



UNIVERSITÀ DEGLI STUDI DI FIRENZE  
SCUOLA DI INGEGNERIA - DIPARTIMENTO DI INGEGNERIA  
DELL'INFORMAZIONE

---

Tesi di Laurea Triennale in Ingegneria Informatica

**SVILUPPO DI SISTEMA DI INDICIZZAZIONE E  
RICERCA SEMANTICA DI DOCUMENTI TESTUALI**

*Candidato*  
Alban Haka

*Relatore*  
Prof. Marco Bertini

---

Anno Accademico 2024/2025

# Indice

<b>Sommario</b>	<b>i</b>
<b>1 Introduzione</b>	<b>1</b>
1.1 Contesto e Motivazioni . . . . .	1
1.2 Problema Affrontato e Proposta Solutiva . . . . .	2
1.2.1 Formulazione del Problema . . . . .	2
1.2.2 Ambito e Delimitazioni del Progetto . . . . .	2
1.2.3 Requisiti Funzionali . . . . .	3
1.2.4 Attori Coinvolti . . . . .	4
<b>2 Strumenti e Metodologie utilizzate</b>	<b>5</b>
2.1 Elaborazione dei Documenti . . . . .	5
2.1.1 Docling . . . . .	5
2.1.2 OCR con Modello DeepSeek . . . . .	8
2.2 Retrieval basato su Embeddings . . . . .	10
2.2.1 Rappresentazioni Vettoriali: gli Embeddings . . . . .	10
2.2.2 Modelli Sentence Transformers . . . . .	11
2.2.3 Database vettorizzati, ChromaDB . . . . .	13
2.3 Framework e Strumenti di Sviluppo . . . . .	14
2.3.1 Django Framework . . . . .	15

2.3.2	Celery e Redis per Task Asincroni . . . . .	16
2.3.3	Infrastruttura GPU Remota con Lambda Cloud . . . . .	19
2.3.4	PDF.js per Visualizzazione Documenti . . . . .	20
<b>3</b>	<b>Architettura del Sistema</b>	<b>21</b>
3.1	Overview . . . . .	21
3.1.1	Componenti e Ruoli . . . . .	23
3.1.2	Relazioni tra Componenti . . . . .	25
3.1.3	Organizzazione del Progetto . . . . .	28
3.2	Struttura Logica del Progetto . . . . .	29
3.2.1	App Users . . . . .	29
3.2.2	App Doc_Manager . . . . .	29
3.2.3	Pipeline RAG . . . . .	31
3.2.4	Risorse Statiche . . . . .	34
3.2.5	Page Nativagation Diagram . . . . .	34
3.3	Database . . . . .	35
3.3.1	Tabella auth_user . . . . .	36
3.3.2	Tabella users_profile . . . . .	36
3.3.3	Tabella doc_manager_document . . . . .	37
3.3.4	Tabella docseek_collection (ChromaDB) . . . . .	37
3.4	Implementazione Lambda Cloud . . . . .	38
3.4.1	Architettura del Server GPU . . . . .	39
3.4.2	Modello DeepSeek-OCR . . . . .	39
3.4.3	API REST del Server . . . . .	40
3.4.4	Processo di Elaborazione . . . . .	40
3.4.5	Integrazione con Django . . . . .	41
3.4.6	Gestione della Connattività . . . . .	42

<b>4 Risultati</b>	<b>43</b>
4.1 Panoramica dell’Interfaccia Utente . . . . .	43
4.1.1 Pagina di Autenticazione . . . . .	44
4.1.2 Home Page . . . . .	45
4.1.3 Pagine Uploader . . . . .	46
4.1.4 Pagina Searcher . . . . .	48
4.2 Ricerca Semantica: Esempi Pratici . . . . .	48
4.2.1 Test di Ricerca su Documento Digitale Nativo . . . . .	48
4.2.2 Test di Ricerca su Documento Scansionato (OCR) . . .	54
4.2.3 Indicatore di Rilevanza Semantica . . . . .	56
4.3 Test Sperimentale del Modello OCR . . . . .	57
4.3.1 Performance Medie Complessive . . . . .	57
4.3.2 Distribuzione della Qualità di Estrazione . . . . .	58
4.3.3 Analisi Dettagliata per Documento . . . . .	59
4.4 Considerazioni sui Risultati . . . . .	61
4.4.1 Riepilogo dei Test Effettuati . . . . .	61
4.4.2 Analisi dei Tempi di Elaborazione . . . . .	62
4.4.3 Analisi dei Chunks Generati . . . . .	64
4.4.4 Qualità dell’Estrazione . . . . .	64
4.4.5 Limitazioni Osservate . . . . .	65
<b>Bibliografia</b>	<b>66</b>

# Sommario

Il presente lavoro di tesi si concentra sullo sviluppo di un **Sistema di Indicizzazione e Ricerca Documentale Semantica** con l'obiettivo di abbattere i tempi di consultazione in contesti ad alto volume documentale, come gli archivi di studi medici o legali, spesso composti da referti e analisi scansionati in formato PDF.

Pertanto, è stata progettata una **pipeline di analisi semantica** che sfrutta l'**Intelligenza Artificiale** per processare e comprendere il significato contestuale non solo del **testo**, ma anche di **tabelle e immagini** all'interno dei documenti.

Attraverso la creazione di **rappresentazioni vettoriali** (*embedding*), il motore di ricerca confronta la richiesta dell'utente (*prompt*) con i documenti in base al loro significato contestuale. Ciò consente il recupero di informazioni anche da sinonimi o concetti correlati, trasformando i vecchi e noiosi flussi di lavoro manuali in un semplice ed intuitivo motore di ricerca digitale ed intelligente. L'elaborato descrive l'architettura di sistema, l'implementazione della **pipeline RAG**, il design e le scelte progettuali adottate, con lo scopo di ottenere un'applicazione ottimizzata in termini di tempo ed efficienza.

# Capitolo 1

## Introduzione

### 1.1 Contesto e Motivazioni

Nel corso degli ultimi decenni, la quantità di dati digitali è aumentata in modo **esponenziale**, il che rende la gestione e la consultazione dei dati estremamente impegnativa. Spesso, gli utenti devono consultare un gran numero di file e documenti di **grandi dimensioni**, ciascuno dei quali potrebbe contenere **centinaia di pagine**. La ricerca manuale è un processo meccanico ed è molto costosa in termini di tempo, che potrebbe essere impiegato in attività più produttive. Inoltre, i metodi tradizionali di ricerca, che si basano sulla corrispondenza precisa della **parola chiave** o della formulazione del testo nella barra di ricerca (il noto tasto *Ctrl + F*), funzionano solo se l'utente ricorda con precisione la parola o la formulazione del testo da cercare. Tuttavia, quando si utilizzano sinonimi o si cerca un concetto di cui non si conosce la terminologia esatta (ad esempio, chiedendosi "*dove si parla dell'argomento X?*"), si riscontra il limite di questo strumento.

## 1.2 Problema Affrontato e Proposta Solutiva

### 1.2.1 Formulazione del Problema

Il presente lavoro di tesi si propone di affrontare il problema **dell'inefficienza** nei processi di ricerca e recupero delle informazioni. Tali difficoltà si manifestano soprattutto in contesti con un'elevata numero di documenti da gestire. In questi scenari, la ricerca dei documenti e delle informazioni contenute al loro interno comporta ad un'enorme spreco di tempo. Continuare con metodi manuali nel 2025 è inconcepibile, considerando le innumerevoli soluzioni tecnologiche disponibili.

### 1.2.2 Ambito e Delimitazioni del Progetto

L'analisi di questo lavoro si concentra nello sviluppo di una **pipeline di tipo RAG<sup>1</sup>**, che si distribuisce nelle seguenti fasi:

1. **Processamento:** Estrazione del testo, tavole e immagini dai file **PDF** tramite tecniche di *OCR* e librerie Python, seguita dalla **segmentazione in chunk**.
2. **Indicizzazione Semantica:** Creazione degli *embeddings* per ogni chunk e loro indicizzazione in un *database vettoriale*.
3. **Ricerca:** Fase in cui il sistema recupera i frammenti più rilevanti rispetto alla richiesta dell'utente e fornisce la risposta finale.

Pertanto, l'obiettivo principale consiste nella realizzazione di un motore di ricerca documentale accessibile tramite un'applicazione dedicata, volta a offrire all'utente una gestione efficace dei documenti e una ricerca intelligente.

---

<sup>1</sup>Retrieval-Augmented Generation, architettura che genera una risposta basata sulla conoscenza dei documenti forniti in input

### 1.2.3 Requisiti Funzionali

Il sistema deve soddisfare i seguenti requisiti funzionali primari:

- **Gestione Upload Documenti:** Il sistema deve consentire agli utenti con permesso di upload di caricare documenti in formato PDF, inclusi sia i file digitali che quelli scansionati.
- **Selezione Metodologia di Processamento:** L'utente responsabile dell'upload deve poter selezionare la metodologia di pre-elaborazione del file in base alla sua tipologia:
  1. Processamento tramite **OCR** (Optical Character Recognition) per i file scansionati.
  2. Processamento standard tramite **Docling** per i file nativi.
- **Funzionalità di Ricerca Semantica:** Il sistema deve permettere agli utenti con permesso di ricerca di interrogare il database mediante *prompt* e ricevere risultati basati sulla rilevanza semantica.
- **Gestione Eliminazione Documenti:** Il sistema deve fornire una funzionalità che consenta l'eliminazione dei documenti caricati e la conseguente rimozione dall'indice semantico.
- **Gestione Rinominazione Documenti:** Il sistema deve fornire una funzionalità che consenta di rinominare i documenti caricati.
- **Visualizzazione Contestuale dei Risultati:** Una volta effettuata la ricerca, l'utente deve poter cliccare un bottone (*View*) che apre il documento PDF sorgente alla pagina corrispondente al frammento di testo più rilevante.

### 1.2.4 Attori Coinvolti

Gli attori che interagiscono con il sistema sono:

- **Uploader:** Utente con l'autorizzazione a inserire nuovi documenti nell'archivio. Le sue responsabilità includono la gestione (caricamento, eliminazione e rinominazione) dei file e la scelta della metodologia di processamento (OCR, Docling) per la successiva indicizzazione semantica.
- **Searcher:** Utente autorizzato all'interrogazione del sistema. Il suo ruolo è incentrato sull'esecuzione di ricerche semantiche tramite *prompt* e sulla visualizzazione dei risultati del file PDF sorgente.

Nota: un utente può possedere entrambi i ruoli, svolgendo sia il ruolo di Uploader che quello di Searcher a seconda dell'azione richiesta.

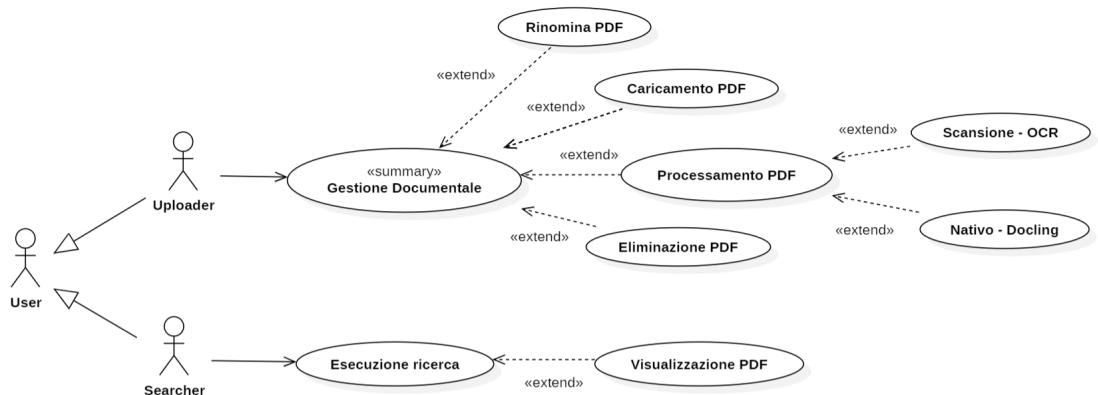


Figura 1.1: Use Case Diagram

Il seguente diagramma 1.1 illustra le interazioni tra i due attori principali e i casi d'uso fornendo una visione completa delle funzionalità del sistema.

# **Capitolo 2**

## **Strumenti e Metodologie utilizzate**

### **2.1 Elaborazione dei Documenti**

Questa sezione analizza le tecnologie utilizzate per il processamento delle diverse tipologie di PDF in rappresentazioni strutturate adatte all'indicizzazione semantica.

#### **2.1.1 Docling**

I documenti PDF digitali, sebbene contengano testo selezionabile, presentano una struttura complessa che va oltre la semplice sequenza di caratteri. Elementi come tavole, intestazioni, paragrafi, elenchi e immagini contribuiscono al significato complessivo del documento e devono essere preservati durante le operazioni di estrazione.

## Limiti delle librerie tradizionali

Le librerie Python più comuni per l'elaborazione di PDF presentano diverse limitazioni:

- **PyPDF2**: estrae esclusivamente testo grezzo senza preservare la struttura del documento. Le tabelle vengono linearizzate in modo incoerente e gli elementi grafici vengono ignorati.
- **pdfplumber**: offre un migliore supporto per l'estrazione delle tabelle, ma richiede una configurazione manuale e non identifica automaticamente la struttura gerarchica del documento.
- **PyMuPDF**: sebbene sia veloce ed efficiente, fornisce principalmente informazioni a livello di pagina senza un'analisi semantica della struttura.

Queste librerie risultano buone per l'estrazione di testo semplice, ma si rivelano inadeguate quando è necessaria una **comprendione strutturata** del documento per applicazioni di retrieval semantico.

## Docling

Per superare le precedenti limitazioni, nel progetto è stato adottato Docling [7], un framework open-source sviluppato da IBM Research. Docling non è un semplice *parser*, ma è un sistema che combina *Document Understanding* e analisi con modelli di *Computer Vision*. Il suo punto di forza risiede nell'utilizzo di modelli di intelligenza artificiale specializzati per l'analisi del layout e il recupero strutturale:

- **Dataset DocLayNet**: consiste in un dataset curato da IBM che include documenti complessi (report finanziari, articoli scientifici, manuali

tecnici), utilizzati per addestrare i modelli di segmentazione di Doceling. Questo consente una classificazione precisa ed affidabile degli elementi della pagina identificando le vari sezioni di una pagina(Titolo, Paragrafi, Indici, Tabelle, Immagini).

- **Struttura di Riconoscimento Tabellare:** Per la gestione delle tabelle Doceling utilizza un modello avanzato chiamato **TableFormer**. Questo modello permette di estrapolare la struttura logica della tabella e le relazioni tra le celle, superando il limite dei parser basati sulle righe e colonne.
- **Approccio Ibrido:** Doceling adotta un approccio "*ibrido*". Quando possibile estrae le informazioni direttamente dai metadati in modo da non commettere errori ed avere il massimo della precisione. Mentre, quando i metadati non sono sufficienti utilizza i modelli di riconoscimento ottico.
- **Gestione di immagini e figure:** estrae le immagini dal documento associandole alle relative didascalie, quando presenti, permettendo di mantenere il contesto visivo.

## Pipeline di elaborazione con Doceling

Il processo di elaborazione mediante Doceling segue i seguenti passaggi:

1. **Ingestione:** Il documento viene processato da modelli di *Object Detection* che generano *bounding box* per ogni elemento logico, segmentando la pagina in aree semantiche.
2. **Recupero Tabelle e Immagini:** Le aree identificate come tabelle vengono processate dal modulo TSR (Struttura Riconoscimento Ta-

bellare) per ricostruirne la griglia logica. Parallelamente, le immagini vengono estratte e associate alle relative didascalie per preservare il contesto visivo.

3. **Costruzione del Docling Document:** I dati estratti vengono normalizzati in una rappresentazione strutturata (rappresentazione ad albero). Ogni nodo dell'albero contiene:

- *Etichetta Semantica*: (testo, tabella, immagine, intestazione, elenco, ecc.).
- *Payload*: Il contenuto testuale o i dati strutturati della tabella.
- *Metadati*: Coordinate originali e numero di pagina (fondamentali per citare le fonti nelle risposte del sistema).

4. **Serializzazione:** Infine, il documento strutturato viene esportato diversi formati tra cui *Markdown* o *JSON*. Questi formati di output sono ottimi per la successiva fase di *chunking*, permettendo di suddividere il testo rispettandone i confini logici (evitando ad esempio di spezzare una tabella a metà) e migliorando significativamente la qualità del retrieval.

### 2.1.2 OCR con Modello DeepSeek

Per quanto riguarda i **PDF scansionati**, quindi immagini di pagine cartacee, per la loro manipolazione ed estrazione di testo è necessario un processo di **Optical Character Recognition (OCR)**, che traduce pixel in caratteri leggibili.

I recenti progressi nel campo del **Deep Learning** hanno portato allo sviluppo di **modelli vision-language multimodali**, capaci di elaborare contemporaneamente informazioni visive e testuali.

Il modello utilizzato in questo progetto è **DeepSeek-OCR** [10], un modello open-source, rilasciato recentemente che supera in termini di prestazioni e precisione i classici sistemi OCR. Le caratteristiche principali sono:

- **Vision Encoder (DeepEncoder)**: Un codificatore visivo che compri me l'immagine del documento in una rappresentazione latente ad alta densità, riducendo drasticamente il numero di token necessari senza perdere i dettagli dei glifi o della formattazione, permettendo in questo modo una maggiore velocità e una riduzione dei costi di memoria.
- **Decoder MoE (Mixture-of-Experts)**: Un modello linguistico specializzato (basato su un'architettura da 3 miliardi di parametri) che "traduce" i token visivi direttamente in testo strutturato.
- **Output Strutturato Nativo**: Il modello non si limita a riconoscere i caratteri, ma restituisce direttamente il contenuto in formato **Markdown** o **LaTeX**, preservando semanticamente tabelle, elenchi e intestazioni.
- **Gestione di Formule e Layout Complessi**: Grazie al training specifico su documenti tecnici, il modello eccelle nella linearizzazione di formule matematiche e nel mantenimento della struttura logica di documenti multi-colonna.
- **Alta Risoluzione e Multilingua**: Supporta l'elaborazione di pagine intere ad alta densità informativa, gestendo efficacemente anche font misti e caratteri speciali tipici della letteratura tecnica.

## 2.2 Retrieval basato su Embeddings

Il Retrieval consiste nel cercare e recuperare, all'interno di una vasta selezione di documenti, le informazioni che si avvicinano maggiormente alla richiesta dell'utente. Le tecniche moderne si basano sul concetto di **ricerca semantica**, ovvero una ricerca effettuata in base al contesto delle informazioni analizzate.

### 2.2.1 Rappresentazioni Vettoriali: gli Embeddings

I motori di ricerca semanticci hanno bisogno di stabilire e trovare relazioni tra dati complessi, come testi, tabelle e immagini. Per farlo, è necessario che gli algoritmi sappiano identificare le caratteristiche essenziali di questi dati per poi confrontarle.

È qui che entra in gioco il concetto di **embedding**. L'idea di base consiste nel proiettare i dati in uno **spazio vettoriale** (chiamato *spazio di embedding*), dove ogni elemento è rappresentato come un vettore. In questo modo, confrontare due elementi complessi diventa più semplice perché basta calcolare la distanza geometrica tra i due.

Lo spazio di embedding permette di preservare le relazioni semantiche, perché elementi con significato simile saranno più vicini, mentre elementi diversi saranno più distanti.

#### Misure di Similarità

Una volta ottenute le rappresentazioni vettoriali, è necessario definire una metrica per quantificare la similarità tra due embeddings. Le misure più comuni sono:

- **Distanza Euclidea:** Misura la distanza geometrica "in linea retta" tra i due punti nello spazio.

$$d_{\text{euclidea}}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^n (a_i - b_i)^2} = \|\mathbf{a} - \mathbf{b}\| \quad (2.1)$$

- **Similarità del Coseno (Cosine Similarity):** Misura il coseno dell'angolo  $\theta$  compreso tra i due vettori. Questa è la metrica più diffusa per le applicazioni di *Semantic Retrieval*, in quanto è insensibile alla magnitudine (lunghezza) del vettore e si concentra sulla direzione (contenuto semantico).

$$\text{sim}_{\cos}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} = \frac{\sum_{i=1}^n a_i b_i}{\sqrt{\sum_{i=1}^n a_i^2} \cdot \sqrt{\sum_{i=1}^n b_i^2}} \quad (2.2)$$

Il valore risultante si trova nell'intervallo  $[-1, 1]$ , dove 1 indica massima corrispondenza semantica.

### 2.2.2 Modelli Sentence Transformers

I **Sentence Transformers** (SBERT) [6] sono un framework e una famiglia di modelli addestrati per generare in modo efficiente gli embeddings di interi paragrafi.

Utilizzano una **Rete Siamese**, cioè un'architettura di rete neurale, dove due o più sottoreti identiche lavorano in parallelo e condividono gli stessi parametri, con l'obiettivo di confrontare due input distinti e misurare la loro similarità.

Nel progetto questi modelli vengono utilizzati per elaborare la query corrispondente alla domanda dell'utente e confrontarla con l'intero set di documenti ricercando i risultati con "distanza" minore, utilizzando la Similarità del coseno come metrica.

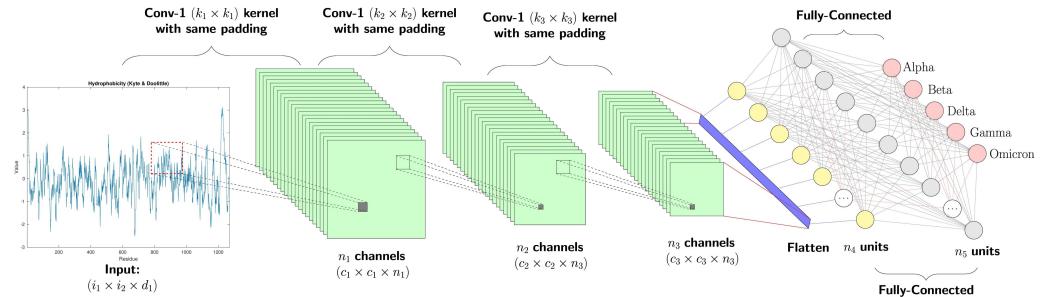


Figura 2.1: Architettura di una Rete Siamese per Sentence Transformers

## Modelli Pre-addestrati

Esistono numerosi modelli Sentence Transformers pre-addestrati su vasti dataset, ciascuno ottimizzato per diversi casi d'uso:

- **all-MiniLM-L6-v2**: Modello **leggero** (circa 80 milioni di parametri) e veloce. È ottimo per applicazioni in tempo reale dove la latenza è un fattore critico, sacrificando leggermente la qualità della comprensione semantica in favore della velocità.
- **all-mpnet-base-v2**: Modello **performante** (circa 110 milioni di parametri) che rappresenta un buon compromesso tra qualità semantica degli embedding e velocità di inferenza.
- **paraphrase-multilingual-MiniLM-L12-v2**: Modello **multilingue** e robusto (circa 116 milioni di parametri). Supporta oltre 50 lingue ed è addestrato su coppie di parafrasi multilingue, il che lo rende ideale per la comparazione semantica in contesti internazionali.

Il modello selezionato per questo progetto è il *paraphrase-multilingual-MiniLM-L12-v2*, in quanto è stato progettato per contesti multilingue, che permette l'utilizzo ad utenti che non conoscono l'inglese. Questa caratteristica consente di elaborare efficacemente documenti in italiano, garantendo una maggiore通用性 (generalità).

do una migliore comprensione delle richieste ed inoltre offre il vantaggio di comprendere contenuti in lingua inglese.

### 2.2.3 Database vettorizzati, ChromaDB

Una volta generati gli embeddings per un insieme di documenti, è necessario un sistema efficiente per **memorizzarli** e **recuperarli** in base alla similarità semantica con una query. I tradizionali database relazionali non sono adatti per questo tipo di operazioni, da cui nasce la necessità di usare un **Database vettorizzato**.

#### ChromaDB

ChromaDB [9] è un database vettorizzato open-source progettato per archiviare, indicizzare e interrogare gli embeddings. Rappresenta una soluzione ottima per applicazioni di **ricerca semantica** e **RAG**. Le sue caratteristiche principali sono:

- **Efficienza**: gestisce i dati in formato vettoriale, permettendo richERCHE basate sulla similarità del coseno in tempi molto brevi.
- **Server**: permette a più applicazioni di accedere agli stessi dati attraverso un'API.
- **Persistenza locale**: supporta modalità persistenza su disco, senza necessità di server esterni.
- **Funzioni di Embeddings Integrate**: supporto nativo per Sentence Transformers e altri modelli di embedding, con gestione automatica della codifica.

- **Metadata filtering:** permette di filtrare i risultati in base a metadati (es. data di creazione, autore, categoria), combinando ricerca semantica e filtri strutturati.

### Operazioni Fondamentali

Le operazioni principali di ChromaDB sono:

1. **Creazione collezione:** inizializzazione di uno spazio di memorizzazione per embeddings con una specifica funzione di embedding.
2. **Inserimento (Add/Upssert):** aggiunta di nuovi documenti con i loro embeddings e metadati. L'operazione di *upsert* permette di aggiornare documenti esistenti o inserirne di nuovi.
3. **Query:** ricerca dei  $k$  documenti più simili in risposta ad una query. Il database:
  - Calcola l'embedding della query
  - Esegue ANN search tra gli embeddings memorizzati
  - Restituisce i  $k$  risultati più simili con i relativi score di similarità
4. **Eliminazione (Delete):** rimozione di documenti in base a ID o metadati (es. eliminazione di tutti i chunk relativi a un documento cancellato dall'utente).

## 2.3 Framework e Strumenti di Sviluppo

Questa sezione descrive i framework e gli strumenti utilizzati per costruire l'infrastruttura web, la gestione delle elaborazioni asincrone, l'integrazione della GPU remota e la funzione di visualizzazione dei documenti.

### 2.3.1 Django Framework

Per la progettazione dell’infrastruttura web è stato utilizzato Django [1], un framework open-source scritto in Python, progettato per favorire lo sviluppo rapido di applicazioni web sicure e manutenibili, basato sul pattern architettonico **MTV** (Model-Template-View). Django fornisce un ecosistema completo di strumenti che coprono tutti gli aspetti dello sviluppo web.

#### Architettura MTV

L’architettura MTV di Django è una variante del classico pattern MVC (Model-View-Controller), adattata al contesto web:

- **Model:** definisce la struttura dei dati e gestisce l’interazione con il database attraverso un ORM (Object-Relational Mapping) che permette di definire e manipolare dati senza scrivere SQL direttamente.
- **Template:** gestisce la presentazione dei dati all’utente. I template sono file HTML con un linguaggio di templating che permette di inserire logica dinamica e supportano l’ereditarietà per definire layout riutilizzabili.
- **View:** contiene la logica di business dell’applicazione. Le view ricevono richieste HTTP, interagiscono con i model e passano dati ai template per il rendering.

#### Caratteristiche Principali

Ho scelto **Django** per i seguenti motivi:

- **ORM potente:** fornisce un livello di astrazione sicuro, con protezioni contro SQL injection e un sistema di migrations che facilita l'evoluzione del database.
- **Autenticazione e permessi:** include un sistema completo per la gestione di utenti, gruppi e permessi, facilmente estendibile.
- **Gestione dei file:** supporto nativo per upload, storage e validazione dei file, con backend configurabili.
- **Admin interface:** un pannello amministrativo auto-generato che accelera le attività di sviluppo, debugging e manutenzione.
- **Scalabilità:** un'architettura collaudata che permette di scalare da applicazioni single-server a infrastrutture distribuite.

Queste caratteristiche rendono Django una soluzione ottima per la creazione di un'applicazione web che richiede la manipolazione di documenti, la gestione degli utenti, l'integrazione di processi e le elaborazioni.

### 2.3.2 Celery e Redis per Task Asincroni

L'elaborazione di documenti PDF, specialmente quando richiede OCR su server remoti, può richiedere diversi minuti. Eseguire queste operazioni in modo sincrono bloccherebbe il thread del web server, impedendo di servire altre richieste e causando una pessima esperienza utente. Per risolvere questo problema, nel progetto si utilizza **Celery** per l'esecuzione asincrona di task in background.

### Celery: gestione di code asincrone

Celery [5] è un sistema di code di attività distribuito, progettato per gestire l'esecuzione asincrona di operazioni computazionalmente pesanti, delegando queste attività a processi in background, affinché l'utente non debba aspettare e possa continuare a navigare liberamente. L'architettura di Celery si basa su tre componenti principali:

- **Producer:** l'applicazione che genera task da eseguire e li inserisce nella coda. Nel contesto web, tipicamente il server applicativo (es. Django, Flask) che riceve richieste dagli utenti.
- **Message Broker:** intermediario per la comunicazione tra producer e worker. Gestisce le code di task, garantendo la consegna affidabile e permettendo la distribuzione del carico.
- **Worker:** processi indipendenti che prelevano task dalla coda ed li eseguono. I worker possono essere distribuiti su macchine diverse per scalabilità orizzontale e possono eseguire task in parallelo utilizzando multiprocessing o threading.

### Redis come Message Broker

Redis [4] è un database in-memory (risiede nella RAM) ad alte prestazioni. Le caratteristiche che lo rendono ideale per questo ruolo sono:

- **Velocità:** essendo completamente in-memory, Redis offre latenze nell'ordine dei microsecondi per operazioni di lettura/scrittura, minimizzando l'overhead nella comunicazione producer-worker.

- **Strutture dati avanzate:** supporta liste, set, hash e altri tipi di dati che Celery sfrutta per implementare code efficienti e gestire metadati dei task.
- **Pub/Sub:** meccanismo di publish/subscribe per notifiche in tempo reale, utile per avvisare quando un processo è terminato.
- **Persistenza opzionale:** può salvare periodicamente i dati sul disco per non perderli in caso di crash o riavvio del server.

### Gestione degli Errori nei Task Asincroni

Un aspetto critico dei sistemi di task queue è la gestione affidabile degli errori. Celery fornisce meccanismi robusti:

- **Automatic retry:** task che falliscono possono essere automaticamente ri-eseguiti dopo un delay configurabile
- **Max retries:** limite al numero di tentativi per evitare loop infiniti su task permanentemente falliti.
- **Task states:** tracciamento dello stato del task attraverso il suo ciclo di vita (PENDING, STARTED, SUCCESS, FAILURE, RETRY), permettendo monitoring e debugging.

La combinazione di Celery e Redis fornisce un'infrastruttura robusta e scalabile per gestire elaborazioni asincrone in applicazioni web, essenziale per task computazionalmente intensivi come OCR, elaborazione di grandi volumi di dati e interazioni con servizi esterni.

### 2.3.3 Infrastruttura GPU Remota con Lambda Cloud

L’elaborazione OCR richiede GPU potenti per eseguire tutti i complessi calcoli computazionali. Non disponendo dell’hardware adatto per poter utilizzare questa tecnologia, ho optato per una soluzione basata su GPU remota, noleggiata su **Lambda Cloud** [3].

#### Lambda Cloud Cloud GPU

Lambda Cloud è un provider di infrastruttura cloud specializzato in GPU per machine learning e deep learning. Offre istanze pre-configurate con:

- **GPU di ultima generazione:** NVIDIA A100 (40GB/80GB), H100, RTX A6000, ottimizzate per inferenza di modelli transformer di grandi dimensioni.
- **Stack software pre-installato:** CUDA, cuDNN, PyTorch, TensorFlow, container Docker con ambienti ML pronti all’uso.
- **Pricing on-demand:** fatturazione oraria, permettendo di attivare istanze solo quando necessario e ridurre i costi.
- **Storage persistente:** filesystem condiviso tra riavvii per salvare modelli e configurazioni.
- **Networking:** IP pubblici e porte configurabili per esporre API HTTP.

Nel sistema sviluppato, viene utilizzata un’istanza Lambda Cloud con GPU NVIDIA A10 (24GB) ed è stato configurato un filesystem per garantire la persistenza dell’ambiente configurato all’interno.

### 2.3.4 PDF.js per Visualizzazione Documenti

La visualizzazione di documenti PDF direttamente nel browser è una funzionalità essenziale per un sistema di document management. PDF.js [2] è una libreria JavaScript open-source sviluppata da Mozilla che permette di renderizzare PDF nel browser senza plugin esterni.

#### Caratteristiche di PDF.js

PDF.js è una implementazione completa del formato PDF scritta interamente in JavaScript:

- **Rendering nativo:** converte le istruzioni PDF in operazioni Canvas HTML5, permettendo visualizzazione cross-browser senza dipendenze native.
- **Viewer completo:** include un’interfaccia utente predefinita con funzionalità di navigazione, zoom, ricerca nel testo, stampa e download.
- **API programmatica:** permette di integrare il rendering PDF in applicazioni personalizzate con controllo fine sul comportamento.
- **Prestazioni ottimizzate:** rendering progressivo delle pagine, caching, e supporto per Web Workers per non bloccare il thread principale.
- **Navigazione Diretta alle Pagine:** una funzionalità chiave del sistema è la capacità di aprire automaticamente un PDF alla pagina contenente un risultato di ricerca.

# Capitolo 3

## Architettura del Sistema

### 3.1 Overview

La figura 3.1 illustra l’architettura logica del sistema, rappresentata tramite le principali componenti collegate da relazioni di dipendenza. Il sistema adotta una netta separazione tra operazioni sincrone e asincrone: il **Django Web Server** gestisce le interazioni dirette con l’utente (autenticazione, upload, ricerca), mentre i **Celery Workers** si occupano dell’elaborazione pesante in background (estrazione testo, OCR, indicizzazione vettoriale). Questa architettura garantisce tempi di risposta rapidi all’utente anche durante l’elaborazione di documenti complessi.

La persistenza dei dati è affidata a due database specializzati: **SQLite** memorizza i metadati strutturati (utenti, documenti), mentre **ChromaDB** gestisce gli embeddings vettoriali necessari per la ricerca semantica. Il coordinamento tra i componenti asincroni avviene tramite **Redis**, che funge da message broker per le code di task Celery.

Per l’elaborazione OCR di documenti scansionati, il sistema si integra con un server GPU remoto esterno (**Lambda GPU Server**) che esegue il mo-

dello **DeepSeek-OCR**. Questa scelta architetturale permette di mantenere contenuti i costi infrastrutturali, attivando risorse computazionali costose solo quando è necessario.

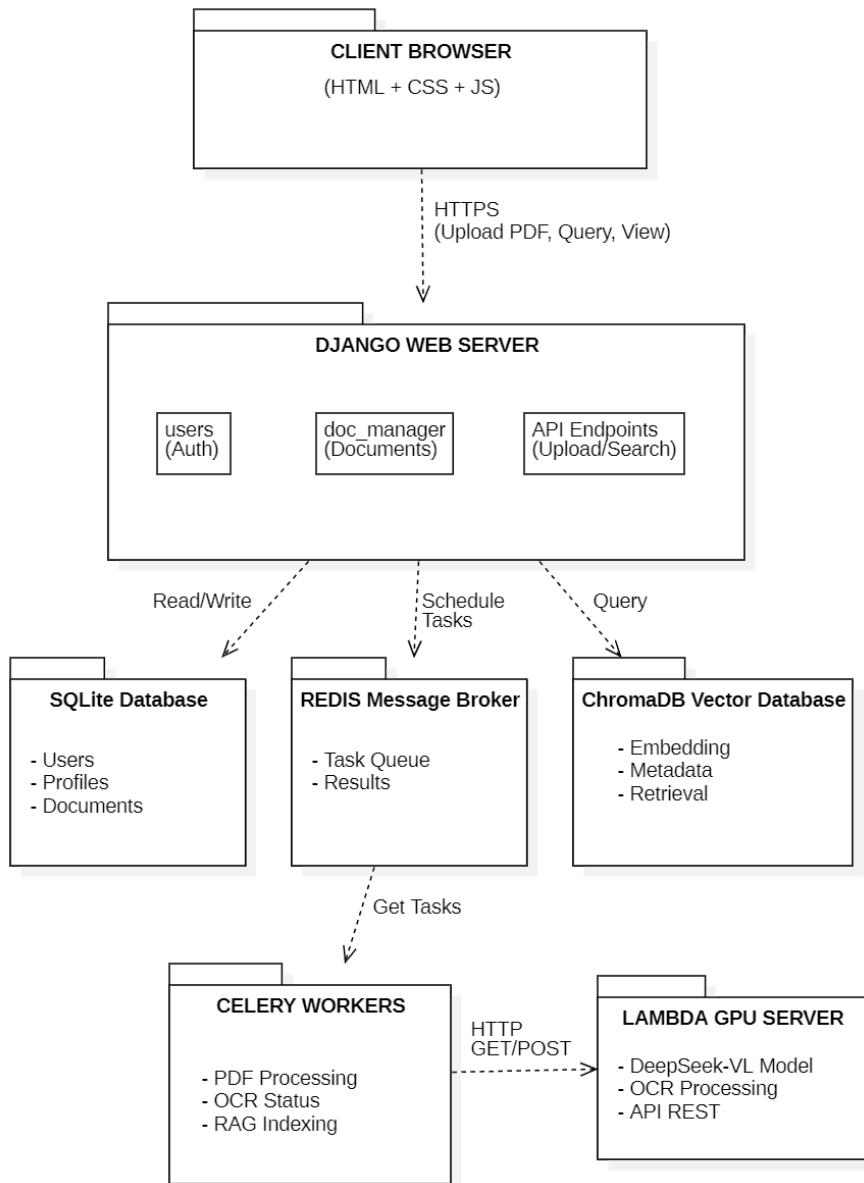


Figura 3.1: Architettura generale

### 3.1.1 Componenti e Ruoli

#### 1. Client Browser

- Interfaccia utente basata su **HTML, CSS e JavaScript**.
- Permette **upload** di documenti, esecuzione di **query** e **visualizzazione PDF**.

#### 2. Django Web Server

- Gestisce **autenticazione** (`users` app) e **manipolazione dei documenti** (`doc_manager` app).
- Implementa **API REST** per upload, ricerca e visualizzazione.
- Effettua **validazione** dei file, scheduling dei **task Celery** e controllo accessi ai file serviti.

#### 3. SQLite Database

- Database relazionale per **dati strutturati**.
- Memorizza:
  - **Utenti e profili**: credenziali di accesso (tabella `auth_user`), ruoli e permessi (tabella `users_profile`).
  - **Metadati documenti**: per ogni documento caricato vengono memorizzati titolo, uploader, timestamp, tipo (nativo/-scansionato), stato di elaborazione (pending, processing, completed, failed) e percorsi dei file.
  - **Dati task Celery**: risultati e stati dei task asincroni per debugging e monitoraggio.

#### 4. Redis Message Broker

- Funge da **broker di messaggi** per il sistema di task asincroni **Celery**.
- Gestisce **code distinte**:
  - **default**: per task parallelizzabili (indicizzazione RAG documenti nativi)
  - **ocr**: serializzata per task OCR che richiedono GPU remota
- Utilizzato per **caching** risultati task e **routing** verso i worker appropriati in base alla coda.

## 5. ChromaDB Vector Database

- Database specializzato per la **ricerca semantica**.
- Memorizza **Embedding vettoriale**, **metadati** e **testo** per ogni chunk.
- Abilita il retrieval per la pipeline **RAG**.

## 6. Celery Workers

- Processi Python indipendenti per l'esecuzione di **task in background**.
- Responsabili di:
  - **Elaborazione PDF nativi**: estrazione del testo tramite libreria Docling, suddivisione in chunk, generazione embeddings.
  - **Elaborazione OCR**: invio di PDF scansionati al server Lambda GPU, polling periodico dello stato, recupero del testo estratto.

- **Indicizzazione RAG:** generazione degli embeddings tramite Sentence Transformers e memorizzazione in ChromaDB con i metadati associati.

## 7. Lambda GPU Server

- Infrastruttura **esterna** che fornisce una **GPU** per l'OCR avanzato.
- Utilizza un modello **OCR** (DeepSeek-VL) per l'estrazione di testo da PDF scansionati.
- Accessibile tramite **API REST** (FlaskAPI).

### 3.1.2 Relazioni tra Componenti

**Client Browser ↔ Django Web Server** La comunicazione avviene tramite protocollo HTTPS con richieste sincrone. Il browser invia tre tipi principali di richieste:

- **Upload:** richiesta POST con il file PDF allegato.
- **Query di ricerca:** richiesta GET con query contenente la domanda. Django interroga ChromaDB e restituisce i risultati ordinati per rilevanza.
- **Visualizzazione:** richiesta GET per scaricare il PDF o aprirlo se già salvato.

**Django Web Server ↔ SQLite Database** Django interagisce con SQLite tramite il suo ORM, che astrae le query SQL.

**Django Web Server → Redis Message Broker** Django agisce come *producer* di task Celery. Dopo aver salvato un documento nel database, Django:

1. Determina il tipo di elaborazione necessaria.
2. Invoca il task Celery appropriato.
3. Riceve un `task_id`.

Redis riceve il task serializzato e lo inserisce nella coda corretta.

**Redis Message Broker → Celery Workers** Redis opera come intermedio tra producer (Django) e consumer (Workers). I worker:

1. Effettuano polling continuo delle code Redis.
2. Prelevano task dalla coda FIFO.
3. Deserializzano i parametri del task ed eseguono la funzione corrispondente.
4. Salvano il risultato su Redis per permetterne il recupero.

Le due code separate permettono di gestire diversamente i task OCR (che devono essere serializzati per non sovraccaricare il server GPU) dai task di indicizzazione (che possono essere parallelizzati).

**Celery Workers ↔ ChromaDB Vector Database** I worker interagiscono con ChromaDB in due modalità:

- **Indicizzazione:** dopo aver generato gli embeddings con Sentence Transformers, i worker invocano il metodo `collection.add()` di ChromaDB passando embeddings, documenti testuali e metadati.

- **Eliminazione:** quando un documento viene cancellato da Django, si attiva un task che rimuove tutti i chunk associati da ChromaDB tramite il metadata `document_pk`.

**Celery Workers ↔ Lambda GPU Server** Questa comunicazione avviene tramite API REST con un pattern asincrono:

1. Il task `process_scanned_document` invia una richiesta POST con il PDF encoded in base64
2. Lambda Server risponde con un `task_id` e inizia l'elaborazione in background
3. Il task Celery avvia un secondo task che periodicamente interroga l'endpoint GET `<task_id>`
4. Quando lo stato diventa "completed", il task recupera il testo estratto tramite GET `<task_id>` e procede con l'indicizzazione RAG

**Django Web Server ↔ ChromaDB Vector Database** Durante una ricerca semantica, Django:

1. Riceve la query dell'utente.
2. Genera l'embedding della query utilizzando lo stesso modello Sentence Transformers usato per indicizzare i documenti.
3. Invoca `collection.query()` di ChromaDB passando l'embedding e il numero di risultati desiderati.
4. ChromaDB esegue similarity search (cosine similarity) sull'indice e restituisce i K chunk più simili con distanza e metadati

5. Presenta i risultati ordinati per rilevanza con preview del testo e link al PDF

### 3.1.3 Organizzazione del Progetto

```
docseek/
  config/
    settings.py
    urls.py
    wsgi.py
  users/
    models.py
    views.py
    forms.py
    templates/users/
  doc_manager/
    models.py
    views.py
    forms.py
    tasks.py
    rag_pipeline/
      processing.py
      embedding.py
      search.py
      templates/doc_manager/
  database/
    db.sqlite3
    chromadb_data/
  media/
    documents/
    img/
  static/
    css/
    js/
    pdfjs/
  requirements.txt
  manage.py
```

## 3.2 Struttura Logica del Progetto

Questa sezione descrive l'implementazione dei componenti principali del sistema, analizzando l'organizzazione logica delle app Django, la pipeline RAG e l'integrazione tra i diversi moduli.

### 3.2.1 App Users

L'app `users` estende il sistema di autenticazione integrato di Django implementando un modello di autorizzazione basato su ruoli. Il modello principale è `Profile`, collegato al modello `User` di Django tramite una relazione uno-a-uno:

```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    is_uploader = models.BooleanField(default=False)
    is_searcher = models.BooleanField(default=False)
```

Il controllo degli accessi è implementato tramite mixin personalizzati nelle view Django che verificano i permessi prima di eseguire le operazioni.

### 3.2.2 App Doc\_Manager

L'app `doc_manager` costituisce il nucleo del sistema e gestisce l'intero ciclo di vita dei documenti. Il modello centrale è `Document`:

```
class Document(models.Model):
    uploader = models.ForeignKey(User, on_delete=models.CASCADE)
    file = models.FileField(upload_to='documents/%Y/%m/%d/')
    title = models.CharField(max_length=255)
    document_type = models.CharField(max_length=20, choices=[('native', 'PDF Nativo'),
```

```
('scanned', 'PDF Scansionato')
])
processing_state = models.CharField(max_length=20)
ocr_text = models.TextField(blank=True)
processed_file = models.FileField(blank=True)
uploaded_at = models.DateTimeField(auto_now_add=True)
```

Il campo `processing_state` traccia lo stato di elaborazione del documento attraverso otto possibili valori: `pending`, `ocr_queued`, `ocr_processing`, `ocr_completed`, `ocr_failed`, `rag_processing`, `completed`, `failed`. Questa macchina a stati permette di monitorare il progresso dell'elaborazione e gestire eventuali errori.

**View e Form** Le view principali sono:

- **DocumentUploadView**: gestisce l'upload, salva il documento nel database e programma il task Celery appropriato.
- **SearchView**: riceve query, genera l'embedding, interroga ChromaDB e mostra i risultati.
- **DocumentDetailView**: fornisce PDF salvati, verificando che l'utente richiedente abbia i permessi necessari.

**Task Celery** L'elaborazione asincrona è gestita da tre task principali definiti in `tasks.py`:

- **index\_document\_rag**: elabora PDF estraendo il testo con Docling o Deepseek-OCR, suddividendolo in chunk, generando embeddings e indicizzando in ChromaDB.

- **process\_scanned\_document**: invia PDF scansionati al server Lambda GPU tramite richiesta HTTP POST, riceve un `task_id` e schedula il polling dello stato.
- **check\_ocr\_status**: effettua polling periodico del server Lambda per verificare il completamento dell'OCR.

### 3.2.3 Pipeline RAG

La pipeline RAG è il cuore del sistema per l'indicizzazione e la ricerca semantica dei documenti. È composta da tre moduli principali che operano in sequenza, come illustrato nella Figura 3.2.

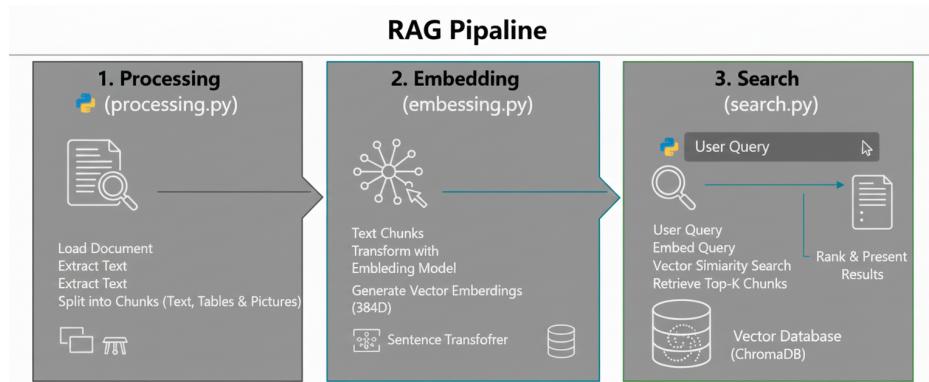


Figura 3.2: Architettura della Pipeline RAG implementata nel sistema.

**Elaborazione Documenti (processing.py)** Questo modulo gestisce l'acquisizione e la preparazione dei documenti per l'indicizzazione. Le fasi principali includono:

- **Caricamento del Documento**: Conversione del PDF in formato elaborabile preservando la struttura originale

- **Estrazione del Testo:** Estrazione del contenuto testuale con riconoscimento di elementi strutturati (titoli, tabelle, immagini)
- **Suddivisione in Chunk:** Frammentazione del testo in porzioni di 500 token con overlap di 50 token per preservare il contesto

```
def convert_pdf_to_doc(filename: str):
    options = PdfPipelineOptions(do_table_structure=True,
                                 generate_picture_images=True)
    pdf_format = PdfFormatOption(pipeline_options=options)
    converter = DocumentConverter(format_options={"pdf": pdf_format})
    print(f"Inizio analisi strutturata di: {filename}")
    result = converter.convert(filename)
    return result.document
```

**Generazione Embedding** (`embedding.py`) Il modulo si occupa della trasformazione del testo in rappresentazioni vettoriali attraverso il modello `paraphrase-multilingual-MiniLM-L12-v2`, scelto grazie alla sua velocità e leggerezza, ma soprattutto grazie alla possibilità di fare ricerca semantica in italiano.

```
def init_chromadb(collection_name=None):
    db_path = os.path.join(settings.BASE_DIR, "database", "chromadb_data")
    client = chromadb.PersistentClient(path=db_path)
    COLLECTION_NAME = get_collection_name()
    MODEL_NAME = get_embedding_model()
    embedding_function = embedding_functions.SentenceTransformerEmbeddingFunction(
        model_name=MODEL_NAME)
    collection = client.get_or_create_collection(
        name=COLLECTION_NAME,
        embedding_function=embedding_function)
    return collection
```

**Ricerca Semantica (search.py)** Implementa l’interfaccia con ChromaDB per operazioni di retrieval basate su similarità vettoriale. La funzione principale processa le query restituendo risultati in ordine di rilevanza.

```
def run_queries(collection, queries, n_results=DEFAULT_N_RESULTS):
    results = collection.query(query_texts=queries, n_results=n_results)
    all_formatted_results = []

    for q_idx, query in enumerate(queries):
        docs = results['documents'][q_idx]
        metas = results['metadatas'][q_idx]
        dists = results['distances'][q_idx]

        query_results = {'query': query, 'chunks': []}
        for i, (doc, meta, dist) in enumerate(zip(docs, metas, dists)):
            query_results['chunks'].append({
                'distance': dist,
                'document_title': meta.get('source_title', 'N/A'),
                'document_id': meta.get('document_id'),
                'page': meta.get('page', 'N/A'),
                'type': meta.get('type', 'N/A'),
                'content': doc.strip()
            })
        all_formatted_results.append(query_results)

    return all_formatted_results
```

I risultati includono metadati completi (titolo documento, pagina, tipologia) e sono strutturati in formato JSON per l’integrazione con l’interfaccia Django.

### 3.2.4 Risorse Statiche

La directory `static/` organizza le risorse front-end secondo la seguente struttura:

- **CSS:** Fogli di stile personalizzati che estendono Bootstrap 5 [8]
- **JavaScript:** Script client-side per la gestione delle interazioni asincrone
- **PDF.js:** Libreria per la visualizzazione integrata dei documenti nel browser

### 3.2.5 Page Navigation Diagram

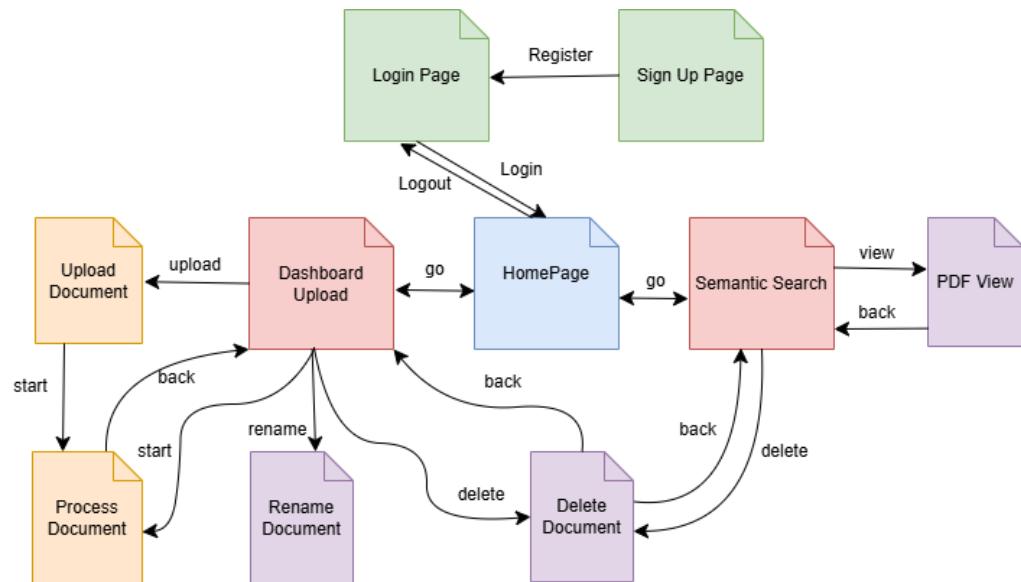


Figura 3.3: Diagramma di navigazione dell'interfaccia utente

### Legenda

- **Verde:** Autenticazione

- **Blu:** Home
- **Arancione:** Gestione Documenti
- **Rosso:** Pagine principali (Ricerca e Upload)
- **Viola:** Funzioni Ausiliarie

### 3.3 Database

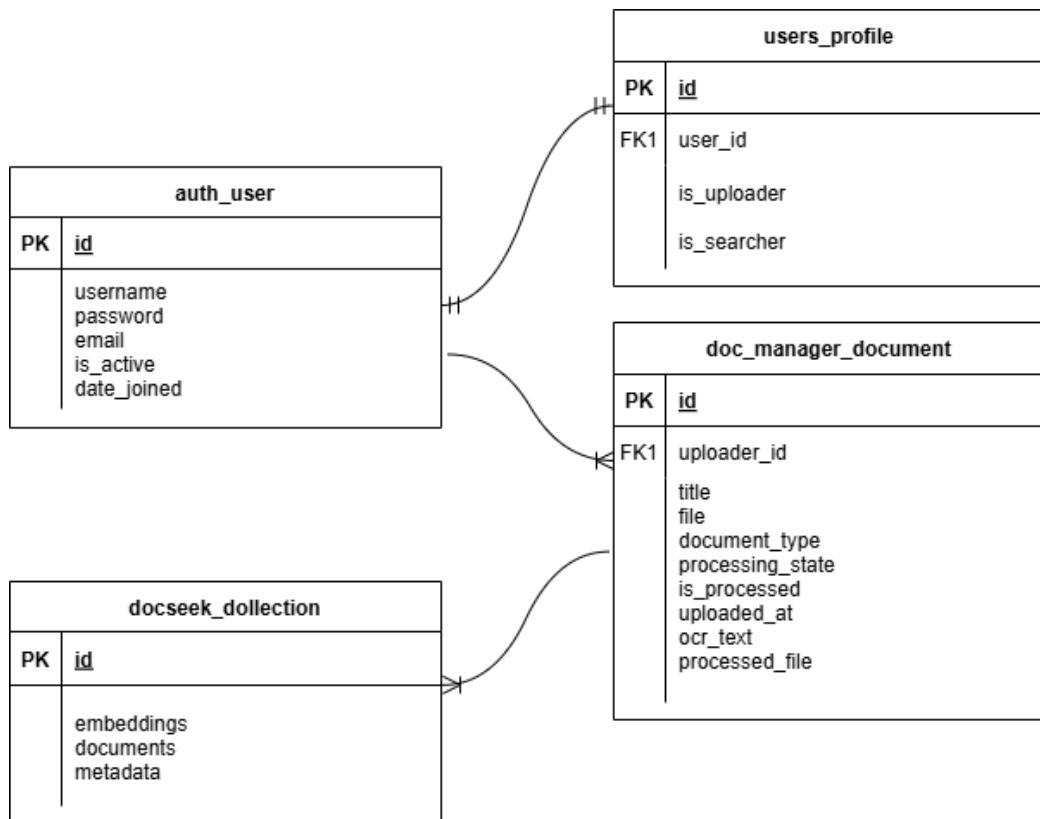


Figura 3.4: Entity-Relationship Diagram

Il sistema DocSeek utilizza un'architettura a **database doppio**: SQLite per i dati dell'applicazione e ChromaDB per gli embeddings vettoriali. La Figura 3.4 illustra lo schema Entity-Relationship completo.

### 3.3.1 Tabella auth\_user

Tabella standard di Django per la gestione dell'autenticazione. Memorizza le credenziali degli utenti e i flag di sistema necessari per il controllo degli accessi.

#### Campi principali:

- **id** (PK): Identificatore univoco dell'utente
- **username**: Nome utente univoco per il login
- **password**: Password hashata con algoritmo PBKDF2
- **email**: Indirizzo email dell'utente
- **is\_active**: Flag che indica se l'account è attivo
- **date\_joined**: Timestamp di registrazione

### 3.3.2 Tabella users\_profile

Estensione del modello **User** per gestire i **ruoli applicativi** specifici di DocSeek. È collegata tramite relazione **one-to-one** con **auth\_user**.

#### Campi principali:

- **id** (PK): Identificatore univoco del profilo
- **user\_id** (FK): Riferimento all'utente in **auth\_user**
- **is\_uploader**: Booleano che abilita il ruolo da caricatore
- **is\_searcher**: Booleano che abilita il ruolo da ricercatore

**Vincolo applicativo:** Durante la registrazione, almeno uno tra **is\_uploader** e **is\_searcher** deve essere **True**. Questo vincolo è implementato nel form di registrazione (**UserRegistrationForm**).

### 3.3.3 Tabella doc\_manager\_document

Tabella centrale per la gestione dei documenti PDF caricati nel sistema. È collegata a `auth_user` tramite relazione **many-to-one**.

**Campi principali:**

- `id` (PK): Identificatore univoco del documento
- `uploader_id` (FK): Riferimento all'utente che ha caricato il file
- `title`: Titolo del documento (deve essere univoco)
- `file`: Percorso relativo al file PDF originale
- `document_type`: Tipo di documento ('native' o 'scanned')
- `processing_state`: Stato corrente nella pipeline di elaborazione
- `is_processed`: Flag booleano che indica se il documento è completamente elaborato
- `uploaded_at`: Timestamp di caricamento
- `ocr_text`: Testo estratto tramite OCR (solo per documenti scansionati)
- `processed_file`: Percorso al PDF con layer OCR sovrapposto

### 3.3.4 Tabella docseek\_collection (ChromaDB)

Collection vettoriale di ChromaDB dedicata alla memorizzazione degli **embeddings** e dei chunk testuali estratti dai documenti. È collegata logicamente a `doc_manager_document` tramite il campo `document_pk` nei metadati.

**Struttura:**

- **id** (PK): UUID univoco per ogni chunk
- **embeddings**: Vettore generato dal modello  
`paraphrase-multilingual-MiniLM-L12-v2`
- **documents**: Contenuto testuale del chunk
- **metadata**: Dizionario JSON contenente:
  - **document\_pk**: Riferimento logico all'ID del documento in SQLite (FK logica)
  - **source\_title**: Titolo del documento sorgente
  - **page**: Numero di pagina del chunk
  - **type**: Tipologia del chunk (`text`, `table`, `image`)
  - **uploader**: Username dell'uploader

**Relazione con SQLite:** Ogni documento in `doc_manager_document` genera chunk multipli in ChromaDB (1:N). Il campo `document_pk` funge da **foreign key logica** non vincolata, permettendo operazioni di filtraggio ed eliminazione. Quando un documento viene cancellato da SQLite, tutti i chunk associati vengono rimossi da ChromaDB filtrando per `document_pk`.

## 3.4 Implementazione Lambda Cloud

Per l'elaborazione di documenti PDF scansionati, il sistema integra un server GPU esterno ospitato su **Lambda Cloud**. Questa scelta è stata intrapresa per necessità di risorse GPU per eseguire modelli OCR avanzati.

### 3.4.1 Architettura del Server GPU

Il server OCR è implementato come applicazione **Flask** eseguita su un’istanza Lambda Cloud equipaggiata con GPU **NVIDIA A10** (24GB VRAM). L’architettura è composta da tre livelli principali:

1. **API Layer**: Endpoint REST per la ricezione delle richieste e il monitoraggio dello stato
2. **Processing Layer**: Logica di conversione PDF-immagini ed elaborazione del modello OCR
3. **Model Layer**: Modello DeepSeek-OCR caricato tramite il framework **vLLM** per inferenza ottimizzata

### 3.4.2 Modello DeepSeek-OCR

Il modello viene caricato all’avvio del server tramite il framework **vLLM**, che ottimizza l’inferenza attraverso tecniche di *continuous batching* e *paged attention*:

```
llm = LLM(  
    model="deepseek-ai/DeepSeek-OCR",  
    enable_prefix_caching=False,  
    mm_processor_cache_gb=0,  
    logits_processors=[NGramPerReqLogitsProcessor],  
    gpu_memory_utilization=0.9,  
    trust_remote_code=True  
)
```

Il parametro `gpu_memory_utilization=0.9` configura vLLM per utilizzare il 90% della VRAM disponibile, massimizzando il throughput mantenendo un margine di sicurezza per evitare errori di memoria.

### 3.4.3 API REST del Server

Il server espone tre endpoint principali per la gestione delle richieste OCR.

#### Health Check

L'endpoint `GET /health` verifica lo stato del server e la disponibilità della GPU, restituendo informazioni diagnostiche sul modello caricato e sulle risorse disponibili.

#### Process OCR

L'endpoint `POST /process` avvia l'elaborazione OCR. Accetta richieste multipart contenenti il file PDF, l'identificatore del documento Django e il titolo. La risposta restituisce immediatamente un `task_id` univoco:

```
{  
    "task_id": "550e8400-e29b-41d4-a716-446655440000",  
    "status": "queued",  
    "message": "OCR processing started"  
}
```

#### Check Status

L'endpoint `GET /status` permette il polling dello stato di elaborazione, restituendo la percentuale di completamento e, al termine, il testo estratto o eventuali messaggi di errore.

### 3.4.4 Processo di Elaborazione

L'elaborazione di un documento segue un flusso in quattro fasi:

## Conversione PDF in Immagini

Il PDF viene convertito in una sequenza di immagini PNG utilizzando la libreria **PyMuPDF**, con risoluzione di 200 DPI in modo da ottenere un buon compromesso tra qualità dell'OCR e velocità di elaborazione.

## Elaborazione Batch

Le immagini vengono elaborate in batch di 4 pagine alla volta, sfruttando la capacità della GPU A10 di processare più immagini in parallelo, permettendo una maggiore velocità nell'elaborazione di PDF voluminosi.

## Inferenza con DeepSeek-OCR

Ogni batch viene processato dal modello con il prompt "`<image>\nFree OCR.`", che istruisce il modello a estrarre tutto il testo visibile.

## Aggregazione e Restituzione

I testi estratti vengono concatenati con separatori che indicano il numero di pagina, facilitando la successiva suddivisione in chunk nella pipeline RAG.

### 3.4.5 Integrazione con Django

Lato Django, l'integrazione avviene attraverso due task Celery che implementano un pattern di comunicazione asincrona.

## Invio del Documento

Il task `process_scanned_document` invia il PDF al server Lambda e schedula il polling dello stato:

## Polling dello Stato

Il task `check_ocr_status` effettua polling periodico fino al completamento. Quando lo stato diventa `completed`, il testo estratto viene salvato nel database e viene avviata automaticamente l'indicizzazione RAG.

### 3.4.6 Gestione della Connettività

La comunicazione con il server Lambda avviene tramite **SSH tunnel**:

```
ssh -N -L 8000:localhost:8000 ubuntu@<LAMBDA_IP>
```

Questo comando crea un tunnel che mappa la porta 8000 locale alla porta 8000 del server remoto, permettendo a Django di comunicare con l'endpoint `http://localhost:8000` come se il server fosse in esecuzione localmente.

# Capitolo 4

## Risultati

In questo capitolo vengono rappresentati i risultati ottenuti dall’implementazione di **DocSeek**, mediante diversi screenshot dell’interfaccia utente che illustrano le principali funzionalità del sistema. Si effettuano dei test che mirano a dimostrare l’efficacia della pipeline RAG, mostrando i risultati delle ricerche semantiche. Infine, viene eseguita una dettagliata considerazione in merito alle performance.

### 4.1 Panoramica dell’Interfaccia Utente

L’interfaccia utente è stata progettata utilizzando principalmente **Bootstrap 5** come framework CSS, presentando un’interfaccia **responsiva** che sia adatta sia su computer sia su dispositivi mobili. Successivamente verranno mostrate le schermate principali dell’applicazione per illustrarne le funzionalità ed il design.

### 4.1.1 Pagina di Autenticazione

The screenshot shows the 'Log In to DocSeek' page. It features a light gray header with the title 'Log In to DocSeek' in blue. Below the header are two input fields: 'Username' and 'Password', both with placeholder text 'Enter your username' and 'Enter your password' respectively. A large blue button labeled '➡ Log In' is centered below the password field. At the bottom of the form, there are links for 'Don't have an account? [Sign Up](#)' and 'Forgot password?'. The entire form is set against a white background.

Figura 4.1: Schermata di login

The screenshot shows the 'Create DocSeek Account' page. It has a light gray header with the title 'Create DocSeek Account' in blue. Below the header are two input fields: 'Username' and 'Email Address', both with placeholder text. There are two checkboxes: 'Can Upload & Process Documents (Uploader Role)' and 'Can Search & View Documents (Searcher Role)'. Below these checkboxes is a 'Password' input field with a note about password requirements: 'Your password can't be too similar to your other personal information.', 'Your password must contain at least 8 characters.', 'Your password can't be a commonly used password.', and 'Your password can't be entirely numeric.' To the right of the password field is a 'Password confirmation' input field with the note 'Enter the same password as before, for verification.'. A blue button labeled '👤 Sign Up' is at the bottom. At the very bottom, there is a link 'Already have an account? [Log In](#)'. The entire form is set against a white background.

Figura 4.2: Form di registrazione

### 4.1.2 Home Page

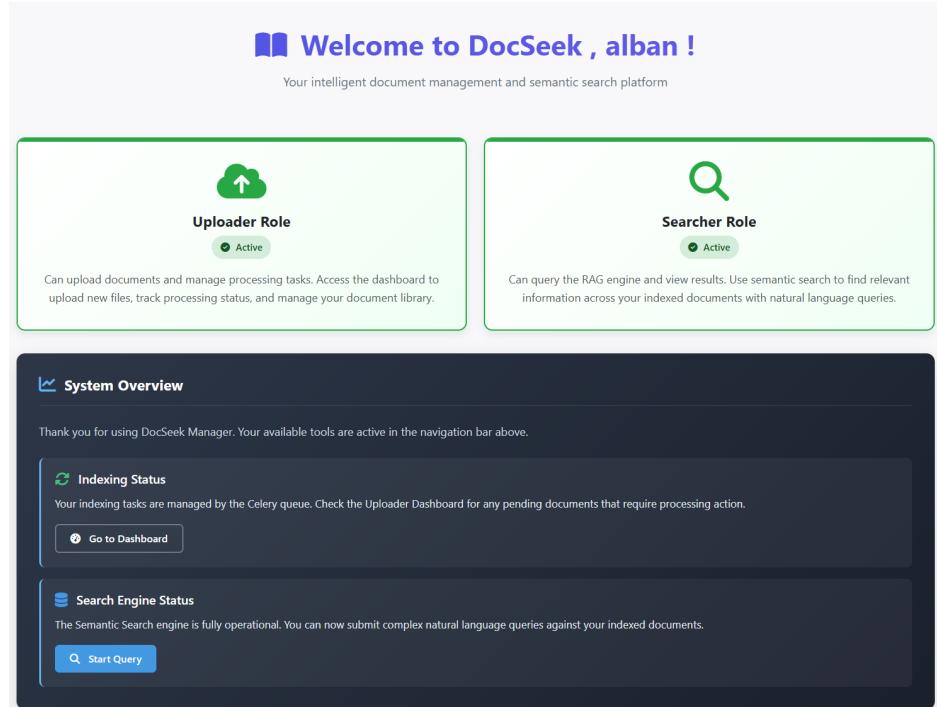


Figura 4.3: Home page del sistema

### 4.1.3 Pagine Uploader

The screenshot shows the Uploader Dashboard interface. At the top, there are three summary cards:

- PROCESSED DOCUMENTS:** 2 documents ready for Semantic Search.
- OCR IN PROGRESS:** 0 documents being processed on the GPU Server.
- AWAITING ACTION:** 1 document requiring processing.

Below these cards is a section titled "Documents Awaiting Processing" containing a single item:

- Articolo\_Conf\_IEEE\_Semantic\_Search\_2024.pdf** (Native) - Status: Native  
Uploaded Nov 26, 2025 16:12

Below this is a section titled "Processed Documents" listing two items:

- Fattura\_N\_45\_Dicembre\_2024\_Scansionata.pdf** (Scanned) - Status: Processed  
Uploaded Nov 17, 2025 09:55  
Info: ✓ Indicizzati 2 chunks in ChromaDB. Documento pronto per ricerca semantica.
- Rapporto\_Trimestrale\_Q3\_2025.pdf** (Native) - Status: Processed  
Uploaded Nov 14, 2025 10:19  
Info: ✓ Indicizzati 14 chunks in ChromaDB. Documento pronto per ricerca semantica.

At the bottom right of the dashboard is a blue button labeled "Upload New Document".

Figura 4.4: Dashboard dell'uploader

**Upload New Document**

Document Title  
Articolo\_Conf\_IEEE\_Semantic\_Search\_2024.pdf

Document Type

Native PDF (Digital)  
PDF documents with selectable text, created digitally from Word, Excel, or other applications. Processing is fast (~10 seconds per document).

Scanned PDF (Images)  GPU OCR  
Scanned documents, photos, or image-based PDFs that require OCR text extraction. Processed with DeepSeek-VL AI on GPU server. Processing time: ~1-3 minutes per page.

Select PDF File  
**test\_document.pdf**

Max file size: 50MB. Supported format: PDF

**Upload Document**

[Back to Dashboard](#)

Figura 4.5: Form di upload documento

**Process Document**  
Start document processing

**Articolo\_Conf\_IEEE\_Semantic\_Search\_2024.pdf**  
Nov 26, 2025 16:12 | alban | Download

**RAG Processing**  
The document will be queued for processing with Celery

**Processing Notes**

Optional: Add any notes or comments about this document before processing

**Start Processing**

[Back to Dashboard](#)

Figura 4.6: Schermata di avvio del processamento

#### 4.1.4 Pagina Searcher

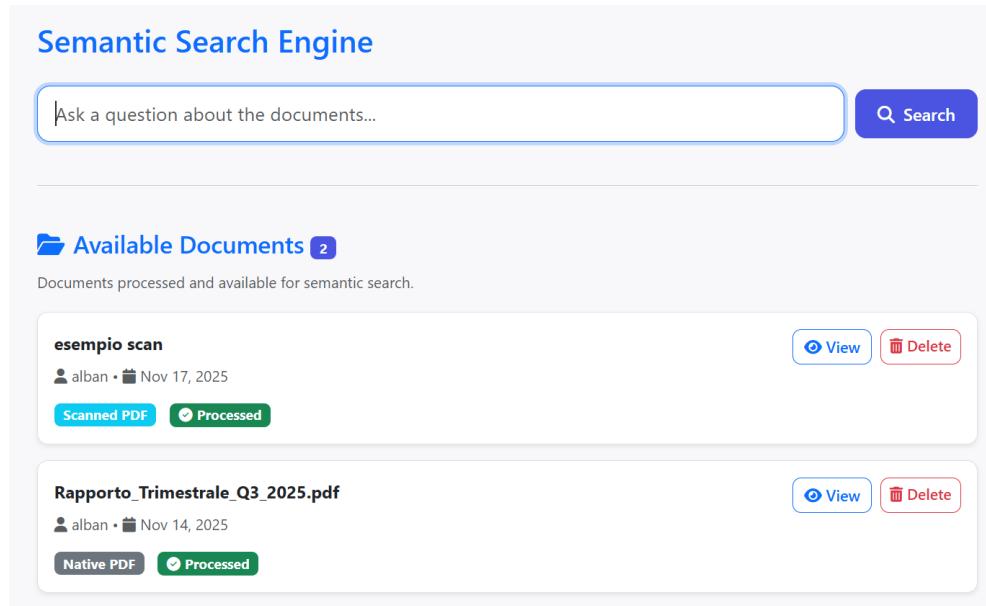


Figura 4.7: Motore di Ricerca Semantica

## 4.2 Ricerca Semantica: Esempi Pratici

In questa sezione si illustrano esempi pratici di query di ricerca e i loro corrispettivi risultati. Vengono mostrati non solo i risultati ottenuti, ma anche la funzionalità che permette di **aprire il PDF direttamente nella pagina** in modo da poter confrontare direttamente la qualità dell'estrazione. I test verranno eseguiti in modo da testare entrambe le funzionalità di processamento (Docing e OCR) ed analizzare l'output di estrazione per quanto riguarda testo, tavole e immagini.

### 4.2.1 Test di Ricerca su Documento Digitale Nativo

In questi test si effettuano domande specifiche su un PDF **nativo (digitale)** per dimostrare la capacità di DocSeek di estrarre e interpretare in-

formazioni strutturate e non strutturate senza utilizzare modelli sofisticati come l'OCR. In particolare, è stato utilizzato un articolo scientifico proprio su Docling, ritenuto ottimo per la ricerca semantica grazie alla sua **struttura complessa** che presenta testo diviso su due colonne, sezioni formattate, immagini e tabelle.

### Esempio 1: Ricerca su Testo

Questa query verifica la capacità del sistema di recuperare risposte specifiche da testo.

Query 1

"Pipeline di Docling?"

Il sistema ha restituito, come si può vedere in Figura 4.8, esattamente il chunk desiderato con un'elevata rilevanza semantica (**0,6485**) e metadati importanti, come il numero di pagina del file sorgente e il tipo di estrazione ("**text**"). Questo risultato è molto importante perché nonostante la domanda posta fosse sintetica, il motore di ricerca è riuscito a individuare il pdf corretto e recuperare il paragrafo desiderato con una buona rilevanza.

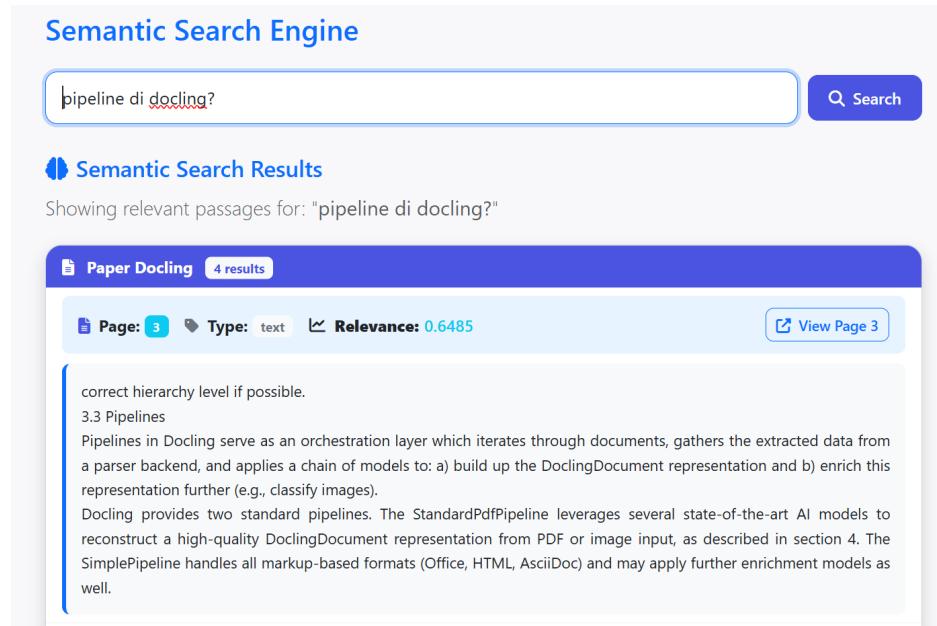


Figura 4.8: Risultato della ricerca su testo

Confrontando il testo estratto dal sistema con il PDF originale, si evince che la qualità dell'estrazione è ottima. La **Figura 4.9** illustra la funzionalità di visualizzazione del documento, dove il sistema apre il PDF direttamente alla pagina indicata nel risultato, permettendo un confronto diretto con il chunk ottenuto.

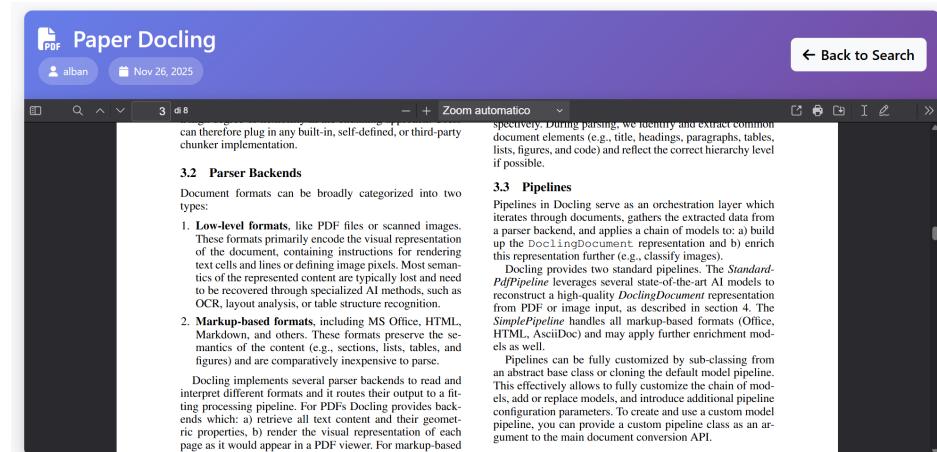


Figura 4.9: Visualizzazione del PDF

## Esempio 2: Ricerca di Tabelle

Questa query dimostra l'efficacia del sistema non solo nell'estrazione della tabella, ma anche nell'interpretazione dei dati in essa contenuti, superando la semplice ricerca testuale.

Query 2  
 "Tabella delle versioni e configurazioni docling?"

Il sistema è stato in grado di recuperare l'intera tabella presente nel documento che riassume le configurazioni utilizzate per il *benchmarking*. Questo risultato verifica la capacità di DocSeek di identificare e processare dati strutturati presenti in una griglia.

La **Figura 4.10** mostra il frammento di tabella estratto dal sistema, corrispondente alla query posta dall'utente con una buona rilevanza, corrispondete al **0,6423**.

**Semantic Search Engine**

Search

**Semantic Search Results**

Showing relevant passages for: "tabella delle versioni e configurazioni docling?"

Paper Docling 5 results																													
Page:	5	Type:	table	Relevance: 0.6423																									
<table border="1" style="width: 100%; border-collapse: collapse; text-align: left;"> <thead> <tr> <th>Asset</th> <th>Version</th> <th>OCR</th> <th>Layout</th> <th>Tables</th> </tr> </thead> <tbody> <tr> <td>Docling</td> <td>2.5.2</td> <td>EasyOCR *</td> <td>default</td> <td>TableFormer (fast) *</td> </tr> <tr> <td>Marker</td> <td>0.3.10</td> <td>Surya *</td> <td>default</td> <td>default</td> </tr> <tr> <td>MinerU</td> <td>0.9.3</td> <td>auto *</td> <td>dockeyout yolo</td> <td>rapid table *</td> </tr> <tr> <td>Unstructured</td> <td>0.16.5</td> <td></td> <td>hi res with table structure</td> <td>hi res with table structure</td> </tr> </tbody> </table>					Asset	Version	OCR	Layout	Tables	Docling	2.5.2	EasyOCR *	default	TableFormer (fast) *	Marker	0.3.10	Surya *	default	default	MinerU	0.9.3	auto *	dockeyout yolo	rapid table *	Unstructured	0.16.5		hi res with table structure	hi res with table structure
Asset	Version	OCR	Layout	Tables																									
Docling	2.5.2	EasyOCR *	default	TableFormer (fast) *																									
Marker	0.3.10	Surya *	default	default																									
MinerU	0.9.3	auto *	dockeyout yolo	rapid table *																									
Unstructured	0.16.5		hi res with table structure	hi res with table structure																									

Figura 4.10: Risultato della ricerca di tabelle

Dalla Figura 4.11 si può osservare che la tabella estratta è al quanto particolare e presenta un livello di difficoltà maggiore, perché il design utilizzato per tabella è composto solamente da righe delimitatrici, senza l'ausilio di colonne che separino nettamente le celle in modo da non confondere il testo. Ancora una volta il sistema si è dimostrato eccellente, perché nonostante l'ulteriore livello di difficoltà è riuscito a ricostruire fedelmente la struttura della tabella. L'unico errore commesso corrisponde alla ripetizione nell'ultima riga della stessa frase in due colonne diverse, ma esso è comprensibile perché come si può notare dal pdf originale quel pezzettino di testo è compreso in entrambe le colonne (Layout e Tables) quindi giustamente l'ha ristampato due volte.

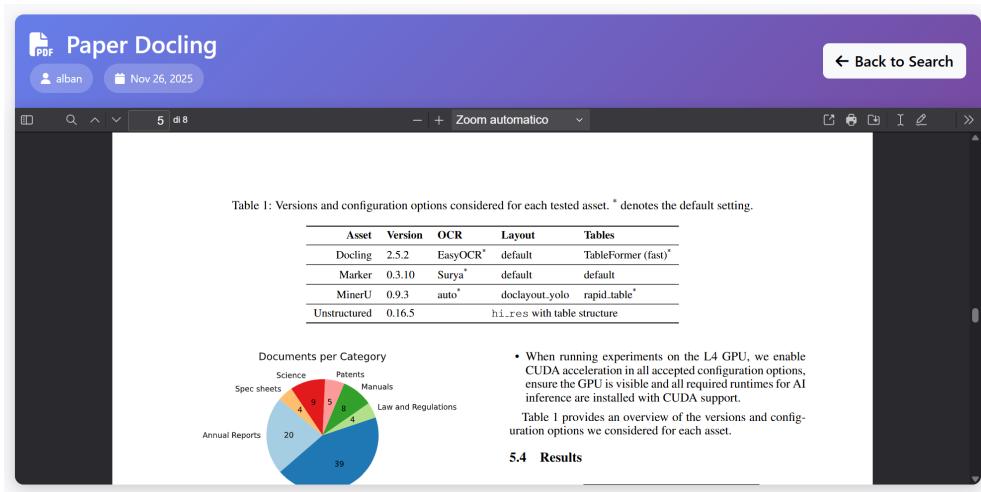
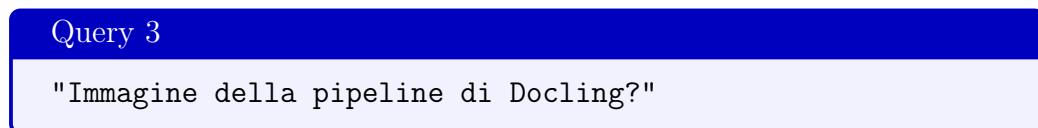


Figura 4.11: Visualizzazione del PDF contenente la tabella.

### Esempio 3: Ricerca di Immagini

Questo esempio valuta la capacità del sistema di ricercare immagini all'interno del documento, sfruttando la ricerca semantica basata sulle didascalie associate.



Il sistema recupera le informazioni più corrispondenti alla domanda posta, soprattutto usa le didascalie delle immagini per eseguire la ricerca semantica. La ricerca è stata estremamente efficace, individuando il **chunk** pertinente con un'alta rilevanza semantica (**0,7068**).

La **Figura 4.12** mostra il risultato della ricerca. L'immagine è rappresentata da un **placeholder** che ne indica la presenza e mostra come metadati importanti la pagina di origine e la **didascalia** associata.

A screenshot of a web-based semantic search engine. The title "Semantic Search Engine" is at the top. Below it is a search bar containing the query "Immagine della pipeline di Docling?". A blue "Search" button is to the right. The main area is titled "Semantic Search Results" and shows the message "Showing relevant passages for: "Immagine della pipeline di Docling?"". Below this, a search result card for "Paper Docling" is displayed. It shows "4 results". The first result has a blue header with icons for document, page number (2), type (image), and relevance (0.7068). The result card itself contains an image placeholder icon, the text "Immagine Rilevata", "Pagina: 2", and "Didascalia: Figure 1: Sketch of Docling". A "View Page 2" button is also present.

Figura 4.12: Risultato della ricerca di immagini

Come nelle sezioni precedenti, è possibile visualizzare l'elemento nel contesto originale del documento. La **Figura 4.13** mostra l'apertura del PDF con l'evidenziazione del diagramma richiesto, confermando l'accuratezza del recupero.

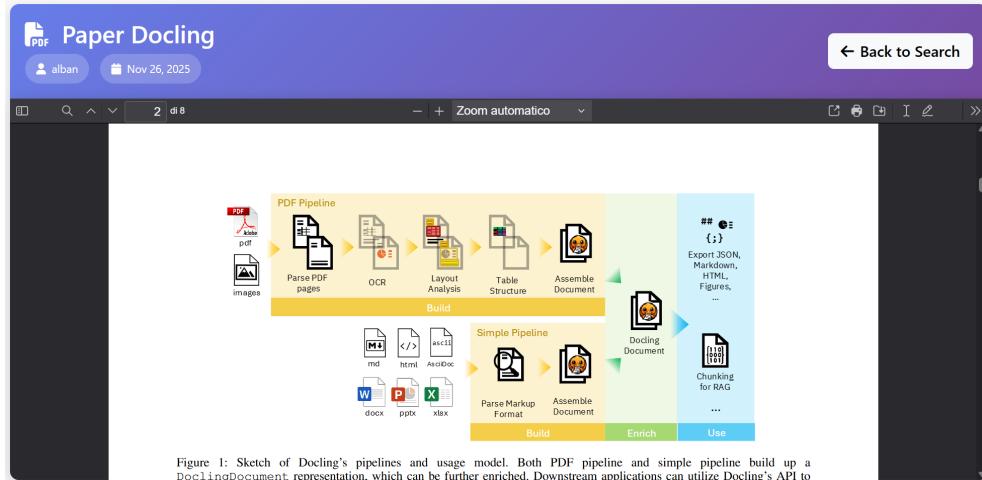


Figura 4.13: Visualizzazione del PDF con immagine

#### 4.2.2 Test di Ricerca su Documento Scansionato (OCR)

Questo test dimostra l'efficacia del modello di **DeepSeek-OCR** nell'estrare informazioni da documenti scansionati. In questo caso viene eseguito un unico test, perché la gestione della ricerca rimane invariata rispetto a quella precedente, cambia solo il metodo di processione, quindi in questo test si punta esclusivamente a valutare la qualità dell'estrazione testuale e tabellare.

##### Query 4

```
"Tabella strumenti con voltmetro?"
```

La **Figura 4.14** mostra il risultato di questa query, posta per ottenere la tabella del documento scansionato.

The screenshot shows a search interface for a scanned PDF. At the top, it says "esempio scan 2 results". Below that, the first result is displayed with the following details: "Page: 1", "Type: table", "Relevance: 0.5143", and a "View Page 1" button.

Strumento	Portata	Sensibilità	Precisione
Voltmetro	0 - 30 V	0.01 V	$\pm 0.5\%$ della lettura
Amperometro	0 - 1 A / 0 - 200 mA	0.1 mA	$\pm 0.5\%$ della lettura
Multimetro digitale	0 - 1000 V, 0 - 10 A, 0 - 6000 $\Omega$	0.001 V, 0.01 mA, 0.1 $\Omega$	$\pm 0.1\%$ della lettura
Generatore di tensione	0 - 30 V, max 5 A	0.01 V	$\pm 0.2\%$ della lettura

Below the table, the second result is shown with the following details: "Page: 1", "Type: text", "Relevance: 0.4062", and a "View Page 1" button.

**MATERIALI E STRUMENTI UTILIZZATI**

Componenti:

1. Resistori di vari valori (100, 10, 220, 1000) $\Omega$ .
2. Generatore di tensione (12V e 24V).
3. Breadboard per il montaggio del circuito.

Strumenti di misura:

1. Voltmetro.
2. Amperometro.
3. Multimetro digitale.

Software:

1. Multisim per la simulazione dei circuiti.

Figura 4.14: Risultato di ricerca su un PDF scansionato.

Dalla **Figura 4.14** si può notare come il modello sia stato in grado di individuare il contenuto testuale e tabellare dal documento scansionato, fornendo una risposta pertinente con una rilevanza di **0,51143**. La rilevanza minore del secondo risultato (che consiste nel testo dello stesso documento) è dovuta al fatto che nella query è stata richiesta specificamente la tabella. Successivamente, cliccando sul risultato, si accede alla schermata di visualizzazione.

La **Figura 4.15** illustra la funzionalità di visualizzazione del PDF scansionato, con il sistema che evidenzia automaticamente la sezione individuata

dal motore di ricerca come più rilevante.

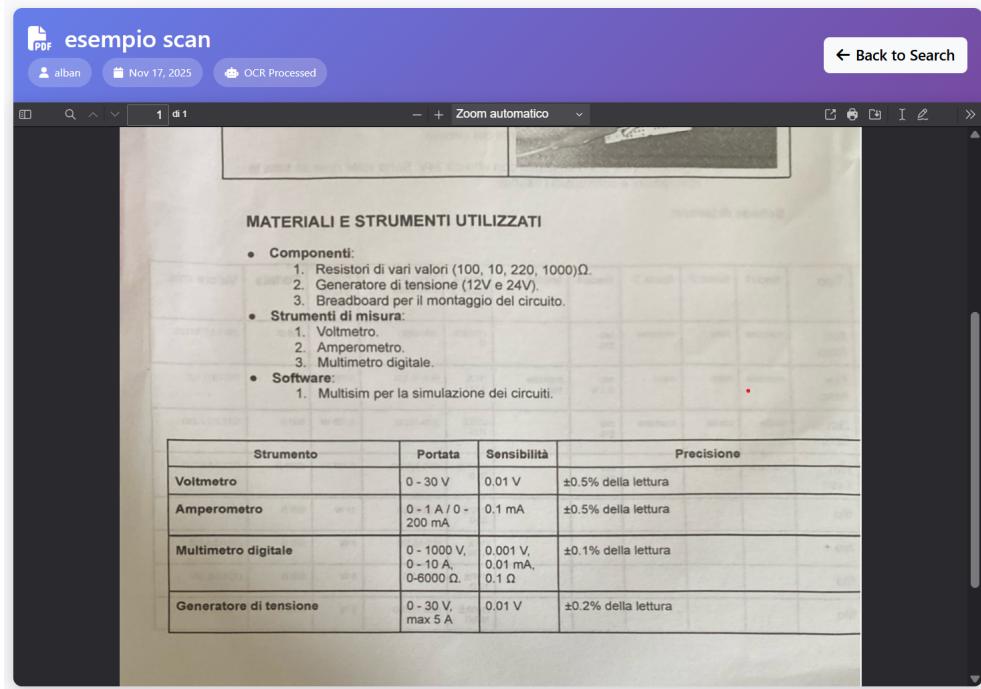


Figura 4.15: Visualizzazione di un PDF Scansionato

Per quanto riguarda la qualità dell'estrazione, il documento utilizzato presentava diversi fattori di intralcio come ombre, riflesso di una tabella posta nella facciata opposta e luce giallastra. Nonostante questi fattori, il modello OCR ha ottenuto prestazioni notevoli, riuscendo ad estrarre con alta precisione tutti i caratteri e simboli del documento.

#### 4.2.3 Indicatore di Rilevanza Semantica

Ogni risultato include un indicatore di rilevanza calcolato come  $1 - d$ , dove  $d$  è la distanza coseno tra l'embedding della query e quello del chunk. Il valore viene visualizzato con colorazione che varia dal verde (alta rilevanza) al rosso (bassa rilevanza):

- **Verde** ( $> 0.7$ ): Molto rilevante.
- **Azzurro** ( $0.5 - 0.7$ ): Rilevante.
- **Arancione** ( $0.3 - 0.5$ ): Moderatamente rilevante.
- **Rosso** ( $< 0.3$ ): Poco rilevante.

## 4.3 Test Sperimentale del Modello OCR

Al fine di validare l'efficacia del modello **DeepSeek-OCR**, è stata condotta un'analisi quantitativa su un dataset di documenti scansionati. L'obiettivo di questa sperimentazione è misurare la precisione dell'estrazione del testo, che costituisce il prerequisito fondamentale per una corretta indicizzazione RAG.

Le metriche utilizzate per la valutazione sono:

- **CER (Character Error Rate)**: La percentuale di caratteri errati rispetto al testo originale.
- **WER (Word Error Rate)**: La percentuale di parole errate.
- **Accuracy**: La precisione complessiva del riconoscimento ( $1 - CER$ ).

### 4.3.1 Performance Medie Complessive

In prima analisi, si osservano le prestazioni medie ottenute sull'intero dataset. Come illustrato nella Figura 4.16, il sistema ha dimostrato un'elevata affidabilità, con il **86,32%** di precisione.

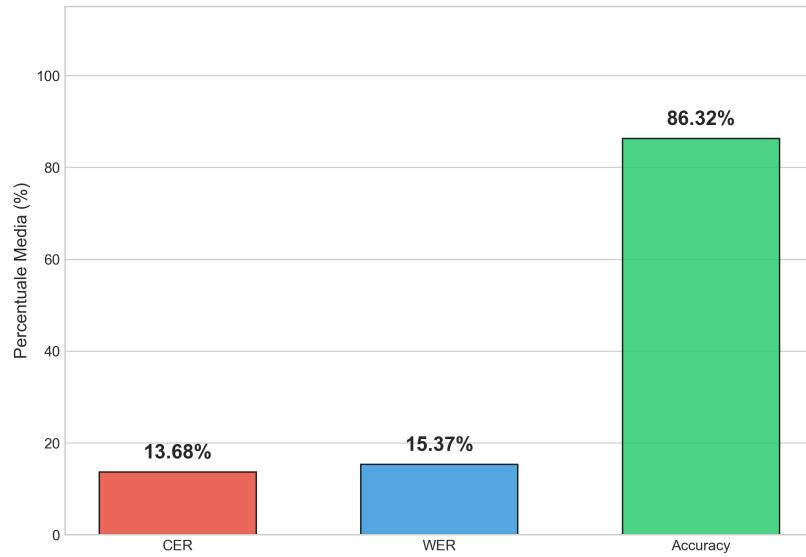


Figura 4.16: Medie complessive del dataset

I risultati evidenziano un’accuratezza media molto alta, confermando che il modello DeepSeek-OCR è in grado di gestire efficacemente la transcodifica da immagine a testo strutturato markdown, minimizzando gli errori di lettura.

### 4.3.2 Distribuzione della Qualità di Estrazione

Per comprendere meglio l'affidabilità del sistema, i documenti sono stati classificati in fasce di qualità (*Tiers*) basate sulla precisione ottenuta. Questa stratificazione permette di isolare i casi critici da quelli ottimali, offrendo una visione chiara del rendimento del modello.

I criteri di classificazione adottati sono i seguenti:

- **Eccellente ( $\geq 98\%$ ):** Il testo estratto è quasi identico all'originale.
- **Buono (90% – 98%):** Presenza di lievi refusi (es. punteggiatura o singoli caratteri) che raramente alterano il contesto globale della frase.

- **Sufficiente** (80% – 90%): Il testo è comprensibile ma presenta rumore evidente.
- **Insufficiente** (< 80%): La qualità dell'estrazione è compromessa, con rischio elevato o perdita di informazioni chiave.

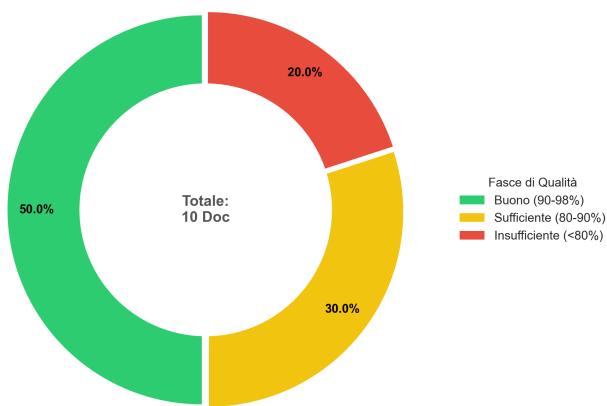


Figura 4.17: Distribuzione qualità

Come mostrato nella Figura 4.17, la maggioranza dei documenti ricade nelle fasce *Buono* e *Sufficiente*. Questo dato è fondamentale perché un'alta percentuale di documenti nelle fasce accettabili garantisce che i *chunks* generati siano semanticamente fedeli all'originale, riducendo il rischio di fornire motore di ricerca un contesto errato o corrotto.

### 4.3.3 Analisi Dettagliata per Documento

Infine, è stata eseguita un'analisi granulare per identificare eventuali criticità su specifici documenti. La Figura 4.18 mette a confronto CER, WER e Accuracy per ogni singolo file processato.

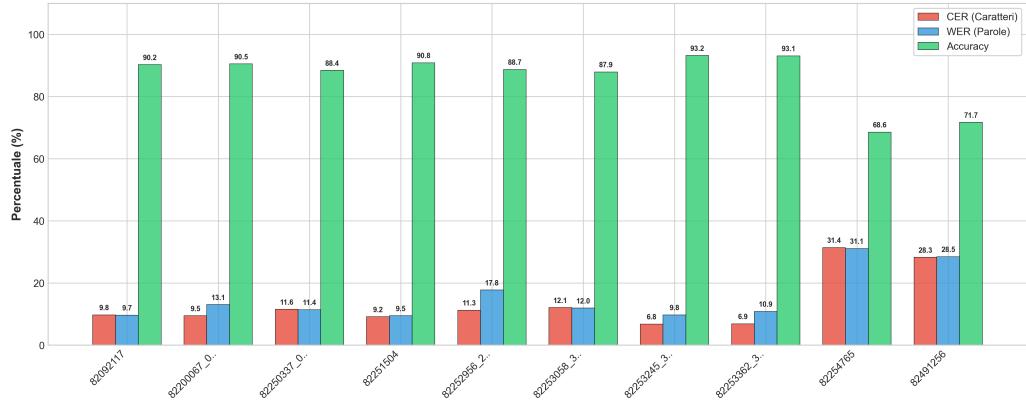


Figura 4.18: Performance OCR per Documento

Si nota che il WER tende ad essere leggermente superiore al CER, un comportamento fisiologico negli OCR: un singolo carattere errato compromette l'intera parola. Tuttavia, l'Accuracy si mantiene costantemente su livelli elevati, dimostrando la robustezza del modello anche su layout variabili.

La Tabella 4.19 riporta i dati puntuali dell'esperimento, includendo il confronto tra la lunghezza del testo originale (GT Len) e quella estratta (OCR Len), utile per verificare l'assenza di tagli o generazioni di testo eccessive.

Documento	CER	WER	Accuracy	GT Len	OCR Len
82092117	9.75%	9.69%	90.25%	1323	1356
82200067_0069	9.49%	13.07%	90.51%	1391	1472
82250337_0338	11.57%	11.44%	88.43%	1383	1492
82251504	9.16%	9.47%	90.84%	1397	1467
82252956_2958	11.30%	17.78%	88.70%	1336	1451
82253058_3059	12.10%	11.96%	87.90%	1174	1248
82253245_3247	6.80%	9.77%	93.20%	2398	2497
82253362_3364	6.91%	10.90%	93.09%	2576	2664
82254765	31.43%	31.11%	68.57%	649	751
82491256	28.30%	28.48%	71.70%	470	603
MEDIA	13.68%	15.37%	86.32%	-	-

Figura 4.19: Tabella riassuntiva dei risultati

## 4.4 Considerazioni sui Risultati

In questa sezione vengono analizzati i risultati ottenuti dai test presentati nelle sezioni precedenti, fornendo una valutazione complessiva delle performance del sistema DocSeek in termini di qualità dell'estrazione, tempi di elaborazione ed efficacia della ricerca semantica.

### 4.4.1 Riepilogo dei Test Effettuati

La **Tabella 4.1** riassume i risultati delle query di ricerca semantica eseguite durante la fase di testing, evidenziando la rilevanza ottenuta per ciascuna tipologia di contenuto.

Query	Tipo Contenuto	Tipo PDF	Rilevanza
Pipeline di Docling?	Testo	Nativo	0.6485
Tabella versioni e configurazioni?	Tabella	Nativo	0.6423
Immagine della pipeline di Docling?	Immagine	Nativo	0.7068
Tabella strumenti con voltmetro?	Tabella	Scansionato	0.5114

Tabella 4.1: Riepilogo dei risultati delle query di test

Dai risultati emerge che:

- La **ricerca di immagini** tramite didascalia ha ottenuto la rilevanza più alta (0.7068), dimostrando l'efficacia dell'indicizzazione basata sulle caption.
- La **ricerca testuale e tabellare** su PDF nativi ha prodotto risultati consistenti, con rilevanze superiori a 0.64.

- La **ricerca su documenti scansionati** presenta una rilevanza leggermente inferiore (0.5114), attribuibile alle caratteristiche del testo estratto via OCR.

#### 4.4.2 Analisi dei Tempi di Elaborazione

I test hanno permesso di raccogliere dati quantitativi sui tempi di elaborazione delle due pipeline implementate. La **Tabella 4.2** riporta i risultati ottenuti.

Metrica	PDF Nativo (Docling)	PDF Scansionato (OCR)
Documento di test	Paper Docling	Relazione di Laboratorio
Numero pagine	9	5
Tempo totale elaborazione	24.61 s	4 min 31 s
Caratteri estratti	21340	6779

Tabella 4.2: Confronto dei tempi di elaborazione

#### Dettaglio Elaborazione PDF Nativo

L'elaborazione del paper scientifico su Docling ha prodotto i seguenti risultati temporali:

- **Conversione e Parsing Docling:** 18.52 secondi
- **Creazione Chunks:** < 0.01 secondi
- **Indicizzazione ChromaDB:** 6.07 secondi
- **Tempo Totale:** 24.61 secondi

La maggior parte del tempo (circa il 75%) è dedicata alla fase di conversione tramite Docling, che include l'analisi della struttura del documento,

il riconoscimento delle tabelle e l'identificazione delle immagini con relative didascalie.

### Dettaglio Elaborazione PDF Scansionato

L'elaborazione OCR del documento scansionato di 5 pagine ha richiesto un tempo complessivo di **4 minuti e 31 secondi**, suddiviso nelle seguenti fasi:

**Fase 1: Elaborazione OCR su GPU (4 min 29 s)** Il processo OCR, monitorato attraverso i log del server GPU, ha mostrato:

- **Conversione PDF in immagini:** preparazione delle 5 pagine come immagini per l'elaborazione.
- **Elaborazione batch pagine 1-4:** Consiste nella processazione in parallelo di 4 pagine alla volta, in modo da diminuire i tempi di esecuzione sfruttando al massimo la potenza di calcolo della GPU.
- **Caratteri totali estratti:** 6779.

**Fase 2: Indicizzazione RAG (1.89 s)** Dopo il completamento dell'OCR, la fase di indicizzazione ha richiesto:

- **Creazione Chunks:** < 0.01 secondi
- **Indicizzazione ChromaDB:** 1.86 secondi
- **Tempo Totale RAG:** 1.89 secondi

Il tempo complessivo significativamente maggiore rispetto ai PDF nativi è dovuto principalmente alla comunicazione di rete con il server GPU remoto e all'inferenza del modello DeepSeek-OCR, che analizza ogni pagina come immagine.

### 4.4.3 Analisi dei Chunks Generati

La **Tabella 4.3** riporta la distribuzione dei chunks generati per tipologia di contenuto.

Documento	Chunks Totali	Testo	Tabelle	Immagini
Paper Docling (Nativo)	51	44	1	6
Relazione di Laboratorio (OCR)	10	9	1	0

Tabella 4.3: Distribuzione dei chunks generati

### 4.4.4 Qualità dell’Estrazione

#### PDF Nativi con Docling

L’elaborazione tramite Docling ha dimostrato eccellenti capacità di estrazione, anche su documenti con layout complessi:

- **Testo su colonne multiple:** il paper scientifico presentava un layout a due colonne, gestito correttamente mantenendo la coerenza semantica dei paragrafi;
- **Tabelle con struttura non standard:** come evidenziato nella Figura 4.11, il sistema ha ricostruito fedelmente tabelle delimitate solo da righe orizzontali;
- **Riconoscimento immagini:** sono state identificate 6 figure con relative didascalie, tutte correttamente indicizzate.

#### PDF Scansionati con DeepSeek-OCR

Il modello DeepSeek-OCR ha mostrato robustezza nell’elaborazione di documenti con imperfezioni tipiche della scansione:

- **Gestione del rumore visivo:** ombre, riflessi e illuminazione non uniforme non hanno compromesso la qualità dell'estrazione.
- **Alta precisione:** 6779 caratteri estratti con riconoscimento accurato di simboli e formattazioni.
- **Strutturazione tabelle:** ricostruzione della struttura tabellare in formato Markdown.

#### 4.4.5 Limitazioni Osservate

Durante i test sono emerse alcune limitazioni del sistema:

- **Ricerca immagini:** l'indicizzazione si basa esclusivamente sulle didascalie. Immagini prive di caption risultano difficilmente recuperabili tramite query semantiche sul contenuto visivo.
- **Dipendenza dalla qualità di scansione:** per i PDF scansionati, la qualità dell'estrazione OCR è correlata alla risoluzione e alle condizioni del documento originale.
- **Latenza OCR:** l'elaborazione richiede comunicazione con un server GPU remoto, con tempi nell'ordine dei minuti rispetto ai secondi dei PDF nativi.
- **Assenza riconoscimento immagini in OCR:** la pipeline OCR non implementa il riconoscimento di figure e didascalie, limitando la ricercabilità di contenuti visivi nei documenti scansionati.

# Bibliografia

- [1] Django Software Foundation. Django: The web framework for perfectionists with deadlines. <https://www.djangoproject.com/>, 2024.
- [2] Mozilla Foundation. Pdf.js: A general-purpose, web standards-based platform for parsing and rendering pdfs. <https://mozilla.github.io/pdf.js/>, 2024.
- [3] Lambda Labs. Lambda labs: Gpu cloud for ai. <https://lambdalabs.com/>, 2024.
- [4] Redis Ltd. Redis: The open source, in-memory data store. <https://redis.io/>, 2024.
- [5] Celery Project. Celery: Distributed task queue. <https://docs.celeryq.dev/>, 2024.
- [6] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-Networks. <https://arxiv.org/abs/1908.10084/>, 2019.
- [7] IBM Research. Docling: Document understanding framework. <https://github.com/DS4SD/docling>, 2024.

- [8] Bootstrap Team. Bootstrap. <https://getbootstrap.com/docs/5.3/getting-started/introduction/>, 2024. *Versione 5.x*.
- [9] Chroma Team. Chroma: Open-source embedding database. <https://www.trychroma.com/>, 2023.
- [10] Haoran Wei, Yaofeng Sun, and Yukun Li. Deepseek-ocr: Contexts optical compression. <https://github.com/deepseek-ai/DeepSeek-OCR>, 2025.