Politecnico di Torino

Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica

A.a. 2024/2025

Sessione di laurea Aprile 2025

Workflow assistito da AI per la creazione documentale: un approccio conversazionale basato su RAG

Relatori:

Luigi De Russis

Daniele Sabetta

Candidato:

Vincenzo Dalia

Ringraziamenti

Ringrazio il mio relatore, il Professore Luigi De Russis, per la sua immensa disponibilità e il suo prezioso supporto durante la realizzazione di questa tesi.

Ringrazio Daniele, Alessandro e tutti i colleghi di Orbyta per avermi concesso la loro fiducia e per avermi sostenuto nella realizzazione del progetto, è stato un piacere condividere con loro questa esperienza.

Ringrazio tutti i compagni di corso conosciuti durante questi anni, con i quali è nata una splendida amicizia e con cui ho condiviso momenti indimenticabili.

Ringrazio i miei amici più preziosi per avermi sempre incoraggiato e tirato su nei momenti peggiori, alleggerendo con scroscianti risate giornate in cui credevo di meritare nient'altro che silenzio.

Ringrazio Mamma e Papà per il loro amore incondizionato e per avermi accompagnato lungo questo percorso credendo costantemente in me, soprattutto quando ero io il primo a non esserne in grado. Siete le mie fondamenta, vi voglio bene.

Infine vorrei ringraziare Giorgio, mio fratello, l'unica persona in grado di sincronizzarsi con me con un solo sguardo. Grazie per la tua estrema sensibilità, per le nostre interminabili conversazioni e per avermi sempre spronato a dare il meglio, grazie a te ho trovato la forza di reagire anche nelle situazioni più difficili.
Mekaja.

Indice

Elenco delle tabelle

Elenco delle figure

Listings

Acronimi

LLM
: Large Language Model

NLP
: Natural Language Processing

RAG
: Retrieval Augmented Generation

API
: Apllication Programming Interface

UUID
: Universally unique identifier

WYSIWYG
: What You See Is What You Get

Capitolo 1

Introduzione

1.1 Contesto

Viviamo in un'epoca caratterizzata da una crescita esponenziale delle informazioni digitali, un panorama in cui l'accesso e l'elaborazione dei dati rappresentano non solo una sfida, ma anche una straordinaria opportunità [1]. Le tecnologie basate sull'intelligenza artificiale, in particolare i modelli di linguaggio di grandi dimensioni (LLM), hanno iniziato a plasmare profondamente il modo in cui individui e organizzazioni interagiscono con i contenuti [2]. Tuttavia, questo progresso tecnologico non è privo di problematiche. Gli utenti si trovano spesso disorientati di fronte all'enorme quantità di informazioni disponibili, desiderosi di strumenti che non solo facilitino l'accesso ai dati, ma che lo rendano anche più mirato, affidabile e trasparente. In particolare, in un contesto lavorativo sempre più dinamico e orientato alla produttività, professionisti come legali, consulenti delle risorse umane e altre figure che devono gestire grandi quantità di documenti e informazioni nel loro lavoro quotidiano, richiedono soluzioni tecnologiche intuitive e performanti, capaci di adattarsi alle loro esigenze in continua evoluzione. Per questi utenti, la sfida non è soltanto accedere ai dati, ma integrarli efficacemente nei propri flussi di lavoro, ottimizzando i tempi e garantendo qualità del risultato finale.

Mentre gli LLM sono straordinariamente potenti nel generare testo coerente e articolato, essi non garantiscono sempre la tracciabilità delle informazioni, sollevando interrogativi sull'affidabilità dei contenuti prodotti, i quali, talvolta, sono inesatti o inventati di sana pianta, producendo un fenomeno noto come "allucinazioni". Tale fenomeno risulta particolarmente deleterio e difficilmente arginabile [3]. Questo aspetto diventa ancora più rilevante in ambiti professionali, dove è imprescindibile verificare l'origine dei dati utilizzati e garantire che le risposte siano ancorate a contenuti solidi e verificabili.

Parallelamente, lo sviluppo di database vettoriali, che gestiscono dati rappresentati in forma, per l'appunto, vettoriale, detti embeddings, ha aperto nuovi orizzonti per il recupero e l'organizzazione delle informazioni. Tecnologie come il Retrieval-Augmented Generation (RAG) rappresentano una svolta in questo ambito, consentendo di combinare modelli di linguaggio avanzati con sistemi di retrieval che recuperano dati direttamente da fonti specifiche e rilevanti. Questo approccio non solo migliora la qualità e l'accuratezza delle risposte fornite, ma introduce anche un elemento di trasparenza che riduce il cosiddetto "effetto black box", potendo ricostruire il contesto utilizzato per fornire una determinata risposta.

In questo scenario, il progetto descritto in questa tesi si colloca come risposta a un'esigenza ben definita: offrire un sistema, con un design user-centric ispirato ai modelli più utilizzati e diffusi del momento, che unisca in un'unica piattaforma la potenza dei modelli di linguaggio avanzati e la possibilità di creare documentazione, automatizzando il processo di ricerca di informazioni all'interno di documenti testuali pre-caricati dall'utente, tramite il supporto dell'intelligenza artificiale.

Il seguente progetto di tesi è stato svolto in collaborazione con l'azienda Orbyta [4], all'interno del gruppo Orbyta Tech, ramo della compagnia che si occupa di consulenza informatica.

1.2 Obiettivi

Il principale obiettivo di questa tesi è lo sviluppo di un sistema informatico per supportare e velocizzare il lavoro di professionisti, come legali, consulenti delle risorse umane e altre figure che lavorano quotidianamente con grandi quantità di informazioni e documentazione strutturata, nella gestione e creazione di documentazione strutturata. In particolare, si intende realizzare una soluzione che automatizzi il recupero e l'inserimento di informazioni da fonti documentali preesistenti, al fine di facilitare la compilazione di documenti come contratti, documenti legali o testi strutturati.

Questo obiettivo principale può essere declinato nei seguenti sotto-obiettivi:

- Creare un sistema che permetta l'accesso rapido ed efficace delle informazioni contenute in documenti testuali
- Garantire la tracciabilità e verificabilità delle fonti utilizzate per l'estrazione delle informazioni
- Facilitare la compilazione di documentazione strutturata da parte degli utenti
- Realizzare un'esperienza utente semplice e intuitiva, misurabile attraverso un test di usabilità che valuterà l'intuitività e la facilità d'uso dell'applicazione sviluppata

Per raggiungere tali obiettivi, l'implementazione prevede lo sviluppo di un prototipo software sotto forma di web application che integri: un **chatbot** basato sul meccanismo di RAG per recuperare informazioni dai documenti caricati dall'utente, con funzionalità di verifica delle fonti, e un **editor** di testo di tipo WYSIWYG per la realizzazione di documenti strutturati che richiedono la compilazione di campi preimpostati (*placeholders*).

L'approccio tecnico prevede l'integrazione di soluzioni sia *Open Source* che proprietarie, bilanciando flessibilità e privacy con le prime, e performance ottimali con le seconde, offrendo all'utente un'esperienza personalizzabile in base alle proprie esigenze.

1.3 Struttura della tesi

Questa tesi è suddivisa in un totale di 6 capitoli, ognuno dei quali approfondisce nel dettaglio un determinato aspetto del progetto:

- **Capitolo 2 - Background:** In questo capitolo verrà offerta una panoramica dei domini di studio che forniscono un importante base per lo sviluppo di questo progetto, analizzando le tecnologie attualmente a disposizione. Verranno discussi principalmente i concetti teorici alla base del funzionamento dei Large Language Models e come questi si combinino con l'innovazione della Retrieval Augmented Generation.
- **Capitolo 3 - Progettazione:** In questo capitolo viene affrontata la progettazione dei diversi componenti del progetto, includendo riflessioni sulle strategie utilizzate e sulle scelte effettuate, discutendo le tecnologie impiegate. Verrà inoltre analizzata la progettazione dell'interfaccia finale e i requisiti funzionali dell'applicazione.
- **Capitolo 4 - Implementazione:** Nel quarto capitolo si entrerà nel dettaglio delle scelte implementative, spiegando come le diverse tecnologie trattate in precedenza siano state impiegate per ottenere il risultato finale, includendo immagini rappresentative dell'applicazione e snippet di codice, utili alla comprensione del funzionamento.
- **Capitolo 5 - Risultati e Valutazioni:** Nel quinto capitolo saranno discussi e analizzati i risultati ottenuti, con una valutazione qualitativa dell'output finale. In particolare, verranno esaminate le performance dei modelli impiegati attraverso specifiche metriche e verrà presentata e discussa l'analisi del test di usabilità condotto.
- **Capitolo 6 - Conclusioni:** Infine, nel sesto e ultimo capitolo, verranno inserite le riflessioni finali e saranno discussi possibili sviluppi futuri del

progetto attuale, proponendo spunti e idee per arrichire questo progetto e analizzando le limitazioni del prototipo attuale.

Capitolo 2

Background

Nel presente capitolo verranno introdotti e approfonditi i concetti chiave che costituiscono il fondamento teorico e tecnologico delle soluzioni proposte nel progetto, con particolare riferimento al campo del Natural Language Processing (NLP) e alla sua applicazione nei Large Language Models (LLM). Si partirà da una panoramica generale delle tecniche e degli approcci che hanno consentito ai modelli di linguaggio di evolversi fino a diventare strumenti avanzati per l'elaborazione, la comprensione e la generazione del linguaggio naturale, ambito che ricade all'interno del più generale concetto di Generative AI, il quale verrà discusso a sua volta.

In questo contesto, particolare attenzione sarà dedicata al concetto di rappresentazione vettoriale delle informazioni, attraverso cui dati complessi, come il linguaggio naturale e dunque documenti, vengono tradotti in forme numeriche compatte e manipolabili, comunemente note come embeddings.

Inoltre, verrà affrontato il tema del prompt engineering, ovvero la tecnica di formulare input ottimali per i modelli generativi, al fine di ottenere risposte più accurate, coerenti e mirate.

Sarà infine discusso come tali tecnologie convergano nelle tecniche di Retrieval-Augmented Generation (RAG), un paradigma che combina la capacità di recuperare conoscenze rilevanti da ampie repository di dati con la generazione di risposte precise e contestualizzate, ampliando così le potenzialità e i campi applicativi dei moderni sistemi basati sull'intelligenza artificiale.

2.1 Natural Language Processing

La Natural Language Processing (NLP) è un campo interdisciplinare che combina informatica, linguistica computazionale e intelligenza artificiale con l'obiettivo di consentire ai computer di comprendere, interpretare, generare e rispondere al linguaggio umano in modo efficace. Attraverso la NLP, i sistemi possono analizzare

grandi quantità di testo o parlato, estrapolando informazioni utili, identificando relazioni tra concetti e generando risposte coerenti. La sua importanza risiede nella capacità di rendere il linguaggio, che è intrinsecamente complesso e ambiguo, interpretabile da macchine che altrimenti operano su dati altamente strutturati.

I metodi tradizionali di NLP includevano approcci simbolici, come grammatiche formali e regole linguistiche, affiancati da tecniche statistiche per modellare la probabilità di eventi linguistici [5]. Negli ultimi anni, tuttavia, lo sviluppo di tecnologie basate sul deep learning, come i modelli di trasformatori comunemente detti transformers (ad esempio BERT o GPT), ha rivoluzionato il settore, permettendo una comprensione più approfondita delle sfumature semantiche e contestuali del linguaggio. In particolare, il primo transformer proposto da Vaswani ha introdotto il concetto di "self-attention", che ha migliorato enormemente l'efficienza e la capacità di modellare sequenze lunghe, diventando la base per i moderni modelli di NLP [6]. Questi modelli, pre-addestrati su grandi quantità di dati e successivamente specializzati per specifici compiti, hanno ottenuto risultati senza precedenti in una vasta gamma di applicazioni, tra cui l'analisi del sentiment, la traduzione automatica, la sintesi testuale e la risposta automatica alle domande [7].

Inoltre, la NLP si pone al centro di molte tecnologie di uso quotidiano, come assistenti vocali, chatbot avanzati per il servizio clienti e motori di ricerca. La sua adozione non si limita solo al contesto aziendale o tecnologico, ma ha impatti significativi in ambiti come la medicina, dove viene utilizzata per analizzare cartelle cliniche, o nel settore legale, per l'automazione dell'analisi dei contratti.

2.1.1 Obiettivi della NLP

Le principali attività della NLP si dividono in due grandi categorie: la comprensione e la generazione del linguaggio naturale. Tra le attività di comprensione rientrano compiti come il Part-of-Speech Tagging (POS), che consiste nell'assegnare a ciascuna parola una categoria grammaticale; il Named Entity Recognition (NER), che identifica entità rilevanti come nomi propri, date o luoghi all'interno di un testo; e la Dependency Parsing, che analizza la struttura sintattica di una frase per comprendere le relazioni tra le parole. Un'altra attività chiave è la Sentiment Analysis, utilizzata per determinare il tono emotivo di un testo, applicabile in ambiti come il marketing o l'analisi delle opinioni. Inoltre, la Text Classification permette di categorizzare documenti o frasi in base a criteri specifici, un esempio molto pratico e utilizzato al giorno d'oggi è lo spam detection, ossia la classificazione di un testo come spam o meno per filtrare i messaggi di posta elettronica in ingresso.

Sul fronte della generazione, la NLP si occupa di attività come la Text Summarization, che sintetizza contenuti estesi in versioni più brevi e coerenti; la Machine Translation, che converte automaticamente testi da una lingua a un'altra (es. Google Translate); e la Text Generation, utilizzata per creare contenuti a partire da una

data richiesta in input posta dall'utente, questo meccanismo, implementato sotto forma di dialogo, è il concetto alla base degli odierni chatbot. Un'altra area cruciale è l'Information Retrieval e Extraction, che consente di individuare e strutturare informazioni rilevanti in grandi quantità di dati testuali, ad esempio per motori di ricerca o analisi di documenti legali. Infine, compiti come lo Speech-to-Text e il Text-to-Speech completano il quadro, favorendo l'interazione uomo-macchina tramite linguaggio parlato.

Queste attività, grazie all'integrazione di modelli avanzati basati su deep learning e trasformatori, stanno rendendo la NLP sempre più efficace e pervasiva, con applicazioni che spaziano dall'automazione aziendale alla ricerca scientifica, migliorando l'accessibilità e l'interazione con la tecnologia.

2.1.2 Embeddings

Nel contesto della NLP, gli embeddings sono rappresentazioni vettoriali di parole, frasi o documenti in uno spazio continuo a bassa dimensionalità, progettati per catturare informazioni semantiche e sintattiche del linguaggio. Il concetto di embedding nasce dall'esigenza di rappresentare il linguaggio naturale in un formato che possa essere elaborato efficacemente da algoritmi di machine learning, specialmente quelli basati su reti neurali.

Un embedding associa a ciascuna parola un vettore di un certo numero di dimensioni, in cui le relazioni semantiche e contestuali tra le parole sono preservate. Ad esempio, parole simili o correlate tra loro (come "re" e "regina" o "gatto" e "cane") tendono ad avere vettori vicini nello spazio degli embeddings. Questo avviene grazie a tecniche di addestramento che sfruttano contesti testuali. Tra gli approcci principali per generare embeddings troviamo:

- Word2Vec [8]: Un modello che utilizza due architetture, Skip-Gram e CBOW (Continuous Bag of Words), per prevedere le parole circostanti o il contesto, rispettivamente.

- GloVe [9]: Un modello che sfrutta una matrice di co-occorrenze globale delle parole per apprendere relazioni semantiche.

- FastText [10]: Un'estensione di Word2Vec che considera anche i sotto-elementi (n-gram) delle parole, migliorando la rappresentazione di parole rare o non presenti nel vocabolario.

Con l'avvento dei modelli basati su trasformatori, come BERT e GPT, gli embeddings sono diventati contestuali, ovvero la rappresentazione di una parola varia a seconda del contesto in cui è utilizzata. Ad esempio, la parola "apple" avrà un embedding diverso se appare in un contesto culinario rispetto a un contesto

tecnologico. Questo rappresenta un passo avanti significativo rispetto agli embeddings statici, che assegnano un solo vettore per ogni parola, indipendentemente dal contesto.

2.2 Generative AI

L'intelligenza artificiale generativa è una tipologia di intelligenza artificiale in grado di produrre contenuti nuovi ed originali, come immagini, testi o video, a partire da un dataset col quale il modello è stato addestrato. A differenza dei modelli di intelligenza artificiali precedenti, il cui focus era principalmente la classificazione o regressione attraverso apposite tecniche di Machine Learning, e dunque utilizzabile in contesti limitati, la Generative AI ha impattato in modo profondo diversi settori, dalla produzione di contenuti alla ricerca scientifica.

L'IA generativa si basa su modelli di deep learning, reti neurali che vengono addestrate durante la cosiddetta fase di *training* su enormi quantità di dati, in questo modo la rete "impara" ad individuare dei pattern tra i diversi samples.

La fase di training è un processo computazionalmente oneroso in cui i vari samples, trasformati in rappresentazioni numeriche, attraversano la rete e attraverso complesse funzioni matematiche, di cui un esempio è la *Back-propagation*, si definiscono i *pesi* dei vari nodi che costituiscono la rete neurale.

Una volta terminata la fase di training, e dunque ricavati i pesi del modello, questo sarà pronto per generare dati nuovi partendo da un determinato input, che nella maggior parte delle applicazioni reali attualmente in uso sono costituiti da *prompt* di testo, ossia indicazioni precise sul tipo di contenuto da generare.

2.2.1 Large Language Model

Nel campo dell'intelligenza artificiale generativa, l'avvento degli LLM (Large Language Models) ha segnato una svolta significativa. Questi modelli, come suggerisce il nome stesso, vengono addestrati su vastissime quantità di dati testuali e sono in grado di generare linguaggio naturale con un livello di fluidità e coerenza in continuo aumento negli ultimi anni.

Il percorso verso gli attuali LLM ha avuto inizio con lo sviluppo di architetture relativamente semplici, un esempio sono i modelli *n-gram*, in cui, per generare la risposta dato un input, si procedeva con l'analisi di un contesto limitato, composto dagli n elementi precedenti alla parola da generare. In questi modelli, basati sulla predizione probabilistica della parola successiva, le principali limitazioni erano dovute all'incapacità di mantenere delle dipendenze relative all'intero contesto semantico globale del testo, proprio perchè il contesto tenuto in considerazione si muove come una *sliding window*, eslcudendo di volta in volta le parole meno recenti.

Per provare a superare le limitazioni dei modelli n-gram, sono state introdotte le *Recurrent Neural Network*, reti neurali progettate per mantenere una memoria degli input precedenti attraverso, appunto, connessioni ricorrenti. Il funzionamento si basa sull'utilizzo di un *livello nascosto*, realizzato mediante dei *loop*, il quale consente di reimmettere nella rete informazioni provenienti da passaggi precedenti e dunque di conservare informazioni sul contesto globale.

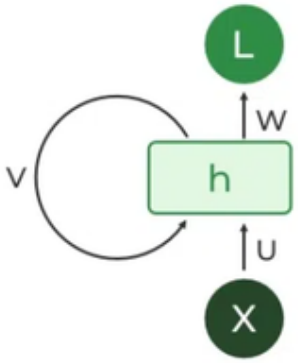

Figura 2.1: Neurone Ricorrente nelle RNN

Tuttavia, le RNN tradizionali presentano il problema del *vanishing gradient*, problema che si verifica durante la fase di addestramento in reti con molti strati e che si traduce nell'incapacità di apprendere ad un livello soddisfacente per la cattura delle dipendenze di lungo termine all'interno della rete.

Successivamente, sono state sviluppate le architetture LSTM (Long-Short Term Memory), una variante delle RNN classiche, progettate per gestire in modo più efficace le dipendenze a lungo termine. Questa tipologia di architettura affronta efficacemente il problema del vanishing gradient, migliorando la capacità di lavorare con sequenze più lunghe.

A differenza di una RNN classica, che si basa esclusivamente sull'output del nodo precedente — anche se derivato da più strati passati — per affinare la previsione, le LSTM introducono celle di memoria e meccanismi di gate, progettati per controllare il flusso delle informazioni all'interno della rete.

Nonostante i progressi apportati, queste architetture presentavano ancora problematiche nell'addestramento su larga scala e nella cattura di dipendenze molto lunghe. La vera svolta fu introdotta con lo sviluppo dell'architettura *Transformer*, già citata nel paragrafo 2.1. Il meccanismo di *self-attention* sul quale i transformer si basano, consente al modello di elaborare in modo parallelo, e non più sequenziale, tutte le parole di una certa sequenza, determinando per ciascuna il grado di rilevanza rispetto alle altre.

Questa architettura è composta da due elementi principali:

- **encoder**: elabora l'input nella sua interezza, effettuandone una *tokenizzazione* e generando i corrispondenti embedding, che sono sia relativi alle parole, codificando il significato semantico di ciascun token, sia posizionali, in modo da preservare informazioni sulla collocazione delle parole all'interno della sequenza

- **decoder**: a partire dall'output generato dall'encoder, applicando un mascheramento alle posizioni future della sequenza da generare, produce un output che sia contestualmente rilevante per poi essere reintrodotto come input nel decoder stesso per la generazione della parola successiva

I transformer hanno costituito la base per gli odierni LLM e hanno cambiato radicalmente il panorama dell'elaborazione del linguaggio naturale, superando le limitazioni riscontrate nei modelli precedenti riguardanti la capacità di catture dipendenze a lungo raggio e la velocità di addestramento.

2.2.2 Prompt Engineering

Nonostante le ottime prestazioni, in costante miglioramento nel corso del tempo, i Large Language Models non sono esenti da problematiche. Tra le principali, vi è quella relativa alle *"allucinazioni"*, ossia quel fenomeno per cui questi modelli generano informazioni all'apparenza plausibili, sia dal punto di vista sintattico che semantico, ma che in realtà non sono basate su dati reali. In altre parole, l'LLM produce contenuti inventati o inesatti, presentandoli come se fossero fatti concreti.

Per cercare di limitare questo fenomeno si è sviluppata una tecnica denominata *Prompt Engineering*, ossia l'arte di ingegnerizzare la richiesta da rivolgere, formulando un set di istruzioni efficaci, per ottenere i risultati desiderati dai modelli linguistici. Fornendo delle istruzioni chiare e specifiche, si riduce l'ambiguità della richiesta, inoltre, tramite questa tecnica, è possibile dare indicazioni precise rispetto al "comportamento" che il modello deve adottare per rispondere alle richieste. Tra gli approcci possibili del prompt engineering troviamo tecniche come few-shot learning (inclusione di esempi dimostrativi), chain-of-thought prompting (guida del modello attraverso passaggi di ragionamento espliciti) e role prompting (assegnazione al modello di un ruolo specifico). Tutte queste tecniche hanno dimostrato di migliorare significativamente le prestazioni dei modelli, tanto che il prompt engineering si sta evolvendo in una vera e propria disciplina formalizzata, con l'obiettivo di standardizzare le best practices da adottare per ottenere un risultato ottimale.

Tuttavia, è importante evidenziare che il prompt engineering non elimina completamente il rischio che il modello risponda in modo inesatto con delle allucinazioni, per questo motivo è sempre necessario verificare le risposte ottenute

2.3 Retrieval-Augmented Generation

Come già discusso, addestrare un modello di grandi dimensioni, ossia contenente un'elevata quantità di parametri da regolare, comporta un costo computazionale non indifferente. Quando si vuole adattare un modello pre-addestrato ad uno specifico dominio o compito, è possibile utilizzare un approccio chiamato *fine-tuning*, tecnica specifica di transfer learning che consiste nell'addestramento di alcuni o tutti i livelli del modello stesso con un nuovo dataset, spesso con un tasso di apprendimento più basso. Questa tecnica, in alcuni contesti, si rivela essere molto efficace, perché riduce drasticamente i tempi di adattamento del modello a nuovi contesti. Tuttavia, un modello fine-tuned potrebbe non riflettere informazioni aggiornate se non viene periodicamente riaddestrato, per questo motivo potrebbe essere poco indicato per applicazioni più dinamiche che richiedono un aggiornamento dei dati a disposizione estremamente veloce.

La **RAG** (Retrieval-Augmented Generation) rappresenta un approccio alternativo al fine-tuning, si tratta di una tecnica utilizzata per ottimizzare l'output di un modello generativo linguistico, combinando all'utilizzo del modello linguistico un sistema di estrazione delle informazioni contenute in una *knowledge base* esterna ai dati utilizzati per la fase di training del modello. Invece di modificare i parametri interni del modello esistente, la RAG integra all'interno del prompt un contesto di informazioni estratto da un database apposito, grazie al recupero dei documenti più pertinenti alla richiesta dell'utente. La RAG rappresenta dunque un approccio più flessibile e conveniente in scenari in cui le informazioni evolvono molto rapidamente.

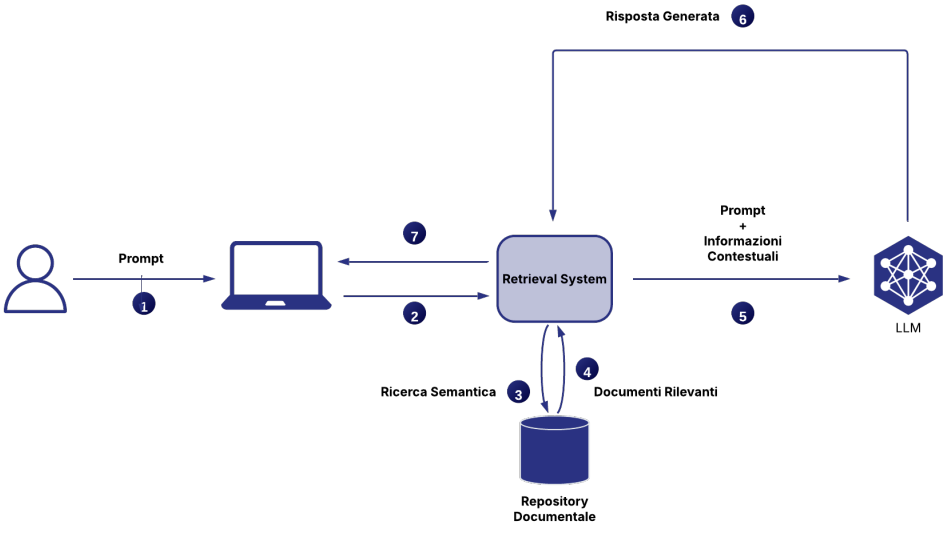

Figura 2.2: Schema funzionamento RAG

Il funzionamento della tecnica RAG, di cui è possibile visualizzare uno schema nella Figura 2.2, si basa sull'utilizzo di un modulo specifico, che prende il nome di **Retrieval System**. Questo modulo si frappone tra la richiesta dell'utente e l'LLM che deve fornire una risposta, andando ad arricchire la richiesta effettuata con un contesto recuperato da una **Repository Documentale**, che si trova all'interno di un database vettoriale. All'interno del suddetto database, vengono conservate le rappresentazioni vettoriali della knowledge-base, ossia gli embeddings, che in questo caso contengono in una singola istanza una porzione di testo del documento originale. La necessità di conservare i documenti in questa forma e non come semplici porzioni di testo all'interno di un comune database di tipo SQL o NoSQL si riconduce al meccanismo di ricerca tra i documenti relati alle domande dell'utente. Quando un utente effettua una richiesta, questa viene trasformata a sua volte in un embedding, subendo lo stesso processo che si occupa di convertire le varie porzioni di testo ricavate dal documento originale in vettori nello spazio. Ovviamente, l'embedding della richiesta avrà le stesse caratteristiche in termini di dimensionalità dei vettori contenuti nella repository documentale, in modo tale da poter performare una successiva ricerca per confronto tra i vettori. Una volta ottenuta la rappresentazione vettoriale della richiesta, viene effettuata una **ricerca semantica** all'interno del database, in modo tale da ottenere i **K** documenti (intesi come segmenti) più rilevanti.

Per implementare il meccanismo di ricerca semantica all'interno della repository documentale, la tecnica più utilizzata è la **similarità del coseno**, una misura che valuta la somiglianza tra due vettori nello spazio n-dimensionale, calcolando il coseno dell'angolo compreso tra essi. Questa metrica è ampiamente utilizzata nell'analisi testuale e nel recupero delle informazioni per confrontare documenti o segmenti di testo. Di seguito la formula per calcolare la similarità del coseno tra due vettori **A** e **B**:

$$\text{Similarità del coseno} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}} \tag{2.1}$$

In questa formula, **A** e **B** rappresentano i due vettori da confrontare. Il numeratore $\mathbf{A} \cdot \mathbf{B}$ indica il prodotto scalare dei due vettori, mentre il denominatore $||\mathbf{A}|| \times ||\mathbf{B}||$ è il prodotto delle loro norme (o lunghezze). Il risultato varia tra -1 e 1, dove 1 indica vettori identici, 0 indica vettori ortogonali (nessuna somiglianza) e -1 indica vettori diametralmente opposti.

Nell'ambito dell'analisi testuale, i vettori A e B spesso rappresentano la frequenza dei termini nei documenti. In questo contesto, la similarità del coseno misura quanto due documenti siano simili in base al contenuto delle parole, indipendentemente dalla loro lunghezza totale.

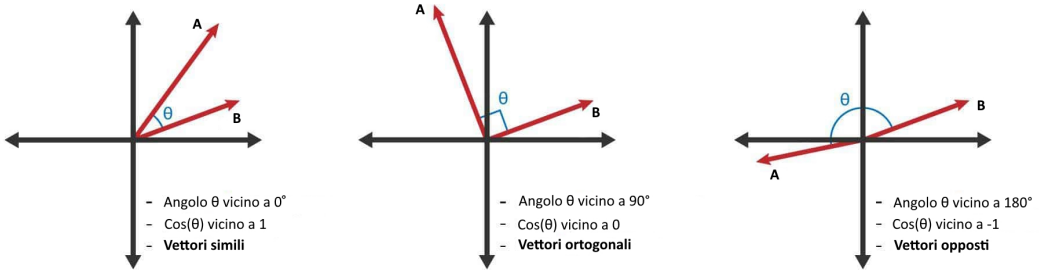

Figura 2.3: Similarità del coseno: 3 differenti casi

Una volta recuperati i K frammenti più rilevanti, viene realizzata una nuova richiesta utilizzando le tecniche di prompt engineering, includendo al suo interno, oltre alla domanda dell'utente, il contesto recuperato dal retrieval system. Si ottiene così una richiesta completa di fonti dal quale ricavare la risposta e con una precisa indicazione, definita a priori, dell'approccio con cui l'LLM dovrà rispondere. Questa richiesta viene finalmente inviata all'LLM, che a questo punto la processerà e restituirà una risposta completa e il più possibile pertinente, che verrà eventualmente ripulita e successivamente consegnata al client, corredata dalle fonti documentali dalle quali è stato possibile ricavare la risposta finale.

Capitolo 3

Progettazione

Il presente capitolo illustra nel dettaglio le fasi di progettazione del sistema, esaminando i diversi aspetti che compongono la soluzione sviluppata. In primo luogo, verranno esaminati i requisiti funzionali che l'applicazione deve soddisfare, delineando le capacità e le funzionalità essenziali del sistema. Successivamente, verrà presentata una panoramica dell'architettura software, evidenziando l'organizzazione dei diversi componenti e le loro interazioni. Particolare attenzione sarà dedicata alle scelte tecnologiche effettuate, giustificando l'adozione di specifiche soluzioni in relazione alle alternative disponibili e ai requisiti del progetto. Il capitolo si concluderà con la presentazione della prototipazione dell'interfaccia utente, illustrando le scelte di design volte a garantire un'esperienza d'uso intuitiva ed efficace.

3.1 Requisiti funzionali

In seguito ad un processo di brainstorming condotto all'interno dell'azienda, che ha coinvolto alcune delle figure target dell'applicazione, sono emerse alcune necessità che hanno successivamente guidato la progettazione del sistema, consentendo di definire i requisiti funzionali dell'applicazione.

Tra le necessità emerse durante queste sessioni sono state individuate:

- La necessità di poter caricare la propria documentazione all'interno della piattaforma e di mantenerla in modo permanente, in modo tale da non dover caricare ogni volta il documento dal quale estrarre le informazioni
- La necessità di accedere rapidamente alle informazioni contenute nei documenti archiviati, senza doverli rileggere per intero
- La possibilità di verificare la fonte di provenienza delle risposte fornite, in modo tale da poter controllare che si tratti di informazioni veritiere

- La possibilità di recuperare informazioni precedentemente richieste al chatbot senza la necessità di effettuare nuovamente le stesse richieste
- La necessità di poter realizzare nuova documentazione direttamente all'interno dell'applicazione, senza utilizzare software esterni, in modo assistito
- L'esigenza di conservare all'interno l'applicazione dei template documentali realizzati, in modo tale da poterli recuperare all'occorrenza

Queste necessità hanno costituito la base per l'elaborazione dei requisiti funzionali descritti di seguito, garantendo che il sistema rispondesse a esigenze concrete.

- **Gestione documentale** - Il sistema permette il caricamento permanente (o l'eliminazione) di documenti da parte dell'utente. Durante questa fase, i documenti vengono automaticamente segmentati in chunks di dimensioni appropriate, ottimizzati per il recupero delle informazioni. Questi segmenti vengono successivamente memorizzati nel database, mantenendo la relazione con il documento originale tramite l'utilizzo di metadati, utilizzati successivamente per la trasparenza delle fonti.
- **Interrogazione dei documenti** - L'applicazione implementa un chatbot basato su una retrieval chain che consente l'estrazione di informazioni dai documenti caricati tramite il meccanismo di Retrieval-Augmented Generation. Il processo prevede la vettorizzazione delle query utente e l'utilizzo di tecniche di prompt engineering per ottimizzare sia le domande in ingresso che le risposte generate.
- **Gestione delle conversazioni** - Le interazioni con il chatbot vengono automaticamente conservate nell'applicazione. Ogni conversazione viene identificata attraverso un titolo generato automaticamente tramite LLM, che riflette il contenuto della richiesta iniziale. L'utente ha la possibilità di gestire queste conversazioni, potendo sia recuperarle secondo necessità che eliminarle.
- **Rich Text Editor** - Il sistema include un editor di testo WYSIWYG (What You See Is What You Get) che permette la creazione e la modifica di documenti. I documenti creati possono essere esportati in formato PDF, garantendo la portabilità dei contenuti generati
- **Gestione dei template** - L'applicazione supporta la generazione di template documentali strutturati, contenenti placeholder nei punti in cui andrebbero inserite le informazioni reali, attraverso l'utilizzo di modelli di linguaggio. Questi template possono essere salvati all'interno dell'applicazione e recuperati successivamente, facilitando la standardizzazione dei documenti prodotti.

- **Sostituzione placeholders tramite ricerca** - Questa funzionalità permette di sostituire automaticamente i placeholder all'interno del testo dell'editor con informazioni pertinenti estratte dai documenti attraverso l'interazione con il chatbot, facilitando la creazione di documenti basati su dati esistenti.

- **Personalizzazione del sistema** - L'applicazione consente all'utente di configurare le proprie preferenze attraverso un pannello di impostazioni dedicato. Le opzioni di configurazione includono la selezione del modello di embedding per la vettorizzazione dei testi, la scelta del modello LLM per la generazione delle risposte. Per entrambe le tipoloogie di modelli, sono rese disponibili alternative open source o della famiglia OpenAI, disponibili qualora l'utente fosse dotato di una API Key. Tra i parametri, vi è a possibilità di attivare una ricerca ibrida che combina BM25 e similarity search semantica, nonché la definizione della lingua di output del chatbot.

3.2 Modelli AI utilizzati

3.2.1 Modelli LLM

Nel progetto sono stati integrati due diversi Large Language Models: Llama 3[11] e ChatGPT 3.5 Turbo[12].

LLama 3 è un modello sviluppato da Meta, appartenente alla categoria dei Large Language Models (LLMs). Questo modello rappresenta un'evoluzione rispetto ai suoi predecessori, con miglioramenti significativi in termini di capacità di generazione del linguaggio naturale, efficienza computazionale e adattabilità a diversi contesti applicativi. LLama 3 è stato addestrato su un vasto corpus di dati testuali ed è ottimizzato per attività come il completamento del testo, la risposta a domande, la traduzione automatica e l'assistenza conversazionale. Llama 3, eseguito localmente tramite Ollama, offre una soluzione gratuita che garantisce privacy dei dati, sebbene con prestazioni inferiori in termini di accuratezza e velocità di risposta, in quanto le prestazioni dipendono dall'architettura sul quale viene eseguito in locale, vi è dunque la necessità di elevate risorse computazionali per ottenere prestazioni adeguate.

Per ottenere prestazioni migliori è stato integrato il modello **ChatGPT 3.5 Turbo** della famiglia OpenAI, utilizzabile solo se l'utente ne sblocca l'utilizzo registrando all'interno della piattaforma la chiave API OpenAI, acquistabile presso il loro sito. Questo modello garantisce migliori performance, riducendo nettamente il tempo necessario per ottenere delle risposte, che sono più accurate rispetto al modello Llama, ma richiede una spesa proporzionale all'utilizzo. Infatti, per utilizzare questo modello ci si serve di apposite API esposte dal modello, che non viene eseguito localmente, bensì su server remoti con capacità computazionali

molto più elevate rispetto all'attrezzatura utilizzata in via di sviluppo. La scelta di non implementare modelli più recenti di ChatGPT, come GPT-4, è stata dettata dall'obiettivo di mantenere costi operativi contenuti, considerando che il modello 3.5 Turbo offre un equilibrio ottimale tra prestazioni e costo per token.

Questa architettura multi-modello permette all'utente di scegliere quello più adatto alle proprie esigenze, bilanciando costi, privacy, qualità delle risposte e il tempo necessario per ottenerle.

3.2.2 Modelli di Embedding

Seguendo la stessa logica che ha guidato la scelta dei modelli LLM, anche per quanto riguarda i modelli di embedding è stato scelto di integrarne diversi, con l'obiettivo di ottimizzare il bilanciamento tra accuratezza, efficienza computazionale e accessibilità. La scelta di utilizzare più modelli nel progetto permette di testare e confrontare le diverse soluzioni, valutando l'impatto delle dimensioni degli embeddings sulle performance e identificando il miglior compromesso tra accuratezza e velocità in base alle esigenze dell'applicazione.

Tra i modelli impiegati vi sono text-embedding-ada-002, LLaMA 2, BAAI/bge-small-en e all-MiniLM-L6-v2.gguf2.f16.gguf.

text-embedding-ada-002, sviluppato da OpenAI, è l'unico modello proprietario tra quelli selezionati ed è noto per la sua elevata qualità nella generazione di embeddings. Questo modello produce vettori di 1536 dimensioni, offrendo una rappresentazione ricca e dettagliata del testo, a scapito però di un maggiore costo computazionale, che in ogni caso non è a carico del dispositivo sul quale viene eseguita l'applicazione, in quanto utilizzato tramite API su server remoti, e di spazio rispetto a modelli con embedding più piccoli.

Gli altri modelli utilizzati sono open-source, garantendo maggiore flessibilità nell'integrazione e l'azzeramento dei costi economici per l'utilizzo, essendo eseguiti in locale.

LLaMA 2, sviluppato da Meta, offre embeddings con un'ottima capacità di generalizzazione, utilizzabile con diverse configurazioni di dimensionalità per gli embeddings. Nel progetto, è stato scelto di settare questo parametro a 4096 dimensioni, decisione motivata sia dalla capacità di rappresentare il testo in modo più dettagliato, sia dalla volontà di esplorare e testare diverse configurazioni. Infatti, questa sperimentazione consente di valutare l'impatto della dimensione degli embeddings sulle prestazioni, analizzando il compromesso tra accuratezza e costi computazionali.

Un'alternativa molto più leggera in termini di costo computazionale è offerta dal modello **all-MiniLM-L6-v2.gguf2.f16.gguf** della famiglia GPT4All. Con embeddings di 384 dimensioni, rappresenta un ulteriore compromesso tra efficienza e accuratezza, essendo ottimizzato per applicazioni a bassa latenza.

Infine, un'altra soluzione leggera è portata **BAAI/bge-small-en**, anch'esso basato su un parametro di dimensionalità pari a 384, dunque simile al modello della famiglia GPT4ALL.

3.3 Panoramica dell'Architettura

L'applicazione è stata progettata come una Single Page Application (SPA) che implementa un'architettura con una netta separazione tra frontend e backend. Le Single Page Application (SPA) rappresentano un'evoluzione rispetto ai modelli tradizionali di sviluppo web, come le Multi Page Application (MPA) e le architetture server-rendered. A differenza delle MPA, che prevedono il caricamento di una nuova pagina per ogni richiesta dell'utente, le SPA operano all'interno di un'unica pagina, aggiornando dinamicamente il contenuto tramite richieste asincrone al server. Questo approccio presenta diversi vantaggi, tra questi vi è il miglioramento significativo dell'esperienza utente, riducendo i tempi di caricamento e offrendo un'interazione più fluida e reattiva. Il sistema si compone di tre macro-componenti principali, a cui si aggiunge un ulteriore layer, composto dai modelli di AI della famiglia OpenAI eseguiti in remoto e accessibili tramite apposite API:

- **Backend**: Modulo centrale che gestisce la logica applicativa e il sistema di RAG, coordinando le diverse funzionalità ed esponendo al client i servizi mediante apposite API.
- **Database**: Layer dati che combina l'utilizzo di due database, uno di tipo relazionale per i dati strutturati e uno vettoriale per la gestione degli embedding dei documenti.
- **Frontend**: Modulo che, attraverso un'interfaccia utente progettata per offrire un'esperienza intuitiva, consente di comunicare con il sistema sottostante tramite un'interazione asincrona.

Nella Figura 3.1 viene mostrato uno schema dell'architettura con le rispettive relazioni tra i vari elementi che la compongono.

Le linee tratteggiate che connettono il Backend ai modelli di AI eseguiti in remoto riflettono la natura selettiva di tale comunicazione, che si realizza, quando necessario, attraverso interfacce API dedicate per il dialogo con i modelli cloud. Qualora la scelta ricada sui modelli Open Source, questi verranno eseguiti interamente all'interno dell'infrastruttura server locale, escludendo così qualsiasi necessità di interazione con ambienti esterni. L'utente si relaziona con l'applicazione mediante l'interfaccia grafica offerta dal frontend, attraverso la quale può elaborare documentazione nell'editor testuale integrato e definire le proprie preferenze di

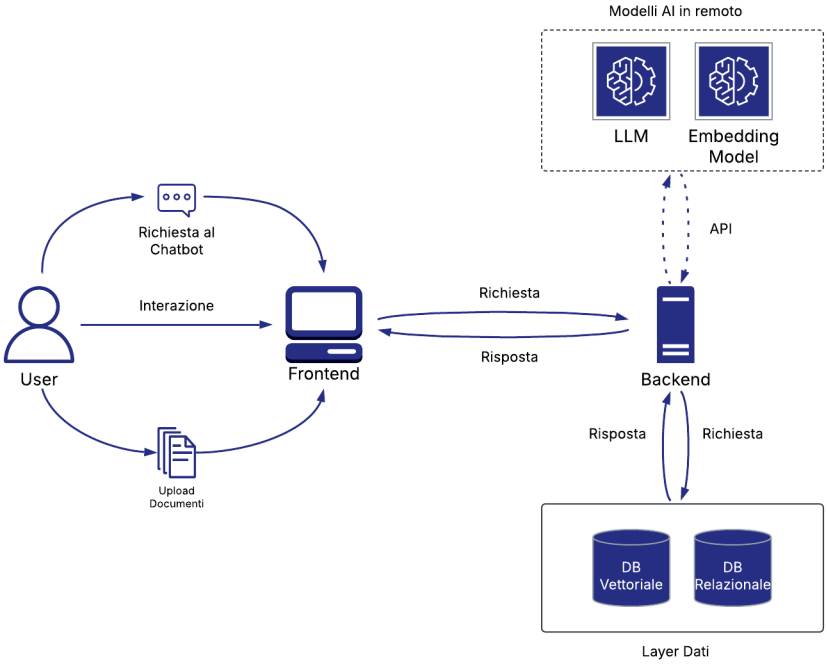

Figura 3.1: Architettura generale

utilizzo, ossia selezionare tra i diversi modelli di intelligenza artificiale, nell'apposita pagina dedicata. Nell'esperienza d'uso si distinguono due flussi principali: il caricamento di documenti testuali all'interno dell'applicazione, finalizzato alla creazione di una knowledge base, fonte di informazioni per successive consultazioni; e l'interazione tramite approccio conversazionale con il chatbot, sia per l'estrazione mirata di contenuti dalla knowledge base, sia per l'assistenza nella realizzazione dei testi all'interno dell'editor, attraverso la sostituzione automatica dei placeholder presenti o la generazione di nuovi template.

Nei successivi sotto-capitoli esploreremo in dettaglio ciascuno dei tre macro-componenti del sistema, analizzando le motivazioni che hanno guidato la selezione delle specifiche tecnologie adottate. L'obiettivo è fornire una panoramica critica delle scelte architetturali, evidenziando i vantaggi delle tecnologie selezionate rispetto alle alternative disponibili e illustrando come tali scelte rispondano ai requisiti funzionali e non funzionali del progetto.

3.4 Backend

L'architettura del backend è stata progettata secondo un approccio modulare, suddividendo i vari elementi in classi e componenti indipendenti. Questo consente una maggiore manutenibilità, facilitando l'aggiunta di nuove funzionalità senza stravolgere l'intero sistema. La suddivisione del codice in moduli, rende anche più semplice effettuare dei test sui singoli componenti, oltre a favorire una maggiore scalabilità dell'applicazione. Il backend dell'applicazione è stato sviluppato in **Python**, una scelta dettata dalla sua flessibilità e dall'ampia disponibilità di librerie specifiche per l'implementazione di meccanismi di RAG. Tra queste, un ruolo centrale è stato affidato a LangChain, una libreria progettata per semplificare l'integrazione tra LLM e sistemi di recupero delle informazioni.

LangChain offre un'astrazione avanzata per la gestione delle pipeline di AI, consentendo la composizione modulare di componenti come retriever, prompt manager, memoria contestuale e interfacce con database vettoriali come Qdrant, utilizzato all'interno di questa applicazione. Rispetto ad alternative come LlamaIndex, LangChain si distingue per una maggiore flessibilità nella gestione di flussi complessi e per una migliore compatibilità con un'ampia gamma di strumenti, rendendolo particolarmente adatto a scenari di RAG avanzati, consentendo per esempio l'utilizzo di retriever combinati.

Per la realizzazione del server è stato utilizzato **FastAPI**, un framework moderno che garantisce elevate prestazioni e una sintassi chiara ed espressiva. Rispetto a soluzioni come Flask e Django, FastAPI offre vantaggi significativi in termini di efficienza e scalabilità. Per esempio, a differenza di Flask, che è più orientato alla semplicità ma richiede estensioni per funzionalità aggiuntive, FastAPI supporta nativamente la gestione asincrona con async/await, migliorando la capacità di gestire richieste concorrenti con tempi di risposta ridotti. Invece, rispetto a Django, FastAPI si distingue per la sua leggerezza e rapidità di esecuzione. Django, sebbene offra un ecosistema robusto con un sistema ORM integrato e strumenti avanzati per la gestione dell'autenticazione e della serializzazione, è meno efficiente per applicazioni che richiedono un'elevata concorrenza e scalabilità. L'approccio sincrono di Django, infatti, può rappresentare un limite per sistemi ad alta intensità di richieste, mentre FastAPI, grazie alla sua architettura asincrona basata su Starlette, consente una gestione più efficiente delle operazioni I/O-intensive.

Tuttavia, nonostante FastAPI non fornisca un Object-Relational Mapping (ORM) di default, come avviene su Django che lo include nativamente tramite Django ORM, nell'applicazione sviluppata la gestione della mappatura tra il database e le entità applicative è stata implementata tramite SQLAlchemy, una delle librerie ORM più potenti e flessibili per Python. **SQLAlchemy** offre sia un'API declarativa, simile a quella di Django ORM, sia un livello più basso per un controllo più dettagliato delle query, rendendolo una scelta ideale per applicazioni

che richiedono prestazioni elevate e una gestione avanzata dei database.

Inoltre, grazie all'integrazione con Pydantic[1], FastAPI garantisce validazione automatica dei dati e una tipizzazione rigorosa.

Infine, per gestire le dipendenze Python del progetto è stato scelto **Poetry**[13], uno strumento moderno di gestione delle dipendenze e packaging che sostituisce l'uso combinato di pip, virtualenv e setuptools. La sua architettura è progettata per risolvere problemi comuni come il "dependency hell" e per fornire un'esperienza di sviluppo più coerente e affidabile rispetto agli strumenti tradizionali. Poetry implementa un sistema di risoluzione delle dipendenze deterministico attraverso il file *pyproject.toml*, che garantisce coerenza tra gli ambienti di sviluppo e produzione grazie al lock file *poetry.lock*. A differenza di pip, Poetry isola automaticamente le dipendenze in ambienti virtuali dedicati, prevenendo conflitti tra pacchetti e semplificando la gestione delle diverse versioni. La sua interfaccia CLI intuitiva facilita operazioni comuni come l'aggiunta, la rimozione e l'aggiornamento delle dipendenze.

3.4.1 Classi Principali

Tra le classi principali che compongono il backend di questo progetto troviamo la classe ***QdrantDBHandler***, che funge da interfaccia per la gestione delle operazioni sul database vettoriale Qdrant, stabilendo una connessione con il server Qdrant e definendo operazioni standardizzate per manipolare i dati, offrendo metodi per creare collezioni, inserire e recuperare dati, gestire impostazioni utente e rimuovere elementi specifici.

Troviamo poi la classe ***BaseEmbedder***, il suo obiettivo principale è gestire in modo flessibile il processo di generazione degli embeddings, ovvero le rappresentazioni vettoriali dei contenuti testuali che consentono la ricerca semantica. Supporta molteplici tecnologie di generazione di embeddings, permettendo di configurare dinamicamente il modello di embedding. Implementa metodi di suddivisione e preprocessamento dei documenti, che trasformano i testi grezzi in chunks, ossia porzioni di testo più piccole, e gestisce l'inserimento degli embeddings nel database vettoriale Qdrant. La dimensione dei chunks indica il numero di token per ognuno dei frammenti di testo realizzati a partire dal documento originale, tale parametro è stato fissato a 1000, per consentire ad ogni chunk di contenere abbastanza informazioni senza eccedere nella dimensione, la quale si sarebbe trasformata in un

[1]Pydantic è una libreria per Python progettata per la validazione e serializzazione dei dati, basata sull'uso delle type hints introdotte nelle versioni più recenti del linguaggio. Il suo obiettivo principale è garantire che i dati gestiti da un'applicazione siano conformi a schemi predefiniti, riducendo il rischio di errori e migliorando la robustezza del codice.

costo computazionale più alto e, nel caso dell'utilizzo del modello ChatGPT 3.5 Turbo, in un maggiore costo anche in termini economici.

Infine troviamo la classe ***RetrievalChain***, la quale rappresenta il cuore della logica di Retrieval-Augmented Generation nel sistema, orchestrando l'intero processo di recupero e generazione delle risposte. Progettata per gestire l'interazione tra il modello di linguaggio, il sistema di retrieval e la cronologia della conversazione, questa classe implementa meccanismi per riformulare le domande basati su tecniche di prompt engineering, recuperare documenti rilevanti e generare risposte contestualizzate. Supporta differenti modalità operative come la ricerca e risposta classica *QA* e la modalità *Find and Replace*, che verranno analizzate nella loro implementazione tecnica nel capitolo successivo, consentendo all'utente di interrogare semplicemente i documenti o di sostituire i placeholders nel testo in via di realizzazione. Il meccanismo alla base della ricerca per similarità è la trasformazione della domanda dell'utente in un embedding a sua volta e la restituzione dei 3 migliori risultati, individuati mediante l'algoritmo di *cosine similarity*, spiegato nel dettaglio nel capitolo 2. I risultati ottenuti costituiscono il *contesto* dato in pasto, insieme alla domanda, al Large Language Model per ottenere una risposta pertinente.

3.5 Struttura del Database

Come anticipato precedentemente, all'interno dell'applicazione sono stati utilizzate due tipologie di database:

- **Database relazionale:** Utilizzato per una gestione elementare degli *users*, *templates* e per alcuni aspetti di *documents* e *conversations*
- **Database vettoriale:** Utilizzato per conservare il contenuto effettivo delle conversazioni, le *settings* e soprattutto il contenuto dei documenti, sotto forma di chunks.

3.5.1 Database Vettoriale

La necessità dell'utilizzo di un database di tipo vettoriale dipende direttamente dall'utilizzo del meccanismo di RAG, alla base del quale vi è il recupero di dati attraverso l'utilizzo di algoritmi di ricerca per similiratà, come la *cosine similarity*, già citata nel Paragrafo 2.3.

All'interno di questa applicazione è stato deciso di utilizzare **Qdrant**[14], un database vettoriale progettato per offrire alte prestazioni nella ricerca Approximate Nearest Neighbor (ANN), grazie all'uso dell'algoritmo HNSW, che garantisce velocità e scalabilità anche su dataset di grandi dimensioni. La sua capacità di

combinare ricerca vettoriale con filtri basati su metadati lo rende particolarmente adatto alla necessità di manipolare gli elementi al suo interno in modo simile ad un classico db. L'integrazione con tecnologie come FastAPI e LangChain è immediata grazie alle API RESTful e gRPC, oltre ai client disponibili per Python e JavaScript, semplificando l'implementazione.

A differenza di FAISS, che opera principalmente in memoria, Qdrant supporta la persistenza su disco, garantendo che i dati vettoriali restino accessibili anche dopo un riavvio, requisito fondamentale per questo progetto. Offre diverse metriche di similarità, come la distanza coseno, Euclidea e il punto scalare, adattandosi a vari tipi di modelli di embedding, e permette di combinare la ricerca ANN con il filtraggio per ottenere risultati più pertinenti. La facilità di deployment su Docker, Kubernetes o macchine virtuali, unita alla possibilità di utilizzare Qdrant Cloud come servizio gestito, lo rende estremamente flessibile per diverse esigenze architetturali.

In Qdrant, le **collections** sono l'equivalente concettuale delle tabelle nei database tradizionali, ma sono ottimizzate per gestire e cercare vettori ad alta dimensionalità. In particolare, una collezione è un set di punti identificati tramite un UUID[2] tra i quali è possibile effettuare ricerche. Per ogni punto sono definiti due campi fondamentali:

- **payload**: qui vengono conservate le informazioni originali dei dati e/o informazioni aggiuntive, suddivise in ulteriori sottocampi
- **vector**: al suo interno vi è invece la rappresentazione vettoriale ricavata dal processo di embedding

```
{
  "id": 129,
  "vector": [0.1, 0.2, 0.3],
  "payload": {"color": "red"}
}
```

Listing 3.1: Esempio punto su Qdrant

Le collezioni presenti all'interno del database Qdrant dell'applicazione sono le seguenti:

- **`documents_collection_user_id`**: ne è presente una per ogni utente (user_id è infatti un campo variabile), per consentire la ricerca solo nei documenti

[2]Un UUID (Universally Unique Identifier) è un identificatore univoco, composto da 128 bit, usato per distinguere oggetti in un sistema distribuito. Viene generato secondo standard definiti (come UUIDv4, basato su casualità) per minimizzare il rischio di collisioni. È rappresentato in formato esadecimale separato da trattini, ad esempio: 550e8400-e29b-41d4-a716-446655440000.

caricati dal singolo utente, e contiene tutti i documenti caricati dall'utente, frammentati in porzioni di testo più piccole

- **chat_history**: in questa collezione sono compresi i messaggi delle conversazioni degli utenti, ogni punto della collezione rappresenta un messaggio

- **settings**: all'interno di questa collezione, ogni punto rappresenta le impostazioni di un utente, con tutti i parametri necessari al funzionamento dell'applicazione, ad esempio modelli e i relativi parametri

Analizziamo adesso nel dettaglio le tre collezioni e i campi che le compongono, in modo da comprendere la loro utilità. Per comodità di visualizzazione, è riportato lo screenshot (ricavato dalla Dashboard UI di Qdrant) di un punto per ogni collezione,

All'interno della collezione **documents_collection_`user_id`** vengono conservati i documenti del singolo utente, i quali, dopo aver attraversato la pipeline di *ingestion* che verrà descritta nel paragrafo successivo, vengono frammentati in porzioni di testo più piccole e trasformati in vettori. Per ogni punto di questa collezione, oltre al testo vengono conservati alcuni metadati descrittivi che verrano successivamente utilizzati per la tracciabilità delle fonti delle risposte.

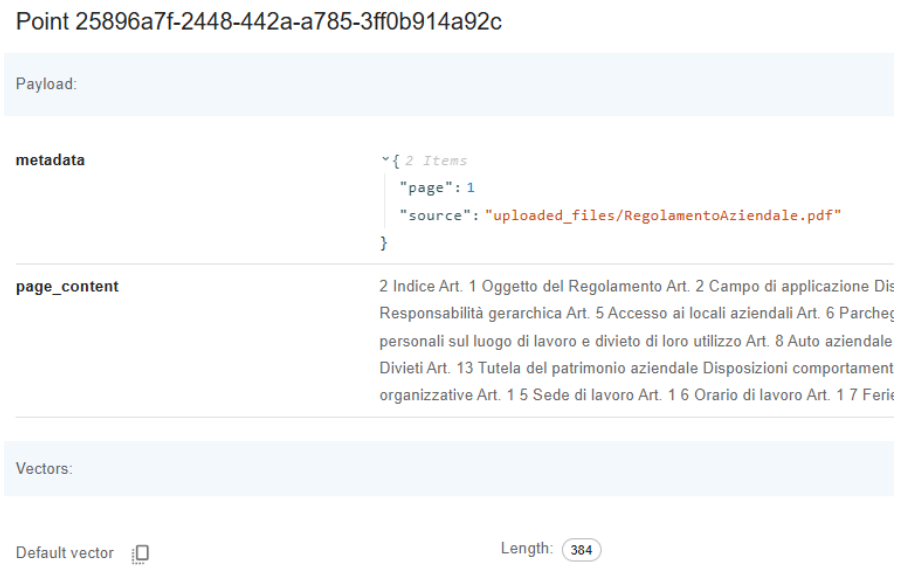

Figura 3.2: Punto della collection *documents_collection_`user_id`*

All'interno del payload, nel campo **page_content**, è conservato il testo effettivo di ogni frammento di documento, mentre nel campo **metadata** sono presenti ulteriori sottocampi:

- **source**: nome del documento all'interno del quale è contenuto il frammento di testo contenuto nel campo page_content
- **page**: indica il numero della pagina del documento nella quale si trova il testo contenuto nel campo page_content

È presente un'altra informazione, ossia la *Length*, cioè la lunghezza del vettore, questa dipende direttamente dal modello di embedding che viene utilizzato durante la fasi di ingestion dei documenti.

All'interno della collezione **settings** vengono conservate invece le impostazioni del singolo utente per l'applicazione. Ogni punto è associato ad un unico utente e all'interno del payload troviamo diversi campi. Come è possibile vedere all'interno della figura 3.3 sono presenti 5 campi relativi al modello di embedding attualmente selezionato dall'utente, nello specifico **embedding_family** rappresenta la famiglia di modelli alla quale appartiene lo specifico modello, identificato attraverso il suo nome con il campo **embedding_name**. In **embedding_dimensions** vi è l'informazione relativa alla dimensionalità dei vettori che il modello di embedding produce. Invece, **embedding_encode_kwargs** e **embedding_model_kwargs** sono dei campi necessari per un unico modello, nello specifico "BAAI/bge-small-en" di HuggingFace e servono a indicare se normalizzare o meno i vettori (tramite un valore booleano) e su quale device eseguire il modello ("cpu" o "gpu"), sono lasciati di default a `"normalize_embeddings": true` e `"device": "cpu"` per questo specifico modello. Per quanto riguarda il modello LLM attualmente selezionato dall'utente, abbiamo **llm_model** in cui è conservato il nome del modello, mentre **llm_temperature** è un parametro numerico di tipo *float* che controlla la casualità delle risposte: valori bassi rendono le risposte più deterministiche, mentre valori alti aumentano la creatività e la variabilità. In **openai_api_key** è conservata, sotto forma di stringa, il valore alfanumerico della chiave necessaria per utilizzare i modelli della famiglia di OpenAI. Troviamo anche il campo **hybrid_search**, un parametro booleano utilizzato per effettuare, quando settato a `true`, una ricerca combinata tra il retriever standard e il retriever BM25 di Langchain. Infine abbiamo il campo **user_id** che contiene l'UUID dell'utente al quale appartengono queste impostazioni, costituendo in un senso più lato un vincolo.

In questo caso, non viene costruito un vettore relativo ai punti della collezione, poiché non necessario ai fini di una successiva ricerca, per tale motivo la Length sarà in questo caso sempre pari a 0.

Figura 3.3: Punto della collection *settings*

Infine, troviamo la collezione **chat_history**, in questa possiamo trovare i messaggi scambiati tra i vari utenti e il chatbot.

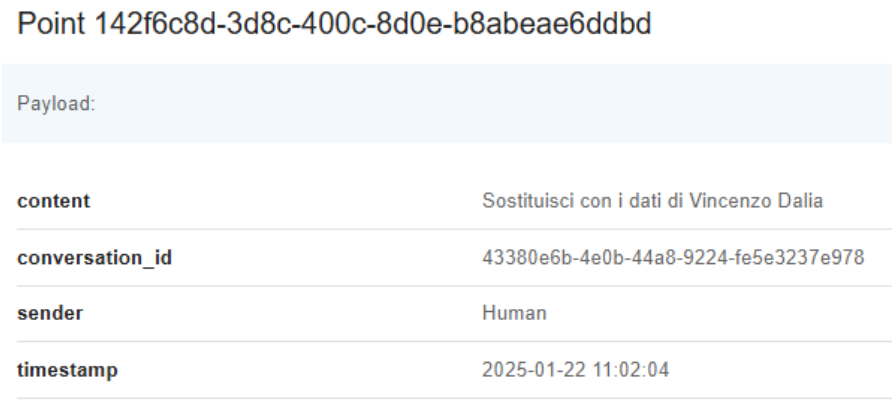

Figura 3.4: Punto della collection *chat_history*

All'interno del payload sono presenti 4 campi:

- **content**: contiene il testo del messaggio
- **conversation_id**: UUID che identifica la conversazione dalla quale proviene il messaggio e lo lega ad uno specifico utente, utile in combinazione alla logica del database relazionale per recuperare tutti i messaggi di una data conversazione da mostrare a schermo
- **sender**: stringa che identifica il mittente del messaggio, può assumere i valori "Human" o "AI"
- **timestamp**: data in cui il messaggio è stato inviato

Anche in questo caso, i punti di questa collezione non hanno una rappresentazione vettoriale, per le stesse motivazioni della collection *settings*.

3.5.2 Database Relazionale

Per quanto riguarda invece il database relazionale, questo è stato pensato per andare a contenere tutte quelle informazioni aggiuntive, più che altro descrittive, che riguardano documenti e conversazioni e per la gestione dei template realizzati dall'utente, in modo tale renderli riutilizzabili; è stata realizzata anche una tabella per gli users per implementare successivamente un meccanismo di autenticazione all'interno dell'applicazione finale, elemento non oggetto di questo progetto di tesi, in quanto aspetto non particolarmente rilevante ai fini dello studio. La tecnologia utilizzata per realizzare il database relazionale è **PostgreSQL**[15], un database relazionale open-source avanzato, noto per la sua robustezza, scalabilità e conformità agli standard SQL.

Come anticipato nel precedente paragrafo, all'interno di questo progetto è stato utilizzato **SQLAlchemy**, una libreria Python per la gestione dei database relazionali basata su ORM (Object-Relational Mapping), una tecnica che permette di interagire con un database relazionale usando oggetti e classi invece di query SQL. Tramite un'astrazione software, l'ORM mappa le tabelle del database su entità Python (e viceversa), semplificando la gestione dei dati e riducendo il rischio di errori manuali nelle query.

Analizziamo meglio le tabelle che compongono il database relazionale e come queste siano in relazione visualizzandone una rappresentazione grafica:

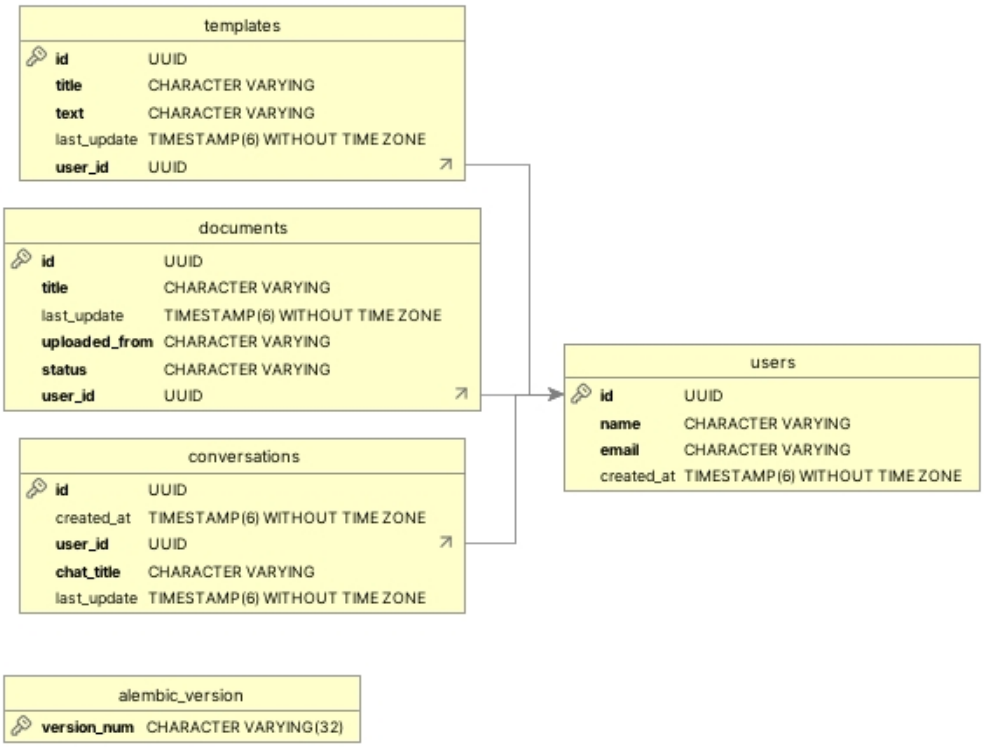

Figura 3.5: Schema tabelle del database

Come `PRIMARY KEY` di ogni tabella è stato deciso di utilizzare un UUID, soluzione che presenta tra i suoi vantaggi unicità distribuita e indipendenza dal database. Come è possibile notare nella figura 3.5, gli unici vincoli presenti nello schema del database sono quelli delle 3 tabelle *templates*, *documents* e *conversations* con la tabella *users*. In ognuna delle 3 tabelle il campo **user_id** è `FOREIGN KEY` e referenzia il campo **id** della tabella *users*, questo per legare ad ogni singolo utente uno o più elementi delle altre entità in modo univoco.

All'interno della struttura della tabella **documents**, è stato introdotto il campo *uploaded_from*, la cui funzione primaria consiste nell'identificare e tracciare la sorgente di provenienza dei documenti caricati nel sistema. Tale implementazione è stata realizzata in un'ottica di espandibilità futura dell'applicazione. Nella configurazione attuale del sistema, la procedura di caricamento dei documenti è circoscritta esclusivamente all'utilizzo del file system locale, pertanto il suddetto campo assume sempre il valore `"local"`. Tuttavia, in previsione di future estensioni funzionali, è stata predisposta la possibilità che il campo possa assumere altri valori (per esempio il valore `"cloud"`), nell'eventualità in cui venga implementata l'integrazione con servizi di archiviazione cloud, ampliando così le modalità di acquisizione dei documenti nel sistema. Il campo *status* è stato implementato

invece con lo scopo di rappresentare lo stato corrente del documento all'interno del processo di ingestion. Nella configurazione attuale, il campo può assumere due valori distinti: `"pending"`, che indica che il processo di ingestion del documento è ancora in corso di esecuzione, e `"active"`, che segnala il completamento con successo della procedura di ingestion. In previsione di future implementazioni, è stata contemplata la possibilità che il campo possa assumere il valore `"to_be_updated"`, specificamente nel caso in cui un documento precedentemente caricato tramite servizi cloud abbia subito modifiche nella sua sorgente originale, richiedendo così un aggiornamento del documento.

Per quanto riguarda i rimanenti campi che costituiscono la struttura delle tabelle del database, si evidenzia come questi siano stati denominati seguendo criteri di immediata comprensibilità e trasparenza semantica. Per tale motivo, sarà omessa una trattazione dettagliata di ciascun campo.

Per la gestione delle migrazioni del Database, è stato utilizzato **alembic**, una libreria Python utilizzata con **SQLAlchemy**. Consente di versionare, aggiornare e gestire la struttura del database in modo sicuro e controllato, generando automaticamente script di migrazione basati sui cambiamenti del modello dati. Per tale motivo, é presente un'ulteriore tabella, ossia **alembic_version**, che contiene un unico campo, *version_num*, contenente una stringa di caratteri che identifica la versione del database attualmente in uso.

3.6 Frontend

Per raggiungere gli obiettivi del progetto, la decisione è stata fin da subito orientata allo sviluppo di una applicazione web, poichè, per sua stessa natura, un'applicazione di tipo mobile sarebbe stata una scelta poco sensata, in quanto l'applicazione è concepita per essere utilizzata sul lavoro, prediligendo la comodità di utilizzo piuttosto che la portabilità. Inoltre, data la presenza di un editor di testo affiancato alla conversazione corrente, si è reso necessario privilegiare display di dimensioni adeguate per garantire una fruizione ottimale dell'applicazione.

L'architettura frontend è stata progettata seguendo i moderni paradigmi di sviluppo web, con un focus specifico sulla modularità e la manutenibilità del codice, utilizzando un approccio basato su componenti riutilizzabili che consentono di costruire interfacce modulari.

Nel valutare quale tecnologia utilizzare, sono state esaminate due tra le più utilizzate ed influenti in ambito di sviluppo web, entrambe realizzate in Javascript: React, una libreria caratterizzata dalla sua flessibilità , e Angular, un framework completo che offre soluzioni strutturate. Entrambe forniscono strumenti avanzati per la creazione di applicazioni single-page (SPA), migliorando l'esperienza utente

attraverso un rendering efficiente dei componenti e la gestione ottimizzata dello stato.

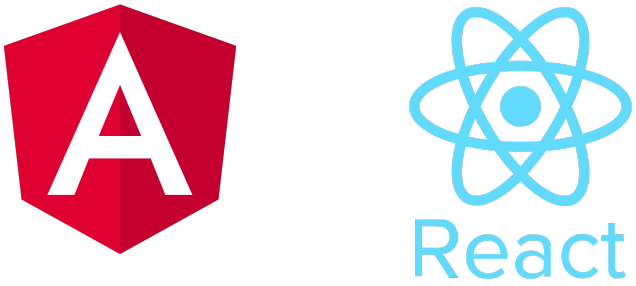

Figura 3.6: Loghi di Angular e React

Angular è un framework completo sviluppato da Google che utilizza TypeScript come linguaggio principale. È basato su un'architettura modulare e segue il paradigma MVC (Model-View-Controller) e fornisce una struttura ben definita per lo sviluppo, includendo strumenti come il data binding bidirezionale, la dependency injection e una gestione avanzata dello stato tramite servizi e osservabili (RxJS). La sua curva di apprendimento è più ripida rispetto ad altre soluzioni, ma offre una maggiore coerenza e scalabilità per progetti di grandi dimensioni.

React, sviluppato da Meta (ex Facebook), è una libreria focalizzata sulla creazione di interfacce utente tramite componenti riutilizzabili. Il suo principale punto di forza è il Virtual DOM, che ottimizza il rendering riducendo il numero di operazioni necessarie per aggiornare l'interfaccia. React utilizza JSX, una sintassi che combina JavaScript e HTML, facilitando la creazione di componenti interattivi. Grazie alla sua flessibilità, React consente di integrare facilmente librerie di terze parti, consentendo l'estensione delle funzionalità dell'applicazione in modo modulare e personalizzabile, senza vincoli imposti da un framework rigido.

Nel confronto tra le due tecnologie, Angular si distingue per la sua struttura rigida e completa, adatta a progetti complessi che richiedono un'organizzazione ben definita e il supporto nativo per molte funzionalità avanzate. Questa rigidità può però essere una limitazione nel caso di progetti che richiedono una maggiore flessibilità. A differenza di Angular, React è più leggero e modulare, permettendo di adottare solo le funzionalità necessarie in base alle esigenze specifiche dell'applicazione, inoltre gode di una ampia community e riceve un supporto continuo da parte di Meta.

Come è possibile vedere nelle figure 3.7 e 3.8, ricavate da due survey [16][17] condotti da StackOverflow[3], le percentuali di popolarità di React e del numero

[3]Stack Overflow è una piattaforma online di domande e risposte dedicata alla programmazione

di domande che lo riguardano, risultano essere nettamente più alte rispetto a quelle di Angular (e di altri frameworks non tenuti in considerazione), dimostrando un maggiore interesse da parte della community degli sviluppatori. Questo dato evidenzia non solo la diffusione di React nel settore dello sviluppo web, ma anche il continuo supporto e aggiornamento della libreria, facilitato dalla vasta disponibilità di risorse, documentazione e soluzioni condivise dagli utenti.

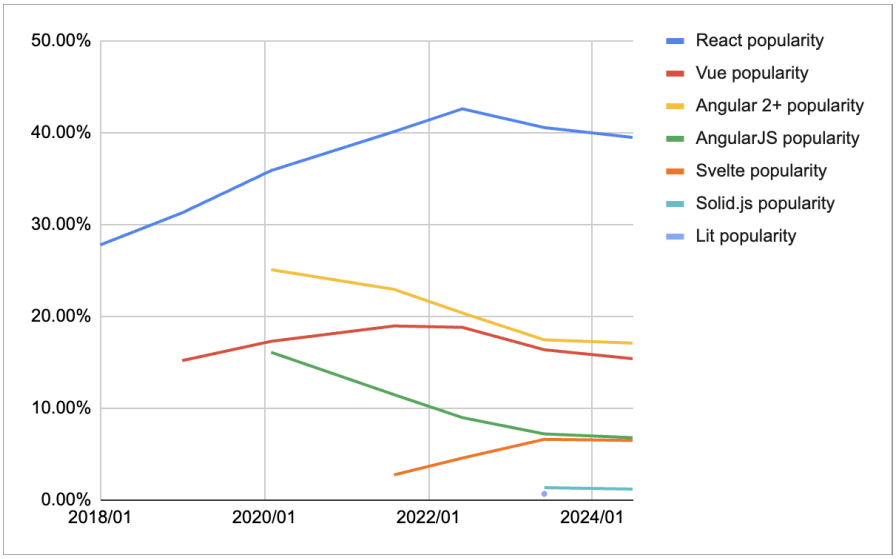

Figura 3.7: Percentuale nel tempo della popolarità di React vs altri Frameworks

Alla luce di queste considerazioni, per lo sviluppo dell'applicazione presentata in questa tesi si è scelto di adottare React, combinato all'utilizzo di Vite, un build tool moderno che offre notevoli vantaggi in termini di velocità e semplicità d'uso. Vite si distingue per la sua capacità di avviare il server di sviluppo in modo pressoché istantaneo, grazie all'utilizzo dei moduli ES nativi che eliminano la necessità di un bundling iniziale. Durante lo sviluppo, questa caratteristica, unita a un Hot Module Replacement estremamente rapido, permette di visualizzare immediatamente le modifiche al codice, migliorando significativamente l'efficienza del processo di sviluppo.

e allo sviluppo software. Gli utenti possono porre domande, fornire risposte e votare i contributi, creando un archivio di soluzioni per problemi tecnici. È una delle risorse più utilizzate dagli sviluppatori per trovare supporto e condividere conoscenze.

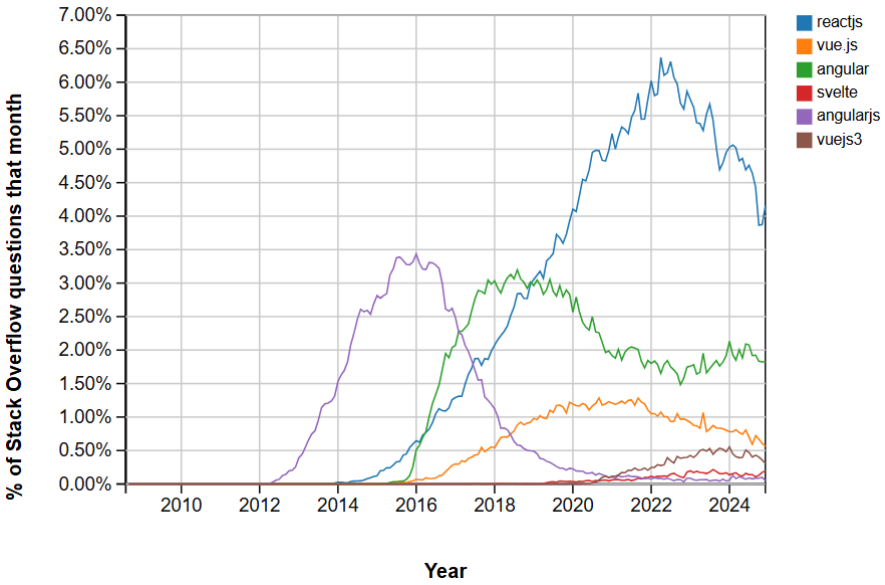

Figura 3.8: Percentuale nel tempo di domande su StackOverflow riguardanti React vs altri Frameworks

Hooks di React

Nello scegliere React come tecnologia per lo sviluppo del frontend, è stata presa fortemente in considerazione la presenza degli Hooks, una delle sue caratteristiche più innovative, funzioni su cui si basa la logica di React.

Gli Hooks, introdotti in React 16.8, hanno reso più efficiente la gestione dello stato e del ciclo di vita nei componenti funzionali. Questo cambiamento ha semplificato la scrittura del codice, migliorandone leggibilità e manutenibilità. Grazie agli Hooks, infatti, è possibile gestire lo stato locale, eseguire effetti collaterali e condividere logica tra componenti in modo più modulare. Inoltre, il loro utilizzo riduce la complessità del codice, evitando l'uso di metodi di lifecycle e facilitando l'organizzazione del flusso dati all'interno dell'applicazione.

Tra gli Hooks principali, quello fondamentale e il più frequentemente utilizzato è **useState**, che permette ai componenti funzionali di gestire il proprio stato interno. Questo Hook ritorna una coppia di valori: lo stato corrente e una funzione per aggiornarlo, consentendo una gestione dello stato chiara e prevedibile. La sua semplicità d'uso non ne limita la potenza, permettendo di gestire sia stati semplici che complessi con la stessa eleganza sintattica.

useEffect rappresenta la soluzione di React per gestire gli effetti collaterali nei componenti funzionali. Questo Hook permette di eseguire operazioni come chiamate API, sottoscrizioni a eventi, o manipolazioni del DOM dopo che React ha

aggiornato il DOM stesso. La sua flessibilità si manifesta attraverso il sistema di dipendenze, che permette di controllare precisamente quando l'effetto deve essere rieseguito, ottimizzando così le prestazioni dell'applicazione. Sostituisce le vecchie funzioni `componentDidMount` , `componentDidUpdate` e `componentWillUnmount`, per gestire l'esecuzione di una porzione di codice in base allo stato del componente relativamente al suo ciclo di vita.

Per la condivisione di stato tra componenti distanti nell'albero dei componenti, **useContext** emerge come una soluzione elegante ed efficace. Questo Hook consente di accedere al contesto React senza introdurre componenti annidati aggiuntivi, semplificando notevolmente la gestione dello stato globale dell'applicazione. useContext offre un'alternativa più leggera e intuitiva rispetto al *prop drilling*[4], migliorando la manutenibilità e la leggibilità del codice.

Infine troviamo **useParams**, hook facente parte di React Router DOM, una libreria utilizzata per la gestione della navigazione tra le diverse routes all'interno di applicazioni sviluppate in React. Grazie a questo hook è possibile accedere ai parametri dinamici dell'URL corrente, per esempio l'id di un elemento, rendendo così più semplice la gestione di contenuti dell'applicazione basati sui parametri dell'URL.

React Bootstrap

Per quanto riguarda invece la realizzazione dell'interfaccia utente dal punto di vista estetico, la scelta di React come framework di sviluppo viene ulteriormente rafforzata dalla possibilità di integrare strumenti che ne potenziano l'efficacia e ne semplificano l'utilizzo. Tra questi strumenti, React Bootstrap emerge come una soluzione particolarmente vantaggiosa per la gestione degli elementi grafici, offrendo un ponte naturale tra la potenza di React e la popolarità di Bootstrap.

React Bootstrap è una completa reimplementazione dei componenti Bootstrap nativi, specificamente progettata per integrarsi perfettamente con React. Questa libreria elimina completamente la dipendenza da jQuery, presente invece nella versione standard di Bootstrap, rendendo il codice più leggero e performante all'interno dell'ecosistema React. Ciò che caratterizza maggiormente React Bootstrap è l'utilizzo di componenti React già pronti all'uso, come ad esempio bottoni o tabelle, che mantengono l'estetica di Bootstrap, consentendo agli sviluppatori di creare interfacce utente moderne riducendo di molto lo sforzo. Questi componenti possono inoltre essere liberamente personalizzati, aggiungendo delle regole specifiche di CSS definite dall'utente. Inoltre, i componenti seguono le best practice di React,

[4]Il prop drilling è una situazione che si verifica quando le props devono essere passate attraverso più livelli di componenti che non ne hanno direttamente bisogno, solo per raggiungere un componente più profondo nell'albero che effettivamente le utilizza.

consentendo così di gestire correttamente gli stati e gli eventi, integrandosi al paradigma di sviluppo di React. La libreria offre anche un'eccellente compatibilità cross-browser e una struttura modulare che permette di importare solo i componenti effettivamente necessari, ottimizzando così le dimensioni finali del bundle dell'applicazione. React Bootstrap semplifica notevolmente il processo di sviluppo grazie alla sua API intuitiva e alla possibilità di utilizzare props per la configurazione dei componenti, eliminando la necessità di scrivere classi CSS personalizzate per le funzionalità più comuni. Anche in questo caso, la manutenibilità del codice è garantita dall'approccio component-based di React Bootstrap, che incoraggia una struttura del codice più organizzata e riutilizzabile. Gli sviluppatori possono facilmente estendere e personalizzare i componenti esistenti per soddisfare requisiti specifici del progetto, come è stato fatto all'interno del progetto per specifiche classi di componenti di cui si voleva ottenere un determinato aspetto estetico, mantenendo al contempo la coerenza visiva e funzionale dell'interfaccia utente.

3.6.1 Editor di testo

Nel contesto del progetto, sono stati valutati diversi editor WYSIWYG per la gestione dei documenti, tra cui Lexical e Quill. L'obiettivo era individuare una soluzione che garantisse buone prestazioni, facilità di integrazione con l'architettura del frontend e un livello adeguato di personalizzazione. Tra i Rich Text Editor più utilizzati, Lexical e Quill rappresentano due soluzioni moderne e altamente performanti.

Quill è un editor di testo di tipo WYSIWYG open source sviluppato in JavaScript, è tra i più popolari ed è conosciuto per la semplicità di implementazione all'interno delle applicazioni. Essendo disponibile da diverso tempo, poichè rilasciato nel 2014, è disponibile un'ampia documentazione ed è facile trovare soluzioni nella community degli utenti per alcuni dei problemi più comuni. Tuttavia, il suo approccio, nonostante sia modulare e dunque personalizzabile, risulta essere poco flessibile in contesti in cui è necessario ampliare le funzionalità di base dell'editor.

Lexical, invece, è un editor di testo open source sviluppato da Meta. Essendo stato rilasciato nel 2022, risulta essere una soluzione più moderna e in costante sviluppo. Anche Lexical è implementato con un approccio di tipo modulare basato su componenti chiamati plugin, ma a differenza di Quill risulta molto più personalizzabile, lasciando allo sviluppatore la possibilità di definire da zero e in modo semplici nuovi plugin per arricchire le funzionalità dell'editor. Lexical utilizza una struttura a nodi per rappresentare il contenuto dell'editor in modo gerarchico. In questa struttura esistono diversi tipi di nodi: i nodi radice che contengono il documento, i nodi elemento che contengono per esempio paragrafi, intestazioni ed elenchi, e i nodi foglia che contengono il testo effettivo. Questa architettura permette di ottenere una manipolazione maggiore del testo rispetto a quanto

possibile con Quill. Tra i due, Lexical risulta più efficiente nel rendering e nella gestione dello stato dell'editor, riducendo il rischio di rallentamenti perchè, quando avviene una modifica, Lexical non aggiorna immediatamente il DOM come fa Quill, ma crea prima una rappresentazione virtuale delle modifiche. In questo modo, può aggregare più cambiamenti e applicarli tutti insieme in un unico aggiornamento del DOM, riducendo il numero di manipolazioni dirette del DOM necessarie.

Dopo un confronto tra le due tecnologie, la scelta è ricaduta su Lexical, grazie alla sua maggiore modularità, efficienza e integrazione con React. Tuttavia, nel progetto è stata adottata un'implementazione minimale di Lexical, scegliendo solo alcuni dei plugin disponibili, in linea con le necessità dell'applicazione, ma la sua predisposizione alla personalizzazione lascia aperta la possibilità, in futuri sviluppi del prototipo, di ampliare le funzionalità dell'editor stesso.

Tra questi, sono stati impiegati i plugin per la formattazione del testo, essenziali per garantire un controllo intuitivo su aspetti come grassetto, corsivo e titoli attraverso una semplice toolbar. Data la natura dell'applicazione, focalizzata sulla produzione di semplici documenti strutturati di tipo aziendale o legale, non è stato necessario integrare funzionalità più avanzate come l'inserimento di elementi multimediali o la collaborazione in tempo reale.

Inoltre, è stato utilizzato il plugin per la gestione del markdown di Lexical, il quale permette di gestire l'input e l'output di testo formattato in markdown all'interno dell'editor. Il funzionamento è piuttosto semplice: il plugin "ascolta" ciò che viene digitato dall'utente o, più in generale, inserito all'interno dell'editor e, quando riconosce una specifica sequenza di caratteri che corrisponde a una regola Markdown, interviene per trasformare il testo in un nodo dell'editor con la formattazione appropriata.

In ottica di cooperazione con il plugin di Markdown di Lexical, è stato pensato da zero un altro plugin di sincronizzazione del testo dell'editor con uno *stato* di React, in modo tale da poter controllare con maggiore facilità la modifica del testo nei momenti in cui vengono utilizzate le funzioni di recupero informazioni o di generazione di template grazie all'approccio conversazionale. In questi casi, i risultati delle api relative, vengono inseriti all'interno di uno stato grazie all'effetto dell'hook useEffect descritto in precedenza e, grazie al plugin di sincronizzazione, inseriti all'interno dell'editor.

3.7 Prototipazione Interfaccia

La prototipazione dell'interfaccia è una fase essenziale nello sviluppo di applicazioni, poiché permette di visualizzare e testare l'esperienza utente prima della realizzazione effettiva. Creare una rappresentazione visiva dell'interfaccia aiuta a individuare criticità, migliorare l'usabilità e facilitare la comunicazione tra sviluppatori e

stakeholder. I prototipi possono essere classificati in base al loro livello di fedeltà: i prototipi a bassa fedeltà sono bozze schematiche (wireframe) realizzate con strumenti semplici, come carta e penna o software dedicati, utili per definire la disposizione degli elementi e i principali flussi di navigazione. I prototipi a media fedeltà, invece, aggiungono dettagli grafici e interattività limitata, fornendo una rappresentazione più realistica dell'interfaccia senza raggiungere la complessità di un prototipo ad alta fedeltà.

3.7.1 Prototipo bassa fedeltà

Per la realizzazione del prototipo a bassa fedeltà sono stati utilizzati penna e carta, una scelta che permette di esplorare rapidamente diverse soluzioni senza vincoli tecnici, favorendo la creatività e la flessibilità nella definizione della struttura dell'interfaccia.

Nelle successive immagini vengono mostrate le schermate più rilevanti della prima prototipazione dell'applicazione.

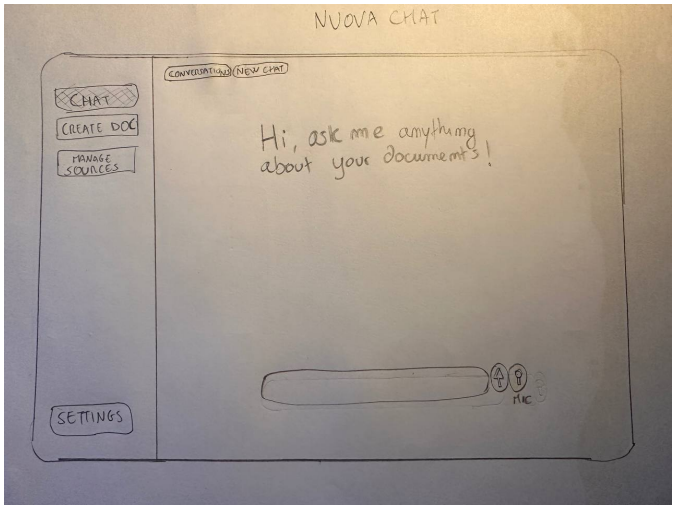

Figura 3.9: Prototipo bassa fedeltà: Schermata iniziale

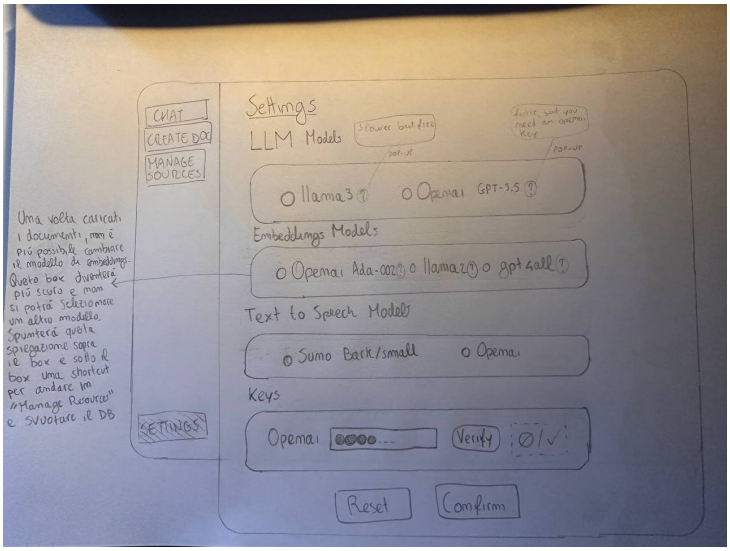

Figura 3.10: Prototipo bassa fedeltà: Schermata Settings

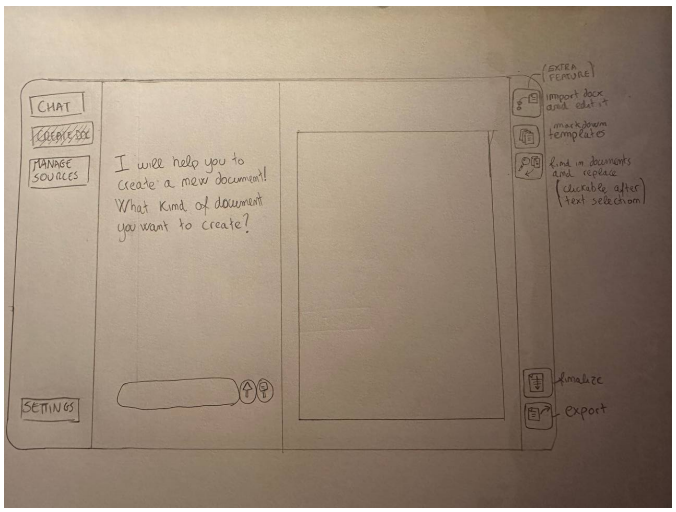

Figura 3.11: Prototipo bassa fedeltà: Schermata Chat + Editor

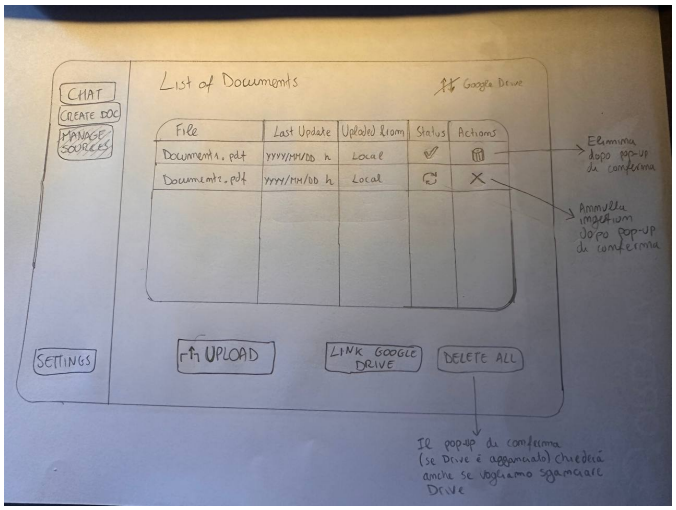

Figura 3.12: Prototipo bassa fedeltà: Schermata Manage Sources

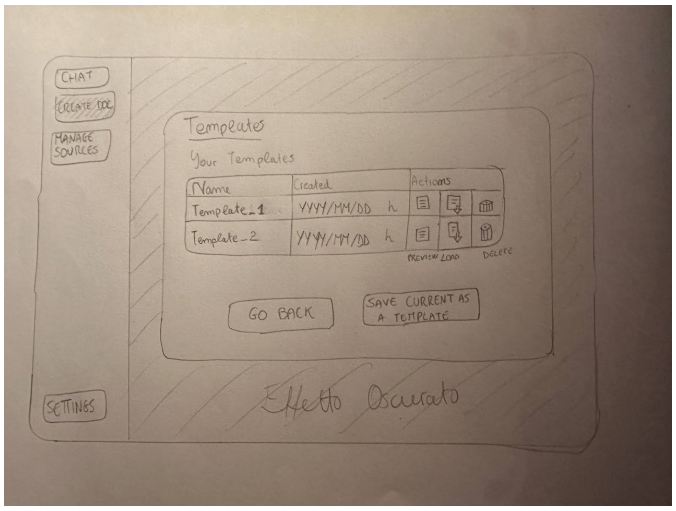

Figura 3.13: Prototipo bassa fedeltà: Schermata Templates

3.7.2 Prototipo media fedeltà

Il prototipo a media fedeltà è stato realizzato utilizzando Figma, un software di progettazione collaborativo basato su cloud, ampiamente utilizzato per la creazione di interfacce utente e prototipi interattivi. Questo strumento offre funzionalità avanzate per la progettazione vettoriale e il design di interfacce, permettendo di definire layout dettagliati e simulare l'interazione tra le diverse schermate dell'applicazione. La realizzazione del prototipo è avvenuta in collaborazione con il reparto UX/UI dell'azienda, che ha contribuito a ottimizzare l'esperienza utente e l'accessibilità dell'interfaccia, assicurando che le scelte progettuali fossero in linea con le best practice di design.

Anche per questo livello del prototipo, vengono allegate le immagini delle schermate più rappresentative.

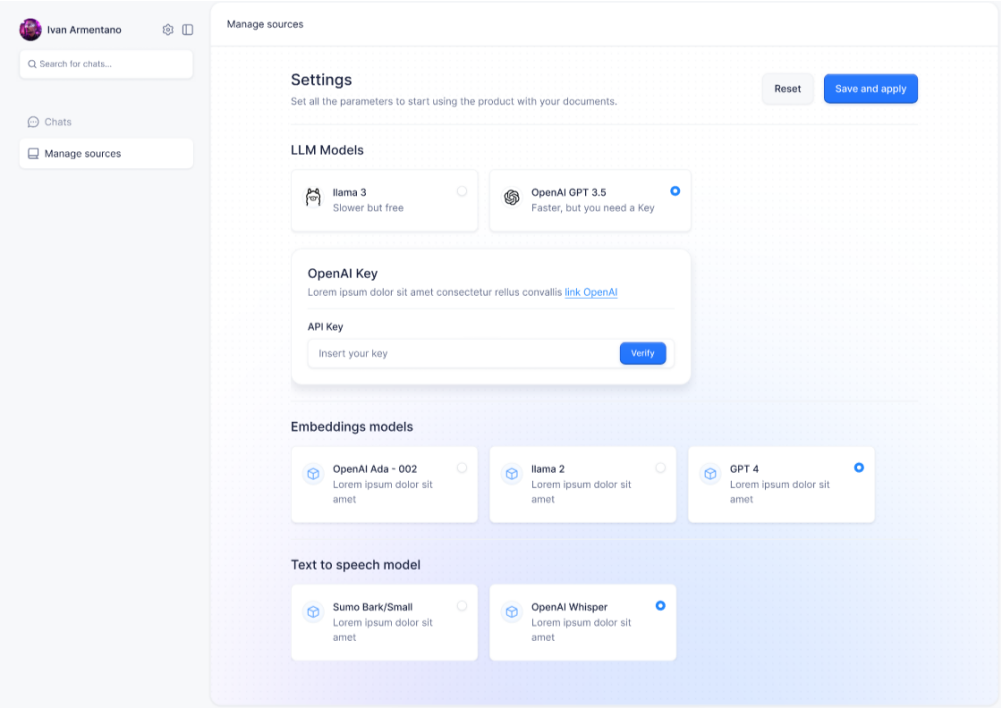

Figura 3.14: Prototipo media fedeltà: Schermata Settings

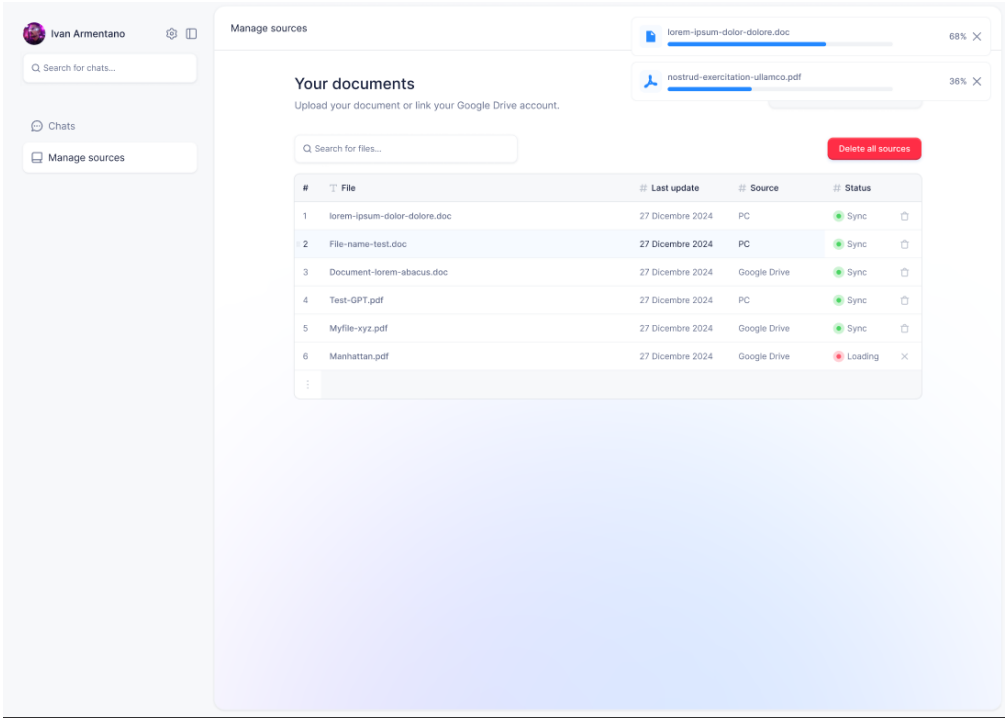

Figura 3.15: Prototipo media fedeltà:Schermata Manage Sources

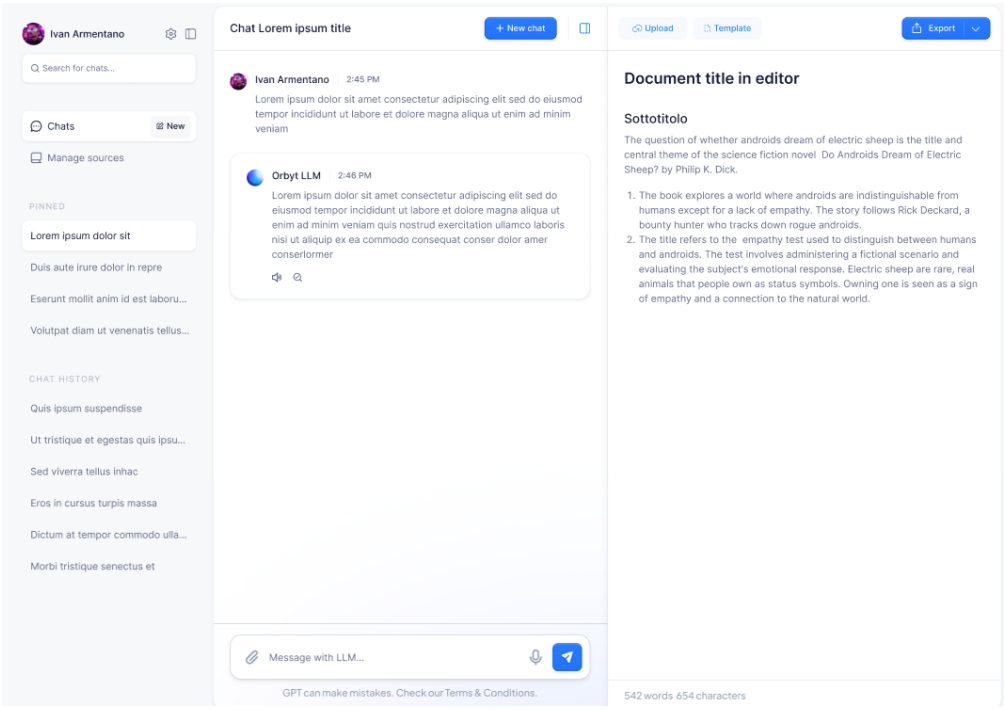

Figura 3.16: Prototipo media fedeltà: Schermata Chat + Editor

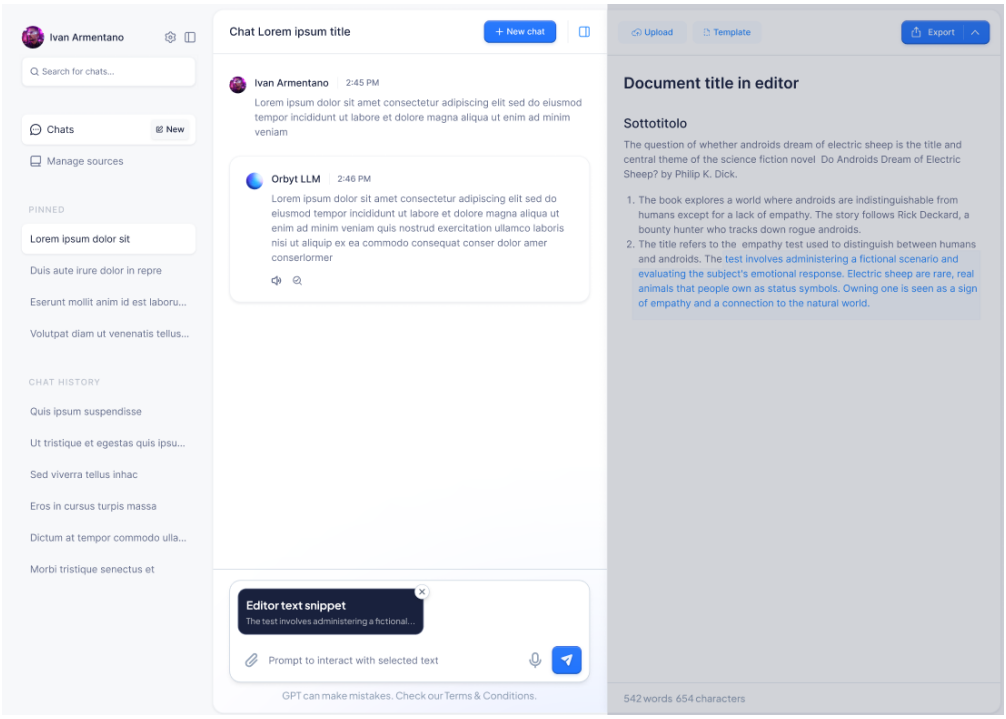

Figura 3.17: Prototipo media fedeltà: Schermata Chat + Editor (2)

Capitolo 4

Implementazione

4.1 Organizzazione del codice

Nel progetto, l'organizzazione del codice è stata strutturata attraverso un sistema gerarchico di cartelle e sottocartelle, finalizzato a garantire una chiara separazione delle responsabilità dei vari elementi e a migliorare la manutenibilità dell'applicazione. Il codice è stato suddiviso in file distinti, ciascuno dedicato a una specifica funzionalità, al fine di favorire la modularità e la riusabilità delle componenti software. Per quanto riguarda lo sviluppo backend in Python, sono state adottate specifiche librerie per garantire la leggibilità e la coerenza dello stile del codice: Black è stato utilizzato per l'autoformattazione, imponendo uno stile uniforme, Flake8 per l'analisi statica del codice e l'individuazione di potenziali errori, mentre Autopep8 ha permesso la correzione automatica delle violazioni delle linee guida PEP 8[1]. L'adozione di tali strumenti ha contribuito a migliorare la qualità del codice, rendendolo più chiaro, manutenibile e conforme alle best practice di sviluppo software.

4.2 Sviluppo del Backend

In questa sezione verrà approfondito l'aspetto implementativo del backend, analizzando nel dettaglio le componenti descritte a livello architetturale nel capitolo dedicato alla progettazione. Attraverso degli snippet di codice, verrà illustrata la

[1]PEP 8 (Python Enhancement Proposal 8) è il documento ufficiale che definisce le linee guida per la scrittura di codice Python leggibile e coerente. È stato scritto da Guido van Rossum, Barry Warsaw e Nick Coghlan per stabilire uno standard di stile che faciliti la collaborazione tra sviluppatori e migliori la qualità del codice.

logica delle principali funzionalità dell'applicazione, offrendo così una panoramica tecnica dell'intero progetto.

Per prima cosa verrà discussa la realizzazione del server contenuto nel file `main.py` e delle API più importanti, successivamente verranno esplorate le pipeline che sono alla base delle API discusse, visualizzando il codice dei più importanti moduli del progetto, contenuti nei seguenti file:

- **`qdrant.py`**: modulo all'interno del quale viene definita la clase QdrantDBHandler, necessaria per poter istanziare un client di Qdrant con cui poter interagire con il database
- **`upload_retrieve.py`**: modulo in cui viene implementata la classe BaseEmbedder, responsabile della creazione degli embeddings a partire da uno o più documenti testuali
- **`prompt.py`**: file contenente i vari prompt testuali utilizzati per ripulire domande e risposte e per fornire indicazioni dettagliate rispetto all'approccio che il chatbot deve tenere per rispondere
- **`chain.py`**: modulo contenente la classe RetrievalChain, all'interno della quale sono implementati i metodi per recuperare il contesto documentale utile alla generazione della risposta e successivamente rivolgere la richiesta all'LLM

4.2.1 Server

Come anticipato nel capitolo precedente, il server dell'applicazione è stato realizzato utilizzando FastAPI, al quale è stato agganciato il middleware CORS per l'abilitazione delle richieste proveniente esclusivamente dal client React, in esecuzione su `http://localhost:5173`. Grazie all'utilizzo dei tags in FastAPI, le varie API sono state aggregate in base al database con cui interagiscono, PostgreSQL o Qdrant, questo è risultato comodo nella fase del loro sviluppo, poichè FastAPI offre una UI molto intuitiva attraverso la quale testare gli endopoint, in cui vengono raggruppati in base al loro tag.

Per interagire con il database PostgreSQL, si è resa necessaria l'implementazione di un apposito modulo per la gestione della connessione, realizzata utilizzando la libreria SQLAlchemy. Come anticipato nel capitolo precedente, questa libreria consente di definire classi Python che vengono poi tradotte in tabelle e altri elementi del database relazionale. Di seguito viene riportato come esempio, la definizione della classe Conversation, la quale viene mappata successivamente nella tabella *conversations*:

```
1 class Conversation(Base):
2     """Conversation table schema. For each User there could be
    multiple Conversations."""
```

```
3
4       __tablename__ = "conversations"
5
6       id = Column(UUID(as_uuid=True), primary_key=True, default=
        uuid4)
7       created_at = Column(DateTime, default=datetime.datetime.utcnow
        )
8       user_id = Column(UUID(as_uuid=True), ForeignKey("users.id"),
        nullable=False)
9       chat_title = Column(String, nullable=False)
10      last_update = Column(DateTime, default=datetime.datetime.
        utcnow)
11
12      user = relationship("User", back_populates="conversations")
13
14      def __repr__(self) -> str:
15          """Method to represent the User object as a string."""
16          return f"<Conversation ({self.id}, {self.user_id})>"
```

Listing 4.1: Esempio di classe SQLAlchemy

Una volta definite le classi che rappresentano le entità del database, è necessario stabilire la connessione con PostgreSQL. Tale funzionalità è contenuta all'interno del file `database.py` e se ne mostra la relativa implementazione nel Codice 4.2. Come è possibile vedere, vi è un primo caricamento delle variabili d'ambiente mediante il pacchetto dotenv, che consente di estrarre in modo sicuro le credenziali d'accesso da un file di configurazione attraverso il metodo `dotenv.load_dotenv()`, invocato per rendere disponibili le variabili definite nel file `.env`. Successivamente, viene costruita la stringa di connessione a PostgreSQL tramite l'oggetto `URL.create()` di SQLAlchemy. Come parametri di input per questa funzione sono presenti le variabili recuperate tramite dotenv, mentre l'host è impostato su "localhost" e la porta su "5432", che rappresenta la porta predefinita per i server PostgreSQL.

Il codice procede con la creazione di un'istanza del motore di database tramite `create_engine(SQLALCHEMY_DATABASE_URL)`. Questo oggetto engine rappresenta il punto di accesso principale per gestire la connessione e l'esecuzione delle query su PostgreSQL, costituendo un intermediario tra l'applicazione e il database.

Viene poi definita una sessione locale attraverso il costruttore di sessioni `sessionmaker()`; una sessione rappresenta una connessione temporanea al database, attraverso la quale vengono eseguite operazioni di lettura e scrittura. Questo metodo riceve come input 3 parametri:

- **`autocommit=False`**: disabilita il commit automatico delle transazioni, richiedendo che l'applicazione lo gestisca esplicitamente. Questo garantisce un maggiore controllo sulle operazioni e consente di eseguire rollback in caso di errori.

- **`autoflush=False`**: impedisce l'invio automatico delle modifiche al database prima dell'esecuzione di una query, evitando comportamenti imprevisti e consentendo di accumulare più operazioni prima di inviarle in un'unica transazione.
- **`bind=engine`**: collega la sessione all'istanza del motore di database, specificando quale connessione utilizzare per l'esecuzione delle query, in questo modo è possibile riutilizzare il motore esistente senza dover creare nuove connessioni per ogni sessione.

Infine, viene dichiarata una classe base per la definizione delle entità del database attraverso la funzione `declarative_base()`. Gli oggetti derivati da Base rappresenteranno le tabelle e le relazioni del database, consentendo una gestione ORM strutturata e flessibile.

```
1
2  """Postgres connection module."""
3
4  import os
5
6  import dotenv
7  from sqlalchemy import URL, create_engine
8  from sqlalchemy.ext.declarative import declarative_base
9  from sqlalchemy.orm import sessionmaker
10
11 dotenv.load_dotenv()
12
13 SQLALCHEMY_DATABASE_URL = URL.create(
14     "postgresql",
15     username=os.getenv("POSTGRES_USER"),
16     password=os.getenv("POSTGRES_PASSWORD"),
17     host="localhost",
18     database=os.getenv("POSTGRES_DB"),
19     port=5432
20 )
21
22 engine = create_engine(SQLALCHEMY_DATABASE_URL)
23 SessionLocal = sessionmaker(autocommit=False, autoflush=False,
       bind=engine)
24
25 Base = declarative_base()
```

Listing 4.2: Modulo connessione DB PostgreSQL

All'interno del server, in `main.py`, viene poi richiamato e utilizzato `SessionLocal()` per creare una nuova sessione del database e, attraverso la definizione di un blocco try-finally ci si assicura che, indipendentemente dall'esito delle operazioni eseguite nella sessione, la connessione venga sempre chiusa tramite db.close(), evitando

sprechi di risorse o connessioni lasciate aperte. Successivamente viene definito l'oggetto `db_dependency`, il quale utilizza `Annotated` e `Depends(get_db)` per dichiarare una dipendenza in FastAPI, in questo modo, ogni volta che un endpoint o funzione richiede un'istanza di sessione del database PostgreSQL, FastAPI eseguirà `get_db()`, fornendo automaticamente una sessione valida e chiudendola una volta completata l'elaborazione della richiesta.
A seguire, una serie degli endpoint più utilizzati all'interno dell'applicazione:

`GET /user`: Dato un codice UUID identificativo dell'utente, viene restituito un dizionario contenente i dati dell'utente ad esso collegato. Non essendo ancora presente un meccanismo di autenticazione, questa funzione viene utilizzata principalmente all'interno di altri endpoint per verificare che la richiesta provenga da un utente realmente esistente all'interno del database, se così non fosse, verrebbe lanciata una `HTTPException` con codice `404`.

`GET /Q/get_user_settings`: Recupera dal DB Qdrant le impostazioni per l'utente di cui viene passato l'user_id come parametro di input e le restituisce sotto forma di dizionario. Una volta accertata la presenza dell'utente relativo al codice UUID fornito come parametro all'interno del DB, viene creata un'istanza di QdrantDBHandler e conservata nella variabile `vectorstore`. Qualora non fosse presente la collezione delle settings, questa verrebbe creata appositamente tramite il metodo `create_settings_collection` della classe QdrantDBHandler, il quale richiama la funzione `create_collection` della classe QdrantClient.

Grazie al metodo `get_settings_point`, viene recuperato dalla collection delle settings, se esistente, il punto corrispondente all'UUID. Il risultato contiene informazioni in questo caso superflue, che sono relative alla rappresentazione vettoriale del dato; per questo motivo, del risultato ricavato, viene estratto solamente il `payload`, che è ciò che contiene le impostazioni dell'utente, e successivamente trasformato in un dizionario attraverso il metodo `model_dump()` per infine restituirlo al client.

`/Q/conversation`: Recupera i messaggi di una conversazione identificata mediante codice UUID. Con la stessa logica della funzione precedente, all'interno del metodo `get_conversation_messages` ci si accerta dapprima, mediante l'uso della funzione `get_conversation`, della presenza all'interno del DB Postgres della conversazione relativa al codice UUID ricevuto come parametro (analogamente al metodo `get_user` di prima). Successivamente, attraverso il metodo `get_conversation`, interno alla classe QdrantDBHandler, si recupera la lista dei messaggi relativi alla conversazione in questione ordinati dal meno recente al più recente e la si restituisce al client.

Nello specifico, il metodo `get_conversation` ritorna una lista di Record, modello standard ricavato dal modulo `qdrant_client.http.models`. Di seguito, uno snippet dettagliato del metodo della classe QdrantDBHandler (contenuta nel file qdrant.py), esemplificativo della possibilità di recuperare informazioni da Qdrant

mediante delle query molto simili alla logica standard SQL:

```
1
2  def get_conversation(self, conversation_id: str) -> list[Record]:
3          """Get a complete conversation oredered by timestamp,
       given a conversation_id.
4
5          Args:
6              conversation_id (str): The conversation id to search
       for.
7
8          Returns:
9              list[Record]: list of Record objects.
10         """
11         query_filter = Filter(
12             must=[
13                 FieldCondition(
14                     key="conversation_id",
15                     match=MatchValue(value=str(conversation_id)),
16                 )
17             ]
18         )
19
20         count = self.client.count(collection_name="chat_history",
       count_filter=query_filter)
21
22         results = self.client.scroll(
23             collection_name="chat_history", scroll_filter=
       query_filter, limit=count.count if count.count > 0 else 1
24         )
25         points = results[0]
26         points = sorted(points, key=lambda x: x.payload["timestamp
       "] if x.payload is not None else float("-inf"))
27
28         return points
```

Listing 4.3: Metodo get_conversation della classe QdrantDBHandler

Come è possibile vedere nel Codice 4.3, viene dapprima definito un oggetto `Filter`, che definisce un criterio di ricerca basato su una condizione `FieldCondition`; entrambi gli elementi vengono importati da `qdrant_client.http.models`. Nello specifico, viene filtrato il campo `conversation_id`, assicurandosi che il valore corrisponda all'identificativo passato come parametro.

Successivamente viene invocato il metodo `count()` del client di Qdrant, applicando il filtro appena definito alla collezione `chat_history`. Questa operazione serve a determinare quanti punti dati soddisfano la condizione e quindi a quantificare il numero di messaggi presenti nella conversazione. Viene poi utilizzato il metodo `scroll()` per recuperare effettivamente i punti corrispondenti. Questa operazione

sfrutta il filtro definito e imposta un limite pari al numero di punti trovati tramite count(), garantendo il recupero di tutti i messaggi pertinenti. Se nessun punto viene trovato, il valore di default del limite è 1, il che evita errori nella query.

Infine, dopo aver ottenuto i punti, viene applicata una funzione di ordinamento basata sul valore del campo `timestamp`, estratto dal payload di ciascun punto. Se un punto non contiene il payload, viene assegnato un valore di - ∞ per garantire che venga ordinato in ultima posizione.

`POST /Q/upload_documents`: Carica all'interno del DB Qdrant la lista di documenti ricevuta, utilizzando le impostazioni dell'utente che ne ha fatto richiesta.

Come è possibile vedere nel Codice 4.4, dopo aver recuperato le settings per un dato utente, identificato per mezzo di un id di tipo UUID, e aver stabilito una connessione con il database Qdrant creando un'istanza del client, la prima cosa che viene verificata è l'esistenza di una collezione di documenti per l'utente in questione. Qualora questa non dovesse esistere, si procederebbe con la creazione della collezione. Una volta fatto ciò, i file ricevuti dal client vengono copiati in modo temporaneo in locale, in modo tale da potervi accedere in modo semplice e, definito un *loader*, leggerne il contenuto e aggiungendolo alla lista dei documenti definita come *docs*.

Recuperato il corretto modello di embedding per mezzo della funzione *get_embedding*, viene creata un'istanza della classe BaseEmbedder, che riceve come parametri il vectorstore definito in precedenza, le configurazioni relative al modello di embedding da utilizzare e il nome della collezione di qdrant in cui successivamente salvare la rappresentazione vettoriale.

Grazie al metodo `recursive_documents_splitter`, i documenti vengono suddivisi in testi composti da 1000 token e successivamente aggiunti alla collezione grazie alla funzione `add_to_vectorstore`.

Infine, dato che questa funzione viene eseguita successivamente al caricamento di alcuni dei metadata dei documenti all'interno del database PostgreSQL, come titolo e data di caricamento, i codici UUID dei documenti sono già disponibili, di conseguenza vengono ricevuti da questa funzione come parametro e utilizzati per sovrascrivere il campo `"status"` dei documenti che sono stati effettivamente caricati su Qdrant, con il valore `"active"`.

```
1 @chatbot_rag.post("/Q/upload_documents", tags=["Qdrant"])
2 async def upload_documents(
3 user_id: UUID, db: db_dependency, files: list[UploadFile] = File
      (...), metadata: str = Form(...)
4 ):
5 """Upload documents, split them in chunks and upload them to
      QdrantDB."""
6 settings = await get_user_settings(user_id=user_id, db=db)
7
8 vectorstore = QdrantDBHandler()
```

```
9  if f"documents_collection_{user_id}" not in vectorstore.
       list_collections():
10     await create_documents_collection(user_id=user_id, db=db)
11
12 docs = []
13
14 if not os.path.exists(UPLOAD_FOLDER):
15     os.makedirs(UPLOAD_FOLDER)
16
17 for file in files:
18     if file.filename:
19         _, file_extension = os.path.splitext(file.filename)
20     else:
21         raise HTTPException(status_code=400, detail="File name is
       empty")
22
23     file_path = f"{UPLOAD_FOLDER}/{file.filename}"
24
25     with open(file_path, "wb") as buffer:
26         shutil.copyfileobj(file.file, buffer)
27
28     loader: TextLoader | PyPDFLoader
29
30     if file_extension:
31         match file_extension.lower():
32             case ".txt":
33                 loader = TextLoader(file_path)
34             case ".pdf":
35                 loader = PyPDFLoader(file_path)
36             case _:
37                 raise HTTPException(
38                     status_code=400,
39                     detail=(
40                         f"File {file.filename} has an invalid
       extension. Allowed extensions are: "
41                         f"{', '.join(ALLOWED_EXTENSIONS)}"
42                     ),
43                 )
44
45     docs.extend(loader.load())
46
47     os.remove(file_path)
48
49 embedder_config = get_embedding(settings)
50 embedder = BaseEmbedder(embedder_config, vectorstore, f"
       documents_collection_{user_id}")
51 docs = embedder.recursive_documents_splitter(docs, chunk_size
       =1000, chunk_overlap=200)
52 embedder.add_to_vectorstore(docs)
```

```
53
54 try:
55     metadata_dict = json.loads(metadata)
56     document_ids = [UUID(doc_id) for doc_id in metadata_dict.get("
   document_ids", [])]
57
58     if document_ids != []:
59         try:
60             for document_id in document_ids:
61                 db.query(Document).filter(Document.id ==
   document_id).update({"status": "active"})
62             db.commit()
63             print("Document status updated for id:", document_id)
64         except Exception as e:
65             db.rollback()
66             raise HTTPException(status_code=500, detail=f"Internal
    server error: {str(e)}")
67     else:
68         print("No document status updated")
69
70     return {"detail": "Documents uploaded to QdrantDB"}
71
72 except (json.JSONDecodeError, KeyError, ValueError) as e:
73     raise HTTPException(status_code=400, detail=f"Errore nel
   parsing dei dati: {str(e)}")
```

Listing 4.4: Endpoint: POST /Q/upload_documents

`POST /Q/ask`: Effettua una domanda al chatbot, interrogandolo sui documenti precedentemente caricati dall'utente che effettua la richiesta.

Tra i parametri di input della funzione, troviamo un input di tipo `QuestionRequest`, un modello definito mediante pydantic che contiene una richiesta testuale dell'utente (question) e una lista dei messaggi più recenti all'interno della chat (chat_history), in modo da costruire una base contestuale migliore per il chatbot.

Anche in questo caso, viene creata un'istanza del client Qdrant e successivamente recuperate le settings dell'utente, in modo tale da avere a disposizione la sua configurazione per i vari modelli da utilizzare. Una volta creata una istanza di BaseEmbedder, attraverso il metodo `get_retriever` viene restituito un retriever, nello specifico `BaseRetriever` importato da `langchain_core.retrievers`. Il metodo accetta un parametro numerico che, com'è possibile vedere nel Codice 4.5, è settato a 3 ed indica il numero di documenti da restituire dopo la ricerca per similitudine, dunque i K migliori risultati.

Se il parametro `hybrid_search` è settato a true, viene istanziato un altro retriever, ossia BM25Retriever, il quale viene combinato al retriever definito in precedenza grazie all'EnsembleRetriever, attribuendo comunque un peso maggiore

alla ricerca per similutidine grazie al parametro `weights`. Sia BM25Retriever che EnsembleRetriever, vengono importati dal modulo `langchain.retrievers`.

A questo punto viene definita una istanza della classe `RetrievalChain`, inizializzata con il retriever precedentemente definito e il modello llm e la lingua di risposta recuperati dalle settings dell'utente.

Alla riga 42 del Codice 4.5 è presente un `if`, dentro al quale si entra se non è presente un codice id riconducibile alla conversazione in corso, che sostanzialmente implica il fatto che quella corrente sia una nuova conversazione. Qualora fosse così, verrebbe avviata una nuova funzione (`await generate_tile`) attraverso la quale, partendo dalla richiesta dell'utente contenuta nella `question_request`, si genererà un titolo che riassuma il senso della richiesta in non più di 5 parole. Questo titolo verrà assegnato alla conversazione, per poi poterla andare a memorizzare all'interno del database grazie alla funzione `await create_new_conversation`. Grazie al titolo, sarà possibile per l'utente consultare in un secondo momento le conversazioni in modo molto più semplice dall'interfaccia utente a disposizione ed, eventualmente, riconoscerle nella sidebar per eliminarle senza dover accedere al contenuto della conversazione. Successivamente, il messaggio contenuto nella richiesta dell'utente viene inserito all'interno del database, etichettandolo come inviato da un sender di tipo *Human*.

A questo punto è tutto pronto per poter rivolgere la richiesta all'LLM, viene dunque chiamata la funzione `ask()` dell'istanza di RetrievalChain, attraverso la quale finalmente parte il meccanismo di RAG per recuperare il contesto e ottenere una risposta pertinente. Il funzionamento di questa pipeline verrà discussa successivamente, nell'apposita sezione.

Ricevuta la risposta, viene dapprima memorizzata all'interno dello storico delle conversazioni, con il corrispondente codice id, e infine restituita al client.

```
1  @chatbot_rag.post("/Q/ask", tags=["Qdrant"])
2  async def ask(
3      question_request: QuestionRequest,
4      user_id: UUID,
5      db: db_dependency,
6      conversation_id: Optional[UUID] = None,
7  ):
8      """Ask a question to the chatbot."""
9      time = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
10     vectorstore = QdrantDBHandler()
11     settings = await get_user_settings(user_id=user_id, db=db)
12     llm = get_llm(settings)
13     language_response = settings["language_response"]
14     hybrid_search = settings["hybrid_search"]
15     embedder_config = get_embedding(settings)
16
17     if f"documents_collection_{user_id}" not in vectorstore.
       list_collections():
```

```
18          await create_documents_collection(user_id=user_id, db=db)
19
20      embedder = BaseEmbedder(embedder_config, vectorstore, f"
    documents_collection_{user_id}")
21      retriever = embedder.get_retriever(3)
22
23      if hybrid_search:
24          documents = vectorstore.get_documents(f"
    documents_collection_{user_id}")
25          if documents == []:
26              raise HTTPException(status_code=400, detail="You must
    upload documents to perform hybrid search")
27          bm25_retriever = BM25Retriever.from_documents(documents)
28          ensemble_retriever = EnsembleRetriever(retrievers=[
    retriever, bm25_retriever], weights=[0.7, 0.3])
29          qa = RetrievalChain(llm, ensemble_retriever,
    language_response)
30      else:
31          qa = RetrievalChain(llm, retriever, language_response)
32
33      if "chat_history" not in vectorstore.list_collections():
34          await create_chat_history_collection()
35
36      if question_request.chat_history is not None:
37          qa.set_history(question_request.chat_history)
38
39      if f"documents_collection_{user_id}" not in vectorstore.
    list_collections():
40          await create_documents_collection(user_id=user_id, db=db)
41
42      if conversation_id is None or conversation_id == "":
43          chat_title = await generate_title(question_request.
    question, llm, language_response)
44          conversation = await create_new_conversation(user_id=
    user_id, db=db, chat_title=chat_title)
45          conversation_id = conversation.id
46
47      await add_message_to_chat_history(
48          ChatMessage(
49              content=question_request.question, timestamp=time,
    sender="Human", conversation_id=conversation_id
50          ),
51          db=db,
52      )
53      response = qa.ask(question_request.question)
54      time = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
55
56      await add_message_to_chat_history(
57          ChatMessage(
```

```
58              content=response["fixed_answer"], timestamp=time,
    sender="AI", conversation_id=conversation_id
59          ),
60          db=db,
61      )
62      response["conversation_id"] = conversation_id
63      return response
```

Listing 4.5: Endpoint: POST /Q/ask

`POST /Q/ask_find_replace`: Data una porzione di testo contenente dei placeholder ed una richiesta dell'utente, restituisce la porzione di testo opportunamente modificata, sostituendo i placeholder con le informazioni richieste.

Il flusso di codice della funzione sottostante è analogo alla funzione `ask` vista in precedenza, ciò che caratterizza la funzione `ask_find_replace` è la tipologia di richiesta. Come parametro in input passato attraverso il body, questa volta è presente un'istanza di `QuestionFindReplaceRequest`, un'estensione del modello `QuestionRequest` della funzione precedente.

Oltre a contenere la richiesta dell'utente e la chat_history, è presente anche un campo contenente il contenuto testuale dell'editor da modificare. Per il resto, il codice si mantiene invariato, fatta eccezione per il metodo da invocare al momento della richiesta vera e propria.
Infatti, in questo caso il metodo invocato è il seguente:

```
response = qa.ask_find_and_replace(
question_request.question, question_request.editor_content
)
```

Nella sottosezione **Retrieval dei Documenti** verranno approfonditi gli elementi impiegati per l'implementazione delle due varianti del metodo ask.

4.2.2 Ingestion dei documenti

Una delle pipeline più importanti all'interno dell'applicazione è quella che si occupa di effettuare la trasformazione in forma vettoriale dei documenti caricati dall'utente e di inserirli all'interno dell'apposita collezione in Qdrant, inserendo al contempo alcuni dati identificativi del documento all'interno del database PostgreSQL. Questo processo è il meccanismo che sta alla base della funzione `upload_documents`, di cui è stato analizzato il flusso nella precedente sottosezione. Di seguito vengono analizzati i componenti che costituiscono la sua implementazione.

QdrantDBHandler

Una delle classi fondamentali alla base del funzionamento dell'intero sistema è `QdrantDBHandler`. Come detto precedentemente, è la classe attraverso la quale

viene creata un'istanza di `QdrantClient`, importato dal modulo `qdrant_client`, e si interagisce con il database. L'istanza del client viene creata tramite la funzione costruttore `__init__` ed assegnata al campo `self.client` utilizzando come parametri di default `host="localhost"` e `port=6333`, porta standard per Qdrant.

I metodi definiti all'interno di questa classe sfruttano le funzioni offerte da `QdrantClient`, eventualmente riadattandole a dei contesti più specifici. Un esempio della composizione di nuovi metodi utilizzando come base quelli messi a disposizione, la si può trovare nel Codice 4.3 analizzato in precedenza.

Tra i metodi di `QdrantClient` impiegati all'interno della classe troviamo:

- **`create_collection`**: Crea una nuova collezione nel DB Qdrant, ricevendo come parametri il nome della collezione, la metriche da utilizzare per calcolare la distanza tra vettori e la loro dimensionalità.
- **`set_payload`**: Modifica il payload di una lista di punti identificati tramite un UUID, di cui si fa il cast a `str`. Riceve inoltre il nome della collezione in cui sono contenuti i punti e un `dict`, che verrà sostituito al payload attualmente settato.
- **`upsert`**: Data una collection e una lista di punti definiti come `PointStruct`, inserisce tali vettori all'interno della collezione.
- **`retrieve`**: Restituisce, se presenti, i corrispondenti `Record` relativi ad una lista di UUID all'interno della collezione passata come parametro di input.

BaseEmbedder

Nel processo di ingestion dei documenti, il componente principale è la classe `BaseEmbedder` contenuta nel modulo `upload_retrieve.py`, già citata precedentemente, al suo interno sono presenti tutti i metodi responsabili dell'ingestion dei documenti.

Ogni qualvolta è necessario caricare dei documenti all'interno del database, viene creata un'istanza della classe BaseEmbedder, di cui è possibile visualizzare il metodo costruttore di seguito:

```
1
2 def __init__(
3         self, emedding_config: EmbeddingConfig, vectorstore_client
    : QdrantDBHandler, vectorstore_collection: str
4     ):
5         """Base Embedder init.
6
7         Args:
8             emedding_config (EmbeddingConfig): The embedding model
    configuration.
```

```
9             vectorstore_client (Qdrant): The vector store used for
    RAG.
10            vectorstore_collection (str): The specific collection
    to be used.
11        """
12        self.emedding_config = emedding_config["config"]
13        self._create_embeddings()
14        self.vectorstore = Qdrant(
15            client=vectorstore_client.client, collection_name=
    vectorstore_collection, embeddings=self.embeddings
16        )
```

Listing 4.6: Costruttore della classe BaseEmbedder

Il costruttore accetta tre parametri principali: `embedding_config` è un oggetto di tipo EmbeddingConfig, un dizionario che contiene la configurazione del modello di embedding - inclusi dettagli come il nome del modello utilizzato per convertire i dati testuali in vettori numerici - la chiave API (se necessaria) e altre impostazioni specifiche; `vectorstore_client` è un'istanza di QdrantDBHandler, classe discussa precedentemente; `vectorstore_collection` è una stringa che specifica il nome della collezione nel database Qdrant in cui gli embedding verranno salvati o ricercati.

Assegnata all'istanza della classe la configurazione da utilizzare, viene chiamato il metodo interno `self._create_embeddings()`, il quale istanzia un modello di embedding in base alla configurazione specificata tramite l'utilizzo delle classi GPT4AllEmbeddings, HuggingFaceBgeEmbeddings o OllamaEmbeddings importate da langchain_community.embeddings o tramite OpenAIEmbeddings importata da langchain_openai.embeddings.

Infine, il costruttore crea un'istanza di `Qdrant`, classe importata dal modulo `langchain_qdrant` che rappresenta il database vettoriale effettivo. Questa istanza viene configurata passando il client di vectorstore_client, il nome della collezione specificata e il modello per la generazione degli embeddings istanziato al passo precedente.

Per effettuare l'ingestion vera e propria dei documenti testuali viene utilizzato il metodo `recursive_documents_splitter` che, a partire da una lista di `Document`, oggetto importato dal modulo `langchain_core.documents`, effettua la suddivisione in porzioni di testo più piccole in base ai valori assunti dai parametri di input `chunk_size` e `chunk_overlap` già discussi in precedenza.

Di seguito è possibile vedere il codice che implementa tale metodo:

```
1
2 def recursive_documents_splitter(
3         self,
4         documents: list[Document],
5         chunk_size: int,
6         chunk_overlap: int,
```

```
7           separators: list[str] | None = None,
8           encoding_name: str = "cl100k_base",
9       ) -> list[Document]:
10          """Return splitted documents using a recursive method.
11
12          Args:
13              documents (List[Document]): List of documents to chunk
14              chunk_size (int): Chunk size (number of tokens)
15              chunk_overlap (int): Token overlap between chunks
16              language (Optional[Language], optional): Programming
    language enum. Defaults to None.
17
18          Returns:
19              List[Document]: List of chunked documents
20          """
21          if separators is not None:
22              separators.extend(["\n\n", "\n", " "])
23          else:
24              separators = ["\n\n", "\n", " "]
25
26          text_splitter = RecursiveCharacterTextSplitter.
    from_tiktoken_encoder(
27              encoding_name=encoding_name,
28              chunk_size=chunk_size,
29              chunk_overlap=chunk_overlap,
30              separators=separators,
31          )
32          splitted_documents = text_splitter.split_documents(
    documents)
33          return self._split_and_strip_documents(splitted_documents)
```

Listing 4.7: Metodo recursive_documents_splitter della classe BaseEmbedder

Come elemento per effettuare l'operazione di suddivisione dei documenti è stato utilizzato `RecursiveCharacterTextSplitter` del modulo `langchain.text_splitter`. A differenza di altri splitter, utilizza un approccio ricorsivo: suddivide il testo in base a delimitatori specificati e, se una parte supera la lunghezza desiderata, la suddivide ulteriormente fino a ottenere segmenti della dimensione appropriata. Questo metodo garantisce che i frammenti risultanti siano coerenti e ben formattati, riducendo la perdita di informazioni tra i segmenti.

Una volta creata una sua istanza, vengono effettivamente creati i chunks attraverso il metodo `split_documents` e successivamente restituiti come risultato, dopo averli opportunamente normalizzati, rimuovendo spazi bianchi superflui e assegnandone il contenuto al campo `"page_content"`.

Una volta realizzati i chunks, questi possono finalmente essere conservati all'interno del database Qdrant. Per fare ciò, viene utilizzato il metodo `add_to_vectorestore`

il quale, ricevuta una lista di `Document` in input, invoca a sua volta il metodo `add_documents` del vectorestore, ossia dell'istanza di `langchain_qdrant.Qdrant`.

4.2.3 Retrieval dei Documenti

Un'altra pipeline di fondamentale importanza all'interno dell'applicazione è quella che consente, una volta che i documenti sono stati archiviati nel database vettoriale, di recuperare gli elementi più rilevanti. Questo processo permette di costruire un contesto informativo adeguato affinché l'LLM possa fornire risposte ottimali. Anche in questo caso, le classi QdrantDBHandler e BaseEmbedder svolgono un ruolo significativo. Tuttavia, la componente centrale per l'implementazione di tale meccanismo è la classe **`RetrievalChain`**, contenuta nel modulo `chain.py`.

Seguendo il flusso del Codice 4.5 analizzato in precedenza, è possibile vedere che, una volta definito un oggetto della classe BaseEmbedder, viene invocato il metodo `get_retriever`, il quale restituisce un'istanza di `BaseRetriever` (ricavata dal modulo `langchain_core.retrievers`) tramite il metodo `as_retriever` messo a disposizione dalla classe Qdrant.

Una volta pronto il retriever, è possibile passarlo come input al metodo costruttore della classe RetrievalChain, il cui codice è il seguente:

```
1
2  def __init__(
3          self,
4          llm: BaseLanguageModel,
5          retriever: BaseRetriever,
6          language_response: str,
7          memory_llm: BaseLanguageModel | None = None,
8          chat_history: list = [],
9          ask_mode: str = "qa",
10     ):
11         """Return a RetrievalChain instance.
12
13         Args:
14             llm (BaseLanguageModel): the LLM we want for RAG.
15             retriever (BaseRetriever): the Retriever we want to
    use for RAG.
16             memory_llm (BaseLanguageModel, optional): the LLM we
    want for summarization, if None the llm for RAG will be used.
17                 Defaults to None.
18             chat_history (list, optional): _description_. Defaults
     to [].
19             language_response (str): The language of the response.
20             ask_mode (str): The mode of the question answering.
    Defaults to "qa".
21         """
22         self.llm = llm
```

```
23          self.memory_llm = memory_llm if memory_llm else llm
24          self.retriever = retriever
25          self.chat_history = chat_history
26
27          self.language_response = language_response
28          self.ask_mode = ask_mode
29
30          self.question_answering_chain = self.create_qa_chain()
31          self.fixing_chain = self.create_fixing_chain()
```

Listing 4.8: Metodo costruttore della classe RetrievalChain

Il risultato del metodo `create_qa_chain()` varia in base al valore assegnato al parametro `ask_mode`, che stabilisce il comportamento del Retriever. Il parametro ask_mode può assumere due valori: **"qa"** per le richieste che richiedono il normale meccanismo RAG e **"find_and_replace"** per le richieste che prevedono la sostituzione dei placeholder contenuti nell'editor, sempre attraverso la RAG.

Nello specifico, ciò che altera il comportamento del retriever è la scelta del prompt da utilizzare, definito all'interno del modulo `prompt.py`. Per esempio, nel caso della modalità "Find and Replace", il prompt viene ricavato dalla seguente funzione:

```
1
2  def create_find_and_replace_prompt(with_history: bool = True) ->
       PromptTemplate:
3       """
4       Args:
5           with_history (bool, optional): If the prompt has to embed
       also chat history. Defaults to True.
6
7       Returns:
8           PromptTemplate: basic prompt template for a generic
       Chatbot.
9       """
10      prompt_template = """ You are an advanced AI agent tasked with
        modifying a given text using information provided in a
       separate context. You receive as input a text containing
       placeholders or sections to be updated, and a context
       containing relevant information. Replace placeholders in the
       document with corresponding data from the context, ensuring
       accuracy and alignment.
11      If a section in the document cannot be updated due to a lack
       of compatible information in the context, leave it unchanged.
       Do not invent or generate any information that is not
       explicitly present in the context.
12      Provide the updated version of the document, reflecting only
       the modifications supported by the context,while leaving all
       other sections unchanged. Preserve the original markdown
       formatting.
```

```
13
14      By using the following context:
15
16      {context}
17
18      update the document text below:
19
20      {editor_content}
21      """
22
23      if with_history:
24          prompt_template = (
25              prompt_template
26              + """
27      Current conversation:
28
29      {chat_history}
30
31      Human: {input}
32      AI:
33      """
34          )
35          prompt = PromptTemplate(
36              template=prompt_template,
37              input_variables=["context", "chat_history", "input", "
    editor_content"],
38          )
39      else:
40          prompt_template = (
41              prompt_template
42              + """
43      Human: {input}
44      AI:
45      """
46          )
47
48      prompt = PromptTemplate(
49          template=prompt_template,
50          input_variables=["context", "input", "editor_content"],
51      )
52      return prompt
```

Listing 4.9: Funzione di prompt engineering per la modalità "Find and Replace"

Successivamente, il risultato di questa funzione verrà utilizzato all'interno del metodo `create_qa_chain()` per creare, a partire dai moduli preposti di langchain, la Runnable chain da invocare successivamente all'interno del metodo `ask` per ottenere una risposta alla richiesta dell'utente.

```
1
2  ## Resto del codice..
3
4  from langchain.chains.combine_documents import
       create_stuff_documents_chain
5  from langchain.chains.retrieval import create_retrieval_chain
6  from langchain_core.runnables import Runnable
7
8  ## Resto del codice..
9
10 def create_qa_chain(self) -> Runnable:
11         """Return a chain for RAG.
12
13         Returns:
14             Runnable: a runnable chain for RAG
15         """
16         if self.ask_mode == "qa":
17             system_prompt = create_basic_prompt()
18         elif self.ask_mode == "find_and_replace":
19             system_prompt = create_find_and_replace_prompt()
20         else:
21             system_prompt = create_basic_prompt()
22
23         question_answer_chain = create_stuff_documents_chain(self.
       llm, system_prompt)
24         rag_chain = create_retrieval_chain(self.retriever,
       question_answer_chain)
25
26         return rag_chain
```

Listing 4.10: Metodo create_qa_chain() della classe RetrievalChain

Quelli sopracitati sono i metodi principali della classe, i quali convergono nella funzione `ask` (o `ask_find_replace`) per poter generare una risposta partendo da una specifica richiesta dell'utente.

```
1
2  def ask(self, question: str) -> Any:
3          """Return the RAG response to the input question.
4
5          Args:
6              question (str): User input question.
7
8          Returns:
9              Any: The RAG response dictionary.
10         """
11         rag_answer = self.question_answering_chain.invoke(
12             {"input": question, "chat_history": self.
       create_history_string(self.chat_history)}
13         )
```

```
14          fixed_rag_answer = self.fixing_chain.invoke({"input":
    rag_answer["answer"]}).strip()
15
16          rag_answer["original_input"] = question
17          rag_answer["fixed_answer"] = fixed_rag_answer
18          rag_answer["chat_history"] = deepcopy(self.chat_history)
19          self.update_history(rag_answer["original_input"],
    rag_answer["answer"])
20
21          return rag_answer
```

Listing 4.11: Metodo ask() della classe RetrievalChain

La risposta generata viene infine sottoposta a un processo di raffinamento attraverso un'ulteriore chain, la quale, mediante un apposito prompt, elimina eventuali imperfezioni presenti nel testo. Il risultato così ottimizzato viene poi memorizzato nel campo "fixed_answer", consentendo di restituire al client un dizionario strutturato contenente la domanda originale, la risposta iniziale e la sua versione corretta.

4.2.4 Generazione titoli conversazioni e template

All'interno del modulo `chain.py` è presente un'ulteriore classe denominata **ConversationalChain**, la cui funzione principale è la generazione dei titoli delle conversazioni e dei template richiesti dagli utenti attraverso la funzionalità "Generation Mode".

Il meccanismo su cui si basa questa classe è concettualmente simile a quello della RetrievalChain descritta in precedenza, anche in questo caso viene utilizzato il prompt engineering per ottere il risultato desiderato. Tuttavia, a differenza di quest'ultima, ConversationalChain non richiede l'impiego di un retriever, poiché l'elaborazione della richiesta avviene esclusivamente interrogando direttamente il modello di linguaggio, senza la necessità di recuperare informazioni aggiuntive dal database vettoriale, rendendo il processo più lineare e immediato.

Nello specifico, il prompt utilizzato per la generazione dei titoli è il seguente:

```
1     prompt_template = """
2           Given the following question, generate a concise title of
    no more than five words that summarizes
3           the main topic or theme. The title should be clear,
    relevant to the question, and capture its essence.
4           The title must be direct and to the point, avoiding
    unnecessary words.
5           It will be used as a title for a chat, in a chatbot
    interface, so it should be informative and
6           also not too much specific, in order to be a good starting
     point for a conversation.
```

```
        \n\n
        Question: {question}
        """
```

Listing 4.12: Prompt per la generazione del titolo di una conversazione

Mentre quello utilizzato per generare un template a partire dalla richiesta dell'utente, in modo che possa modificarlo successivamente inserendo i dati opportuni, è il seguente:

```
    prompt_template = """You are an AI assistant specialized in generating structured legal documentation templates.
    Your task is to create precise, well-organized responses formatted in Markdown, suitable for formal legal use.

    ### Instructions for Generating Legal Documentation:
    1. **Clarity and Formality:**
    - Use precise legal terminology and formal language.
    - Avoid colloquial expressions or ambiguous wording.

    2. **Structure and Readability:**
    - Organize content using appropriate headings, subheadings, and numbered sections.
    - Include sections such as "Definitions," "Scope," "Obligations," "Liabilities," and "Signatures" as applicable.
    - Use bullet points, numbered lists, and tables to improve clarity and readability.

    3. **Customizable Placeholders:**
    - Provide placeholders (e.g., `[Insert name here]`, `[Insert date here]`) where users need to input specific details.
    - Use consistent formatting for placeholders, such as brackets or italic text.

    4. **Legal Accuracy:**
    - Ensure the template adheres to general legal standards and is adaptable to specific jurisdictions.
    - Add a disclaimer (e.g., "This template is for informational purposes and does not constitute legal advice.").

    5. **Comprehensive Coverage:**
    - Cover all critical sections necessary for the document's purpose.
    - Include optional sections for flexibility and additional details if needed.

    ### Output Requirements:
    - Format the entire response in Markdown.
    - Begin with an introductory section summarizing the document's purpose.
```

```
29      - Include placeholders for user-specific data.
30      - Ensure the document is concise, logically organized, and
        suitable for legal purposes.
31
32      ---
33
34      **Question:** {input}
35      """
```

Listing 4.13: Prompt per la generazione di un template testuale

Anche in questo caso, analogamente alla classe RetrievalChain, viene utilizzato il parametro `ask_mode` per discriminare i due possibili casi, mediante i valori `"title"` o `"template"`.

4.3 Sviluppo del Frontend

Come per il backend, il codice sviluppato per il frontend è strutturato gerarchicamente in cartelle e sotto-cartelle, seguendo un principio di separazione logica che garantisce maggiore chiarezza e manutenibilità.
La cartella che contiene tutti gli elementi necessari all'implementazione dell'interfaccia è denominata *"frontend"* e di fianco, nella Figura 4.1, se ne mostra il contenuto.
Nel file `package.json` sono elencate le dipendenze necessarie, con una definizione degli script di esecuzione, mentre il file `package-lock.json` viene generato automaticamente quando si installano le dipendenze (`npm install`). Contiene versioni esatte dei pacchetti, i cui file effettivi sono immagazzinati, opportunamente suddivisi, nella cartella `node_modules`.
Il cuore dell'applicazione è contenuto nella cartella `src`, all'interno della quale i componenti delle pagine che compongono l'interfaccia sono organizzati in cartelle individuali.

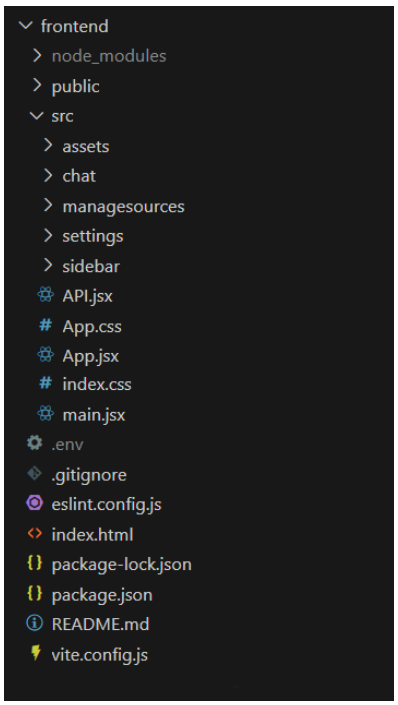

Figura 4.1: Organizzazione gerarchica Frontend

Nelle successive sotto-sezioni verranno analizzati i componenti principali che costituiscono l'interfaccia, descrivendo il loro ruolo, la suddivisione in apposite routes e la loro interazione all'interno dell'applicazione. Verranno inoltre descritte

le API implementate per consentire la comunicazione tra il frontend e il backend, contenute nel file `API.jsx`.

4.3.1 Suddivisione in Routes

La configurazione delle routes dell'applicazione è stata realizzata mediante l'utilizzo di React Router DOM, una libreria per la gestione della navigazione che permette di definire rotte e gestire il cambio di pagina senza ricaricare l'intero sito, migliorando l'esperienza utente. Il contenuto del componente principale App.jsx è racchiuso all'interno di un `<BrowserRouter>`, che abilita la gestione della navigazione lato client. Le rotte sono organizzate all'interno di un contenitore `<Routes>`, e ogni percorso è definito da un `<Route>` con uno specifico path. Il layout generale prevede una sidebar (`ChatSidebar`) fissa e un'area principale che cambia dinamicamente in base al percorso selezionato.

```
1
2   <BrowserRouter>
3     {/* resto del codice per i provider di Context */}
4     <div style={{display:"flex", width:"100vw", overflow:"hidden",
     height: "100vh"}} >
5       <ChatSidebar/>
6       <div style={{flex:1, overflowY:"auto"}}>
7         <Routes>
8           <Route path='/' element={<ChatLayout />} />
9           <Route path='/chat' element={<ChatLayout />} />
10          <Route path='/chat/:param_conversation_id' element={<
     ChatLayout />} />
11          <Route path='/settings' element={<SettingsPage/>} />
12          <Route path='/managesources' element={<DocumentsPage />}
     />
13          <Route path='*' element={<Navigate replace to='/' />} />
14        </Routes>
15      </div>
16    </div>
17    {/* resto del codice per i provider di Context */}
18  </BrowserRouter>
```

Listing 4.14: Suddivisione in Routes in App.jsx

Gli elementi associati alle diverse *routes* sono incapsulati all'interno di specifici *Provider* di contesto, consentendo loro di ereditare determinati stati dell'applicazione senza ricorrere al tradizionale meccanismo di passaggio delle *props*, che in alcuni casi può risultare complesso e di difficile lettura a livello di codice. Un esempio significativo è rappresentato dal contesto relativo allo stato `dirty`, un valore booleano che assume il valore `true` quando determinati dati vengono recuperati dal server tramite un'operazione di *fetch*. In tale scenario, i componenti che dipendono

direttamente da questo stato, la cui gestione è affidata all'hook `useState`, attivano un'operazione di *re-rendering* al fine di aggiornare correttamente la pagina.

4.3.2 Componenti

Per la realizzazione delle pagine è stato adottato un approccio basato su componenti e sotto-componenti, privilegiando anche in questo caso una struttura modulare che consente una migliore organizzazione del codice e ne facilita la manutenzione e l'estensibilità. Al loro interno, quando si è reso necessario accedere agli stati ereditati dai componenti di livello superiore, si è utilizzato in minima parte il meccanismo di passaggio delle props, ma soprattutto l'hook `useContext`, che ha consentito di ereditare i contesti definiti in precedenza. Per la realizzazione del layout dei vari componenti sono stati utilizzati gli elementi di react-bootstrap combinati all'utilizzo di alcune regole CSS, definite all'interno di appositi file.

La pagina principale dell'applicazione costituisce il punto di accesso primario alle funzionalità offerte, ed è implementata attraverso il componente React `<ChatLayout>`. In questa sezione, l'utente ha la possibilità di interagire con il chatbot e, se necessario, creare nuova documentazione avvalendosi di un editor di testo WYSIWYG. La versione estesa di questa pagina è illustrata in Figura 4.2.

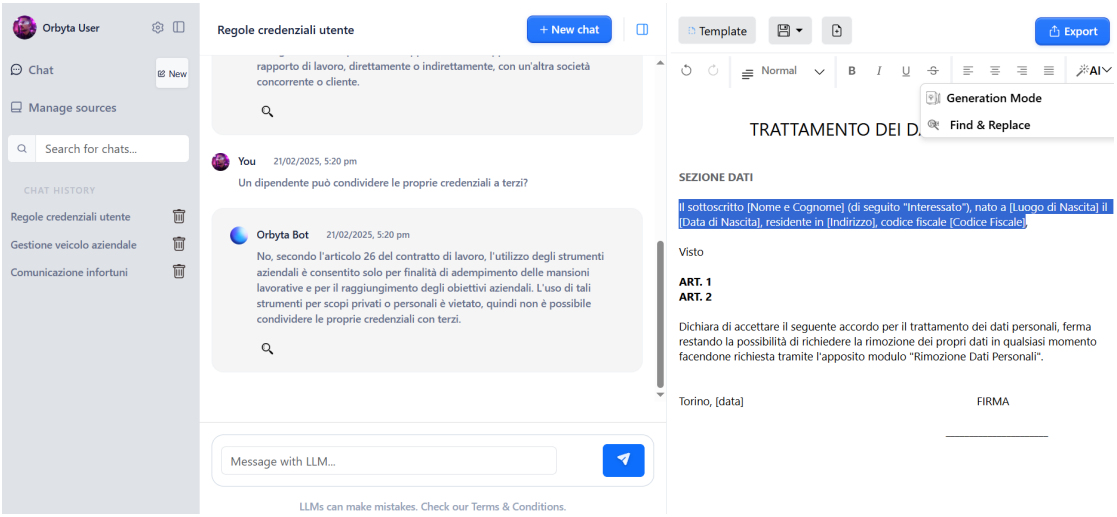

Figura 4.2: Schermata Frontend: Pagina principale

La schermata principale è strutturata in due sezioni distinte. Sul lato sinistro è presente l'area dedicata alla conversazione con il chatbot, dove ogni messaggio ricevuto consente di esplorare le relative fonti attraverso un pulsante con l'icona di una lente, questa funzione è tuttavia disponibile solo durante lo svolgimento della conversazione e non in un secondo momento, in quanto le fonti della risposta fornita

non vengono immagazzinate nel database. L'attivazione di questo pulsante apre un *Modal*, che permette di consultare i frammenti di documenti utilizzati per costruire il contesto della risposta generata. Sul lato destro si trova invece la sezione riservata all'editor di testo. Nella parte superiore sono disponibili il pulsante *Template*, che consente di visualizzare i modelli precedentemente salvati dall'utente e di caricarli nell'editor (Figura 4.3), il pulsante per il salvataggio del contenuto come nuovo template o come versione aggiornata di un modello esistente, e il pulsante per l'esportazione del testo in formato PDF, funzionalità realizzata mediante la libreria `react-pdf/renderer`. Al di sotto, è presente una toolbar contenente gli strumenti per la formattazione del testo, nonché le funzionalità basate su *AI*, tra cui la generazione di template tramite modelli LLM e la sostituzione automatica di dati nel testo attraverso il meccanismo RAG. Nella sua versione compatta, l'editor di testo risulta nascosto, permettendo di ampliare l'area dedicata alla conversazione con il chatbot. Il controllo della visibilità dell'editor è gestito tramite il pulsante *Create Document*, il quale assume la forma dell'icona posizionata accanto al tasto "*+ New Chat*" quando l'editor è visibile a schermo (Figura 4.2).

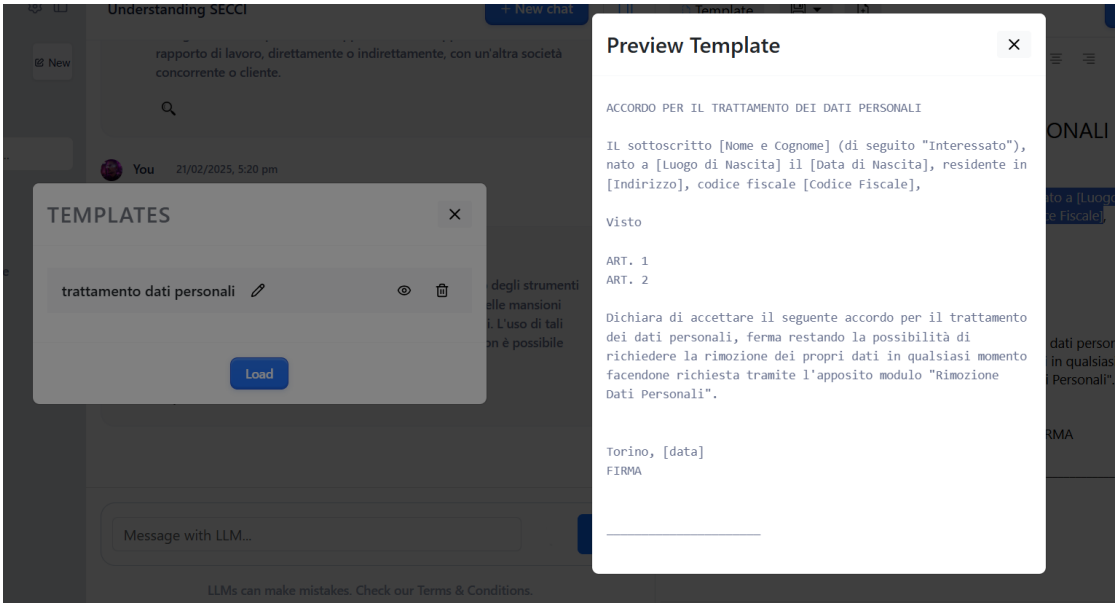

Figura 4.3: Schermata Frontend: Anteprima Template

L'editor di testo dell'applicazione è implementato come un componente React denominato *Editor* basato su Lexical, il quale si occupa della configurazione dell'editor e dell'integrazione con le diverse funzionalità offerte. L'elemento principale è il `LexicalComposer`, inizializzato con una configurazione specifica (editorConfig), che fornisce il contesto per la gestione dell'editor e delle sue estensioni. Il LexicalComposer rappresenta il cuore dell'editor di testo, fungendo da contenitore principale per

la gestione dello stato e delle funzionalità dell'editor. Questo componente fornisce un contesto condiviso tra i vari elementi che compongono l'editor, permettendo di configurarlo in modo modulare e flessibile attraverso l'integrazione di plugin, ossia quei componenti che estendono le capacità dell'editor, aggiungendo funzionalità senza modificare la struttura di base.

```
export default function Editor(props) {

  {/* resto del codice*/}

  return (
    <LexicalComposer initialConfig={editorConfig}>
      <div className="editor-toolbar d-flex justify-content-
   between align-items-center">
        <div className="toolbar-left d-flex align-items-center"
   style={{ gap: "8px", marginBottom: "8px" }}>
          <Button
            variant="light"
            className="toolbar-btn"
            onClick={() => setShowTemplatePopup(true)}
          >
            <Image src="/assets/template" />
            <span>Template</span>
          </Button>

          <div className="d-flex" style={{ gap: "4px" }}>

            {/* Codice DropDown Salva Template*/}

            {currentTemplate === null ? null :
              <Button
              variant="light"
              className="toolbar-btn"
              onClick={() => {
                setCurrentTemplate(null);
                setEditorText("");}}
            >
              <FileEarmarkPlus />
            </Button>
            }
          </div>
        </div>

        <div className="ml-auto" style={{marginBottom: "5px"}}>
          <PDFDownloadButton />
        </div>
      </div>

      <TemplatePopup
```

```
42          show={showTemplatePopup}
43          onHide={() => setShowTemplatePopup(false)}
44        />
45        <SaveTemplateModal
46          show={showSaveModal}
47          handleClose={() => setShowSaveModal(false)}
48          handleSave={handleSaveTemplate}
49        />
50        <ToolbarPlugin />
51        <div className="editor-container overflow-auto">
52          <div className="editor-inner">
53            <RichTextPlugin
54              initialEditorState={() => }
55                $convertFromMarkdownString(str, TRANSFORMERS);
56              }}
57              contentEditable={<ContentEditable className="editor-
    input" />}
58              placeholder={<Placeholder />}
59            />
60
61            <HistoryPlugin />
62            <CodeHighlightPlugin />
63            <ListPlugin />
64            <LinkPlugin />
65            <OnChangeMarkdown onChange={props.onChange} />
66            <ReadOnlyPlugin isDisabled={props.isDisabled} />
67            <ListMaxIndentLevelPlugin maxDepth={7} />
68            <MarkdownShortcutPlugin transformers={TRANSFORMERS} />
69
70            <SelectionHandlerPlugin onTextSelect={
    setTempSelectedText} />
71           <SyncContentPlugin content={editorText} onChange={
    setEditorText} />
72         </div>
73       </div>
74     </LexicalComposer>
75   );
76 }
```

Listing 4.15: Codice Editor Lexical in EditorContent.jsx

Per coordinare i vari plugin e accedere al contenuto dell'editor, al loro interno viene utilizzata la funzione `useLexicalComposerContext` importata da `lexical/react/LexicalComposerContext`. Una volta importato il contesto dell'editor, tramite l'utilizzo di alcune funzioni della libreria lexical (`$getSelection`, `$isNodeSelection`, `$getRoot`, `$createTextNode`, ...) è possibile accedere a determinati nodi che compongono il testo, crearne di nuovi o manipolare quelli esistenti.

L'unico elemento costantemente visibile nell'interfaccia è la sidebar laterale, che svolge un ruolo centrale nella navigazione dell'applicazione. Attraverso di essa, l'utente può accedere alle conversazioni precedenti, le quali sono elencate in ordine cronologico.
Selezionando il titolo di una conversazione presente nella sidebar, è possibile recuperare e visualizzare lo storico dei messaggi, permettendo all'utente di riprendere la conversazione interrotta. Inoltre, è disponibile un'operazione di eliminazione delle chat precedenti tramite un apposito pulsante contrassegnato dall'icona di un cestino. Quest'ultima, al passaggio del cursore, si ingrandisce per segnalare all'utente la possibilità di interagire con essa.
Oltre alla gestione delle conversazioni, la sidebar include collegamenti ad altre sezioni dell'applicazione. In particolare, sono presenti un'icona a forma di ingranaggio per accedere alle impostazioni, un collegamento alla pagina Manage Sources, dove è possibile caricare documenti testuali per il meccanismo di RAG, e un pulsante per avviare una nuova conversazione.

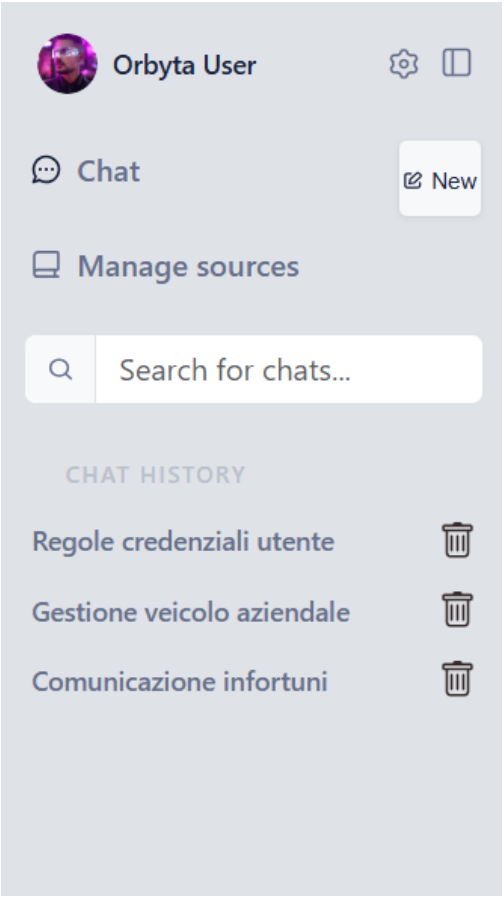

Figura 4.4: Sidebar laterale

Nella pagina Manage Sources, visibile in Figura 4.5, è possibile caricare i propri file di testo tramite il selettore accessibile tramite il pulsante "*Choose your file*" o trascinandolo all'interno del box, utilizzando il meccanismo di *Drag and Drop*. Come per le conversazioni nella sidebar, anche in questo caso è presente la stessa icona per l'eliminazione di un documento dal database, in alternativa, qualora si volesse svuotare l'intera collezione con un'unica interazione, sarebbe possibile farlo attraverso il bottone "*Delete all sources*".

É inoltre possibile cercare tra questi documenti tramite apposita barra di ricerca o ordinarli in base ai valori delle colonne che li descrivono, per esempio per data di ultimo aggiornamento. Lo status indica quando un documento è effettivamente

disponibile per il meccanismo di RAG. In un primo momento, quando l'utente carica i propri documenti, è necessaria un'attesa per il completamento di tale operazione, in quel lasso di tempo lo status dei documenti sarà *pending* e il cerchio, che in questa figura è di colore verde per segnalare la disponibilità all'utilizzo, sarà di colore rosso, con una freccia interna che gira per indicare il caricamento.

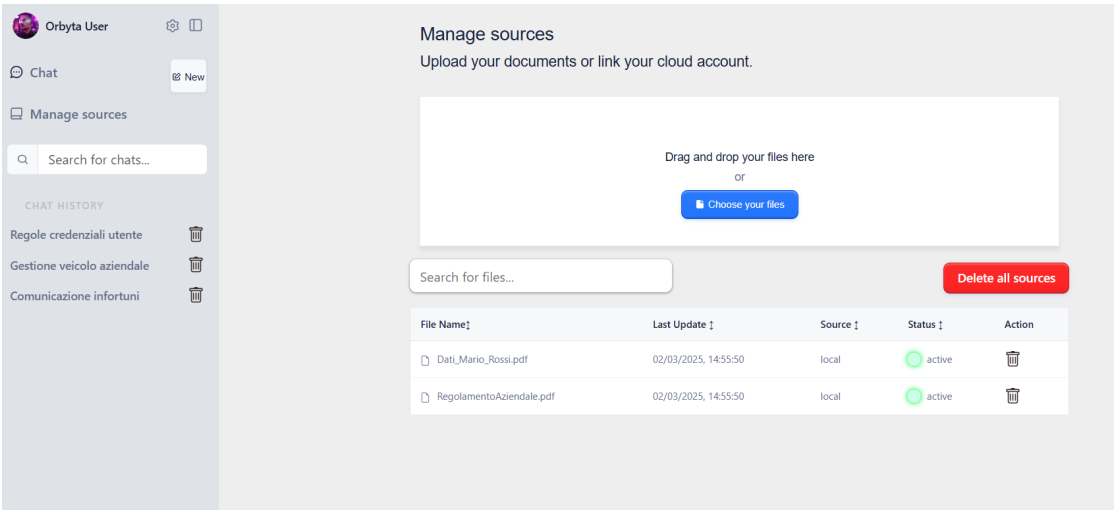

Figura 4.5: Schermata Frontend: Manage Sources

Dalla schermata delle *Settings*, accessibile tramite l'icona dell'ingranaggio presente nella parte superiore della sidebar, è possibile per l'utente selezionare i modelli da utilizzare, i relativi parametri, come lingua di risposta, temperatura o ricerca ibrida. Per inserire la chiave OpenAI è necessario utilizzare l'apposito box per l'input del testo e cliccare sul tasto *Verify*, attraverso il quale verrà prima verificata la validità della chiave fornita e successivamente, se valida, salvata tra le impostazioni dell'utente. Una volta inserita, renderà selezionabili i modelli OpenAI disponibili all'interno dell'applicazione.

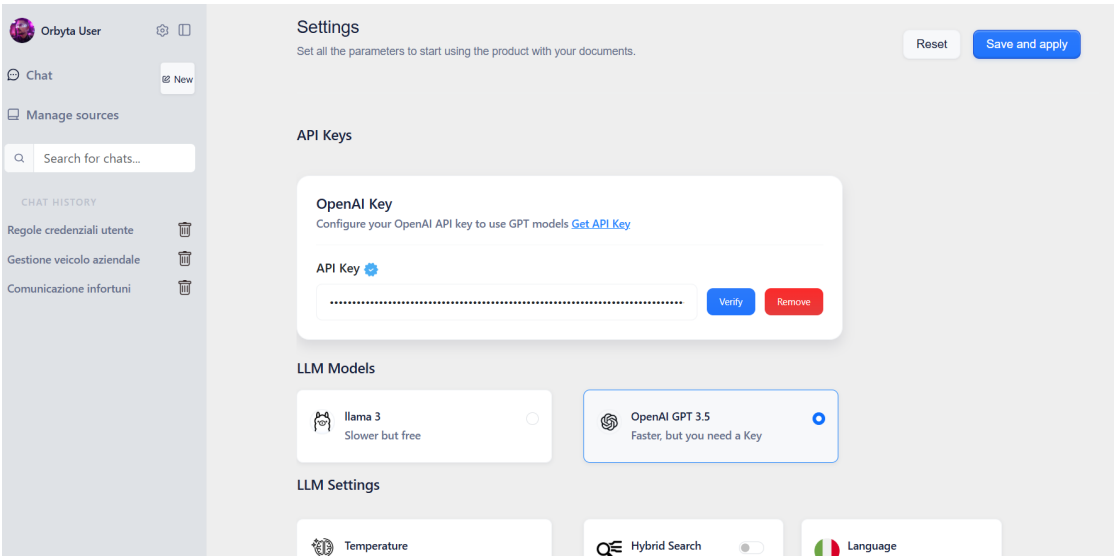

Figura 4.6: Schermata Frontend: Pagina Settings

4.3.3 API

Nel contesto dello sviluppo frontend, le API rappresentano il punto di accesso ai dati e alle funzionalità fornite da un server. Attraverso chiamate HTTP, il client può inviare richieste per recuperare, modificare o eliminare informazioni, rendendo possibile l'interazione con servizi esterni o backend. Per effettuare richieste HTTP dal frontend è stato utilizzato Axios, una libreria JavaScript basata su *Promise* tra le più utilizzate che semplifica le chiamate API rispetto alla funzione nativa fetch. Axios offre numerosi vantaggi, tra cui la gestione automatica della conversione dei dati in JSON, il supporto per le richieste e risposte intercettabili (interceptors), una gestione avanzata degli errori e una sintassi più chiara e intuitiva. Grazie a queste caratteristiche, Axios è spesso preferito a fetch, che richiede una gestione manuale più complessa per il parsing delle risposte e il trattamento degli errori.

Come anticipato, tutte le APIs, definite come funzioni asincrone tramite la keyword `async`, si trovano all'interno del file `API.jsx` e vengono esportate mediante `export`, in modo tale che siano accessibili tramite `import` all'interno dei componenti React nei quali si rendono necessarie.

Analizzando il tab di rete disponibile su *Firefox Developer Edition*, una versione avanzata del browser Firefox progettata specificamente per gli sviluppatori, è possibile vedere sia le richieste HTTP effettuate che le risposte ottenute in seguito a tali richieste. A titolo esemplificativo di tutte le altre API si riporta di seguito la risposta ottenuta in caso di successo della richiesta HTTP `POST /Q/ask`, richiamata all'interno del componente `ChatInput.jsx` al momento dell'invio del messaggio, attraverso la quale è possibile per l'utente interrogare il chatbot. La funzione

che viene eseguita in relazione a tale API è `sendMessage`, funzione che accetta come parametri della richiesta l'identificativo dell'utente e della conversazione, una domanda e la cronologia della chat, inseriti all'interno del body della richiesta.


```
{
    "input":"Quali sono gli orari di lavoro per un dipendente?",
    "chat_history":[],
    "context":[
        {"id":null,
         "metadata":
          {"page":9,
          "source":"uploaded_files/RegolamentoAziendale.pdf",
          "_id":"b1fff5a7-5f0e-4015-90bb-8dbf67d18e9a",
          "_collection_name":"documents_collection_f3fbb000-122d-4994-8205-8b87891dd050"},
          "page_content":"...contenuto...", "type":"Document"},
        {"id":null,
         "metadata":
         {"page":10,
         "source":"uploaded_files/RegolamentoAziendale.pdf",
          "_id":"66787aa2-c3b8-4e2a-9ae5-2edec554dc50",
          "_collection_name":"documents_collection_f3fbb000-122d-4994-8205-8b87891dd050"},
          "page_content":"...contenuto...","type":"Document"},
        {"id":null,
         "metadata":
         {"page":5,
         "source":"uploaded_files/RegolamentoAziendale.pdf",
         "_id":"40ebfb57-ee7a-4001-8340-8b902178f098",
         "_collection_name":"documents_collection_f3fbb000-122d-4994-8205-8b87891dd050"},
          "page_content":"...contenuto...","type":"Document"},
    "answer": "...Prima versione risposta...",
    "original_input": "Quali sono gli orari di lavoro per un dipendente?",
    "fixed_answer": "...Versione finale risposta...",
    "conversation_id": "1d7cea17-d5ef-4228-9490-b945bafff929"
}
```


La risposta visualizzata dall'utente corrisponde al contenuto del campo "fixed_answer", la quale viene affinata attraverso un meccanismo di pulizia basato sulle tecniche di prompt engineering descritte in precedenza, garantendo così una formulazione il più possibile priva di imperfezioni. Le informazioni presenti nel campo "context" rappresentano invece le fonti utilizzate per la generazione della risposta. Dal lato frontend, tali informazioni vengono opportunamente mappate e rese accessibili all'utente tramite un apposito *Modal*, il quale viene reso visibile

quando l'utente clicca sull'icona della lente d'ingrandimento, situata all'interno del box del messaggio ricevuto.

Capitolo 5

Risultati e Valutazioni

5.1 Analisi delle performance dei modelli

In questa sezione verranno analizzate le performance delle varie configurazioni possibili tra modelli di embedding e LLM, in modo tale da fornire una panoramica dei punti deboli e di forza di entrambi modelli.

Per effettuare le seguenti valutazioni, è stato utilizzato **RAGAs** [18], un framework ideato appositamente per valutare delle pipeline basate su RAG. RAGAs fornisce diverse metriche per la valutazione delle pipeline RAG, che possiamo raggruppare in due blocchi, quelle utili alla valutazione del componente di retrieval e quelle utili alla valutazione del componente di generazione della risposta.

A seguire una lista delle metriche utilizzate per la valutazione:

- **Context Precision**: misura la proporzione di chunks rilevanti nei contesti recuperati. Viene calcolata come la media delle Precision@k per ciascun frammento nel contesto.

$$\text{Precision@k} = \frac{\text{TruePositives@k}}{\text{TruePositives@k} + \text{FalsePositives@k}}$$

  La precision@k rappresenta il rapporto tra il numero di frammenti rilevanti al rango k e il numero totale di frammenti al rango k.

$$\text{C\_Precision@K} = \frac{\sum_{k=1}^{K} (\text{Precision@k} \times v_k)}{\text{Numero totale di elementi rilevanti nei primi } K \text{ risultati}}$$

  Dove K è il numero totale di frammenti nei contesti recuperati, mentre $v_k \in \{0,1\}$ è l'indicatore di rilevanza al rango k.

- **Context Recall**: valuta la capacità del retriever di recuperare tutti i documenti o le informazioni rilevanti necessarie per rispondere a una domanda. Un

alto recall significa che il sistema non ha tralasciato informazioni importanti. Si calcola come:

$$\text{Context Recall} = \frac{|\text{Documenti Rilevanti Recuperati}|}{|\text{Documenti Rilevanti Totali}|}$$

Dove il numeratore rappresenta l'intersezione tra le informazioni necessarie per rispondere alla domanda e le informazioni effettivamente recuperate nel contesto, mentre il denominatore rappresenta il totale delle informazioni necessarie per rispondere completamente alla domanda

- **Faithfulness**: misura la coerenza fattuale della risposta generata rispetto al contesto fornito. Indica quanto la risposta sia fedele alle informazioni disponibili, senza aggiungere dettagli non supportati ed evitando allucinazioni o informazioni non presenti nel contesto. Per calcolarla si usa la seguente formula:

$$\text{Faithfulness} = \frac{\text{N° affermazioni supportate dal contesto}}{\text{N° totale affermazioni nella risposta}}$$

- **Answer Rilevancy**: valuta quanto la risposta generata sia pertinente e direttamente correlata alla domanda posta dall'utente, dipende dalla capacità dell'LLM di comprendere la domanda e fornire risposte appropriate. Per poterla calcolare si tengono in considerazione 2 elementi, la domanda dell'utente e la risposta ottenuta, successivamente:
  - si genera un set di domande artificiali (il valore predefinito è 3) progettate per riflettere il contenuto della risposta
  - si calcola la similarità del coseno tra l'embedding dell'input dell'utente (E_o) e l'embedding di ognuna delle domande generate (E_{g_i})
  - si ricava la media dei punteggi di queste similarità

  Dunque la formula finale per poterla calcolare, sarà la seguente:

$$\text{Answer Relevancy} = \frac{1}{N}\sum_{i=1}^{N}\frac{E_{g_i}\cdot E_o}{\|E_{g_i}\|\|E_o\|}$$

  Dove:
  - E_{g_i} è l'embedding della i-esima domanda generata
  - E_o è l'embedding dell'input dell'utente
  - N è il numero di domande generate

Per eseguire queste valutazioni è stato utilizzato un dataset composto da 20 domande e le relative risposte considerate ottimali, dette *ground truth*, utilizzando come documento di riferimento il documento *RegolamentoAziendale.pdf*, contenente il regolamento interno dell'azienda Orbyta. Fatta eccezione per i modelli di OpenAI, ossia *text-embedding-ada-002* per gli embedding e *ChatGPT-3.5-turbo-instruct* come LLM, tutti gli altri modelli sono stati eseguiti localmente su un notebook con le seguenti caratteristiche:

- **Sistema Operativo**: Windows 10 Home 22H2 (64 bit)
- **CPU**: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 2.59 GHz
- **GPU**: Nvidia GeForce GTX 950M
- **RAM**: DDR3 16 GB
- **Archiviazione**: SSD Samsung 970 Evo Plus (500 GB)

Di seguito possiamo visualizzare i risultati ottenuti dopo la fase di evaluation con RAGAs:

Modello Embedding	Context Precision	Context Recall	Faithfulness	Answer Relevancy
all-MiniLM-L6-v2.gguf2.f16.gguf	0.7135	0.6875	0.4925	0.7921
text-embedding-ada-002	0.9114	0.9166	0.7264	0.6815
llama2-70b-4096	0.7552	0.4375	0.5563	0.2901
BAAI/bge-small-en	0.7864	0.7291	0.6743	0.5948

Tabella 5.1: Metriche RAGAs con Llama3

Modello Embedding	Context Precision	Context Recall	Faithfulness	Answer Relevancy
all-MiniLM-L6-v2.gguf2.f16.gguf	0.6510	0.7500	0.7025	0.9593
text-embedding-ada-002	0.8750	0.9666	0.7988	0.9586
llama2-70b-4096	0.8177	0.4375	0.6915	0.9544
BAAI/bge-small-en	0.7708	0.7291	0.7302	0.9109

Tabella 5.2: Metriche RAGAs con ChatGPT 3.5 Turbo

Come era facilmente intuibile, i risultati migliori sono stati ottenuti utilizzando i modelli di OpenAI. Vediamo infatti che il modello di embedding *text-embedding-ada-002* ottiene i migliori risultati in ognuna delle 4 metriche, fatta eccezione per il valore di *Answer Relevancy* usando Llama 3, risultato che tuttavia non è particolarmente significativo, poichè questa metrica, insieme alla *Faithfulness*, dipende particolarmente dalla capacità dell'LLM. Infatti, nella Tabella 5.2 è possibile vedere come in questa metrica, utilizzando *ChatGPT 3.5 turbo*, si ottengano dei risultati nettamente migliori rispetto al caso in cui viene utilizzato *LLama 3*. Il modello di embedding *llama2-70b-4096* ha mostrato prestazioni inferiori in entrambe le

configurazioni di LLM. A differenza degli altri modelli, che utilizzano embedding a 384 e 1536 dimensioni, questo modello impiega embedding a 4096 dimensioni. Questa differenza potrebbe spiegare i risultati, in quanto l'elevata dimensionalità potrebbe introdurre rumore e informazioni irrilevanti, che andrebbero a disturbare il calcolo della similarità, rispetto a modelli con dimensionalità più contenuta.

5.2 Test di usabilità

Per valutare l'efficacia dello sviluppo del progetto e ottenere una visione più chiara della qualità del risultato finale, è stato condotto un test di usabilità.

Un test di usabilità consiste nell'osservare persone mentre interagiscono con l'app per identificare eventuali problemi di usabilità, difficoltà d'uso o punti di frustrazione. Quando viene condotto un test di usabilità, si sottopongono ai partecipanti dei compiti, detti "*task*", da svolgere con l'applicazione oggetto di test.

Un test di usabilità serve a verificare quanto bene sia stata progettata un'applicazione, valutando quanto la sua interfaccia sia facile, intuitiva ed efficiente per gli utenti reali. Può essere condotto secondo diverse modalità, in presenza o da remoto, usando l'apparecchiatura più adatta allo specifico contesto. In ognuno dei possibili casi, oltre alla figura del tester, troviamo l'osservatore, ossia colui che *registra* ciò che avviene durante l'intero processo, acquisendo una traccia audio o video del test, oppure appuntando manualmente delle note riguardanti gli aspetti più rilevanti che emergono durante l'esecuzione del test.

L'obiettivo è quello di ottenere un feedback diretto da parte degli utenti, così da ottimizzare, eventualmente, l'esperienza utente (UX) migliorando la navigazione, l'interfaccia e le funzionalità, rendendo l'app più accessibile ed efficace.

Per condurre in modo efficace un test di usabilità è necessario strutturarlo nelle sue diverse fasi, così da avere un piano d'azione da rispettare.

5.2.1 Pianificazione

La prima fase di un test di usabilità è la pianificazione, fase in cui vengono delineate le modalità con cui verranno condotte le varie interviste e i candidati ai quali verranno sottoposti i test.

Per questo test è stato selezionato un gruppo di 5 tester, un numero che, secondo gli studi condotti da Jakob Nielsen [19], informatico danese pioniere nel campo dell'usabilità web, consente di ottenere informazioni qualitative sufficienti per intercettare la maggior parte dei problemi di usabilità. L'età dei partecipanti va dai 21 ai 30 anni e la caratteristica richiesta affinché un candidato possa essere considerato valido è la familiarità con software di creazione documentale (*Microsoft Word* o simili) e una minima confidenza con le piattaforme di intelligenza artificiale (*ChatGPT* o simili).

I test sono stati condotti tutti in presenza, modalità che ha semplificato notevolmente lo svolgimento consentendo di utilizzare come unica apparecchiatura un computer su cui eseguire in locale l'applicazione, un registratore per acquisire la traccia audio da analizzare successivamente e un blocco note per annotare manualmente elementi ed eventi particolarmente rilevanti. Per facilitare la comprensione del ragionamento dietro le scelte dei vari utenti, è stato chiesto di ragionare ad alta voce, spiegando le motivazioni alla base delle interazioni con l'interfaccia.

Successivamente allo svolgimento delle task delineate nella Tabella 5.3, è stato somministrato un questionario SUS (System Usability Scale), uno strumento ideato da John Brooke utilizzato per valutare l'usabilità di un prodotto, il quale si articola in 10 affermazioni alle quali gli utenti rispondono indicando quanto si trovano d'accordo su una scala da 1 a 5. Grazie a questo metodo, è possibile ottenere un feedback rapido sulla facilità d'uso percepita, permettendo ai progettisti di identificare aree di miglioramento nell'interfaccia utente.

5.2.2 Task da svolgere e Metriche

Successivamente alla pianificazione del test relativa alle modalità di svolgimento e al bacino di utenza a cui sottoporlo, si passa alla definizione dei task che gli utenti dovranno completare durante il test di usabilità.

I task sono stati pensati per testare le principali funzionalità dell'applicazione, seguendo un flusso coerente tra le diverse operazioni da svolgere. Per ognuno dei task presenti nella Tabella 5.3, vengono definiti i seguenti campi:

- **Numero del Task**: identificativo del task individuare rapidamente il task.
- **Testo del task**: indicazioni del compito da eseguire da comunicare al tester. Contiene le informazioni strettamente necessarie per eseguire il task, senza influenzare l'utente attraverso suggerimenti o dettagli che possano facilitare l'esecuzione.
- **Contesto**: breve panoramica offerta all'utente, in cui viene presentato qual è lo scopo alla base del task da svolgere.
- **Obiettivi**: insieme delle azioni da eseguire per completare con successo il task in questione.

Task N	**Testo del Task**	**Contesto**	**Obiettivi**

T1	Apri la pagina Settings, seleziona i modelli Llama 3 come LLM e GPT4All come embedding e salva le modifiche.	L'utente avvia l'applicazione la prima volta e deve settare le proprie impostazioni, selezionando un modello tra quelli disponibili per Embeddings e LLM	Entrare nella sezione delle settings cliccando sull'apposito pulsante nella sidebar; Selezioniare il modello LLM Llama3; Selezionare il modello GPT4ALL tra i modelli di embeddings già disponibili; Salvare le impostazioni tramite l'apposito tasto per rendere permanenti le modifiche;
T2	Apri la pagina Manage Sources e fai l'upload dei documenti "Dati_MarioRossi.pdf" e "RegolamentoAziendale.pdf".	L'utente deve caricare la propria documentazione (fornita da chi conduce il test) all'interno dell'applicazione per fornire una knowledge base dalla quale estrarre successivamente dati e/o informazioni	Entra nella sezione Manage Sources dalla sidebar; Effettuare la selezione dei documenti da caricare nella piattaforma tramite apposito bottone o trascinando i documenti nel riquadro apposito tramite Drag and Drop; Completare il caricamento tramite il pulsante Upload All;
T3	Avvia una nuova chat e chiedi al chatbot: "Cosa succede se non si comunica al datore di lavoro un infortunio?". Successivamente verifica le fonti della risposta.	L'utente vuole recuperare, attraverso una domanda al chatbot, un'informazione dal documento "RegolamentoAziendale.pdf" caricato precedentemente nella piattaforma e verificarne le fonti	Entrare nella pagina principale della chat tramite sezione sulla sidebar; Fare una domanda al chatbot riguardo la clausola specifica d'interesse utilizzando l'apposito input form; Una volta ricevuta la risposta, cliccare sul pulsante con lente d'ingrandimento per visualizzare fonti;

T4	Apri la pagina settings, inserisci e attiva la OpenAI API Key, successivamente seleziona ChatGPT 3.5 turbo e salva le modifiche.	L'utente ottiene una KEY (fornita da chi conduce il test) per sbloccare i modelli di OpenAI, vuole dunque cambiare le proprie settings per utilizzare un LLM diverso e ottenere performance migliori	Entrare nella sezione delle settings; Inserire il codice alfanumerico della chiave fornita nell'apposita sezione; Attivare la key tramite l'apposito tasto; Selezionare il modello ChatGPT 3.5 per LLM; Salvare le impostazioni tramite l'apposito tasto per rendere permanenti le modifiche;
T5	Apri la pagina Manage Sources e svuota la collezione di documenti, torna nella pagina delle Settings e seleziona il modello embedding OpenAI-ada-002, successivamente salva le modifiche. Al termine ripeti T2.	Inserendo la chiave di OpenAI, l'utente ha sbloccato un nuovo modello per embedding, vuole dunque utilizzarlo per ottenere performance migliori, ma è necessario svuotare la collezione di documenti per cambiare modello	Una volta letto l'alert, cliccare sulla shortcut che rimanda alla pagina Manage Sources; Svuotare la collezione di documenti tramite il pulsante Delete All; Entrare nella sezione delle settings cliccando sull'apposito pulsante nella sidebar; Selezionare il nuovo modello OpenAI di embedding; Salvare le impostazioni tramite l'apposito tasto per salvare e rendere permanenti le modifiche; Ripetere T2;
T6	Avvia una nuova chat e passa in modalità "Create Document", tra le funzioni della barra degli strumenti dell'editor attiva "Generation Mode" e chiedi al chatbot: "Genera un template per un contratto di apprendistato".	L'utente vuole creare un nuovo template da poter riutilizzare successivamente (es: contratto di apprendistato) utilizzando il supporto dell'AI, senza dunque scriverlo manualmente	Entrare nella pagina principale della chat tramite sezione sulla sidebar; Premere il tasto Create Document per splittare lo schermo in Chat + Editor; Avviare la modalità di generazione di template dalla barra dell'editor; Chiedere al chatbot di generare il template per un "Contratto di apprendistato";

T7	Salva il template generato nominandolo "Contratto di apprendistato".	L'utente è soddisfatto del template realizzato, vuole adesso salvarlo tra i suoi template in modo da poterlo riutilizzare successivamente	Cliccare sul tasto con l'icona del Floppy (Salvataggio); Dal menu a tendina che compare, cliccare "Save as"; Inserire un nome per il template; Completare il salvataggio cliccando sull'apposito bottone;
T8	Apri la sezione Template, visualizza l'anteprima di "Trattamento Dati Personali", successivamente caricalo all'interno dell'editor.	Si suppone che l'utente abbia utilizzato l'applicazione per un po' di tempo e che abbia salvato diversi template nell'applicazione. Adesso vuole recuperare un template tra quelli già salvati per poterlo compilare	Sempre dalla schermata Chat + Editor, premere il tasto Template; Visualizzare la preview di "Trattamento Dati Personali" cliccando sull'apposito bottone con l'immagine di un occhio; Chiudere il pop-up della preview; Selezionarlo e premere il tasto "Load";
T9	Seleziona la porzione di testo contenente i placeholder, attiva la modalità "Find & Replace" dall'editor e chiedi al chatbot di sostituirli con i dati di Mario Rossi. Al termine, esporta il pdf.	Completato il task precedente, l'utente vuole compilare il template con i dati di Mario Rossi (contenuti in un documento tra quelli caricati) con il supporto dell'AI ed esportarne il pdf	Selezionare, evidenziandola, la parte di testo contenente i placeholder dei dati della persona; Avviare la funzione "Find and Replace" dalla barra dell'editor; Chiedere al chatbot di sostituire i placeholder con i dati di Mario Rossi; Effettuare l'export del documento generato premendo l'apposito bottone "Export" dalla schermata dell'editor;

Tabella 5.3: Elenco delle Task per usability test

Per valutare il test di usabilità verranno utilizzate delle metriche definite in precedenza, in modo tale da poter discutere i risultati per i singoli task. Nello specifico, le metriche utilizzate sono:

- **Tempo**: espresso in secondi, indica la quantità di tempo necessaria all'utente per completare il task
- **Errori Critici**: conteggio degli errori di particolare rilevanza che allontanano l'utente dal raggiungimento degli obiettivi del task in questione
- **Successo**: valore numerico da 0 a 3 per indicare il tasso di successo per il task rispetto agli obiettivi che lo compongono, nello specifico:
  - 0 (Fallimento): L'utente non riesce a completare il task, si blocca e non sa come procedere, richiedendo un aiuto per completare il task
  - 1 (Successo con errore critico): L'utente riesce a completare il task, seppur commettendo degli errori che rallentano significativamente lo svolgimento del task o che si traducono in un risultato diverso da quello richiesto
  - 2 (Successo con lievi problemi): L'utente riesce a completare il task, seppure in modo non ottimale o commettendo qualche piccola imprecisione
  - 3 (Successo senza errori): L'utente riesce a completare perfettamente il task senza commettere nessuno tipo di errori

Per ognuno dei partecipanti, verranno discussi gli eventuali errori critici compiuti, sottolineando cosa ha portato l'utente a seguire un path diverso rispetto a quello pensato per il task e il tasso di successo, analizzando eventuali errori non critici e portando alla luce i ragionamenti e sensazioni riportare dai tester. Infatti, al termine dell'esecuzione del test e del completamento del questionario SUS, viene chiesto all'utente di esprimere un proprio parere riguardo l'interfaccia e il flusso di esecuzione, permettendo così di catturare spunti di riflessione per eventuali punti critici.

5.2.3 Risultati dell'esecuzione

Nelle successive sotto-sezioni vengono riportati i risultati dei singoli test condotti e, successivamente, il risultato complessivo del questionario SUS. Il tempo per lo svolgimento delle task è conteggiato al netto del tempo necessario per ottenere una risposta dai modelli di intelligenza artificiale alla base del programma, in quanto queste tempistiche, nel caso dell'esecuzione dei modelli open-source, è influenzata dalle prestazioni dell'hardware utilizzato per eseguire i test e non dall'effettiva attività del tester.

Partecipante 1

Task N	Tempo	Errori Critici	Successo
T1	30 s	0	3
T2	25 s	0	3
T3	133 s	0	2
T4	31 s	0	3
T5	19 s (+ 15 s per T2)	0	3
T6	86 s	1	1
T7	17 s	0	3
T8	15 s	0	3
T9	40 s	0	3

Tabella 5.4: Risultati task: Partecipante 1

Il Partecipante 1, fatta eccezione per il task T6, riesce a completare con successo tutti i task senza commettere alcun errore critico. Nello specifico, nel task T6, alla richiesta di *Generare un nuovo template*, clicca impulsivamente sul bottone *Template* nella sezione immediatamente superiore alla toolbar dell'editor. Una volta esplorata brevemente la schermata che gli si presenta davanti, capisce di non essere nella sezione corretta dell'applicazione. Chiude dunque il pop-up e torna nella schermata precedente, a questo punto pone maggiore attenzione alla toolbar e si accorge del pulsante con menu a tendina *"AI"*, a quel punto riesce a completare correttamente il task.

Durante lo svolgimento del task T3, perde del tempo per trovare il pulsante con la lente d'ingrandimento per *verificare le fonti* della risposta generata. Infatti, ragionando ad alta voce, in un primo momento esprime la sua perplessità riguardo al come eseguire l'operazione, chiedendosi se non sia necessario formulare un'altra domanda al chatbot per chiedere le fonti. Tuttavia non procede con l'effettuare la nuova domanda poichè valuta l'operazione poco pratica, si prende del tempo per guardare meglio la schermata e infine trova il pulsante cercato.

Dopo aver completato il test, si dice soddisfatto dell'interfaccia e del funzionamento dell'applicazione, suggerendo di apportare una modifica nel campo di input del chatbot:

"Immagino che io non debba inviare un messaggio al chatbot mentre sta ancora processando la richiesta precedente, ma secondo me sarebbe meglio fare in modo che questo non fosse possibile nemmeno volendo, bloccando il campo di input quando si è in attesa di una risposta. In questo modo si potrebbe evitare a monte un possibile errore da parte dell'utente"

Partecipante 2

Task N	Tempo	Errori Critici	Successo
T1	27 s	0	3
T2	33 s	0	3
T3	51 s	0	3
T4	39 s	0	3
T5	15 s (+ 30 s per T2)	0	3
T6	51 s	1	0
T7	14 s	0	3
T8	11 s	0	2
T9	58 s	0	2

Tabella 5.5: Risultati task: Partecipante 2

Nel test effettuato con il Partecipante 2, tutti i task sono completati con successo, eccetto il task T6 nel quale sono stati riscontrati dei problemi che hanno impedito il completamento. Nello svolgimento di T6, una volta cliccato il bottone *Create Document*, il tester clicca immediatamente il pulsante *Template*, senza soffermarsi sugli strumenti presenti nella toolbar. Una volta aperto il pop-up in cui sono presenti i template salvati sulla piattaforma per l'utente in questione, che nel caso del test contiene solamente il template "*Trattamento dei dati personali*"), clicca sul bottone con la matita e rinomina il suddetto template in "*Contratto di apprendistato"*, lo seleziona e preme il tasto "*Load*". Quando compare il template sull'editor, si accorge che non corrisponde a quello desiderato, decide di cancellare il contenuto dell'editor e chiede un supporto per continuare, per tale motivo ritengo di doverlo contrassegnare il completamento del task come *fallimento*.

Tuttavia, dopo aver dato come unica indicazione quella di prendersi qualche secondo in più per guardare la toolbar, il tester chiede di ripetere il task da capo e riesce a completarlo in 56 secondi. Gli altri task completati non in modo ottimale sono T8 e T9:

- in T8 il tester dimentica di visualizzare l'anteprima prima di caricare il template all'interno dell'editor, dunque lo carica senza accertarsi del contenuto

- in T9, nel tentativo di sostituire prova dapprima a sostituire i placeholder singolarmente, si accorge che ciò non è possibile, dunque procede selezionando l'intera porzione di testo in cui sono contenuti e completa il task

In generale, il tester si ritiene soddisfatto dell'applicazione, evidenziando però la poca intuitività della funzione *Generate Template*:

"D'istinto, per generare un nuovo template viene da cliccare sul tasto Template, anche se dando uno sguardo più attento si capisce che essendo una funzionalità supportata dall'AI si debba cliccare su quel tasto nella toolbar. Tuttavia potrebbe essere più intuitivo se, queste due funzionalità offerte dall'AI venissero messe

singolarmente nella parte superiore alla toolbar, sullo stesso livello dei tasti per l'export e per salvare, accanto al tasto Template"

Partecipante 3

Task N	Tempo	Errori Critici	Successo
T1	24 s	0	3
T2	35 s	0	3
T3	50 s	1	0
T4	35 s	0	3
T5	45 s (+ 22 s per T2)	0	3
T6	115 s	0	3
T7	21 s	0	3
T8	26 s	0	3
T9	122 s	0	2

Tabella 5.6: Risultati task: Partecipante 3

Il task più critico per il Partecipante 3 si è rivelato essere il task T3: una volta ricevuta la risposta dal chatbot, l'utente interpreta l'icona della lente d'ingrandimento come un bottone per cercare delle parole all'interno della risposta, per tale motivo non tiene minimamente in considerazione la possibilità di utilizzare questo bottone per verificare le fonti. Per raggiungere l'obiettivo ritiene necessario fare un'altra domanda al chatbot per chiedere esplicitamente di citare le fonti della risposta, ed una volta ricevuta la risposta successiva, nonostante non fosse pienamente convinto del risultato, ritiene di aver completato il task.

Tutti gli altri task vengono completati senza commettere errori o seguendo percorsi alternativi, fatta eccezione per T9, in cui, per selezionare il testo contenente i placeholder, inizia dapprima ad evidenziare i singoli placeholder, costatando l'impossibilità di selezionarli simultaneamente suppone di dover selezionare l'intera porzione di testo da compilare e poi procede, ottenendo il raggiungimento degli obiettivi preposti.

Al termine dell'esecuzione e del questionario, il partecipante dice di aver gradito la fluidità dei percorsi da seguire per compiere un'azione e suggerisce una modifica:

"Cambierei l'icona per visualizzare le fonti della risposta o la sostituirei con un bottone testuale, vedendo la lente d'ingrandimento ho pensato subito che fosse collegata ad un qualche tipo di ricerca. Per tutto il resto, mi sono trovato a mio agio nella navigazione, penso che l'applicazione possa essere un ottimo supporto nel lavoro quotidiano."

Partecipante 4

Task N	Tempo	Errori Critici	Successo
T1	26 s	0	3
T2	25 s	0	3
T3	40 s	1	0
T4	62 s	0	2
T5	21 s (+ 24 s per T2)	0	3
T6	64 s	0	3
T7	27 s	0	3
T8	24 s	0	3
T9	158 s	1	1

Tabella 5.7: Risultati task: Partecipante 4

Analogamente al test precedente, il Partecipante 4 non riesce a completare con successo il task T3, seguendo lo stesso comportamento precedentemente descritto. A differenza del test precedente, il Partecipante 4 non si sofferma minimamente sul box della risposta. Non vedendo l'icona della lente d'ingrandimento, si dirige immediatamente verso il box di input per formulare la nuova domanda. Conclude il task allo stesso modo del Partecipante 3, non raggiungendo l'obiettivo.

Nella task T4, dopo aver inserito la chiave di OpenAI, invece di cliccare direttamente sul nuovo modello LLM, clicca sul modello LLama3 per deselezionarlo, quando vede che non è possibile farlo clicca sul nuovo modello per selezionarlo e poi salva le modifiche. Riesce dunque a completare con successo il task, commettendo un errore lieve che non compromette la buona riuscita.

Nel completamento del task T9 viene registrato un errore critico: tenta in prima battuta di attivare la modalità *Find and Replace* senza aver evidenziato il testo dal menu a tendina del tasto *AI*, letto il banner che suggerisce di selezionare il testo, chiude il menu ed inizia a ed esplorare la toolbar dell'editor. Clicca sul tasto per sottolineare il testo, pensando servisse a evidenziare la porzione di testo da sostituire tramite la funzionalità, quando capisce che il tasto serve a sottolineare fisicamente il testo torna sui suoi passi, evidenzia un solo placeholder e avvia la modalità Find and Replace, ottenendo un risultato parziale rispetto all'obiettivo fissato.

Terminato il test, esprime le sue impressioni riguardo l'interfaccia e il test appena svolto:

"Credo che questa applicazione possa essere utilizzata anche in ambiti diversi da quelli aziendali o legali, potrebbe essere comoda anche per studenti che vogliono consultare i propri appunti in modo più rapido e comodo, anche se in quel caso le

funzioni di AI integrate nell'editor non sarebbero molto compatibili, ne servirebbero di nuove. Per quanto riguarda la visualizzazione delle fonti non ho proprio preso in considerazione la possibilità di farlo tramite un bottone, pensavo di poter interagire solo tramite domande, dunque non ho nemmeno guardato se ci fosse un bottone all'interno del box di risposta."

Partecipante 5

Task N	Tempo	Errori Critici	Successo
T1	35 s	0	3
T2	32 s	0	3
T3	43 s	0	3
T4	45 s	0	3
T5	30 s (+ 20 s per T2)	0	3
T6	48 s	0	3
T7	19 s	0	3
T8	27 s	0	2
T9	82 s	0	2

Tabella 5.8: Risultati task: Partecipante 5

Il Partecipante 5 completa tutte le task rapidamente e con successo, senza nessun errore critico. L'unico errore lieve riscontrato, che può assumere importanza in relazione agli altri test, si riscontra nel task T9, in cui l'utente prova prima ad evidenziare i placeholders singolarmente per poi accorgersi che ciò non è possibile. Una volta effettuato questo tentativo, evidenzia l'intero testo d'interesse e completa il task con successo.

Al termine del test, l'utente si dichiara complessivamente soddisfatto dell'esperienza d'uso, descrivendo l'interfaccia come intuitiva e accessibile. Inoltre, propone alcune idee per migliorarne ulteriormente l'interfaccia e l'efficacia del sistema:

"Per rendere più gradevole l'aspetto visivo dell'applicazione, cambierei il colore della barra superiore dell'applicazione e del box di input del testo, in modo da creare uno stacco tra le diverse sezioni dell'interfaccia. Per rendere più comoda l'azione di salvataggio delle impostazioni farei in modo di rendere fisso in alto il bottone anche quando l'utente scorre tra i diversi parametri, non visualizzandolo costantemente sullo schermo l'utente potrebbe dimenticare di applicare le modifiche prima di uscire dalla pagina. Per concludere, per migliorare ulteriormente l'applicazione, aggiungerei altre funzionalità all'interno dell'editor per renderlo più completo, come Microsoft Word."

Questionario SUS

Successivamente allo svolgimento delle task, è stato somministrato ai 5 utenti il questionario SUS. Per ottenere il risultato finale, che va da 0 a 100, vengono eseguite delle operazioni ben precise. Come anticipato, il punteggio che l'utente può attribuire alla singola affermazione va da 1 a 5, una volta ottenuto questo punteggio, se si tratta di una affermazione con indice pari (Q2, Q4, Q6, Q8 e Q10) si sottrae a 5 il punteggio dell'utente, se invece si tratta di una delle affermazioni con indice dispari (Q1 , Q3, Q5, Q7 e Q9), al punteggio dell'utente si sottrae 1. Si sommano tutti i valori ottenuti e infine si moltiplicano per un fattore *2.5*. Il punteggio finale può dunque essere espresso con la seguente formula:

$$Punteggio\_Sus = 2.5 \times \left(\sum_{i=1,3,5,7,9} (X_i - 1) + \sum_{j=2,4,6,8,10} (5 - X_j) \right)$$

dove X_i rappresenta il punteggio assegnato alla domanda i-esima.

Di seguito viene riportata in forma tabellare l'esito del questionario SUS dei vari partecipanti, per ognuna delle affermazioni viene riportato il punteggio fornito dall'utente, utile per ricavare il risultato finale.

Utente	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Risultato**
P1	5	2	4	1	5	1	5	1	5	1	95
P2	5	1	5	1	5	1	5	1	5	4	92.5
P3	5	1	4	1	4	1	4	1	4	1	90
P4	5	1	5	3	4	1	5	1	4	1	90
P5	4	1	5	1	4	1	5	1	5	1	95

Tabella 5.9: Risultati del questionario SUS

Un punteggio SUS superiore a 68 è generalmente considerato indicativo di una buona usabilità. Il sistema in esame ha ottenuto un punteggio medio di 92,5, superando significativamente la soglia di 68, questo risultato evidenzia dunque che il sistema soddisfa ampiamente i criteri di usabilità, garantendo un'esperienza utente di alta qualità.

Capitolo 6

Conclusioni

L'obiettivo del progetto di tesi era quello di sviluppare il prototipo di un'applicazione che integrasse al suo interno un chatbot basato su meccanismo di RAG con un'interfaccia intuitiva e semplice da utilizzare, che potesse supportare tramite l'AI professionisti che lavorano con una grande mole di documenti testuale nella fase di realizzazione di nuova documentazione.

Durante la prima fase, è stato condotto uno studio approfondito del meccanismo di RAG, in modo tale da poter comprendere appieno il funzionamento, analizzando i diversi modelli e strumenti disponibili per la sua implementazione. Questo primo studio preliminare è stato fondamentale ed indispensabile per porre le basi per la realizzazione dell'intero progetto.

Successivamente, è stata condotta un'analisi delle funzionalità da includere all'interno dell'applicazione per raggiungere l'obiettivo preposto, analisi necessaria per passare ad una prima bozza dell'interfaccia utente.

La fase successiva, infatti, è stata dedicata alla progettazione dell'interfaccia utente, partendo da un primo prototipo low-fidelity, per poi passare ad uno a mid-fidelity, quello che avrebbe guidato successivamente la realizzazione pratica dell'interfaccia e, dunque, del frontend.

Una volta ricavata una visione d'insieme delle funzionalità e dell'interfaccia, sono state prese in esame le diverse tecnologie attualmente a disposizione per poter realizzare il progetto. Parallelamente a questa fase, sono state definite le caratteristiche concettuali dei componenti che avrebbero composto l'intero sistema, delineando un piano preciso per la successiva implementazione.

Successivamente, si è passati allo sviluppo vero e proprio del codice dell'intera architettura, implementando Database, Backend e Frontend, seguendo quanto programmato in fase di progettazione.

Infine, nell'ultima fase del progetto, è stata condotta una fase di valutazione della piattaforma, analizzando le performance delle vaie configurazioni di modelli possibili ed effettuando un test di usabilità per l'interfaccia, testando gli scenari

d'uso più rilevanti, in modo tale da verificarne l'efficacia e di indentificare eventuali punti critici o elementi da migliorare.

6.1 Sviluppi Futuri

Miglioramento Editor

La principale limitazione all'interno del progetto è relativa alla presenza del testo formattato in Markdown. Talvolta, quando all'interno dell'editor è presente del testo formattato (Rich Text), per esempio alcuni elementi come titoli, sottotitoli o testo in grassetto, capita che la funzione *Find and Replace* non riesca ad iniettare correttamente il testo che l'API restituisce come risposta. Infatti, analizzando la *response* dell'API dall'apposito tab di rete del browser, è possibile vedere come il testo sia ricevuto correttamente. É possibile che questo bug dipenda dalla gestione dei nodi in Lexical e dalla sincronizzazione, tramite l'apposito Plugin realizzato, del testo presente all'interno dello stato di React e quello presente all'interno dello stato di Lexical.

Inoltre, per migliorare l'editor, potrebbe essere sviluppata la capacità di importare al suo interno template realizzati con *Microsoft Word*, in formato `.docx`, molto utilizzati in ambito aziendale. Questo, se realizzato nel modo corretto, potrebbe consentire agli utenti di importare documenti con immagini (per esempio loghi o filigrane) e di ottenere così un risultato più professionale.

Autenticazione

Tra gli sviluppi futuri di primaria importanza per il seguente progetto, vi è sicuramente l'implementazione del meccanismo di autenticazione. Come anticipato nei capitoli precedenti, l'architettura dell'applicazione è stata concepita con una predisposizione multiutente, pensando ad una futura implementazione in questo senso, per tale motivo la strada è già aperta a questo sviluppo, facilitandone l'integrazione e riducendo al minimo la necessità di modifiche strutturali.

L'introduzione dell'autenticazione rappresenta un passo fondamentale e necessario verso una piattaforma completa, per implementarla potranno essere prese in considerazione diverse opportunità tecnologiche in base al contesto di utilizzo dell'applicazione, che potrebbe essere destinata ad uso interno dell'azienda oppure ad una messa in produzione.

Ingestion Documenti dal Cloud

Un ulteriore sviluppo possibile per il progetto potrebbe essere l'integrazione con piattaforme di cloud storage, come *OneDrive* o *Google Drive*, in modo tale da automatizzare il processo di acquisizione dei documenti e facilitare la gestione

collaborativa degli stessi. In contesti lavorativi moderni, è sempre più comune che i team utilizzino cartelle condivise su cloud per archiviare e condividere documenti, l'implementazione di questa funzionalità eviterebbe all'utente finale il caricamento manuale da file system locale e garantirebbe allo stesso tempo la possibilità di importare molto più agilmente documenti realizzati da altre persone all'interno del proprio team di lavoro, anche quando questi vengono aggiornati da diversi utenti.

Una possibile idea per l'implementazione di tale funzionalità potrebbe essere un sistema di monitoraggio dei metadata dei file presenti su cloud, nello specifico si potrebbe effettuare un controllo sulla data di ultima modifica del documento e confrontarla con quella memorizzata nel database dell'applicazione. Qualora venisse rilevata una differenza tra le due date, il sistema provvederebbe ad aggiornare, automaticamente o su richiesta dell'utente in seguito ad una segnalazione relativa alla versione non aggiornata dei file in questione.

Bibliografia

[1] B.O. Omoyiola. *Overview of the Social Implications, Risks, Challenges, and Opportunities of Big Data*. 2023. eprint: Vol.1No.4. URL: https://doi.org/10.1108/EOR-04-2023-0014 (cit. a p. 1).

[2] Yu Xie e Sofia Avila. *The Social Impact of Generative LLM-Based AI*. 2024. arXiv: 2410.21281 [cs.CY]. URL: https://arxiv.org/abs/2410.21281 (cit. a p. 1).

[3] Ziwei Xu, Sanjay Jain e Mohan Kankanhalli. *Hallucination is Inevitable: An Innate Limitation of Large Language Models*. 2024. arXiv: 2401.11817 [cs.CL]. URL: https://arxiv.org/abs/2401.11817 (cit. a p. 1).

[4] Orbyta. URL: https://orbyta.it/ (cit. a p. 2).

[5] Christopher D. Manning e Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. Cambridge, Massachusetts: The MIT Press, 1999. URL: http://nlp.stanford.edu/fsnlp/ (cit. a p. 6).

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser e Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cit. a p. 6).

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee e Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805 (cit. a p. 6).

[8] Tomas Mikolov, Kai Chen, Greg Corrado e Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781 (cit. a p. 7).

[9] Jeffrey Pennington, Richard Socher e Christopher Manning. «GloVe: Global Vectors for Word Representation». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. A cura di Alessandro Moschitti, Bo Pang e Walter Daelemans. Doha, Qatar: Association

for Computational Linguistics, ott. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162/ (cit. a p. 7).

[10] Piotr Bojanowski, Edouard Grave, Armand Joulin e Tomas Mikolov. *Enriching Word Vectors with Subword Information*. 2017. arXiv: 1607.04606 [cs.CL]. URL: https://arxiv.org/abs/1607.04606 (cit. a p. 7).

[11] LLama. URL: https://www.llama.com/ (cit. a p. 16).

[12] ChatGPT. URL: https://chatgpt.com/ (cit. a p. 16).

[13] Poetry. URL: https://python-poetry.org/ (cit. a p. 21).

[14] Qdrant. URL: https://qdrant.tech/documentation/ (cit. a p. 22).

[15] PostegreSQL. URL: https://www.postgresql.org/ (cit. a p. 27).

[16] Stackoverflow. URL: https://docs.google.com/spreadsheets/d/1kODyUrTPWvz5n0fpUovRdxfXsSikVHz6T3h9Kspuk8g/edit?gid=81955775#gid=81955775 (cit. a p. 30).

[17] Stackoverflow. URL: https://trends.stackoverflow.co/?tags=reactjs,vue.js,angular,svelte,angularjs,vuejs3 (cit. a p. 30).

[18] Shahul Es, Jithin James, Luis Espinosa-Anke e Steven Schockaert. *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. 2023. arXiv: 2309.15217 [cs.CL]. URL: https://arxiv.org/abs/2309.15217 (cit. a p. 74).

[19] Jakob Nielsen. *Why You Only Need to Test with 5 Users*. 200. URL: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/ (cit. a p. 77).