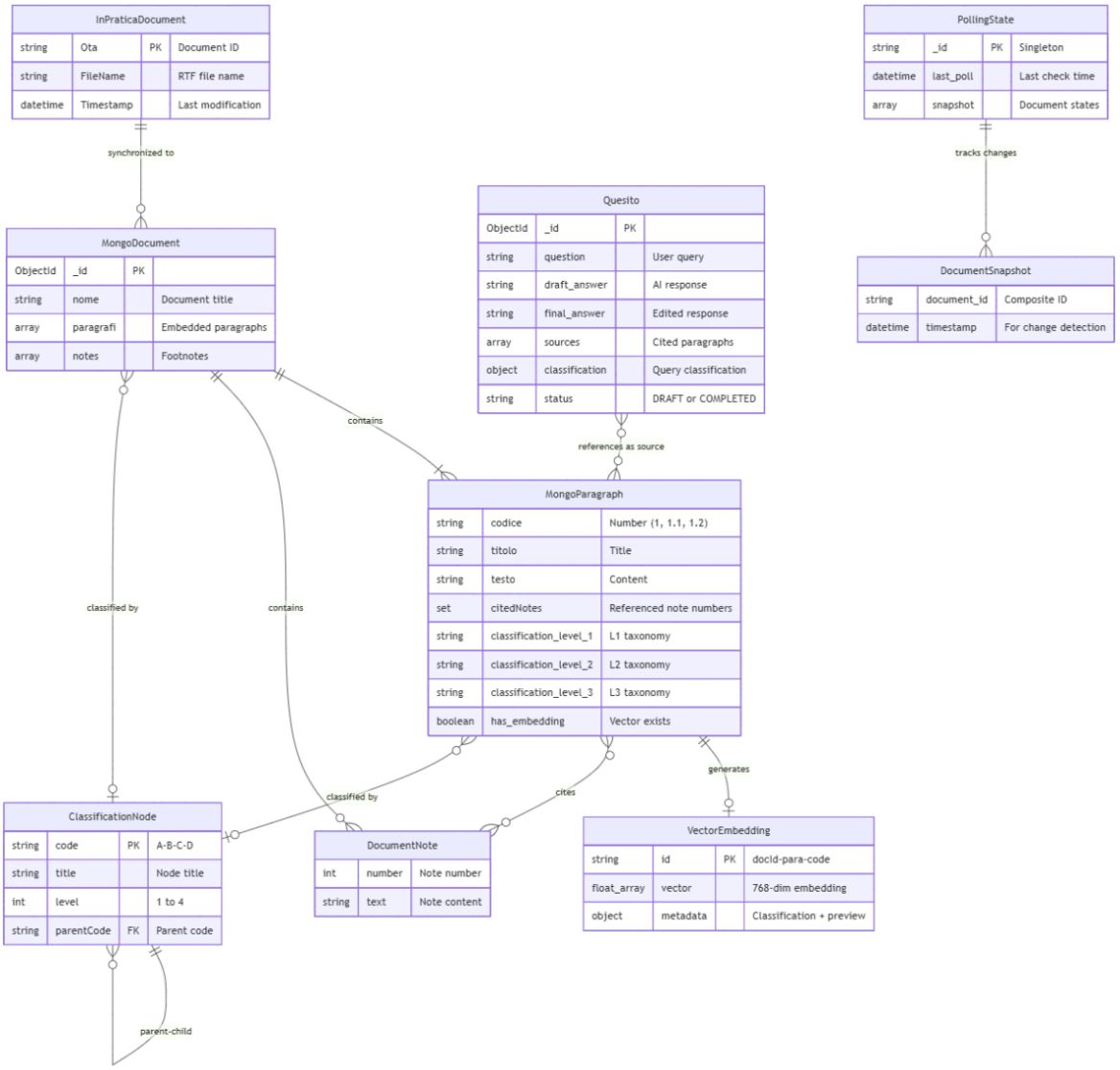
**Network Distribution:**

All 10 containers talk to each other through a private Docker network. Docker's built-in DNS means services can reference each other by name without hardcoding IP addresses. No load balancer since the moderate work load.

**Service Readiness and Health Checks:**

Docker Compose health checks solve service dependencies by letting containers declare when they're truly ready to accept connections. Each service runs a simple diagnostic command periodically.
These health checks create a dependency chain, so Docker Compose won't start dependent services until their dependencies report healthy.

## Modelling




## Domain Model Overview

The system is built around a set of entities that describe the entire lifecycle of documents—from their origin in the legacy system to their use in semantic search and in generating responses to user queries. The previous diagram illustrates how these entities relate to one another.

**InPraticaDocument** entries represent the primary data source. They originate from the legacy system and are read periodically from a read-only MySQL database. Any detected change—creation, update, or deletion—triggers a domain event that starts the downstream processing pipeline.

When a document changes, the **DocumentProcessingService** converts it from RTF into structured text and creates the corresponding **MongoDocument**, stored in the

`raginpratica.docs_v2` collection. This enriched version contains extended metadata, the four-level classification, and the list of extracted paragraphs.

Paragraphs, represented as **MongoParagraph** entities, are the fundamental unit for semantic search. Each paragraph includes a title, content, hierarchical structure, and classification. They are embedded directly within the MongoDocument because they form a single logical entity.

To support RAG workflows, every paragraph produces a vector representation: the **VectorEmbedding**. These vectors—stored in the Qdrant collection, contain both the 768-dimensional embedding and the metadata identifying the source document and paragraph. Their generation is asynchronous and handled by the **EmbeddingConsumer**, which reacts to text-processing completion events.

The system also relies on a structured taxonomy represented by **ClassificationNode** entities. These define the classification tree (2,142 nodes across four levels), imported every 24 hours from the InPratica system and kept cached in memory. This taxonomy is used both to classify paragraphs and to filter search results.

Users interact with the system through the management of **Quesito** entities. Each Quesito contains the user's question, the automatically generated draft response, the final answer, the sources used, and the search metadata.

Finally, the **PollingState** entity tracks the progress of synchronization with the legacy system. It stores the last processed timestamp, how many documents were processed, and how many were deleted.

Overall, domain events exchanged via RabbitMQ orchestrate the workflow without distributed transactions: each step is asynchronous, eventually consistent, and modeled as a domain event. The synchronous part occurs only when users perform searches through the APIs, while the document pipeline runs continuously in the background, maintaining the corpus—currently around 1,500 documents and 120,000 paragraphs, with an equal number of vector embeddings.

## Interaction

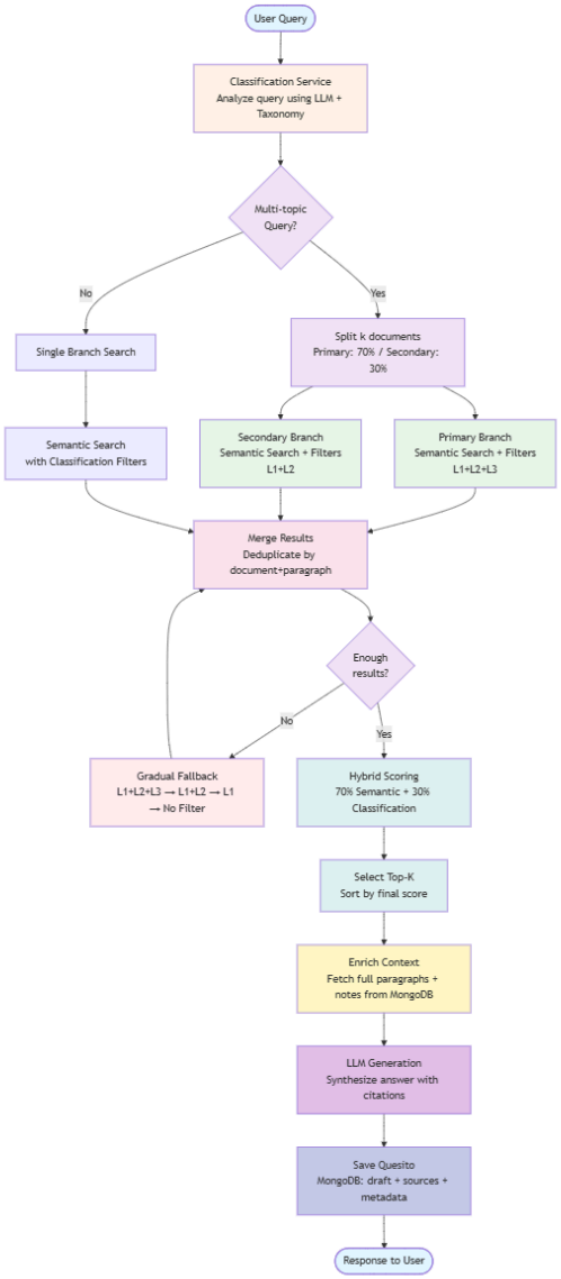
### 1. Document Synchronization Flow (Event-Driven Chain)

This is how documents get from the legacy InPratica system into the modern infrastructure. Every 5 minutes, InPraticaPollingService wakes up and queries the MySQL database looking for documents that have changed since the last check. When it finds changes, it publishes a DocumentChangedEvent to RabbitMQ.
DocumentProcessingService picks up these events, downloads the actual RTF document files from InPratica's HTTP service, and uses LibreOffice to convert them to plain text. It then parses the text into structured paragraphs and stores everything in MongoDB. Once that's done, it publishes a TextProcessingEvent.
EmbeddingConsumer receives that event, reads the paragraphs from MongoDB, and generates 768-dimensional semantic embeddings using a HuggingFace transformer model.
Below you can see the diagram that represents this flow.


Timer: Every 5 minutes
Load Last Snapshot from MongoDB
Poll InPratica MySQL Query current document list
Compare with Snapshot
Changes Detected
Classify Changes ADDED / UPDATED / DELETED
No Changes
Publish Events to RabbitMQ
Update Snapshot Save timestamp
Event Type?
Wait for next cycle
ADDED/UPDATED
DELETED
Download RTF File from InPratica HTTP
Convert RTF to Text LibreOffice Converter
Parse Document Structure Extract paragraphs, hierarchy, notes
Save to MongoDB Upsert document + paragraphs
Delete from MongoDB Delete vectors from Qdrant
Publish Text Ready Event to RabbitMQ
Event Processed
Read Document from MongoDB
For Each Paragraph
Generate Embedding 768-dim vector using HuggingFace
Save Vector to Qdrant with classification metadata
Update has_embedding flag in MongoDB
More paragraphs?
Yes
No
Document Indexed Ready for search

## 2. RAG Query Flow (Synchronous Request-Response)

When a user submits a question through the web interface, it hits the RAG API's endpoint. The API performs a semantic search in Qdrant to find the most relevant paragraph vectors. It fetches the full content and classification metadata from MongoDB, constructs a prompt with this context, and sends it to Claude for response generation. The generated answer gets saved as a Quesito in MongoDB with status "DRAFT", and the response returns to the user with the draft text and source citations.




*Rag flow with query classification.*

**3. Ticketing Integration Flow (Hybrid Sync/Async)**

When a legal author submits a query through the ticketing platform, it calls the Email Automation Service which then proxies the request to the RAG API. The RAG pipeline executes exactly like flow 2, but instead of returning directly to a web interface, the Email Service formats the response as an email and sends it to the ticketing system via SMTP. The ticket gets automatically populated with a draft answer without the author needing to switch tools.

**Communication Patterns:**

The architecture uses three distinct communication patterns depending on the scenario:

- Publish-Subscribe through RabbitMQ for the asynchronous document processing pipeline
- Request-Response over HTTP REST for user-facing queries and external integrations
- Polling the MySQL database every 5 minutes to detect document changes (since the legacy system doesn't support push notifications)

## Behaviour

**Stateful Components:**

MongoDB's replica set maintains document state across three nodes. The PRIMARY node handles all writes while SECONDARYs replicate asynchronously. If the PRIMARY crashes, the remaining nodes hold an election and promote a SECONDARY to PRIMARY - this takes about 30 seconds but happens automatically.

Qdrant keeps the entire vector index in memory for fast similarity searches. When you shut it down, it persists everything to disk and reloads on startup.

RabbitMQ uses durable queues that survive restarts. If the broker crashes with unacknowledged messages in queues, those messages get redelivered when it comes back up.

The PollingStateService maintains a simple in-memory cursor tracking which documents we've already processed. It syncs this to MongoDB after each polling cycle so restarts don't cause reprocessing of old documents.

**Stateless Components:**

DocumentProcessingService treats each RabbitMQ message as an independent task. It only acknowledges the message after successfully writing to MongoDB, ensuring we never lose track of documents that need processing.

EmbeddingConsumer is idempotent - if it processes the same document twice, Qdrant's upsert operation just overwrites the existing vectors rather than creating duplicates.

## Data and Consistency Issues

**Data Storage Strategy:**

| Data Type | Storage | Format | Why |
|---|---|---|---|
| Processed Documents | MongoDB | Document-oriented | Hierarchical paragraphs, variable metadata, flexible schema |
| Vector Embeddings | Qdrant | In-memory + disk | Fast similarity search (HNSW algorithm), 768-dim vectors |
| Source Documents | MySQL (external) | Relational | Legacy system, read-only access |
| Classification Tree | MongoDB + API cache | Document + in-memory | 2142 nodes, 24-hour TTL cache for read-heavy workload |
| Quesiti | MongoDB | Document-oriented | CRUD operations, embedded arrays (fonti, metadatiRicerca) |
| RabbitMQ Messages | Disk (durable queues) | JSON | Survive broker restarts, 7-day DLQ retention |

**Database Access Patterns:**

*MongoDB*

- **Writers**: DocumentProcessingService (docs), RAG API (quesiti, classifications), ClassificationPollingService (classifications)
- **Readers**: RAG API (all queries), EmbeddingConsumer (docs for embedding generation)
- **Concurrency**: Single consumer on document-changes queue eliminates write conflicts. Quesiti collection allows concurrent writes (different documents).
- **Queries**: Find by docId (indexed), classification filtering (4-level hierarchy), full-text search on titles

*Qdrant*

- **Writers**: EmbeddingConsumer exclusively
- **Readers**: RAG API for similarity search (k-NN queries with classification filters)
- **Concurrency**: Idempotent upserts by document ID. No read-write conflicts (MVCC

internally).
- **Queries**: Cosine similarity search with metadata filtering

*MySQL (InPratica)*

- **Writers**: None (read-only access)
- **Readers**: InPraticaPollingService (SELECT Ota, Timestamp WHERE Timestamp > ?)
- **Queries**: Index scan on Timestamp column for incremental polling

**Data Sharing Between Components:**

- **Classification Tree**: Shared via MongoDB. ClassificationPollingService writes once daily, RAG API caches in memory for 24 hours, DocumentProcessingService reads during document parsing.
- **Document Metadata**: Shared via MongoDB docs collection. DocumentProcessingService writes, EmbeddingConsumer and RAG API read.
- **Events**: Shared via RabbitMQ. Producers publish, consumers subscribe independently.

**Consistency Model (CAP Analysis):**

The system prioritizes **Availability + Partition Tolerance** (AP) with eventual consistency:

1. **Eventual Consistency Across Pipeline**: MySQL → MongoDB (5min lag) → Qdrant (seconds lag). Total propagation time: 5-15 minutes (requirement NFR3).

2. **MongoDB Replica Set**: Asynchronous replication from PRIMARY to SECONDARYs. Read preference: PRIMARY (strong consistency for writes). On partition, minority partition rejects writes (sacrifices consistency in minority for availability in majority).

3. **Network Partition Behavior**:

    - If MySQL unreachable: System continues serving queries with existing MongoDB/Qdrant data. Polling resumes when connectivity is restored.
    - If MongoDB PRIMARY fails: Automatic election promotes SECONDARY (~30s).
    - If Qdrant unavailable: RAG queries fail (semantic search unavailable). Document processing continues storing to MongoDB.

4. **Idempotency for At-Least-Once Delivery**: RabbitMQ redelivers unacknowledged messages (NACK on consumer failure). Services implement idempotent processing:

    - DocumentProcessingService: MongoDB upsert by docId
    - EmbeddingConsumer: Qdrant upsert by point ID

## Fault-Tolerance

**Data Replication:**

1. **MongoDB Replica Set**

    - Configuration: 1 PRIMARY + 2 SECONDARYs with majority write concern
    - Replication: Oplog-based asynchronous replication (< 1s lag typical)
    - Failover: Automatic election on PRIMARY failure. SECONDARYs vote, majority elects new PRIMARY (~30 seconds downtime).
    - Recovery: Failed node rejoins as SECONDARY, syncs oplog automatically
    - Why: Satisfies NFR1 (99% availability) and NFR4 (data integrity during failures)

2. **RabbitMQ Durable Queues**

    - Message Persistence: All queues durable, messages written to disk before acknowledging publisher
    - Consumer Acknowledgment: Manual ACK after successful processing. NACK triggers redelivery.
    - Dead Letter Queue: Failed messages (after max retries) routed to DLQ with 7-day TTL for manual inspection
    - Why: Ensures no message loss during broker restarts or consumer crashes

**Retry Mechanisms:**

**RabbitMQ Connection Recovery**
The RabbitMQ .NET client is configured with `automatic recovery` and a 10-second network recovery interval. When a connection drops, the client re-establishes the TCP connection and reconstructs all channels, consumers, and topology, ensuring message flow resumes without manual intervention.

**MongoDB Connection Handling**
The MongoDB driver automatically tracks replica-set topology and routes operations to the current PRIMARY. On transient network faults or leadership changes, it retries operations as needed and rebinds to the new PRIMARY once elected, minimizing query and write disruption.

**HTTP Client Retry (Claude API)**
The RAG API applies an exponential-backoff strategy for upstream HTTP calls. Requests returning rate limiting or server errorsare retried up to three times, with progressive delays.

**Error Handling Strategies:**

| **Failure Scenario** | **Impact** | **Recovery** |
|---|---|---|
| LibreOffice conversion fails | Document not indexed | Message to DLQ, manual retry after fixing document format |

| Failure Scenario | Impact | Recovery |
|---|---|---|
| Embedding generation fails | Paragraph not searchable | Message to DLQ, idempotent retry after model reload |
| MongoDB write fails | Processing halted | RabbitMQ redelivers message, automatic retry |
| Qdrant unavailable | RAG queries fail with 503 | Document processing continues, vectors indexed when Qdrant returns |
| Claude API rate limit | Query delayed | Exponential backoff, retry after cooldown |
| MySQL polling timeout | Synchronization paused | Next polling cycle (5min) retries from last successful timestamp |

## Availability

**Caching Mechanisms:**

The classification tree lives in the RAG API's memory with a 24-hour lifetime. Since this 2142-node taxonomy (~500KB) changes maybe once a week, there's no point querying MongoDB for it on every search request.

Qdrant keeps all 120,000 vectors (about 460MB) in memory using a HNSW index structure. HNSW stands for Hierarchical Navigable Small World, which organizes vectors so you can quickly find nearest neighbors without checking all 120,000 entries. Qdrant periodically snapshots to disk so it doesn't lose everything on restart.

**Network Partition Behavior:**

If the MySQL database becomes unreachable, polling stops but queries keep working with whatever data is already in MongoDB and Qdrant. Since documents only change 3-5 times per day, serving slightly stale data is better than going offline entirely. We chose availability over immediate consistency.

If MongoDB's PRIMARY node loses contact with the SECONDARYs, MongoDB's majority consensus kicks in. Whichever partition has the majority continues accepting writes; the minority partition refuses writes to prevent split-brain scenarios.

If Qdrant becomes unreachable, queries fail with a 503 error. We don't have a fallback search mechanism because returning wrong results would be worse than admitting the system is temporarily unavailable. This leans toward the C (consistency) choice - we'd rather be correct or

unavailable than available but incorrect.

Similarly, if Claude's API goes down, queries fail with a 502 error. We don't have a backup LLM configured, so we're dependent on Anthropic's availability. For the current use case, this is acceptable since users can retry later.

## Security

### Authentication and authorization

Authentication is delegated to Keycloak, the existing corporate SSO platform (requirement IR5). Incoming requests include a JWT issued by Keycloak, and the RAG API validates the token's signature, issuer, and expiration before processing the request.

The system does not implement fine-grained, role-based authorization. All authenticated users are granted full access to all endpoints. This model is justified by the small, homogeneous user group and the presence of network-level perimeter controls, which together provide sufficient access governance for the current scope.

# Implementation

## Network Protocols

### HTTP/1.1 with JSON

All user-facing and inter-service communication uses HTTP/1.1 with JSON payloads. The RAG API service exposes REST endpoints using FastAPI, while the Email Automation Service provides an ASP.NET Core Web API. The stateless nature of HTTP aligns with the CAP theorem trade-offs made earlier, supporting horizontal scaling of the RAG API service when needed.

The web frontend communicates with the RAG API using standard XMLHttpRequest wrapped in Axios, with all requests authenticated via JWT tokens passed in Authorization headers.

### AMQP (Advanced Message Queuing Protocol)

RabbitMQ implements AMQP for all asynchronous event-driven communication between microservices. Messages are published as durable JSON payloads to topic exchanges with routing keys following the pattern `{domain}.{entity}.{action}` . AMQP provides persistent message queues survive broker restarts (satisfying NFR4 data integrity requirements), negative acknowledgments (NACK) enable automatic retry with dead-letter queue routing for failed messages, and the publisher-confirms extension ensures producers know when messages are safely persisted.

The .NET services use RabbitMQ.Client  with automatic connection recovery enabled, while Python services use the aio-pika library for asynchronous consumption. All consumers use manual acknowledgment mode, only ACKing messages after successful processing to

MongoDB or Qdrant. This at-least-once delivery guarantee, combined with idempotent operations on the receiving end, ensures no data loss during service crashes or network partitions.

### SMTP (Simple Mail Transfer Protocol)

The Email Automation Service forwards enriched responses to the ticketing platform using plain SMTP on port 25.

### MySQL Wire Protocol

The polling approach was necessary because the legacy InPratica system does not support triggers or change data capture. Using the timestamp column with an indexed scan keeps query performance acceptable even as the document corpus grows beyond the current documents.

### MongoDB Wire Protocol

All document storage operations use MongoDB's binary wire protocol through official drivers. Write operations use majority write concern to ensure data is replicated to at least two nodes before acknowledging, balancing durability with latency.

## Data Representation

### JSON

All HTTP APIs use JSON for request and response payloads. The choice of JSON simplifies debugging and allows inspection of message queues through RabbitMQ's management UI.

FastAPI automatically serializes Pydantic models to JSON using Python's built-in json module with UTF-8 encoding. ASP.NET Core uses System.Text.Json with case-insensitive property matching enabled to handle variations in client request formats.

MongoDB stores documents that are structured with embedded arrays for paragraphs and notes.

### Binary Vector Format (Qdrant)

Embeddings are stored as binary float32 arrays (768 dimensions × 4 bytes = 3,072 bytes per vector). The sentence-transformers model outputs embeddings as NumPy arrays, which the Qdrant client serializes directly to the binary format expected by the vector database's HNSW index. Metadata associated with each vector (document ID, paragraph number, classification code) is stored as JSON within the same Qdrant point structure, enabling filtered similarity searches.

## Database Querying

**MongoDB Queries**: the system uses both simple find operations and complex aggregation pipelines. The RAG API service executes queries asynchronously using Motor.
All collections have indexes on frequently queried fields:

- *docs* collection: index on `docId` (unique), index on `classificazione.codice`
- *quesiti* collection: index on `createdAt` (descending), compound index on `status + createdAt`
- *classifications* collection: index on `code` (unique)

**Qdrant Vector Search**: similarity searches use cosine distance with optional classification filtering.

**MySQL Read-Only Queries**: the polling service executes a single parameterized SQL query against the InPratica database.

## Component Authentication

## Authentication and Security Overview

The system uses multiple authentication mechanisms. These approaches ensure secure access to the API, the frontend, and the vector database, while keeping internal service communication lightweight within the trusted network environment.

**JWT token validation** is used for all RAG API endpoints. Tokens are issued by the corporate Keycloak instance, and FastAPI's dependency injection layer performs the authentication and validation process. This ensures that only authenticated corporate users can access the API.

On the frontend, **Keycloak OAuth2/OIDC** is integrated using the `keycloak-js` library. The Vue.js application follows the Authorization Code Flow, enabling seamless SSO integration with other corporate systems and providing a consistent login experience for all users.

For vector storage, **Qdrant** uses an API key for authentication. The key—generated via `openssl rand -base64 32` and passed through environment variables—prevents unauthorized access to the vector database.

## Technology Stack Summary

#### Backend Services

*C# .NET 9 Microservices:*

- RagConsumerService: Document processing pipeline (LibreOffice, paragraph parsing, classification enrichment)
- EmailAutomationService: Email processing and SMTP forwarding

*Python 3.11 Services:*

- RAG API: FastAPI application with 15+ endpoints
- EmbeddingConsumer: Standalone service for vector generation

**Frontend**

- Framework: Vue.js 3 with Vite build tool
- Authentication: keycloak-js for OAuth2/OIDC
- Deployment: Nginx static file server in Docker

**Data Storage**

- MongoDB: Three-node replica set for document storage
- Qdrant: Vector database with HNSW index
- MySQL: Legacy InPratica system (read-only access)
- RabbitMQ: Message broker with management plugin

**Development Tools**

- Docker Compose: 10-container orchestration
- Git: Version control using git-flow style

**LLM Provider**

- Primary: Claude Opus 4 via Anthropic API
- Architecture: Abstraction layer using LangChain BaseChatModel enables switching to OpenAI or Ollama without code changes
- Configuration: Provider selected via `config.json` with factory pattern

# Validation

## Automatic Testing

### Unit Testing (C# Services)

All .NET microservices use xUnit with Moq and FluentAssertions. Tests organized in three levels:

*Unit Tests:* Isolated component testing with mocked dependencies

- EmailParserTests: 12 tests for HTML parsing, regex extraction, edge cases
- SmtpSenderTests: 8 tests for SMTP operations, connection failures, timeouts
- RagApiClientTests: 10 tests for HTTP communication, error handling
- ParagraphParserTests: 15 tests for document structure parsing

*Integration Tests:* Multiple components together

- EmailPipelineOrchestratorTests: 12 tests for full workflow (parse → RAG → enrich → SMTP)
- DocumentProcessingServiceTests: 8 tests for MongoDB and RabbitMQ integration

*API Tests:* Controller-level testing

- EmailProcessingControllerTests: 6 tests for HTTP endpoints

Run: `dotnet.exe test --nologo --verbosity minimal`

### End-to-End Testing

Python scripts validate cross-service integration:

*test_e2e_pipeline.py:* Document processing flow

- Publishes DocumentChangedEvent to RabbitMQ
- Verifies RagConsumerService processes message
- Validates document structure in MongoDB

*test_e2e_embedding_pipeline.py:* Vector generation flow

- Inserts test document, triggers EmbeddingConsumer
- Verifies embeddings in Qdrant

Run: `python3 scripts/test_e2e_pipeline.py`

## Acceptance Testing

### Manual Testing

*Keycloak Integration:* OAuth2 login, token refresh, role-based authorization

*SMTP Delivery:* Email rendering in Outlook/Gmail, BCC verification

*Document Migration:* all 1,500 documents migrated

# Release

## Module Organization

The system is organized as **10 independent Docker containers** rather than a monolith:

- Infrastructure layer
  - MongoDB replica set nodes
  - RabbitMq, message broker
- Data layer
  - Qdrant
- Application layer
  - rag_consumer: .NET document processor
  - embedding_consumer: Python vector generator
  - rag_api: FastAPI backend service
  - email_automation: .NET email processor
  - web: Vue.js frontend (Nginx)

## Distribution Strategy

**Docker Images:** Each service builds its own Docker image locally:

- .NET services: Multi-stage Dockerfile (build with SDK, run with runtime)
- Python services: Single-stage with Poetry dependency management
- Frontend: Node build stage + Nginx runtime

The deployment uses local builds orchestrated by docker-compose.

# Deployment

## Prerequisites

- Docker Engine
- Git
- Ports available: 5672, 15672, 27017-27019, 6333, 8000, 3000, 8002

## Installation Steps

### 1. Clone Repository

```
cd /your/deployment/path

git clone <repository-url> AI-RagInPratica

cd AI-RagInPratica
```

### 2. Configure Environment Variables

```
cp .env.example .env

nano .env
```

### 3. Start Application Services

```
docker compose -f docker-compose.all.yml up -d
```

### 4. Verify Deployment

```
docker compose -f docker-compose.all.yml ps
```

Test endpoints:

- RabbitMQ Management: http://localhost:15672 (user/password from .env)
- RAG API Health: http://localhost:8000/health
- Frontend: http://localhost:3002

### 5. Run Document Migration (One-Time)

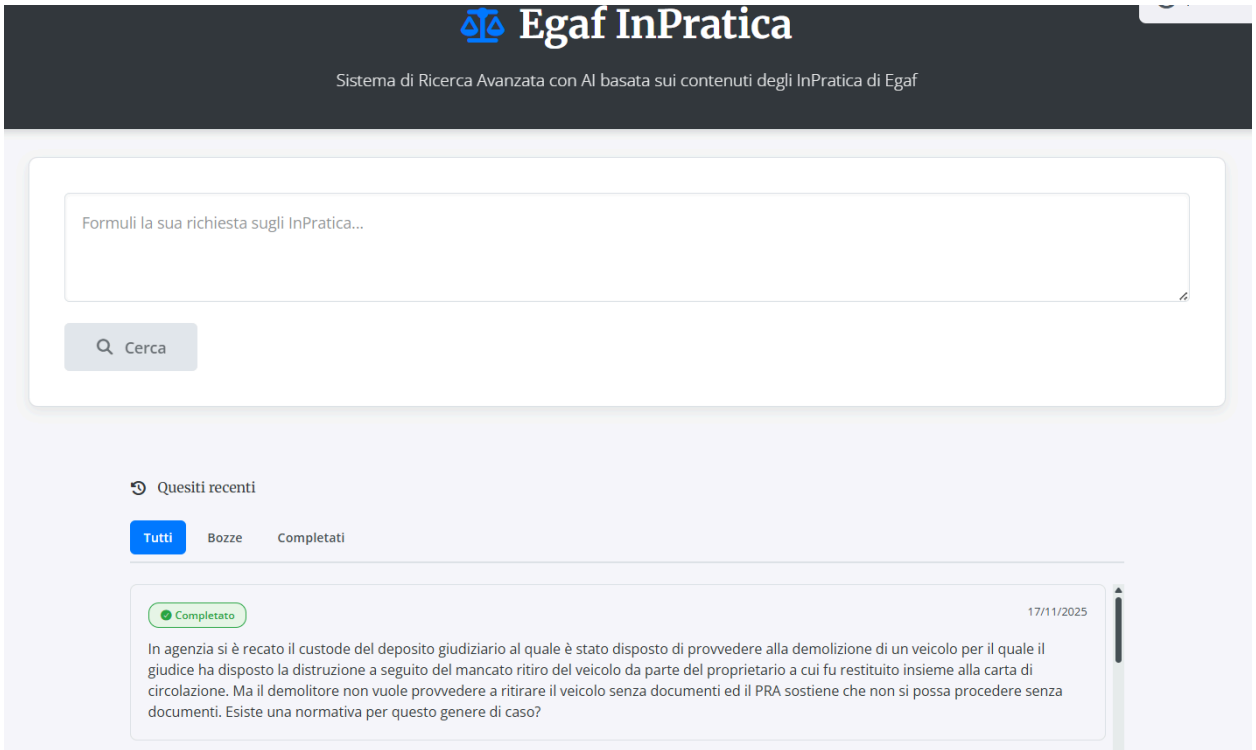
```
docker exec -it rag_consumer dotnet DocumentMigrationTool.dll
```

The system will start publishing messages on Rabbit and indexing documents to Qdrant via the EmebeddingConsumer. This process can be CPU/GPU intensive and may take several hours since there are about 120,000 paragraphs. To avoid overloading both the production and test machines with a full massive import and vector creation, we decided to copy the existing vector directory onto the machines and mount it directly inside the Docker containers. In this way, only a few new documents modified or added each day (around 4-5) need to be processed, which we have verified does not cause any performance issues.
After that the system will be ready.

# User Guide

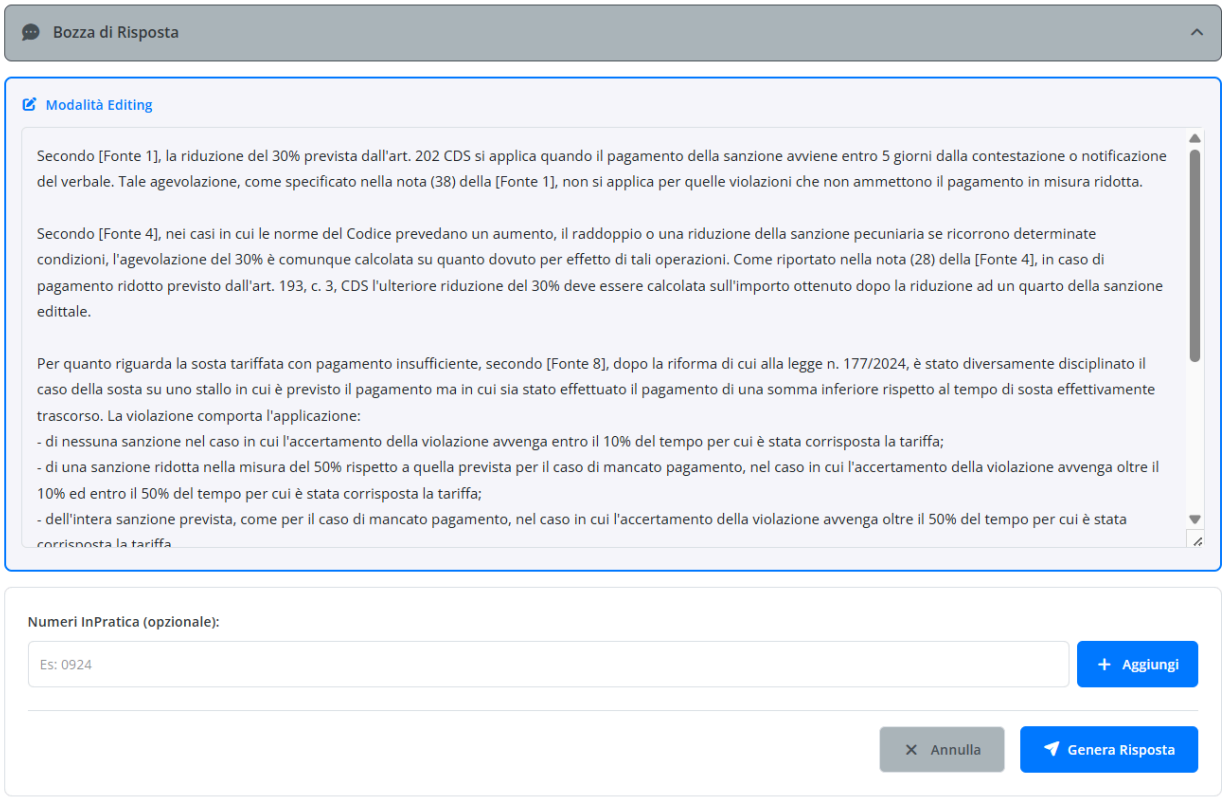
## Accessing the System

After logging into the system, the user is presented with the main dashboard. In the upper section, it is possible to enter a new query or search request, while the lower section displays previously submitted questions and allows the user to complete those still in draft status.




*Initial dashboard, used to search, see and edit Quesiti*

When querying the system or completing a draft answer, the user is redirected to a page where the proposed draft can be reviewed and edited. Additional documents can also be attached through a search over the MongoDB collection containing all *InPratica* records. Once the user selects **Genera Risposta**, the system performs the last processing step and produces the final response in Egaf style, including citations and a formal, technical tone.




*Edit draft response and add new InPratica*

Within the draft section, when not in editing mode, it is possible to click on the cited sources to read the exact referenced paragraph or the full document.

## Email Automation

Legacy systems can trigger automated responses via HTTP POST:

```
curl -X POST http://email-automation:8002/api/email/process \

-F "htmlBody=<b>Testo quesito</b>: La mia domanda..." \

-F "from=gruppo@egaf.it" \

-F "to=lorenzo.leoni@egaf.it" \

-F "subject=Quesito Automatico" \

-F "attachment=@documento.pdf"
```

# Self Evaluation

## Project Outcomes and Development Experience

This project successfully transitioned from proof-of-concept to production system. The key success factor was implementing a non-invasive integration strategy that preserved existing workflows while adding AI capabilities to legacy systems. This approach enabled deployment without modifications to production codebases that have operated for years.

The most significant technical challenge involved processing unstructured data from heterogeneous sources. The legacy InPratica system stores documents as RTF files composed dynamically from multiple sources, with no centralized database or standardized schema. Developing a robust parsing pipeline required extensive testing with real production documents and multiple iterations to achieve reliable processing of approximately 1,500 documents with varying structural patterns.

This project provided the opportunity to design and deploy a production-grade distributed system with full autonomy over technical decisions. Unlike academic projects where complexities can sometimes be avoided, this required addressing real-world challenges: integrating with undocumented legacy databases, handling production data quality issues, securing JWT-based authentication with corporate infrastructure, and ensuring zero data loss through idempotent message processing. Working independently through the full software lifecycle - from requirements gathering to production deployment and monitoring - was a really great experience.

## Product Strengths and Weaknesses

The system's main strengths lie in its architectural approach. The non-invasive integration preserves existing workflows while significantly reducing research time for editorial staff through natural language query interfaces. Automatic draft generation through ticketing platform integration streamlines author workflows without requiring behavioral changes. Comprehensive test coverage and Docker-based deployment provide reliability and reproducible environments across different contexts.

The single-consumer architecture on the document-changes queue creates a potential bottleneck for high-volume scenarios, and the in-memory Qdrant index limits horizontal scaling options. The 5-minute polling interval introduces eventual consistency lag, and manual intervention is required for documents that fail LibreOffice conversion.

## Future Development Directions

The hierarchical classification tree (2,142 nodes across 4 levels) represents an underutilized asset currently used only for document indexing. Future work will implement specialized agents capable of intelligently navigating this taxonomy to improve retrieval precision by traversing the classification hierarchy before performing vector similarity operations.

A new requirement has emerged to incorporate sanctions and sanctioning scenarios from a

separate database system. The existing migration architecture provides a proven pattern that can be replicated for this new data source, demonstrating the system's flexibility for incorporating additional legal content types.

More broadly, the document synchronization pipeline developed for InPratica will serve as a foundation for modernizing other data sources within the organization. With multiple legacy systems storing unstructured data in various formats, establishing MongoDB and Qdrant as a unified data platform represents a crucial step toward Legal Tech infrastructure modernization. The patterns implemented - polling-based change detection, event-driven processing, and idempotent operations - provide reusable components for future migration initiatives.