ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Progettazione, sviluppo e validazione di un ChatBot basato su LLM con RAG su documentazione aziendale

Relatore:
Prof.
Federico Chesani

Correlatore:
Dott.
Nicolò Donati

Presentata da:
Giulia Fattori

Sessione dicembre 2025

Anno Accademico 2024/2025

A chi non c'è più
e a chi è rimasto fino alla fine,
vi voglio bene

Abstract

Il rilascio sul mercato globale di un numero sempre maggiore di modelli linguistici di grandi dimensioni, con performance accuratamente valutate e studiate, ha portato molte aziende a tentare di integrarli all'interno dei propri processi aziendali.

Il seguente elaborato spiega le diverse fasi di progettazione, sviluppo e validazione di un *Chatbot* di assistenza aziendale, realizzato tramite l'impiego di uno dei suddetti modelli linguistici di grandi dimensioni. Tramite l'implementazione di una pipeline di *Retrieval-Augmented Generation*, è stato possibile creare una base di conoscenza specifica, basata su documenti di manualistica aziendale e di normativa legale; essa fornisce al modello le informazioni e la conoscenza necessarie per generare output in linguaggio naturale a partire da una domanda utente. Il chatbot è stato implementato come servizio fruibile dagli utenti tramite API, disponibile come immagine Docker all'interno di un *cluster* Kubernetes.

Si pianifica di rilasciare il servizio ad una prima serie di clienti *beta*, al fine di testare le performance generali, prima di renderlo effettivamente disponibile sul mercato.

Indice

Abstract **i**

1 Introduzione **1**

2 Preconoscenze Teoriche **4**

2.1 Natural Language Processing 4

2.2 Large Language Models 6

2.2.1 Caratteristiche 6

2.2.2 Architettura 7

2.2.3 Addestramento 7

2.3 RAG 9

2.3.1 Estrazione 9

2.3.2 Recupero 10

2.3.3 Generazione 10

3 Progetto di Base **12**

3.1 Framework 12

3.1.1 Langchain 13

3.1.2 Llamaindex 14

3.2 Server e API 15

3.3 GUI 16

4 Processamento dei dati **20**
4.1 Estrazione e Chunking 21
4.1.1 Prima Versione 21
4.1.2 Semantic Chunking 23
4.1.3 Migliorie Successive 24
4.2 Embedding 26
4.2.1 Modelli 26
4.2.2 Test 27
4.3 Database 29
4.3.1 ChromaDB 30
4.3.2 PostgreSQL e pgVector 31
4.4 Immagini 33
4.4.1 Prima Versione 33
4.4.2 Generazione Descrizione 34
4.4.3 Estrazione Testo Alternativo 35
4.5 Normativa 37

5 Chatbot Business Logic **39**
5.1 Chatbot 39
5.2 Retriver 41
5.3 Rag Executor 43
5.3.1 Conversazioni 43
5.3.2 Risposte 45

6 LLM e Prompt **47**
6.1 LLM 47
6.1.1 Mistral 48
6.1.2 Llama3.2 49
6.1.3 Cohere Command R 50
6.1.4 Oracle 51

6.2 Prompt 53
6.2.1 System Prompt 55
6.2.2 User Prompt 56
6.2.3 Prompt di Riformulazione 57

7 Test Automatici **60**
7.1 Benchmarks 60
7.1.1 Correttezza Risposte 61
7.1.2 Tempi di Elaborazione e Numero Massimo Interazioni . 63
7.1.3 Conclusioni 64
7.2 Macchina Linux 65

8 Limitazioni Funzionali **67**
8.1 Metriche Automatiche 67
8.2 Documenti 68
8.3 Retrieval 70
8.4 Prompt di Riformulazione 71

9 Salvataggio Conversazioni **73**
9.1 Conversation Manager 74
9.2 Server 76
9.3 Interfaccia Utente 77

10 Redis **80**
10.1 Problematiche 80
10.1.1 Concorrenza 80
10.1.2 Persistenza 81
10.2 Redis 82
10.2.1 Conclusioni 84

11 Docker **86**
11.1 Immagini 87
11.2 Docker Compose 88
11.3 Macchina Linux 89

12 Autenticazione **90**
12.1 Implementazione 92
12.1.1 API Key 92
12.1.2 API Key Manager 93

13 Function Calling **95**
13.1 Tool 97
13.2 Prima Versione 98
13.3 Seconda Versione 100
13.4 Terza Versione 102

14 Deployment **104**
14.1 Kubernetes 104
14.2 Helm 105
14.3 Istio 107

Conclusioni **112**

Bibliografia **120**

Elenco delle figure

3.1 Diagramma di sequenza del chatbot come servizio. 17
3.2 Homepage dell'interfaccia utente. 18

5.1 Diagramma delle classi per *Chatbot*. 40
5.2 Diagramma di sequenza della fase finale della pipeline RAG. . 46

9.1 Struttura della tabella *chats*. 75
9.2 Struttura della tabella *messages*. 75
9.3 Tabella delle conversazioni. 78
9.4 Tabella dei messaggi. 78

12.1 Diagramma di sequenza con autenticazione positiva. 94

13.1 Diagramma di sequenza con *function calling*. 103

Elenco delle tabelle

4.1 Confronto tra modelli di *embedding*. 29

7.1 Confronto tra Cohere Command R e Llama3.2 sulle metriche esaminate. 62

Capitolo 1

Introduzione

Negli ultimi anni, l'Intelligenza Artificiale ha avuto un'esplosione, diffondendosi radicalmente non più solo tra gli esperti del settore ma tra la popolazione generale, sia a livello privato che commerciale.

Grazie ai continui sviluppi del *Natural Language Processing* (NLP), un campo dell'AI generativa che consente ai computer di riconoscere, comprendere e generare testo [61], è stato possibile addestrare sempre più *Large Language Models* (LLM), una categoria di modelli in grado di comprendere e generare contenuti e linguaggio naturale, e che sono ora in grado di svolgere un serie di funzionalità di base per gestire un'ampia gamma di casi d'uso [31]. Chatbot come ChatGPT di OpenAI e Gemini di Google sono chiare applicazioni di questi concetti, ora alla portata di tutti e più potenti che mai [62].

Creare e addestrare LLM è un compito estremamente oneroso in termini di tempo e risorse fisiche e finanziarie, in quanto il processo richiede quantità esorbitanti di dati ed energia e può durare anni [6]; solo i colossi tecnologici possono permettersi di sviluppare da zero questi modelli [7], ma ciò non preclude alle aziende più piccole il loro utilizzo, in quanto alcuni di essi sono direttamente open-source, mentre molti altri sono utilizzabili dietro pagamento.

Questa disponibilità ha portato molti fornitori di servizi a sviluppare propri chatbot e assistenti virtuali, specializzando il modello di partenza con i propri dati specifici; gli utenti possono rivolgergli direttamente le loro domande e, grazie al modello generativo sottostante e alla base di conoscenza, essi sono in grado di fornire risposte più o meno precise. Non è raro vedere questo tipo di assistenza nei siti web o nelle applicazioni mobili [63].

Il lavoro discusso in questo elaborato è stato svolto durante il periodo di tirocinio curriculare presso la sezione Ricerca e Sviluppo di Gruppo Finmatica, uno dei principali produttori di software e servizi ICT per la Pubblica Amministrazione e le Aziende Sanitarie [59]. Tra i loro prodotti principali vi sono diverse suite per la gestione dei flussi e dei processi aziendali, tra cui **Affari Generali* per la Pubblica Amministrazione e **H-ERP* per la sanità, fortemente collegati, anche se non in maniera diretta, con questo progetto.

Il tirocinio è stato dedicato allo sviluppo di un chatbot basato su intelligenza artificiale, non per uso aziendale interno, bensì per fornire assistenza ai clienti che già utilizzano i loro prodotti. Il chatbot andrebbe parzialmente a sostituire il sistema di assistenza corrente, basato sull'invio di ticket a cui un operatore umano deve rispondere manualmente: le domande più semplici, le cui risposte si possono trovare nei manuali utente degli applicativi, verrebbero poste direttamente all'assistente virtuale senza necessità di inviare ticket, con notevoli guadagni in termini di tempo ed efficienza sia per gli utilizzatori che per il reparto assistenza. Si sottolinea che il chatbot in questione non è pensato come uno strumento *general purpose*, ovvero capace di rispondere a domande di carattere generale su qualsiasi argomento, ma come un sistema specializzato nell'assistenza relativa ai prodotti aziendali.

Il primo passo è stato partire da un LLM open-source di base e specializzarlo, fornendogli come base di conoscenza specifica i manuali utente e di funzionamento dei vari applicativi tramite *Retrieval-Augmented Generation* (RAG). Sono state sviluppate due versioni del chatbot, una per ciascuna sui-

te sopracitata; esse sono identiche in tutto tranne che per base di conoscenza del modello, che cambia da prodotto a prodotto. Il primo ad essere sviluppato è stato quello per Affari Generali, e per semplicità si farà riferimento ad esso e ai relativi applicativi e manuali nel corso dell'elaborato.

Una volta ottenuto un chatbot funzionante, l'obiettivo è stato assicurarsi che rispettasse determinati standard di qualità e di correttezza nelle risposte che forniva. Sono state implementate caratteristiche come il salvataggio delle conversazioni, la multiutenza, l'autenticazione, e la containerizzazione. Infine, è stato effettuato il deployment del servizio completo sui server dell'azienda, in vista di una prova reale da parte di alcuni utenti *beta*.

Il capitolo 2 presenta un'introduzione teorica più dettagliata degli argomenti già menzionati, mentre i capitoli successivi illustrano in dettaglio le varie fasi progettuali e implementative necessarie per la realizzazione del progetto completo, suddivise per macro argomenti e aree tematiche. Sono inoltre analizzate le limitazioni progettuali e implementative del sistema, mentre l'ultimo capitolo include la conclusione dell'eleaborato e una sezione sui possibili sviluppi futuri.

Capitolo 2

Preconoscenze Teoriche

2.1 Natural Language Processing

Natural Language Processing (NLP) è un sottocampo dell'intelligenza artificiale che utilizza tecniche di machine learning per consentire alle macchine di comprendere, elaborare e comunicare con il linguaggio umano [61].

L'obiettivo di questa disciplina è fornire ai computer le capacità computazionali necessarie per interagire con il linguaggio umano e con gli umani stessi: estrarre informazioni da un testo, tradurre da una lingua all'altra, rispondere a domande e sostenere una conversazione; per rendere possibile tutto ciò, è necessario studiare e analizzare algoritmi e rappresentazioni del linguaggio, una scienza che prende il nome di *linguistica computazionale* [17].

NLP può essere classificato in due rami distinti: *Natural Language Understanding* (NLU) e *Natural Language Generation* (NLG), rispettivamente la comprensione e la generazione di testo. NLU permette alle macchine di comprendere il linguaggio naturale in termini di significato, contesto, emozione e forma stessa della lingua; NLG si concentra sulla produzione di frasi e paragrafi di senso compiuto, in grado di comunicare un messaggio in una qualche rappresentazione condivisa [36].

Combinare questi diversi aspetti presenta una serie di sfide: a parte l'enorme varietà di lingue esistenti, ciascuna di esse presenta ambiguità che rendono difficile stabilire con certezza il significato di una parola o frase; inoltre, i linguaggi umani sono pieni di irregolarità sintattiche e semantiche, sono in continua evoluzione in quanto nuove parole vengono costantemente create o importate, e spesso sono influenzati da fattori esterni come linguaggio del corpo e stato d'animo [61].

Alla linguistica computazionale sono state affiancate varie tecniche di machine learning per creare rappresentazioni sempre più complete. Tra quelle più tradizionali vi sono i modelli basati su regole, come gli alberi decisionali, che richiedono regole pre-programmate, e i modelli statistici, come i classificatori Naive Bayes e modelli di Markov nascosti, che invece sfruttano le probabilità per predire le parole successive [61, 12].

Al giorno d'oggi, le tecniche di deep learning rappresentano l'approccio predominante nell'ambito dell'intelligenza artificiale, grazie allo sviluppo sempre più avanzato di architetture di reti neurali complesse, come le reti neurali convoluzionali (CNN) e le reti neurali ricorrenti (RNN). Le CNN sono particolarmente efficienti nell'elaborazione di dati visivi e immagini, grazie alla loro capacità di catturare caratteristiche spaziali attraverso filtri convoluzionali; le RNN, invece, sono ideali per lavorare con dati sequenziali e temporali, come testi e segnali audio, in quanto mantengono una memoria delle informazioni precedenti nel processo di apprendimento. Queste reti neurali sono in grado di gestire grandi volumi di dati non strutturati, come testi, immagini e audio, permettendo di estrarre conoscenza e modelli predittivi con un alto livello di accuratezza [61, 12].

Le reti neurali sono inoltre la base di modelli di linguaggio di grandi dimensioni (LLM), che hanno rivoluzionato il campo dell'NLP, consentendo applicazioni come chatbot avanzati, traduzione automatica e analisi dei sentimenti.

2.2 Large Language Models

I *Large Language Models* (LLM) sono agenti computazionali progettati per comprendere, processare e generare linguaggio umano [17, 32], addestrati tramite tecniche di machine learning su enormi quantità di dati [44]. Sono in grado di eseguire una pletora di compiti legati a NLP, come interagire con le persone, rispondere a domande e sostenere conversazioni, analizzare contesti, sintetizzare, tradurre e generare testi, e molto altro [56, 32, 44].

2.2.1 Caratteristiche

Una delle caratteristiche più potenti degli LLM è la loro flessibilità: essendo addestrati su grandi quantità di dati non strutturati, essi imparano a identificare schemi e relazioni tra diverse idee e concetti, piuttosto che essere programmati per seguire un insieme fisso di regole [32]. Ciò fornisce loro le funzionalità di base per gestire diversi casi d'uso e applicazioni, rendendoli estremamente versatili: uno stesso modello può essere utilizzato in diversi ambiti, dalla finanza alla sanità all'educazione, senza dover essere totalmente riaddestrato, ma semplicemente specializzandolo [47, 32].

Il principio fondamentale del funzionamento degli LLM è la previsione: un modello linguistico non è altro che un sistema computazionale in grado di prevedere la parola successiva in una sequenza testuale, basandosi sulle precedenti [17]. Dato un contesto, il modello calcola con quale probabilità una certa parola seguirà, scegliendo poi quella con la probabilità maggiore; questa capacità di previsione è ciò che permette agli LLM di generare anche testo originale, sfruttando il loro addestramento precedente per creare una nuova sequenza di senso compiuto [17].

2.2.2 Architettura

Gli LLM hanno avuto molte evoluzioni negli anni. Al giorno d'oggi, i più avanzati utilizzano reti neurali abbinate all'architettura del trasformatore, un framework di deep learning per processare dati sequenziali da diverse fonti [44]. I trasformatori consentono a un modello di considerare simultaneamente tutte le parti di una frase o di un documento, permettendo di cogliere dipendenze a lungo raggio e relazioni contestuali più ampie [32].

Il modello del trasformatore scompone il testo grezzo in unità di testo di base più piccole, i *token*, che possono essere costituiti da parole intere, parti di parole o persino singoli caratteri, a seconda del caso d'uso. Questi vengono convertiti in rappresentazioni numeriche che catturano ordine, significato semantico e contesto, gli *embedding*, che attraversano una serie di livelli costituiti da due elementi: *auto-attenzione* e reti neurali [44].

Il meccanismo di auto-attenzione è un componente chiave: è ciò che consente al modello di concentrarsi su diverse parti di una sequenza di testo e di valutare dinamicamente il valore delle informazioni rispetto ad altri token nella sequenza, indipendentemente dalla loro posizione. Conferisce agli LLM la capacità di catturare le complesse dipendenze, relazioni e sfumature contestuali del linguaggio scritto [44, 47].

2.2.3 Addestramento

L'addestramento di LLM può essere suddiviso in tre fasi: *pretraining*, *finetuning*, e allineamento. Durante il *pretraining*, il modello viene allenato su un *corpus* enorme di testo, come pagine Web, libri, documenti pubblici, articoli, e vari dataset, imparando la grammatica, la semantica, la struttura, e le relazioni contestuali [17, 47]. Questo processo permette al modello di "imparare" una lingua, salvando tutte le informazioni nei propri parametri, che hanno funzione di memoria.

Esistono varie tecniche di apprendimento: gli *autoregressive model* imparano generando, iterativamente, la parola successiva in una sequenza basandosi solo sulle precedenti, da sinistra verso destra; i *masked models*, al contrario, "oscurano" un vocabolo in una frase e imparano a predirlo utilizzando quelli che lo circondano, sia a destra che a sinistra [17]. Dopo la fase iniziale, il modello è nuovamente allenato, questa volta su dataset più piccoli e specifici, al fine migliorare le prestazioni in uno specifico campo o compito [17, 44]. Un *finetuning* efficace bilancia il *pretraining* generico con la conoscenza specifica desiderata, affinché il modello mantenga una certa capacità di generalizzazione e non perda la conoscenza pregressa [47].

La fase finale di allineamento mira a rendere il modello il meno "dannoso" possibile: essendo allenato su testi umani, non privi di pregiudizi, disinformazione, stereotipi negativi e discorsi di incitamento all'odio, esso può assorbire queste tendenze e produrre risposte aggressive e pericolose [47, 56, 17]. Tecniche come il *Reinforcement Learning from Human Feedback* (RLHF) puntano a rendere il modello più sicuro, trasformando i valori umani in un formato comprensibile dalla macchina e addestrandola a rispettarli tramite meccanismi di ricompensa e punizione [47].

In aggiunta, gli LLM tendono ad essere soggetti alle cosiddette *allucinazioni*, ovvero alla generarazioni di informazioni oggettivamente false e fuorvianti. Mentre le moderne tecniche di addestramento insegnano ai modelli a generare testi coerenti e coesi, non sono ancora stati implementati algoritmi che permettano di imporre che questo testo sia anche corretto e veritiero [17]. Ciò porta alla diffusione di contenuti ingannevoli e pericolosi, con conseguenze gravi in ambienti come la sanità e la legalità, in cui l'accuratezza e la veridicità delle infomazioni sono essenziali [32, 17].

2.3 RAG

Retrieval-augmented generation (RAG) è una tecnica di ottimizzazione dell'output generato dagli LLM, in cui il modello viene arricchito da fonti esterne di conoscenza, spesso di carattere specifico, da integrare alle competenze di base apprese durante l'addestramento [4, 42, 34].

Il RAG estende le capacità degli LLM a domini specifici o, come nel caso del progetto preso in esame in questo elaborato, alla conoscenza interna di un'organizzazione, garantendo l'accesso a informazioni esterne verificate e aggiornate senza la necessità di riaddestrare il modello, con un notevole risparmio di costi [4, 43]. Inoltre, permette agli utenti di controllare le fonti utilizzate nella fase di generazione, in modo da verificarne l'accuratezza e ridurre allucinazioni e informazioni fuorvianti [42, 47].

Una *pipeline* RAG tradizionale è composta da tre fasi: estrazione, recupero, e generazione [34].

2.3.1 Estrazione

I dati aggiuntivi da passare al modello possono provenire da fonti eterogenee, come pagine web, archivi e database, e avere formati eterogenei, come file testuali, immagini, video, ecc. È fondamentale avere dati completi e organizzati, con duplicazione minima e metadati sensati, in modo da facilitare il lavoro successivo.

I documenti, specie quelli di testo, vengono suddivisi in sezioni più piccole, i *chunk*, prima di essere memorizzati. Salvare file in formato integrale non è consigliato perché, oltre ad essere costoso, rischia di superare il numero massimo di token a disposizione del modello, oltre che a confonderlo con un contesto troppo ampio e variegato [45].

Esistono diverse strategie di *chunking*: alcune si limitano a dividere il testo in sezioni di lunghezza fissa, senza considerare la formattazione pree-

sistente o il contesto; altre, più avanzate, separano il contenuto su base semantica, creando sezioni di lunghezza variabile ma consistenti nel significato [45].

Una volta preparati i dati, un modello di *embedding* li converte dal loro formato originale in rappresentazioni numeriche mappate in uno spazio ad alta dimensione, i *vector embedding*, che vengono poi salvati in database appositi, chiamati database vettoriali [34].

Questi *embedding* catturano il contesto e il significato semantico del dato originale, il che permette di calcolare matematicamente la distanza tra essi per determinare quanto siano simili semanticamente, un'operazione chiamata *similarity search* [18]. I database vettoriali sono appositamente ottimizzati per salvare, indicizzare e interrogare i vettori, e per svolgere operazioni nello spazio dei vettori, come la ricerca di somiglianza [58].

2.3.2 Recupero

Una volta organizzata la conoscenza specifica, il passo successivo consiste nel recuperare le informazioni desiderate dal database.

Data una query utente, questa viene prima convertita in *embedding* (tramite apposito modello) e poi utilizzata come base di una ricerca di somiglianza all'interno del database vettoriale; questa operazione recupera i *top-k chunk* più rilevanti per l'input corrente [4, 34].

2.3.3 Generazione

Nella fase finale, il modello combina le informazioni recuperate con il prompt utente e la query originale per generare una risposta appropriata, sia a livello di struttura che a livello di contenuti [34, 4, 47].

Come già menzionato, fornire dati aggiuntivi al modello riduce significativamente il rischio di allucinazioni, poiché il sistema può basarsi su in-

formazioni più precise e contestualizzate invece che su inferenze probabilistiche. Questo non solo aumenta l'accuratezza delle risposte, ma migliora anche la trasparenza del processo decisionale: se il modello dispone di fonti chiare, strutturate e pertinenti, diventa più semplice tracciare l'origine delle affermazioni e garantirne l'attendibilità.

Capitolo 3

Progetto di Base

Il seguente capitolo espone l'architettura di base del sistema, progettata e sviluppata per rendere il chatbot disponibile come servizio accessibile via API. La logica di business del chatbot, implementata interamente in linguaggio Python tramite una serie di oggetti e classi, verrà invece discussa nel capitolo 5.

3.1 Framework

Esistono diversi framework che supportano lo sviluppo di applicazioni AI, dalla fase di RAG alla gestione dei modelli. Risultano essere utili e versatili, in quanto forniscono un livello di astrazione uniforme che permette di interagire con i vari componenti attraverso interfacce standard; consentono di isolare la complessità interna dei componenti, nascondendo dettagli implementativi, protocolli o formati specifici. Di conseguenza, eventuali modifiche, *refactoring* o sostituzioni non impongono di riscrivere il codice circostante, garantendo una maggiore manutenibilità e portabilità dell'intero sistema.

Nel corso del progetto sono stati adottati due framework: *LangChain*, inizialmente utilizzato per la sua architettura modulare e per i numerosi strumenti per l'integrazione esterna; in una fase successiva, *LlamaIndex*, scelto

per le sue capacità più avanzate nel *retrieval* di informazioni e per il più alto grado di personalizzazione dei componenti. Le sezioni seguenti espongono più in dettaglio le caratteristiche proprie di ciascun framework, incluse le motivazione che hanno portato al passaggio dall'uno all'altro.

3.1.1 Langchain

LangChain è un framework open-source, progettato per supportare l'intero ciclo di vita delle applicazioni basate su modelli linguistici di grandi dimensioni. Offre un ambiente di sviluppo centralizzato e una serie di interfacce altamente astratte, compatibili con la maggior parte degli LLM più diffusi, e quindi favorendo lo sviluppo di codice agnostici rispetto al modello impiegato. Inoltre, mette a disposizione numerosi moduli dedicati alle diverse fasi della pipeline RAG, comprese delle integrazioni con vari database vettoriali [40, 30].

Nel contesto del progetto, LangChain è stato utilizzato per realizzare l'intera pipeline RAG, dalla fase di indicizzazione dei documenti alla fase di *retrieval* delle informazioni a seguito di domande utente. I componenti nativi del framework hanno permesso di implementare rapidamente il caricamento dei documenti, la costruzione dei vettori e la successiva interrogazione dell'indice, facilitando la prototipazione delle prime versioni del sistema.

Inizialmente, l'elevato livello di astrazione proposto da LangChain ha rappresentato un punto di forza: ha permesso di velocizzare lo sviluppo, ridurre la complessità percepita e concentrare l'attenzione sui comportamenti desiderati del sistema, anziché sulle implementazioni sottostanti. Tuttavia, con l'avanzare del progetto, questa stessa astrazione si è rivelata essere un limite: alcuni passaggi della pipeline risultavano difficili da ispezionare e personalizzare, rendendo complesso intervenire su aspetti critici quali la strategia di *chunking* e la configurazione delle strutture dati per l'indicizzazione. Inoltre,

il debugging era rallentato dal fatto che molte operazioni erano incapsulate all'interno di componenti predefiniti, lasciando poche possibilità di modifica.

Queste limitazioni riflettono un più generale *trade-off* tra astrazione e controllo: strumenti altamente astratti come LangChain permettono di costruire rapidamente prototipi funzionali, ma riducono la trasparenza e la possibilità di adattare i singoli componenti; al contrario, soluzioni più a basso livello richiedono un maggiore lavoro iniziale, ma offrono una visibilità completa sul comportamento del sistema e più personalizzazione.

Per rispondere a queste esigenze e ottenere un controllo più diretto sulle diverse fasi di sviluppo, si è quindi scelto di migrare verso LlamaIndex, un framework più leggero e adattabile alle esigenze specifiche del sistema.

3.1.2 Llamaindex

LlamaIndex è un framework open-source per la realizzazione di applicazioni basate su LLM, con un forte orientamento alla costruzione di pipeline RAG e allo sviluppo di sistemi conversazionali [41]. A differenza della soluzione precedete, esso adotta un approccio maggiormente *data-centric*, ponendo al centro dell'architettura la gestione esplicita dei dati, dei documenti e delle strutture di indicizzazione, mantenendo comunque un certo di livello di astrazione dalle implementazioni di basso livello. Il framework offre componenti modulari e altamente configurabili, che includono diversi tipi di indici e strategie di *chunking* personalizzabili [26].

Nel progetto, LlamaIndex è stato impiegato per ristrutturare e migliorare la pipeline RAG sviluppata in precedenza. Ha permesso di ottenere un controllo più accurato sulle fasi di *pre-processing* dei documenti, sulla costruzione degli indici e sulla logica di *retrieval*, oltre a facilitare l'analisi del comportamento.

La scelta di adottarlo come framework principale deriva dal fatto che, diversamente da LangChain, LlamaIndex consente un livello di personalizza-

zione e di controllo molto più elevato sui componenti e sulle strutture dati sottostanti. Questa maggiore flessibilità ha permesso di superare le limitazioni riscontrate nel framework precedente, permettendo un coinvolgimento diretto nelle scelte implementative, pur comportando un incremento del lavoro necessario per configurare e integrare correttamente i vari moduli.

3.2 Server e API

Per rendere il chatbot accessibile agli utenti e integrabile con altri applicativi, è stata adottata un'architettura client-server in cui la logica del sistema è esposta come servizio tramite API. L'elaborazione principale avviene lato server, mentre il client, indipendentemente dalla sua tecnologia, invia richieste HTTP e riceve risposte in formato standardizzato; ciò rende il chatbot un componente modulare e riutilizzabile, e semplifica l'integrazione con i software aziendali. È stato impostato un *rate limit* per utente di sei richieste ogni dieci secondi, in modo da evitare sovraccarichi e attacchi DoS.

Dal lato client, l'accesso alle funzionalità del chatbot avviene tramite chiamate API verso un server dedicato, implementato nello script *api_module*. Il servizio è stato sviluppato con FastAPI, un moderno framework Python pensato per la creazione di API web ad alte prestazioni e particolarmente adatto a scenari asincroni. L'esecuzione è gestita da Uvicorn, un server ASGI (*Asynchronous Server Gateway Interface*) leggero e ottimizzato, che garantisce un'ottima efficienza quando abbinato a FastAPI.

Il server espone tre endpoint principali, tutti realizzati come metodi POST tramite il decoratore *@app.post*. L'endpoint centrale è *process-query*, che riceve la query dell'utente, il suo identificativo, e la categoria, ovvere l'applicativo aziendale a cui fa riferimento la domanda; attiva quindi la pipeline RAG, interroga l'LLM e restituisce la risposta generata. Per supportare la

fase di sviluppo, l'endpoint fornisce anche i *chunk* di testo recuperati durante la fase di *retrieval*.

Gli altri due endpoint gestiscono il feedback sulle risposte: *process-feedback* registra un giudizio immediato ("positivo" o "negativo"), mentre *process-comment* consente all'utente di inviare un commento testuale più dettagliato. Entrambi ricevono l'id utente e il contenuto del feedback, e, per scelta di design, si riferiscono sempre e solo alla risposta più recente.

La ricezione dei commenti è una carattestica utile in fase di testing, in quanto permette ad alcuni dipendenti selezionati, gli esperti del dominio applicativo, di giudicare la qualità e correttezza delle risposte generate dal modello; i loro commenti e suggerimenti sono fondamentali, sia per implementare migliorie al sistema, sia per confrontare le performance del chatbot a seguito dell'introduzione di modifiche. Tutte le conversazioni, inclusi i feedback, sono salvate in un database SQL dedicato; questo aspetto è trattato in dettaglio nel capitolo 8.

Non si prevede di rendere disponibili questi endpoint agli utenti finali in fase di produzione, ma di manterli attivi solo per uso di sviluppo interno.

L'intero servizio è esposto sulla porta 8000, alla quale il client deve connettersi per inviare le richieste e interagire con il chatbot. Il corretto funzionamento della pipeline è stato verificato sia tramite Postman, sia inviando richieste da una GUI, descritta nella sezione successiva, confermando l'affidabilità del flusso end-to-end.

L'immagine 3.1 mostra la sequenza di azioni necessarie per ricevere una risposta, a seguito di una domanda, e per inviare un feedback generico.

3.3 GUI

Per testare il sistema e visualizzare le risposte in modo intuitivo, è stata sviluppata una semplice interfaccia utente web, realizzata con HTML e CSS

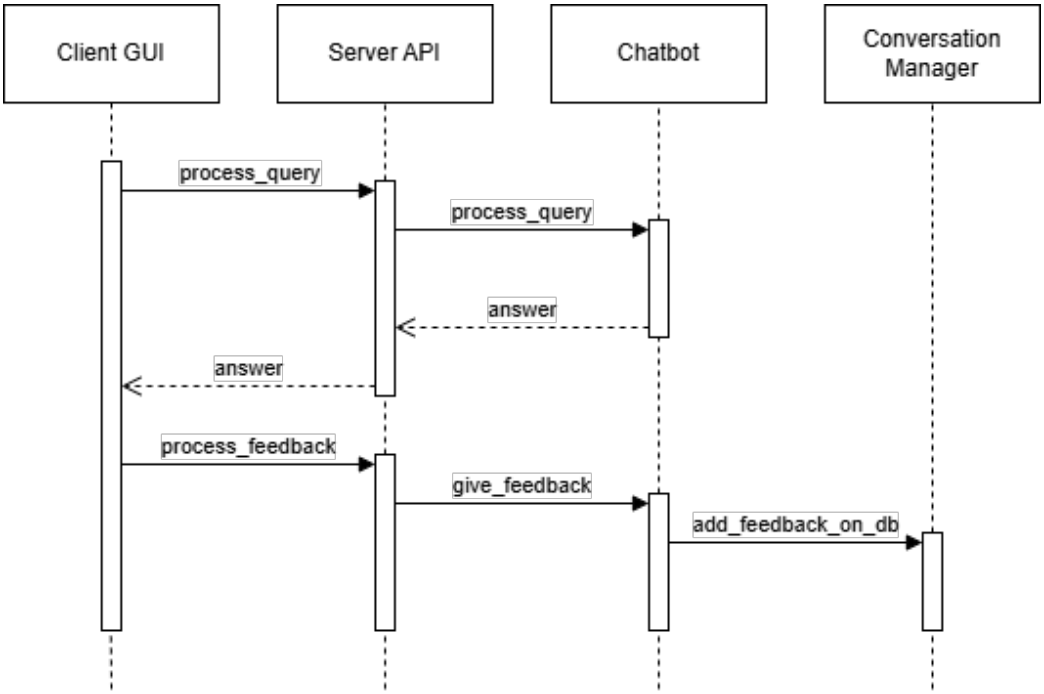

Figura 3.1: Diagramma di sequenza del chatbot come servizio.

per la struttura e lo stile, e JavaScript per la logica e la comunicazione asincrona con il server (tramite *await fetch("endpoint")*). Questa GUI è esclusivamente per uso interno di testing: nella versione finale, il chatbot verrà integrato come *pop-up* direttamente all'interno dei singoli applicativi aziendali.

Come mostrato in Figura 3.2, la pagina iniziale permette di selezionare la categoria della conversazione, che, una volta impostata, rimane costante per l'intera durata della sessione: se l'utente desidera modificarla, è necessario avviarne una nuova ricaricando la pagina.

A questo punto, è possibile inviare domande al chatbot; durante l'elaborazione della query, nella finestra di conversazione compare la scritta "Sto elaborando...", per informare l'utente sullo stato della risposta. Una volta pronta, essa viene riportata all'interno della finestra, mentre sulla destra è possibile lasciare i feedback tramite le icone di due pollici (positivo/negativo)

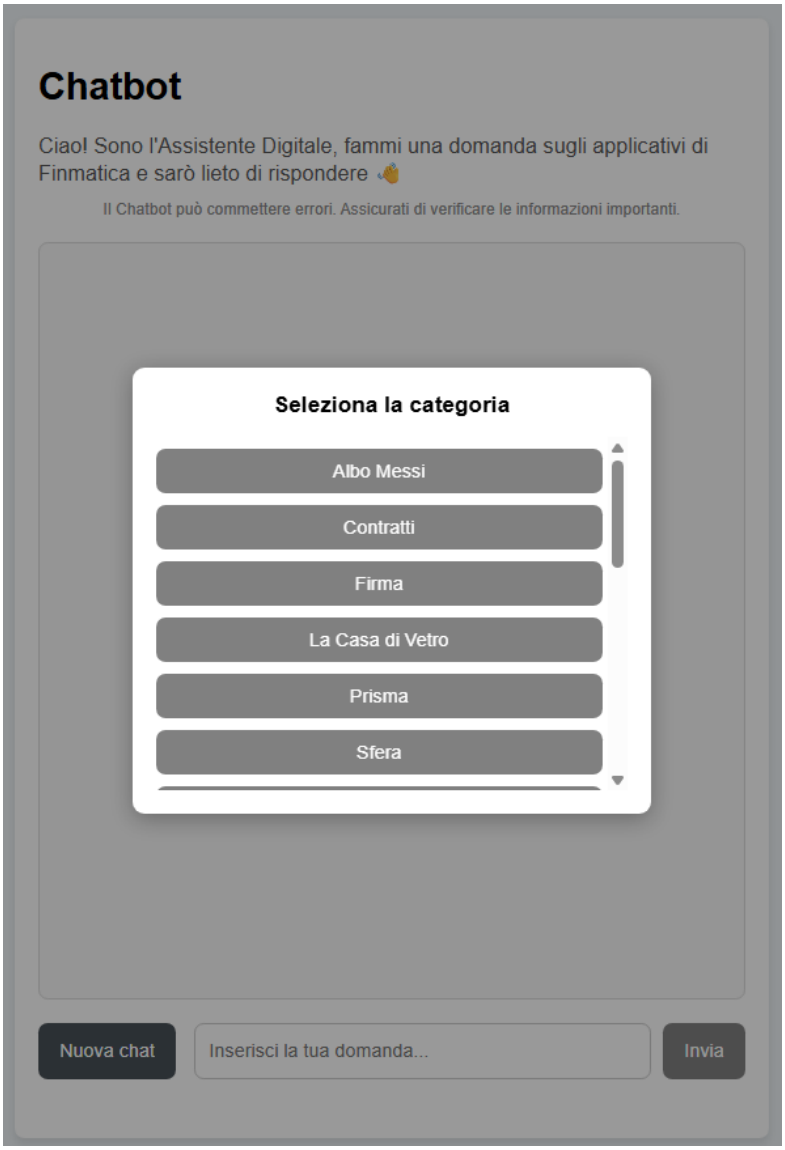

Figura 3.2: Homepage dell'interfaccia utente.

e un campo di testo per i commenti più dettagliati; a sinistra, un menù a tendina mostra i *chunk* di testo utilizzati per generare la risposta. Entrambi gli elementi compaiono automaticamente con la ricezione della risposta e scompaiono all'invio di una nuova domanda.

Ad ogni caricamento della pagina viene generato un nuovo UUID, che

identifica la sessione utente e definisce una nuova conversazione: ricaricando la pagina si avvia automaticamente una sessione distinta e quindi una nuova conversazione. Lato Javascript, viene inoltre verificato che la query non contenga caratteri o combinazioni proibite, prevenendo possibili *command injection* da parte di utenti malevoli.

Questa GUI consente di testare direttamente il servizio API, visualizzando il flusso del sistema e facilitando l'analisi per gli esperti di dominio.

Capitolo 4

Processamento dei dati

Il primo passo nello sviluppo del chatbot ha riguardato la definizione delle modalità di processamento dei dati esterni. L'obiettivo consiste nella realizzazione dello stadio iniziale di una pipeline RAG, volto a estrarre testi e immagini dalle fonti, convertirli in *embedding* e indicizzarli all'interno di un database vettoriale, al fine di fornire al chatbot la base di conoscenza specifica su cui basare le proprie risposte.

L'azienda, infatti, fornisce ai propri clienti un insieme di manuali utente, organizzati per applicativi, contenenti tutte le informazioni necessarie all'utilizzo del relativo software. Il numero di manuali associati a ciascuno di essi varia sensibilmente: in alcuni casi è presente un unico documento, mentre in altri si arriva a superare la ventina.

Le sezioni successive descrivono nel dettaglio le diverse soluzioni implementative, soggette a revisioni e ottimizzazioni successive e quindi presenti in diverse versioni; si tratterà inizialmente la gestione dei contenuti testuali, mentre l'elaborazione delle immagini sarà affrontata in una sezione dedicata.

Dal punto di vista architetturale, si è scelto di preservare la suddivisione per applicativi, trattando ciascuno di essi come una categoria indipendente. Questa decisione risponde a esigenze sia architetturali sia operative: riflette la modalità di intregrazione del chatbot all'interno dei singoli software azien-

dali, in cui ogni istanza del sistema dovrà fornire risposte esclusivamente rilevanti per il contesto funzionale in cui è inserito. Inoltre, la categorizzazione previene la manifestazione di confusione semantica durante la fase di *retrieval*, riducendo il rischio che il modello recuperi contenuti provenienti da manuali non pertinenti rispetto alla query dell'utente. In prospettiva computazionale, poiché l'indice interrogato risulta più compatto e specializzato, la separazione permette di ridurre i costi di indicizzazione e i tempi di interrogazione, migliorando le performance complessive della pipeline RAG.

4.1 Estrazione e Chunking

La pipeline di estrazione delle informazioni dai documenti si articola in una serie di fasi progettate per garantire la coerenza e la qualità del contenuto.

Il testo viene estratto direttamente dai manuali e ripulito da elementi ridondanti o superflui, come intestazioni, piè di pagina o formattazioni non rilevanti, in modo da ottenere contenuti puliti e leggibili. A seguire, esso viene segmentato in unità di dimensioni fisse o variabili, i cosiddetti *chunk*, ciascuno dei quali rappresenta una porzione del documento; ad ognuno di essi vengono associati metadati descrittivi che ne facilitano la gestione e la ricerca a livello di database durante la fase di *retrieval*.

4.1.1 Prima Versione

I manuali utente erano inizialmente disponibili esclusivamente in formato PDF, caratterizzati da una struttura eterogenea che includeva non solo testo continuo, ma anche tabelle, elenchi e numerose immagini incorporate. Per ciascun documento all'interno di ogni categoria applicativa veniva estratto l'intero contenuto testuale, incluse intestazioni, piè di pagina e altri elementi accessori, tramite il componente *PyPDFLoader* fornito da LangChain. Una

funzione dedicata basata su espressioni regolari provvedeva alla rimozione del materiale superfluo dopo l'estrazione, con l'obiettivo di normalizzare il testo e preservare solo le informazioni effettivamente rilevanti.

La prima strategia di segmentazione del contenuto adottata si basava sul numero di caratteri da inserire nel singolo *chunk*, senza considerare la struttura logica del documento (sezioni, paragrafi, elenchi); questa scelta derivava dalla disponibilità di un componente predefinito all'interno di LangChain, il *RecursiveCharacterTextSplitter*, che permette di segmentare ricorsivamente il testo utilizzando un set predefinito di separatori, fino a ottenere frammenti della dimensione desiderata. Dopo una serie di test preliminari, la dimensione dei *chunk* è stata fissata a 800 caratteri, con un *overlap* di 300 caratteri tra segmenti consecutivi, valori ritenuti in grado di bilanciare granularità e continuità informativa.

A ogni *chunk* venivano associati dei metadati essenziali, quali categoria, titolo del manuale e numero di pagina di origine; nel caso in cui un frammento si estendesse su più pagine, veniva registrato unicamente il numero della prima, per evitare ridondanze.

Questo approccio si è rivelato di semplice implementazione, ma sono presto emerse limitazioni significative: il criterio puramente quantitativo ignorava la struttura logica originaria, e portava a spezzare arbitrariamente paragrafi e unità semantiche, compromettendo la continuità del contenuto.

L'*overlap* tentava di mitigare il problema replicando una porzione del testo in *chunk* adiacenti, ma la soluzione si è dimostrata inefficiente: i *chunk* così ottenuti risultavano incoerenti, ripetitivi o non rappresentativi del contenuto, generando *embedding* poco efficaci. Le risposte prodotte dal chatbot su questa base erano spesso incomplete, prive del contesto necessario o, nei casi peggiori, affette da allucinazioni informative.

Queste criticità hanno reso evidente la necessità di esplorare tecniche di segmentazione più strutturate e consapevoli della semantica del documento,

al fine di creare segmenti completi e coerenti.

4.1.2 Semantic Chunking

La soluzione adottata ai problemi precedentemente evidenziati è l'impiego di una diversa tecnica di *chunking*, nota come *chunking semantico*.

Esso mira a suddividere il testo in sezioni di lunghezza variabile sulla base del significato delle frasi e della struttura logica del documento, preservando nel modo più fedele possibile il contesto originale . A differenza del *chunking* basato sul numero di caratteri, l'approccio semantico sfrutta la distribuzione naturale di paragrafi, sezioni ed elementi strutturali per generare frammenti coerenti e rappresentativi; ciò consente ai modelli di catturare meglio le relazioni tra concetti tecnici all'interno della base di conoscenza.

Per implementare questo meccanismo sono disponibili diverse librerie open-source, tra cui *unstructured* e *llm-sherpa*.

La libreria *unstructured*, sviluppata da Unstructured-IO, è una soluzione open-source orientata alla pre-elaborazione di dati non strutturati all'interno di pipeline per applicazioni AI e di machine learning. Supporta l'estrazione di testo e metadati da numerosi formati (PDF, DOCX, HTML, PPTX, ecc.) ed è particolarmente versatile, consentendo di adattare la preparazione dei dati alle esigenze specifiche di etichettatura, addestramento o deployment [64].

Llm-sherpa, invece, è progettata specificamente per l'implementazione di pipeline basate su modelli LLM. Offre API per l'estrazione e la gestione di contenuti provenienti da diversi formati documentali (PDF, DOCX, PPTX, HTML, TXT, XML) e presenta una caratteristica distintiva: la possibilità di associare informazioni spaziali ai blocchi testuali tramite la proprietà *bbox*, utile in applicazioni che richiedono la localizzazione precisa degli elementi nel documento. Il backend è stato recentemente rilasciato come open-source sot-

to licenza Apache 2.0, permettendo l'esecuzione locale tramite un'immagine Docker ufficiale [49].

Entrambe le librerie sono state valutate su un campione ridotto di documenti per confrontarne prestazioni ed efficacia nel contesto del progetto. *Unstructured* tendeva a estrarre tutto il contenuto del PDF, incluse intestazioni e piè di pagina, e generava *chunk* molto brevi, spesso poche centinaia di caratteri, frammentando interi paragrafi. *Llm-sherpa*, al contrario, permette di aggregare paragrafi o sezioni complete in un unico *chunk* tramite l'opzione *sections*, preservando così la coesione semantica e eliminando automaticamente elementi ricorrenti come header e footer.

Sulla base di queste osservazioni, è stato scelto di adottare *llm-sherpa* come strumento principale per la segmentazione semantica. Grazie a ciò, i *chunk* risultanti preservano integralmente il significato originale e migliorano la qualità degli *embedding* e del *retrieval*, riducendo fenomeni di perdita di contesto e minimizzando la probabilità di risposte incomplete o fuorvianti.

4.1.3 Migliorie Successive

Nel corso del progetto sono stati resi disponibili gli stessi documenti in formato DOCX, su cui sono state condotte ulteriori prove di estrazione. A differenza dei PDF, i file DOCX consistono in archivi compressi in formato ZIP contenenti dati strutturati in XML, il che permette di sfruttare i tag del linguaggio per analizzare con maggiore precisione la struttura del documento e generare *chunk* che rispettino fedelmente le suddivisioni originali.

Poiché non era possibile garantire che in futuro tutti i documenti sarebbero stati esclusivamente di un tipo, si è mantenuta una gestione parallela per entrambi i formati. Per ciascun *chunk* viene quindi aggiunto un metadato indicante il formato del documento di origine, garantendo la possibilità di operare eventuali filtri o analisi differenziate.

Un'ulteriore modifica significativa riguarda l'eliminazione degli indici dai documenti. Dalle analisi preliminari svolte dagli esperti di dominio, è emerso che i *chunk* contenenti l'indice, pur venendo frequentemente recuperati dal modulo di *retrieval*, non fornivano alcuna informazione concreta sugli argomenti di interesse; al contrario, la loro presenza riduceva la quantità di dati rilevanti disponibili per il modello e rischiava di produrre risposte incomplete o poco accurate. Per risolvere questo problema, sono state implementate due funzioni distinte, una per ciascun formato di file, in grado di identificare e rimuovere le sezioni di indice prima dell'estrazione del testo.

Per i PDF, la funzione individua e rimuove le pagine contenenti l'indice, basandosi su due criteri: la presenza della parola "Indice" e la corrispondenza di più righe consecutive con la formattazione tipica di un indice, il titolo del paragrafo seguito da dei puntini e dal numero della pagina corrispondente, rilevata mediante espressioni regolari. Per evitare di eliminare qualunque pagina contenente la parola "Indice" all'interno del testo, è stata introdotta una logica di controllo per considerare solo la prima occorrenza in ogni documento.

Nei documenti DOCX, la rimozione dell'indice risulta più complessa, in quanto le pagine non sono entità fisiche come nei PDF: in questo caso, si sfrutta la struttura XML per eliminare le righe contrassegnate dall'apposito tag come parte dell'indice, senza rimuovere fisicamente la pagina.

Entrambe le funzioni restituiscono un nuovo documento modificato, salvato separatamente nel file system. Questo approccio consente di preservare i file originali, utili per aggiornamenti futuri, e assicura che ogni volta che un documento viene modificato o aggiornato, la pipeline di estrazione possa essere riavviata senza compromettere l'integrità dei dati.

4.2 Embedding

Completata la fase di estrazione, i *chunk* ottenuti devono essere convertiti in *embedding* prima di poter essere salvati su un database vettoriale.

La scelta del modello di *embedding* è un passo fondamentale, poichè determina la capacità del sistema di preservare il significato semantico del testo durante la trasformazione in rappresentazione numerica.

4.2.1 Modelli

Il paragrafo seguente presentata una lista dei modelli di *embedding* testati nel contesto del progetto, ciascuno caratterizzato da specifiche proprietà in termini di dimensionalità vettoriale, supporto linguistico e ambiti applicativi.

Per dimensionalità di un modello di *embedding* si intende il numero di dimensioni dello spazio vettoriale utilizzato per rappresentare i dati. Una dimensionalità elevata consente al modello di catturare relazioni e schemi più complessi, con potenziali benefici in termini di accuratezza nella ricerca semantica e nella discriminazione tra concetti simili; tuttavia, comporta anche un incremento dei costi computazionali e dello spazio di archiviazione richiesto. Al contrario, modelli con dimensionalità ridotta risultano più leggeri ed efficienti, ma possono perdere alcune sfumature semantiche e risultare meno efficaci nel rappresentare relazioni più articolate tra dati [46].

I modelli esaminati sono i seguenti:

- **paraphrase-multilingual-MiniLM-L12-v2**: modello della famiglia *sentence-transformers* con una dimensionalità di 384. Supporta più lingue (incluso l'italiano) ed è stato addestrato per generare *embedding* simili per frasi con lo stesso significato. È leggero e veloce, adatto a sistemi multilingua e applicazioni in tempo reale.

- **all-MiniLM-L12-v2**: modello della famiglia *sentence-transformers* con una dimensionalità di 384. È uno dei modelli più diffusi per la lingua inglese, ma può funzionare discretamente anche su testi in altre lingue, ed è apprezzato per il buon equilibrio tra prestazioni e velocità di calcolo.

- **nomic-embed-text-v1.5**: modello sviluppato dal gruppo Ollama (Meta) con una dimensionalità di 768. È progettato per ottenere *embedding* generici e versatili, adatti a una vasta gamma di testi. Offre buone performance su compiti di *clustering*, classificazione e ricerca semantica.

- **bge-base-it-v1.5**: modello della famiglia *sentence-transformers* con una dimensionalità di 768, addestrato specificamente per la lingua italiana. È particolarmente efficace nell'elaborazione di testi tecnici, come documentazione o manuali.

- **bert-base-nli-mean-tokens**: modello della famiglia *sentence-transformers* con una dimensionalità di 768, basato su BERT. È indicato per la ricerca semantica, grazie alla sua capacità di cogliere relazioni logiche e di significato tra frasi.

- **multilingual-e5-large**: modello sviluppato da *intfloat* con una dimensionalità di 1024. Non è specificamente addestrato per l'italiano, ma supporta più lingue e ha mostrato buone performance. È adatto per applicazioni di ricerca semantica su larga scala.

4.2.2 Test

I modelli sono stati confrontati utilizzando lo stesso test di *benchmark*, composto da dieci domande relative a cinque applicativi differenti: per ciascuno di essi è stata formulata una domanda generica e una più specifica,

entrambe valutate in modalità *one-shot* mediante il medesimo LLM (Llama3.2). Le risposte generate sono state analizzate considerando due aspetti principali: la correttezza e completezza dei contenuti, e la presenza del *chunk* corretto tra quelli recuperati dal sistema di *retrieval*, dove per *chunk* corretto si intende il frammento del documento che contiene l'informazione necessaria a formulare la risposta.

È importante sottolineare che la valutazione qualitativa delle risposte non può essere considerata completamente oggettiva: essa dipende sia dal comportamento combinato di LLM e *retriever*, sia dal giudizio soggettivo dell'osservatore che verifica la pertinenza e l'esattezza del contenuto, che nel caso corrente sono gli esperti del dominio. Di conseguenza, pur rappresentando un'indicazione utile, la metrica non può essere interpretata come una misura assoluta o universalmente valida delle prestazioni del modello.

Un ulteriore limite della metodologia adottata riguarda il ricorso a test in modalità *one-shot*, che valutano le prestazioni sulla base di un singolo tentativo per domanda. Sebbene ciò renda il processo più rapido e ripetibile, introduce variabilità nei risultati, poiché modelli come Llama3.2 possono generare risposte leggermente diverse a parità di input, influenzando la percezione della qualità complessiva.

La tabella 4.2.2 riporta i risultati ottenuti. Sulla base delle evidenze raccolte, si è scelto di procedere con il modello che ha ottenuto il punteggio complessivo migliore, *multilingual-e5-large*, il quale presenta anche la dimensionalità vettoriale più elevata tra quelli testati.

Si può ossere una relazione diretta tra dimensionalità del modello di *embedding* e performance rilevate: i modelli con vettori di dimensione maggiore hanno fornito rappresentazioni semantiche più ricche e mostrano una migliore capacità di distinzione tra concetti simili, soprattutto nelle domande specifiche. Ciò suggerisce che, nell'ambito dei manuali tecnici analizzati, la maggiore complessità dello spazio vettoriale contribuisce positivamente al processo

Modello	Punteggio Chunk	Qualità Risposta
paraphrase-multilingual-MiniLM-L12-v2	5/10	Restituisce anche il ragionamento fatto dal modello oltre che i dati.
all-MiniLM-L12-v2	5.5/10	Molte risposte sbagliate.
nomic-embed-text-v1.5	7/10	Non coglie la differenza tra un applicativo (Firma) e un oggetto generico (firma).
bge-base-it-v1.5	8/10	Risposte molto sintetiche, piene di errori grammaticali.
bert-base-nli-mean-tokens	7/10	Risposte precise e dettagliate, ma non sempre corrette.
multilingual-e5-large	9/10	Risposte concise ma corrette.

Tabella 4.1: Confronto tra modelli di *embedding*.

di *retrieval*, giustificando la scelta finale del modello *multilingual-e5-large*.

4.3 Database

Una volta completata la fase di generazione degli *embedding*, è necessario memorizzare i dati ottenuti in strutture dedicate, poiché la gestione efficiente di tali rappresentazioni vettoriali costituisce un elemento centrale all'interno di una pipeline RAG.

I database vettoriali sono progettati per supportare nativamente operazioni tra vettori: a differenza dei database tradizionali, ottimizzati per dati strutturati e query relazionali, essi consentono di memorizzare direttamente vettori numerici e di eseguire ricerche basate sulla distanza semantica tra essi.

Nel contesto di questo progetto, i database vettoriali svolgono un ruolo essenziale: permettono di archiviare e recuperare rapidamente i *chunk* più simili alla query dell'utente, fornendo al modello linguistico il contesto specifico necessario per generare risposte precise e fondate sui documenti originali. La scelta del database vettoriale e delle tecnologie di supporto associate incide direttamente sulla qualità e sulle prestazioni dell'intero sistema.

4.3.1 ChromaDB

ChromaDB è un database vettoriale open-source progettato per applicazioni di intelligenza artificiale. Supporta nativamente le principali operazioni tipiche dei sistemi RAG, come la creazione e la persistenza degli *embedding*, la ricerca vettoriale e il *retrieval* multimodale [8].

Uno dei motivi che ne ha motivato l'adozione iniziale è la totale integrazione con LangChain, che consentiva di utilizzarlo senza necessità di credenziali o configurazioni complesse; è sempre stato eseguito in locale, utilizzando una directory del file system come sistema di persistenza.

LangChain fornisce una classe *wrapper* che semplifica l'interazione con il database: il costruttore *Chroma* crea un riferimento a un *vector_store*, consendo di specificare il nome della collezione, il modello di *embedding* da utilizzare e il percorso di archiviazione dei dati. In maniera speculare all'organizzazione dei documenti in fase di estrazione, ogni applicativo aziendale è stato trattato come una categoria indipendente, con una collezione dedicata in cui memorizzare *embedding* e metadati.

ChromaDB presenta diversi vantaggi: l'integrazione diretta con LangChain ne facilita l'adozione nelle prime fasi di sviluppo, mentre il modello completamente open-source garantisce trasparenza e libertà d'uso; inoltre, la sua leggerezza lo rende adatto a essere eseguito in ambienti locali o di test senza costi aggiuntivi.

Tuttavia, proprio queste caratteristiche ne limitano la possibilità di impiego in produzione. L'essere open-source implica l'assenza di garanzie di sicurezza avanzate o certificazioni, rendendo il sistema potenzialmente vulnerabile in scenari con requisiti elevati di protezione dei dati. Inoltre, l'astrazione fornita da LangChain nasconde molti dettagli implementativi: non permette di personalizzare la struttura del database, il formato di salvataggio o l'estensione dei metadati, limitando la flessibilità necessaria in un contesto professionale. A ciò si aggiungono le prestazioni non ottimali: ChromaDB non è progettato per gestire volumi elevati di query o carichi intensivi, come quelli previsti a pieno regime nel sistema finale.

Per questi motivi, pur essendo una soluzione ideale per prototipazione e sperimentazione, ChromaDB non è risultato adeguato per le esigenze della fase produttiva del progetto.

4.3.2 PostgreSQL e pgVector

PostgreSQL è un DBMS a oggetti basato sull'estensione del linguaggio SQL, che si distingue per robustezza, affidabilità e capacità di gestire carichi di lavoro complessi. Offre funzionalità avanzate per la sicurezza e l'integrità dei dati, tra cui autenticazione a più fattori, controllo granulare degli accessi e supporto nativo alla replica e alla scalabilità, caratteristiche che lo rendono particolarmente adatto a un contesto produttivo con requisiti stringenti, come quello previsto da questo progetto [54].

Per estendere PostgreSQL alla gestione di dati vettoriali è stata utilizzata l'estensione open-source *pgvector*, progettata per memorizzare vettori direttamente all'interno del database e per eseguire le principali operazioni vettoriali, come il calcolo della distaza vettoriale e similarità, essenziali per la ricerca semantica [53]. L'integrazione tra PostgreSQL e pgvector consente quindi di disporre di un database relazionale tradizionale che supporta, al contempo, funzionalità tipiche dei database vettoriali moderni.

L'introduzione di PostgreSQL è anche una delle ragioni del passaggio da LangChain a LlamaIndex. Il primo framework presentava le stesse limitazioni riscontrate con ChromaDB, in particolare una scarsa trasparenza nei meccanismi di persistenza e un livello di astrazione troppo elevato, che ostacolava la personalizzazione necessaria. LlamaIndex, al contrario, offre un controllo più fine su ogni componente della pipeline e una maggiore flessibilità nell'integrazione con database esterni.

Nonostante il cambiamento architetturale, l'organizzazione logica dei dati è rimasta invariata: si mantiene la suddivisione in tabelle per applicativo, continuando a utilizzare lo stesso modello di *embedding*. Il database vettoriale viene creato tramite la classe *PGVectorStore* di LlamaIndex, che richiede credenziali esplicite di accesso (username e password) per connettersi a PostgreSQL. Per ogni tabella destinata al salvataggio degli *embedding* viene istanziato un *vector_store* dedicato, nel quale è possibile memorizzare anche metadati personalizzati, rispondendo così alle esigenze di flessibilità del progetto.

Tramite l'interfaccia grafica pgAdmin è possibile visualizzare in modo diretto gli *embedding*, i metadati associati e il testo originale memorizzato nel database. Ciò ha facilitato l'analisi dei risultati del *chunking* semantico, consentendo di verificare immediatamente come ciascun documento fosse stato segmentato senza dover produrre output a terminale.

In conclusione, l'adozione combinata di PostgreSQL e pgvector ha permesso di superare i limiti emersi nelle fasi preliminari e di ottenere un sistema più robusto, personalizzabile e adatto alla produzione. La possibilità di ispezionare direttamente struttura, insieme a un controllo più fine sui meccanismi di salvataggio, ha garantito una maggiore trasparenza operativa, mentre le funzionalità avanzate del DBMS offrono una base solida per sostenere i carichi di lavoro previsti dal sistema finale. Complessivamente, questa nuova architettura ha migliorato sia la qualità del *retrieval* sia la gestibilità dell'intera

pipeline, consolidando le fondamenta tecniche del progetto.

4.4 Immagini

I manuali contengono numerose immagini, prevalentemente icone e *screenshot* degli applicativi, utilizzate per chiarire le spiegazioni testuali mediante esempi concreti: le icone indicano pulsanti o azioni, mentre gli *screenshot* mostrano come eseguire operazioni specifiche passo-passo. Questi elementi grafici rappresentano parti integranti del discorso e vengono richiamati attivamente all'interno dei testi, contribuendo alla comprensione dei contenuti.

La gestione delle immagini si è rivelata particolarmente complessa: nessuno dei modelli linguistici impiegati supporta nativamente dati multimodali, e quindi non può elaborare direttamente contenuti visivi, anche se questi possono essere memorizzati all'interno dei database vettoriali. Tuttavia, le immagini sono state ritenute essenziali per arricchire la base di conoscenza del chatbot e migliorare la qualità delle risposte. Determinare un metodo efficace per estrarre informazioni utili da questi elementi grafici ha richiesto l'analisi di possibili strategie di preelaborazione e di rappresentazione dei dati visivi in forma compatibile con la pipeline testuale.

4.4.1 Prima Versione

L'estrazione massiva delle immagini dai documenti ha generato problemi pratici, dovuti allo standard aziendale di formattazione dei manuali: ogni piè di pagina, infatti, riportava il logo della società in formato immagine, il quale veniva quindi estratto e salvato una volta per pagina per ogni singolo manuale. Oltre ad aumentare notevolmente i tempi di estrazione e il carico computazionale, essendo impossibile distinguere automaticamente tra conte-

nuti rilevanti e superflui il sistema era costretto a memorizzare centinaia di immagini che non apportavano alcuna informazione utile.

LangChain non offriva metodi di preselezione durante l'estrazione, quindi la soluzione adottata è stata definire un'area "di validità" all'interno di ciascuna pagina, includendo solo la zona centrale ed escludendo intestazioni e piè di pagina: le immagini vengono salvate solo se le coordinate della loro posizione rientrano in quest'area specifica, mentre in caso contrario sono ignorate. Questo approccio, sebbene non ottimale in termini di efficienza, ha permesso di eliminare i loghi e altri elementi non informativi, preservando le immagini rilevanti per la conoscenza del chatbot.

4.4.2 Generazione Descrizione

Una volta salvate, le immagini devono essere accompagnate da una descrizione testuale che ne consenta l'indicizzazione e l'utilizzo all'interno della base di conoscenza. A questo scopo è stato introdotto un *Vision Language Model* (VLM), in particolare BLIP (*Bootstrapped Language-Image Pretraining*).

Un VLM è una rete neurale addestrata su coppie immagine-testo, con l'obiettivo di apprendere uno spazio semantico condiviso tra le due modalità. Questi modelli sono progettati per comprendere il contenuto visivo di un'immagine e generare automaticamente una descrizione coerente in linguaggio naturale, rendendo possibile l'integrazione delle informazioni visive nelle pipeline testuali [60].

BLIP è un VLM open-source sviluppato da Salesforce e introdotto nel 2022, pensato per combinare in modo efficace la comprensione visiva con la generazione di linguaggio naturale, mantenendo al contempo un'architettura efficiente e flessibile [60]. È disponibile tramite la libreria *transformers* di Huggingface e può essere utilizzato sia per la generazione automatica di didascalie sia per compiti di ricerca multimodale.

Le prove iniziali sono state effettuate utilizzando il modello base pre-addestrato, senza alcun *finetuning* specifico sul dominio dei manuali aziendali; al modello è stato chiesto di generare descrizioni per icone generiche, come il classico ingranaggio delle impostazioni o le tre lineette orizzontali che rappresentano il menù.

I risultati si sono rivelati insoddisfacenti: BLIP tendeva a interpretare entrambe le icone come "nuovi loghi" o "icone di siti web", nonostante siano simboli standard facilmente riconoscibili e largamente utilizzati. Le performance sulle immagini specifiche del dominio sono state ancora peggiori, come era prevedibile considerando l'uso del modello base.

Addestrare BLIP per riconoscere in modo affidabile le immagini specifiche dei manuali avrebbe richiesto un notevole impegno, con la necessità di definire standard di descrizione e fornire un gran numero di esempi annotati dagli esperti di dominio, senza alcuna garanzia di miglioramento immediato delle performance. Poiché l'obiettivo principale del progetto non era sviluppare un modello multimodale personalizzato, si è deciso di esplorare soluzioni alternative meno complesse ed onerose, come l'uso di descrizioni predefinite.

4.4.3 Estrazione Testo Alternativo

Le descrizioni generate automaticamente non sono state ritenute sufficienti dagli esperti di dominio, che hanno quindi deciso di fornire manualmente un testo alternativo per ciascuna immagine, direttamente all'interno dei documenti DOCX. Per estrarre queste informazioni è stata utilizzata la libreria *python-docx*.

A seguito di questa modifica progettuale, non è più necessario salvare separatamente i file immagine nel file system, poiché la loro utilità per la pipeline è ormai limitata. La difficoltà principale risiede ora nel collocare correttamente la descrizione all'interno del paragrafo di riferimento, in modo

da mantenere la continuità semantica e rendere l'informazione significativa all'interno del contesto testuale.

Sfruttando i tag XML dei file DOCX, per ciascuna immagine classificata come "elemento grafico" (escludendo quindi loghi e altri elementi non informativi) viene estratto il testo alternativo e il relativo contesto, identificato come il paragrafo immediatamente precedente; nel caso in cui il testo alternativo non sia presente, viene registrato come "Senza descrizione". Queste informazioni vengono salvate temporaneamente in un dizionario Python.

Il contesto dell'immagine viene poi confrontato con il testo estratto tramite *llm-sherpa*, suddiviso in singole frasi: quando la frase finale di un paragrafo e il contesto dell'immagine corrispondono, la descrizione dell'immagine viene inserita direttamente in coda alla frase stessa, e il processo prosegue con le immagini successive.

Un caso particolare si verifica quando più immagini sono inserite in sequenza, e quindi condividono lo stesso contesto: ciascuna di esse ha una descrizione diversa, e l'inserimento della prima in coda al paragrafo modifica la frase finale, impedendo alle immagini successive di trovare la corrispondenza corretta. La soluzione adottata consiste nel pre-processare le descrizioni, concatenando quelle che condividono lo stesso contesto: questo approccio garantisce che tutte le descrizioni vengano inserite nei punti corretti, mantenendo la coerenza semantica del testo e rendendo l'informazione visiva pienamente dispoibili nella base di conoscenza del chatbot.

Una volta integrate correttamente nel testo, le descrizioni delle immagini diventano pienamente sfruttabili nella pipeline RAG: ogni descrizione viene trasformata in *embedding* e salvata nel database vettoriale insieme al testo circostante, in modo che, durante il processo di *retrieval*, il sistema possa considerare tutti i contenuti presenti nei manuali, fornendo risposte più complete.

4.5 Normativa

Oltre ai manuali tecnici sul funzionamento dei vari applicativi, è stato imposto, in un secondo momento, di inserire tra la base di conoscenza anche alcuni documenti di normativa, sia di carattere generale, sia specifica per un singolo applicativo. I documenti di carattere generale, tra cui il testo del GDPR Europeo e di alcune leggi particolarmente rilevanti per la Pubblica Amministrazione, sono da replicare per ogni applicativo, mentre i documenti specifici sono da inserire solo nelle categorie di riferimento.

L'aggiunta di questi dati, in volume largamente superiore rispetto alla manualistica, ha impatti rilevanti e prettemente negativi, sulla qualità dei dati recuperati in fase di *retrieval*, portando allo sviluppo di due approcci di riorganizzazione della base di conoscenza all'interno del database.

Nel primo caso, tutti i dati sono salvati in un'unica tabella per categoria, e un nuovo metadato identifica l'area di appartenenza del singolo *chunk*: "manualistica" per quelli estratti dai manuali tecnici, "normativa" per quelli derivanti dai documenti legislativi. Nel secondo caso, per ogni categoria sono create due tabelle distinte e separate, una per la manualistica, una per la normativa, senza necessità di modificare i metadati.

Entrambi gli approcci presentano degli svantaggi e complicano notevolmente la fase di recupero dei *chunk*. Nel caso in cui si abbia un'unica tabella, la maggior quantità di dati ha un impatto negativo sulla qualità dei *chunk* recuperati, specie in risposta a domande molto generiche che trovano riscontro sia nella manualistica che nella normativa; nonostante l'impiego dei metadati, non è possibile mantenere una separazione netta tra i due ambiti, il che porta all'estrazione di informazioni da entrambi e alla conseguente generazione di risposte confuse e non totalmente pertinenti.

Nel caso siano presenti due tabelle, le complicazioni nascono nella fase ancora precedente: dato che il *retrieval* può essere eseguito concretamente

su una sola tabella per volta, è necessario classificare preventivamente ogni query ricevuta, o come "manualistica" o come "normativa", in modo da indirizzare il recupero dei dati verso la tabella appropriata. È il sistema stesso a dover eseguire la classificazione; di conseguenza, necessita di meccanismi e di conoscenza sufficiente per poter svolgere questa operazione in maniera accurata.

A seguito di alcuni test su un campione ristretto di documenti e *chunk*, è risultato che, se la classificazione dell'ambito risulta corretta, le risposte generate applicando la seconda implementazione sono più precise e corrette. In concomitanza con i suggerimenti degli esperti di dominio, si è deciso di proseguire con la seconda possibilità. Il capitolo 12 analizza in dettaglio l'implementazione del meccanismo di classificazione e le conseguenze che questo ha sull'intero sistema.

Capitolo 5

Chatbot Business Logic

Il capitolo seguente descrive in modo dettagliato il funzionamento della logica di business del chatbot, in particolare della fase finale della pipeline RAG, illustrando il percorso che una richiesta compie dal momento in cui viene ricevuta fino alla generazione della risposta da parte del modello; saranno analizzate fasi come l'interpretazione della query, il recupero dei *chunk* rilevanti, la costruzione del contesto e l'interazione con il modello linguistico di grandi dimensioni.

Nel capitolo verranno introdotti solo i moduli fondamentali per l'implementazione di base, necessari a garantire il funzionamento minimo del sistema. La versione finale del progetto include ulteriori componenti sviluppati nel tempo, che verranno affrontati in capitoli dedicati per evidenziare l'evoluzione graduale dell'architettura e motivare le scelte progettuali adottate.

5.1 Chatbot

La classe *Chatbot* costituisce il nucleo centrale della logica applicativa del progetto. Essa inizializza un'istanza del modello LLM, riceve le query utente dal server, coordina le operazioni necessarie alla loro elaborazione e restituisce la risposta generata dal modello.

Durante lo sviluppo sono stati testati diversi LLM, ciascuno con configurazioni e requisiti specifici, rendendo necessaria un'architettura che consentisse di passare rapidamente da un modello all'altro senza modificare il codice sorgente. Per questo motivo, *Chatbot* è stata progettata come classe base, responsabile esclusivamente della logica generale di elaborazione delle query, senza l'inizializzazione diretta di alcun modello linguistico; i singoli LLM sono implementati come classi derivate che ereditano i metodi di elaborazione della classe base e si occupano dell'inizializzazione del modello corrispondente tramite i parametri richiesti (vedi figura 5.1).

Questo approccio garantisce indipendenza tra logica applicativa e scelta del modello, oltre alla possibilità di sostituire o introdurre nuovi modelli in maniera trasparente, senza modificare il codice centrale.

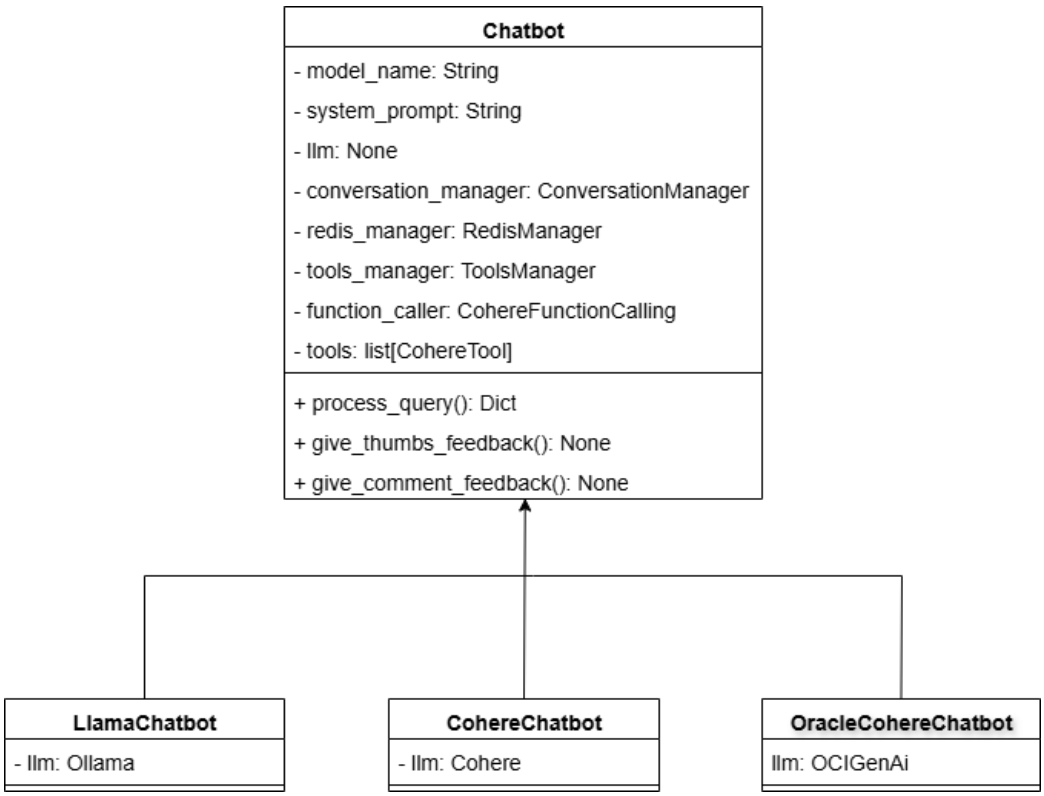

Figura 5.1: Diagramma delle classi per *Chatbot*.

Per rendere il sistema ulteriormente modulare, è stata introdotta una funzione *factory* che, a partire dal nome del modello specificato in un file di configurazione, istanzia la classe corrispondente a tempo di esecuzione, essendo invocata all'interno del modulo *api_module* durante la fase di inizializzazione del server. Grazie ad essa, il codice risulta meno rigido e più adattabile ai cambiamenti: è possibile automatizzare la selezione del modello senza inserire condizioni *hardcoded* e semplificare le operazioni di testing, permettendo di alternare rapidamente modelli diversi.

Poiché la logica di gestione delle query è complessa e articolata in molteplici passaggi, essa non è completamente contenuta nella classe *Chatbot*: in accordo con il principio di Singola Responsabilità, è stata suddivisa in componenti dedicati, ognuno incaricato di una porzione specifica del flusso di lavoro. Ciò contribuisce a rendere il codice più semplice da estendere, ottimizzare e testare, in quanto i singoli moduli possono essere esaminati autonomamente e separatamente. Due di questi componenti, *RagExecutor* e *Retriever*, vengono descritti nelle sezioni successive; essi sono istanziati direttamente dentro *Chatbot*, da cui dipendono per i parametri di inizializzazione.

Oltre alla gestione delle query, *Chatbot* si occupa anche della registrazione dei feedback e dei commenti degli utenti: le informazioni inoltrate tramite il server vengono salvate in apposite tabelle all'interno dello stesso database utilizzato per il RAG, al fine di monitorare nel tempo l'evoluzione delle performance del sistema e rilevare miglioramenti o regressioni dovuti a modifiche del codice o dei modelli. La gestione di questo aspetto è delegata alla classe *conversation_manager*, analizzata in dettaglio nel capitolo 8.

5.2 Retriver

La classe *Retriever* racchiude la logica necessaria per recuperare dal database le informazioni rilevanti per una specifica query utente, al fine di

generare il contesto testuale da fornire al modello. Questo componente costituisce l'intera seconda fase della pipeline RAG ed è uno degli elementi più critici del sistema, poiché la qualità dei risultati prodotti dal chatbot dipende direttamente dalla pertinenza e dalla coerenza dei *chunk* restituiti.

Durante l'inizializzazione, *Retriever* carica dal file di configurazione tutti i parametri necessari: credenziali e indirizzo del database PostgreSQL, nome del modello di *embedding* da utilizzare e lista delle categorie dei documenti; sulla base di queste informazioni, vengono istanziati i relativi *PGVectorStore* tramite LlamaIndex, uno per categoria, e salvati in un apposito dizionario. Questa scelta consente un accesso immediato ai dati vettoriali senza dover ristabilire la connessione o ricostruire gli oggetti a ogni richiesta, riducendo in modo significativo la latenza della pipeline e distribuendo il costo computazionale su un'unica operazione eseguita all'avvio.

Quando il metodo di *retrieval* viene invocato, esso riceve due informazioni fondamentali: la query utente e la categoria a cui appartiene. Dopo aver selezionato dal dizionario il *PGVectorStore* corretto, la query viene trasformata nel relativo *embedding*, utilizzando lo stesso modello usato per indicizzare i documenti, per poi chiamare il metodo di ricerca vettoriale fornito da LlamaIndex, che restituisce i k *chunk* semanticamente più affini alla query. Nel sistema, si è stabilito $k=5$, un valore che bilancia la quantità di informazioni recuperate con il rumore semantico, mantenendo la risposta concentrata sugli elementi più pertinenti.

I *chunk* ottenuti non possono essere passati direttamente all'LLM: devono essere convertiti in *contesto* coerente, leggibile e strutturato. Per questo viene generato un blocco testuale formattato in modo uniforme e gerarchico:

Titolo del documento
Titolo della sezione
Testo

Questa schematizzazione non solo migliora la leggibilità del contenuto, ma soprattutto facilita l'elaborazione da parte del modello, che tende a ottenere prestazioni migliori quando riceve informazioni in forma chiaramente strutturata e ripetibile.

Il contesto ottenuto rappresenta l'output finale della fase di *retrieval* e viene restituito al componente successivo, *RagExecutor*, che si occupa della costruzione del prompt e dell'interazione con l'LLM. Nonostante le performance complessivamente soddisfacenti, l'utilizzo di una funzione di *retrieval* sviluppata da terzi introduce alcuni limiti funzionali, discussi in dettaglio nella sezione corrispondente del capitolo 8.

5.3 Rag Executor

La classe *RagExecutor* implementa la logica centrale per l'esecuzione delle operazioni RAG sui dati indicizzati, a partire dalle query utente. Essa funge da raccordo tra la gestione della conversazione, il recupero dei *chunk* rilevanti e la generazione finale della risposta da parte del modello, costituendo il punto di coordinamento dell'intera pipeline.

5.3.1 Conversazioni

Un primo compito fondamentale della classe è la gestione delle conversazioni utente.

Con il termine *conversazione* si intende uno scambio di messaggi strutturato e continuativo tra utente e chatbot, che include domande, risposte, contesto recuperato, categoria selezionata e, quando presenti, commenti o feedback. Ogni volta che un utente accede all'interfaccia grafica, il sistema genera un UUID univoco che identifica in modo esclusivo la conversazione in corso; l'UUID viene inviato al server, inoltrato a *Chatbot* tramite API e suc-

cessivamente passato ai metodi di *RagExecutor*, garantendo così la corretta associazione tra cliente e sessione.

Nel momento in cui si interroga l'LLM, è importante fornire al modello non solo la domanda corrente, ma l'intera conversazione accumulata fino a quel momento. Ciò consente al modello di disporre di una memoria contestuale più ampia, utile per produrre risposte coerenti e consistenti nel tempo e per interpretare correttamente eventuali riferimenti a passaggi precedenti.

Ogni conversazione è rappresentata come un vettore di messaggi, salvato all'interno di un dizionario Python dove la chiave è l'UUID e il valore è la lista completa dei messaggi scambiati. La coppia UUID-conversazione viene creata dopo la prima query dell'utente; nello stesso momento, nella struttura viene inserito il prompt di sistema, aggiunto una sola volta per conversazione e mantenuto come base fino al termine della stessa. A ogni successiva richiesta, il vettore associato viene recuperato e aggiornato con i nuovi messaggi, preservando uno storico coerente e lineare.

L'uso del dizionario consente di mantenere conversazioni multiple in parallelo e di garantire indipendenza e isolamento tra utenti diversi, requisito essenziale in un sistema destinato a un ambiente lavorativo multiutente; in un contesto di produzione, però, questa struttura dati non è una soluzione accettabile, in quanto non scalabile nè robusta in caso di un numero elevato di uteni. Il capitolo 9 tratta più in dettaglio questi limiti, fornendo anche un'implementazione alternativa migliorata.

I messaggi sono rappresentati tramite oggetti *ChatMessage*, una struttura dati fornita da LlamaIndex ottimizzata per l'interazione con modelli LLM: ogni messaggio contiene un *role* (user, assistant, system) e un *content* testuale. Questa rappresentazione standardizzata assicura che il modello riceva un input nel formato atteso e consente una gestione uniforme della conversazione su tutti i modelli supportati.

Nonostante la struttura risulti sufficientemente modulare, presenta alcu-

ne limitazioni intrinseche: poiché per ogni nuova query l'intera conversazione viene reinviata al modello, la dimensione del prompt cresce linearmente, comportando maggiore latenza e un aumento dei costi computazionali; inoltre, in caso di conversazioni molto lunghe, si rischia di superare il limite di contesto del modello.

5.3.2 Risposte

Una volta recuperata la conversazione, *RagExecutor* utilizza un'istanza di *Retriever* per svolgere le operazioni discusse in precedenza e ottenere le informazioni su cui basare la generazione della risposta. Il contesto rappresenta un messaggio a tutti gli effetti, separato da quello contenente la query e aggiunto al vettore come tutti gli altri.

A questo punto, l'intera struttura dati viene passata al modello come input, costituendo il contesto completo su cui basare la generazione della risposta; per questa operazione si utilizza il metodo *chat* fornito da LlamaIndex, che consente di interagire con diversi modelli LLM in modo uniforme, astraendo dalle implementazioni specifiche e semplificando l'integrazione o la sostituzione del modello sottostante.

Il risultato finale viene restituito al *Chatbot*, che a sua volta lo restituisce al server, che infine lo restituisce alla GUI per essere mostrato all'utente. L'immagine seguente mostra il diagramma di flusso dell'intera fase finale della pipeline RAG, dalla ricezione di una query all'invio della risposta generata, come descritta nelle sezione precedenti.

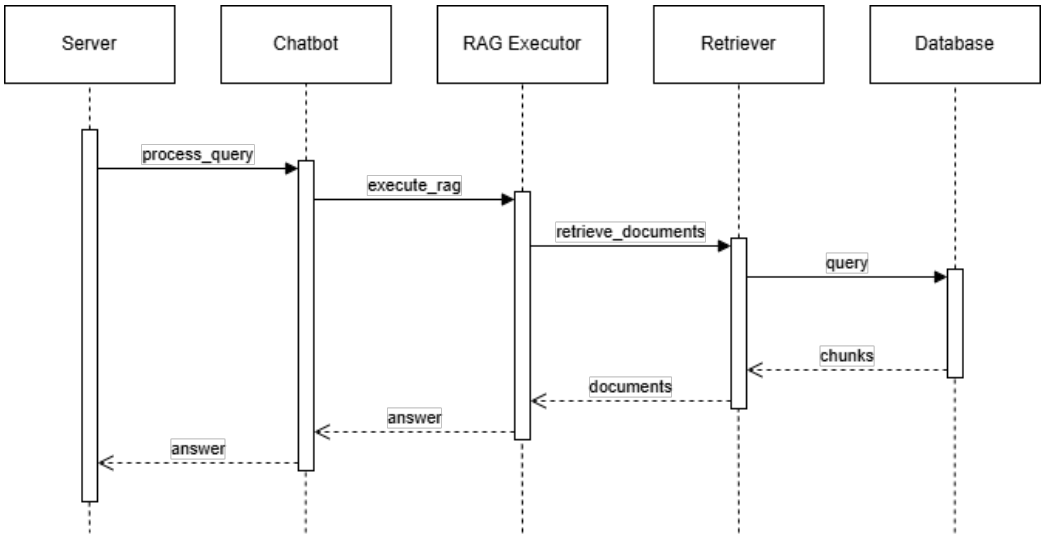

Figura 5.2: Diagramma di sequenza della fase finale della pipeline RAG.

Capitolo 6

LLM e Prompt

6.1 LLM

La scelta del modello linguistico di grandi dimensioni condiziona drasticamente la qualità dei risultati prodotti: a parità di dati iniziali, modelli diversi generano risposte differenti in base alle proprie capacità di elaborazione e comprensione linguistica. Per questo motivo, sperimentare e confrontare le diverse soluzioni disponibili sul mercato è stato un passaggio essenziale per il successo del progetto.

LlamaIndex ha notevolmente semplificato lo sviluppo: le classi e i costrutti offerti del framework garantiscono un livello di astrazione tale che la sostituzione dell'LLM richiede unicamente l'aggiunta della classe dedicata per l'inizializzazione, senza alcuna modifica al codice responsabile del processo di RAG.

Tutti i modelli testati hanno prodotto risultati i migliori con *temperatura* impostata a 0. La temperatura è un parametro che regola il grado di casualità nella generazione del testo durante l'inferenza: valori bassi privilegiano i token più probabili, mentre valori più alti aumentano la possibilità di selezionare token meno prevedibili, introducendo maggiore variabilità e

"creatività" nelle risposte [50]. Dai test condotti, è emerso che anche lievi incrementi rispetto allo zero producevano risposte incoerenti e allucinazioni, risultati non accettabili per un sistema orientato all'affidabilità.

Tra le caratteristiche degli LLM, due risultano particolarmente significative, la *context window* e la *context length*. La prima indica al numero di token che un modello può considerare nella generazione della risposta, includendo il contesto e la memoria delle interazioni precedenti; la seconda, invece, rappresenta il numero massimo di token che il modello può elaborare in una singola interazione o in un singolo messaggio [29]. Ad esempio, un modello con una *context window* di 128k token e una *context length* di 4.096 token è in grado di considerare un numero elevato di scambi conversazionali, ma ogni singola risposta generata non può superare il limite di 4.096.

6.1.1 Mistral

Mistral, sviluppato da Mistral AI, è un LLM open-source a 7B di parametri, pre-addestrato e ottimizzato per l'esecuzione di istruzioni; bilancia i costi dei modelli di grandi dimensioni con prestazioni efficienti e ridotti requisiti di memoria. L'impiego di un tokenizzatore BPE garantisce una gestione più efficace dei token, evitando che singoli caratteri vengano mappati su token non presenti nel vocabolario [20, 21].

Mistral è stato il primo modello impiegato come motore di generazione per il chatbot; è stato eseguito localmente tramite la piattaforma *Ollama*, che permette di scaricare ed eseguire modelli linguistici direttamente sul proprio dispositivo.

Pur non essendo stato addestrato specificamente sull'italiano, ha mostrato una buona padronanza della lingua e delle strutture grammaticali, sebbene in alcuni casi introducesse anglicismi non del tutto pertinenti al contesto. La *context window* limitata a 32k token si è però rivelata rivelata un vincolo significativo nella gestione della memoria conversazionale, compromettendo-

ne l'efficacia nelle interazioni più estese e rendendo necessario il passaggio a modelli più performanti

La scelta di sostituire Mistral è stata motivata non solo dai limiti nella gestione del contesto, ma anche dalla necessità di un modello più robusto nell'italiano, sia in termini di accuratezza sintattica sia di aderenza terminologica. I modelli successivamente impiegati, più avanzati e dotati di finestre contestuali estese, hanno offerto prestazioni nettamente superiori nel mantenimento della coerenza a lungo termine.

Di conseguenza, Mistral è stato impiegato solo per un breve periodo e non sono stati condotti test approfonditi sulle sue capacità.

6.1.2 Llama3.2

Llama3.2 è un modello generativo pre-addestrato e open-source sviluppato da Meta, ottimizzato per l'esecuzione di istruzioni e per il supporto al dialogo. È disponibile in due varianti, rispettivamente a 1B e a 3B di parametri; per il progetto è stata testata la seconda versione, la più potente tra le due. L'italiano è tra le lingue ufficialmente supportate, senza necessità di ulteriore *finetuning* [51]. Anche in questo caso il modello è stato eseguito localmente tramite *Ollama*.

Nonostante disponga di un numero di parametri inferiore rispetto al modello adottato in precedenza, Llama3.2 si è dimostrato complessivamente più performante grazie alla *context window* più ampia, pari a 128k token, che consente di gestire un volume di informazioni significativamente maggiore in fase di generazione.

Tuttavia, il modello presenta alcune limitazioni legate alla *context length*, pari a 2.048 token nella configurazione di base ed estendibile solo fino a un massimo di 4.096 token. Questo vincolo può diventare critico quando il messaggio di contesto, composto dai *chunk* restituiti dal *retriever*, supera la soglia massima consentita: in questi casi, il modello non è in grado di generare una

risposta coerente, ma si limita a restituire una frase predefinita configurata a livello di prompt. Si precisa che il problema riguarda esclusivamente il messaggio di contesto: la risposta generata non è coinvolta nell'errore e i token disponibili sono sempre risultati sufficienti.

Sebbene tale situazione si verifichi raramente, è necessario gestirla correttamente per garantire l'affidabilità del sistema. La soluzione adottata consiste nel conteggiare i token del contesto e, nel caso superino il limite massimo, rimuovere il *chunk* in eccesso meno rilevante, ovvero il quinto; questo intervento ha un impatto minimo sulla qualità della risposta, poiché l'analisi dei dati ha mostrato che, in media, i primi due *chunk* contengono le informazioni più significative, rendendo quindi l'operazione poco invasiva.

Llama3.2 è stato impiegato principalmente nelle fasi iniziali dello sviluppo ed è stato successivamente sostituito da modelli più avanzati, anche perché in ottica di produzione non è sostenibile l'utilizzo un modello eseguito in locale. Nonostante ciò, è stato incluso in numerosi test, descritti nel capitolo seguente, e è stata mantenuta retrocompatibilità con esso tramite la classe *LlamaChatbot*, dedicata all'inizializzazione del chatbot con questo modello.

6.1.3 Cohere Command R

Cohere Command R è un LLM a 35B di parametri sviluppato dall'omonima azienda, ottimizzato per il supporto conversazionale e per attività di RAG. Offre un buon equilibrio tra elevate prestazioni, accuratezza e costi contenuti, risultando adatto anche a contesti produttivi aziendali. Supporta pienamente la lingua italiana senza necessità di training aggiuntivo [9].

Trattandosi di un modello proprietario, Cohere è ospitato sulle infrastrutture della società stessa e non è disponibile per l'uso in locale: per potervi accedere, è necessario iscriversi al loro servizio e ottenere una API key personale. Ne esistono due tipologie principali: una chiave di prova, che consente l'accesso gratuito ma con funzionalità limitate (20 richieste/minuto), e una

chiave di produzione, che richiede l'acquisto di token ma offre minori restrizioni in termini di utilizzo (500 richieste/minuto). Durante la fase di sviluppo, l'azienda ha generato una API key gratuita dedicata al progetto, utilizzata per integrare e testare il modello all'interno del sistema.

Cohere presenta la stessa *context window* di Llama3.2, 128k token, ma con il quintuplo dei parametri e la possibilità di aumentare la *context length* fino a 128k token, risulta essere un modello più performante, con una gestione del contesto molto più robusta, e risolve del tutto i problemi relativi ai limiti di token; le risposte generate risultano più precise delle precedenti, con frequenti riferimenti puntuali ai documenti sorgente. L'integrazione nel sistema avviene tramite la classe *CohereChatbot*, che utilizza questo modello come motore di generazione.

Nonostante tali vantaggi, questa modalità di utilizzo è stata ritenuta adatta solo alla fase di sviluppo e non a quella di produzione. Collaborando con la Pubblica Amministrazione, i prodotti del Gruppo Finmatica devono rispettare standard di sicurezza e garantire completa privacy dei dati, in linea con il GDPR europeo e la certificazione UNI EN ISO 9001:2015, di cui l'azienda è provvista. Uno dei requisiti fondamentali impone che tutti i dati trattati dall'LLM rimangano all'interno dei confini dell'Unione Europea, e il modo più diretto per garantire questa condizione consiste nell'affidarsi direttamente a modelli ospitati in data center europei [10]. Sebbene Cohere offra questa possibilità ai propri clienti, si è comunque scelto di proseguire con un'altra piattaforma, ritenuta più adatta alle esigenze e agli standard aziendali.

6.1.4 Oracle

Oracle, tramite la piattaforma *Oracle Cloud Infrastructure* (OCI), offre l'accesso a *Generative AI*, un servizio completamente gestito che offre una selezione di modelli linguistici di grandi dimensioni all'avanguardia, personalizzabili e adatti a un'ampia gamma di casi d'uso, tra cui conversazione,

generazione di testo, creazione di *embedding* e implementazione di pipeline RAG. La piattaforma consente inoltre di scegliere il data center a cui instradare le richieste, e la disponibilità di un centro dati in Germania permette di garantire la conformità al GDPR e una gestione sicura dei dati sensibili [52].

L'adozione della piattaforma OCI e dei modelli disponibili è stata imposta come requisito aziendale, in quanto il gruppo utilizza già da tempo soluzioni Oracle come componente centrale dei propri sistemi. Questa continuità tecnologica semplifica le integrazioni, sfrutta competenze già presenti internamente e rende più efficiente la manutenzione dei servizi; inoltre, la presenza di accordi commerciali consolidati con Oracle rende economicamente vantaggioso l'accesso a OCI GenAI, il cui utilizzo è a consumo. Affidarsi a un unico fornitore garantisce anche supporto dedicato e continuità operativa, fattori essenziali in un contesto produttivo e in scenari sensibili come quelli della Pubblica Amministrazione, rappresentando vantaggi significativi rispetto a soluzioni che richiederebbero una gestione autonoma dell'infrastruttura o l'integrazione di più *provider*.

D'altra parte, la scelta di OCI comporta anche alcune limitazioni: la scelta del modello generativo è vincolata a quelli disponibili sulla piattaforma, riducendo la flessibilità e la possibilità di sperimentare nuove soluzioni. Al momento della stesura di questo elaborato, OCI GenAI supporta i modelli Cohere, incluso Command R, diversi modelli Meta (sebbene non esattamente Llama3.2 a 7B di parametri già analizzato), nonché i modelli Gemini di Google e GPT di OpenAI in versione *beta*; poiché questi ultimi non erano disponibili durante la fase di sviluppo, non sono stati testati nell'ambito di questo progetto.

Oltre a ciò, utilizzare un'infrastruttura è completamente gestita da Oracle introduce un certo grado di dipendenza dal *provider* e limita il controllo diretto su alcune componenti operative; infine, il modello di *pricing* basato sul consumo può risultare più oneroso nel lungo periodo, specie in caso di

applicazioni ad alto volume o in contesti con molti utenti simultanei.

Questa combinazione di vantaggi e vincoli ha portato a scegliere, all'interno di OCI, Cohere Command R come modello definitivo per il progetto, decisione supportata dai risultati dei test quantitativi descritti nel capitolo 7 e che ha permesso di coniugare prestazioni elevate, supporto per la lingua italiana e conformità ai vincoli di privacy richiesti. Rispetto all'utilizzo diretto tramite Cohere, il modello è ora accessibile tramite OCI: l'invocazione avviene tramite credenziali personali, che includono anche la selezione dell'endpoint geografico, richieste a ogni inizializzazione del servizio. La classe *OracleCohereChatbot* gestisce la creazione delle istanze del modello, sfruttando la classe *OciGenAI* di LlamaIndex.

Non sono state condotte ulteriori sperimentazioni con modelli esterni, anche se in prospettiva futura, potrebbe essere interessante valutare l'integrazione dei nuovi LLM disponibili sulla piattaforma, a condizione che rispettino il vincolo geografico previsto.

6.2 Prompt

Un *prompt* è una sezione di testo o una serie di istruzioni fornite a un LLM per ottenere una risposta o attivare un'azione specifica. Gli sviluppatori e gli utenti comunicano con i modelli tramite il prompt, indicando con precisione cosa desiderano ottenere, e il modello risponde al meglio delle proprie capacità di comprensione e generazione [1].

La scrittura dei prompt è un'arte sottile, influenzata da numerose variabili: prompt chiari e accuratamente formulati guidano i modelli linguistici verso risultati di qualità, mentre prompt ambigui o imprecisi generano output altrettanto incerti, indipendentemente dalle capacità del modello. La scelta delle parole, dello stile, del tono e della struttura del prompt ha quindi un impatto significativo sull'efficacia della generazione [35].

Un altro fattore rilevante è il modello utilizzato: a parità di prompt, modelli diversi possono produrre risposte completamente differenti, a causa dei loro parametri, addestramenti e capacità linguistiche. Per ottenere i migliori risultati, il prompt deve essere adattato alle caratteristiche specifiche del modello.

Nel caso del chatbot sviluppato, il prompt è sempre stato scritto in lingua italiana per due motivi principali. Primo, per mantenere una coerenza linguistica tra istruzioni, contesto e testo generato. Secondo, poiché gli sviluppatori erano madrelingua italiani, scrivere nella propria lingua permetteva di gestire con precisione anche le sfumature più sottili di significato, fondamentali per ottenere risposte di qualità. I modelli impiegati supportavano nativamente l'italiano, quindi questa scelta non ha mai rappresentato un ostacolo.

La strategia iniziale prevedeva l'uso di un unico prompt, contenente tutte le informazioni necessarie: le istruzioni di comportamento generali, il contesto, e la query stessa, fornito al modello ad ogni nuova richiesta.

Tuttavia, questa soluzione si è presto rivelata problematica: oltre ad essere estremamente inefficiente in termini di utilizzo dei token, le risposte generate risultavano di scarsa qualità, incoerenti e inconsistenti tra un'interazione e l'altra, rendendo i risultati inaccettabili.

Dopo ulteriori studi, si è deciso di suddividere le istruzioni in due prompt distinti, *system prompt* e *user prompt*, in base alla loro funzione. Questo approccio ha migliorato la qualità delle risposte, ma la scrittura del prompt, soprattutto quello di sistema, è rimasta un processo lungo e complesso: i modelli linguisti di grandi dimensioni introducono sempre un certo grado di imprevidibilità e, a parità di input, raramente una risposta risulta essere identica alla precedente, rendendo più difficile valutare le modifiche apportate. Inoltre, qualunque tipo di valutazione presenta sempre un certo grado di soggettività, non tanto per i contenuti quanto per la forma con cui essi sono espressi. Infine, non è solo il prompt a stabilire la qualità della risposta, ma

l'intera pipeline RAG, dall'estrazione al *retrieval*; individuare il componente responsabile di un errore richiede tempo ed esperienza, e spesso è necessario adattare il prompt *ad hoc* per correggere specifici problemi.

6.2.1 System Prompt

Un *prompt di sistem*a è un'istruzione fornita dagli sviluppatori di un modello di IA per definire il contesto, il tono, l'identità e i limiti delle risposte generate. Esso funge da guida generale, plasmando il comportamento e lo stile dell'LLM durante l'interazione: il suo obiettivo è garantire che il modello sia allineato a scopi specifici, offra un'esperienza utente coerente, rispetti le linee guida etiche e mantenga la consistenza delle risposte [48]. Il prompt di sistema ha un'influenza duratura sul comportamento del modello, non limitata alla singola interazione: mira ad preservare la consistenza temporale e, una volta impostato, si mantiene stabile fino ad ulteriori aggiornamenti.

Nella sua versione finale, il prompt di sistema del progetto si presenta in forma parametrica, per poter essere utilizzato a seconda dei casi sia dal chatbot dedicato ad Affari Generali, sia da quello per i sistemi ERP.

La parte comune si divide in tre sezioni: *Identità*, *Tono*, e *Gestione dei saluti e delle domande generali*. La suddivisione non è soltanto concettuale, ma anche grafica: il nome di ciascuna sezione è evidenziato tramite l'uso di due simboli #, ed esse sono separate da righe vuote, rendendo la struttura del testo più chiara, diretta e conforme alle preferenze del modello.

La sezione *Identità* definisce il ruolo del chatbot, mentre la sezione *Tono* specifica la lingua e il registro da adottare nelle risposte, indicati rispettivamente come italiano e "formale, educato e coerente".

La parte relativa alla *Gestione di saluti e domande generali*, pur non facendo parte della struttura standard di un prompt di sistema, è stata introdotta per uniformare le risposte del chatbot a questo tipo di input, che in precedenza risultavano molto eterogenee, andando da semplici saluti a lunghe

spiegazioni delle proprie funzionalità. In questo modo, il modello dispone di linee guida chiare per gestire tutti i casi rilevati durante lo sviluppo.

La parte differenziata del prompt, invece, contiene le sezioni *Scopi* e *Gestione delle risposte*. La maggior parte delle istruzioni hanno schema e direttive identiche; la specializzazione è dovuta alla maggiore complessità di Affari Generali, che richiede indicazioni molto specifiche e non applicabili altrove. Inoltre, ciascuna sezione presenta riferimenti ed esempi di risposta pertinenti ai software in questione.

In *Scopi* viene definito l'obiettivo principale del chatbot: fornire risposte pertinenti all'ambito del programma in uso. È esplicitamente vietato al modello inventare o modificare informazioni e conoscenza, al fine di ridurre il fenomeno delle allucinazioni, emerso frequentemente durante le fasi di test; allo stesso modo, gli viene proibito di rispondere a domande di carattere generale, non pertinenti ai software dell'azienda..

Il paragrafo sulla *Gestione delle risposte* fornisce istruzioni specifiche sugli standard da rispettare nella generazione dei testi; in particolare, spiega al modello come trattare i *chunk* e le informazioni in essi contenute, considerando anche diversi casi limite come l'impossibilità di trovare una risposta nella conoscenza recuperata o la presenza di dati discordanti o inconcludenti. Nel caso in cui la domanda non sia inerente al software considerato, viene specificata una frase standard con cui rispondere.

Questo prompt viene caricato una sola volta all'inizio della conversazione tramite un oggetto *ChatMessage* con ruolo system, garantendo così che tutte le interazioni successive siano coerenti con le linee guida definite.

6.2.2 User Prompt

Un *prompt utente* è l'input fornito dall'utente finale durante l'interazione con l'intelligenza artificiale, e solitamente si tratta di una domanda, un comando o un'affermazione che definisce il compito da eseguire. Questo tipo di

prompt rappresenta un'interazione diretta con l'utente ed è adattata di volta in volta in base alle sue esigenze o preferenze. Per sua natura, è altamente dinamico: ogni iterazione può differire significativamente dalle precedenti e non garantisce alcuna consistenza temporale [48].

Nel contesto di questo progetto, i prompt utente non servono solo a porre domande al chatbot, ma forniscono anche al modello le informazioni necessarie per generare risposte pertinenti. Dai test condotti, è emerso che separare chiaramente la domanda dal contesto riduce la confusione del modello, migliorando la precisione e la puntualità delle risposte finali.

Di conseguenza, per ogni query ricevuta, la conversazione e il vettore dei messaggi vengono aggiornate con due oggetti *ChatMessage*, entrambi con ruolo user. Il primo contiene il contesto, formattato come descritto in precedenza, accompagnato da un'istruzione che ne spiega il significato e lo scopo; il secondo include esclusivamente la query, ovvero il prompt utente vero e proprio, da cui ha inizio la generazione della risposta.

In questo modo, il prompt di sistema definisce le regole generali e guida il comportamento del modello, mentre il prompt utente fornisce le informazioni specifiche per la singola interazione. La combinazione dei due permette di bilanciare stabilità, coerenza e adattabilità, garantendo risposte precise e contestualmente rilevanti.

6.2.3 Prompt di Riformulazione

Il chatbot ha evidenziato alcune difficoltà nell'estrapolare correttamente il contesto in presenza di domande formulate in modo generico. Ad esempio, alla richiesta *"Come protocollo un documento?"* il sistema non forniva alcuna risposta, mentre la domanda *"Come protocollo un documento in SmartDesktop?"* restituiva le informazioni corrette tratte direttamente dai manuali di riferimento. L'unica differenza tra le due query è la specifica esplicita

del contesto applicativo nella seconda, che permette al sistema di orientare correttamente la ricerca.

Un'ulteriore fonte di ambiguità è legata alla variazione dei tempi e dei modi verbali. Il chatbot, infatti, risponde in maniera più efficace a domande formulate con il verbo all'infinito (*"Come accedere all'applicativo"*) rispetto a quelle espresse in prima persona singolare (*"Come accedo all'applicativo"*). Poiché gli esperti di dominio ritengono che l'utente medio utilizzi più frequentemente la prima persona presente singolare, si è reso necessario introdurre un meccanismo per gestire questi casi limite, minimizzando così le occorrenze di errore e le risposte mancate.

A tal fine è stato creato un terzo prompt, denominato *prompt di riformulazione*, gestito da una seconda istanza indipendente dello stesso modello Cohere Command R utilizzato per la generazione delle risposte principali, con il compito di adattare le query problematiche, trasformandole in forme che il modello possa interpretare in maniera ottimale.

Il prompt riformula le query degli utenti rendendole il più possibile simili alla query ideale del modello. In particolare, le trasforma da una forma personale a una impersonale, sostituendo il verbo con la sua forma all'infinito e mantenendo invariato il resto della frase. Inoltre, aggiunge in coda alla domanda la categoria di appartenenza, chiarendo ulteriormente il contesto semantico per il chatbot; la categoria è sempre nota, in quanto è uno dei parametri inviati tramite API. Ad esempio, la query *"Come protocollo un documento"*, appartenente alla categoria SmartDesktop, viene riformulata in *"Come protocollare un documento in SmartDesktop"*.

Questa fase di riformulazione avviene prima dell'avvio del processo di RAG, immediatamente dopo la ricezione della domanda dall'endpoint del server. In questo modo, il modulo di *retrieval* elabora esclusivamente la versione riformulata della query, garantendo che la ricerca delle informazioni parta da un input già normalizzato e coerente con l'intento dell'utente.

L'introduzione di questa fase di pre-processamento dell'input ha prodotto un miglioramento nelle prestazioni complessive del sistema: è diminuito il numero di domande prive di risposta e, parallelamente, si è registrato un incremento nella qualità e nella pertinenza delle risposte generate. Tuttavia, l'approccio presenta alcuni limiti intrensechi nella gestione delle domande più articolate e complesse, discussi in dettaglio nel capitolo 8.

Capitolo 7

Test Automatici

Una volta ottenuta una versione stabile del chatbot, risulta fondamentale poterla testare con frequenza e in tempi ridotti, così da confrontare in modo efficace le risposte generate dai diversi LLM analizzati.

A tal scopo, è stata sviluppata una suite di test automatici che semplifica il processo di verifica, consentendo di valutare gli aspetti principali di ciascun modello: la correttezza delle risposte, il tempo di elaborazione e il numero massimo di interazioni gestibili prima dell'esaurimento dei token disponibili.

I modelli presi in esame sono Llama3.2, eseguito in locale, e Cohere Command R, accessibile tramite API, già introdotti nel capitolo precedente.

7.1 Benchmarks

I seguenti test sono stati svolti su una macchina con sistema operativo Windows 11 Enterprise, processore Intel(R) Core(TM) i9-14900K @ 3200 Mhz, 24 Core(s), 32 Logical Processor(s), 64GB di RAM e scheda grafica NVIDIA GeForce RTX 4090.

7.1.1 Correttezza Risposte

La metodologia utilizzata per valutare la correttezza delle risposte riprende quella già impiegata nel confronto tra modelli di *embedding*: dieci domande, due per ciascuna categoria, alle quali ogni modello fornisce una risposta *one-shot*.

Le domande sono raccolte in un file Excel, organizzato secondo la struttura CATEGORIA – DOMANDA – RISPOSTA CORRETTA, che funge da base dati per l'intero processo di test. Le risposte corrette sono state redatte direttamente dagli esperti di dominio, e rappresentano il cosiddetto *golden standard* con cui confrontare le risposte restituite dai modelli.

L'elaborazione avviene tramite due script Python appositamente sviluppati. Il primo ha il compito di gestire la fase di generazione: dopo aver letto le domande dal file Excel, le carica in un *DataFrame* di Pandas, per poi invocare la funzione di elaborazione per ognuna di esse. Le risposte ottenute vengono automaticamente reinserite nello stesso file, arricchendo le informazioni con due nuove colonne: RISPOSTA RICEVUTA, contenente l'output del modello, e CONTESTO, che raccoglie i *chunk* recuperati dal modulo di *retrieval* e utilizzati come base informativa per la risposta.

Completata la fase di generazione, il secondo script esegue la valutazione, confrontando ogni risposta ottenuta con la corrispondente risposta corretta, applicando diverse metriche di similarità per misurare la qualità in modo quantitativo.

Le metriche utilizzate sono le seguenti:

- **BLEU Score** (*Bilingual Evaluation Understudy*): misura la precisione degli *n-grammi* dell'output del modello rispetto al testo di riferimento umano, calcolando quante parole della risposta del chatbot appaiono nel *golden standard* [25].

  - Range: $0 \rightarrow 1$: un valore più alto indica una maggiore somiglianza strutturale tra la risposta generata e la risposta corretta.

- **ROUGE Score** (*Recall-Oriented Understudy for Gisting Evaluation*): metrica focalizzata sul *recall*, confronta la sovrapposizione di *n-grammi*, sequenze e coppie di parole in entrambi i testi [25].

  - Range: $0 \rightarrow 1$: un valore più alto indica una maggiore sovrapposizione tra la risposta generata e quella corretta.

- **Cosine Similarity**: misura la similarità tra due vettori diversi da zero calcolando il coseno dell'angolo tra di essi: una distanza minore indica una similarità maggiore, mentre una distanza maggiore indica una similarità minore, indipendentemente dalle dimensioni dei testi [24].

  - Range: $-1 \rightarrow 1$:
    * 1 = risposte identiche, significato e struttura simile.
    * 0 = nessuna somiglianza.
    * -1 = frasi e significato opposti.

L'insieme di queste metriche consente di ottenere una valutazione completa dei modelli considerati, offrendo una base oggettiva per il confronto tra diversi LLM.

Metrica	**Cohere Command R**	**Llama3.2**	**Variazione**
BLEU Score	0.22	0.12	+83.3%
ROUGE Score	0.41	0.36	+13.9%
Cosine Similarity	0.53	0.54	−1.9%

Tabella 7.1: Confronto tra Cohere Command R e Llama3.2 sulle metriche esaminate.

La tabella 7.1.1 mostra come Cohere Command R ottenga, in media, risultati superiori: i valori medi del ROUGE Score (0.41) e del BLEU Score (0.22) indicano una maggiore somiglianza lessicale e strutturale con le risposte corrette rispetto a Llama3.2, che si attesta rispettivamente su 0.36 e 0.12; la *Cosine Similarity*, invece, evidenzia prestazioni analoghe per entrambi i modelli (0.54 per Llama, 0.53 per Cohere), segno che le risposte generate risultano semanticamente simili alle risposte di riferimento.

7.1.2 Tempi di Elaborazione e Numero Massimo Interazioni

Dopo aver valutato la qualità delle risposte generate dai modelli, si è proceduto al testing delle prestazioni del sistema, con l'obiettivo di verificarne la robustezza nel lungo periodo.

È stato implementato un test automatico per osservare il comportamento dei modelli di fronte a un numero elevato di domande, sufficiente a esaurire i token disponibili. A differenza del test precedente, in cui ogni domanda veniva inviata a una nuova istanza del modello, in questo caso tutte le domande sono state indirizzate alla stessa istanza, in sequenza. Poiché l'obiettivo non era valutare la correttezza delle risposte, e non era noto a priori il limite superiore di domande accettabili, a ciascun modello sono state ripetutamente inviate le stesse quattro domande fino al completo esaurimento dei token.

Parallelamente, sono stati monitorati i tempi di risposta per le singole domande. È rilevante sottolineare come Llama sia eseguito in locale, mentre Cohere richiede l'invio delle richieste tramite API a un modello in cloud; di conseguenza, ci si attende tempi di risposta inferiori per Llama, non essendo previsto alcun trasferimento dati verso l'esterno.

I risultati ottenuti sono soddisfacenti e confrontabili fra i due modelli: a fronte di *context window* simili, Cohere riesce a gestire mediamente 43

domande prima di esaurire i token, mentre Llama 45. Inoltre, entrambi sembrano in grado di rispondere correttamente alla prima domanda che supera il limite di token, ma ovviamente non alle successive.

Per quanto riguarda i tempi di risposta, Llama completa l'elaborazione in media entro quattro/cinque secondi, indipendentemente dal numero di domande inviate o dal consumo dei token. Cohere, al contrario, mostra tempi di risposta crescenti e proporzionali al numero di domande: si passa da circa un secondo per le prime fino a oltre trenta secondi per quelle finali.

Questa differenza significativa può essere attribuita al fatto che Llama opera in locale, mentre Cohere risiede su un server remoto e richiede l'invio dei dati tramite API. Inoltre, nell'esperimento si è utilizzata la versione gratuita dell'API di Cohere, il che può comportare ulteriori ritardi dovuti alla condivisione delle risorse GPU/CPU con altri utenti.

7.1.3 Conclusioni

A seguito dei test svolti, è possibile affermare che, nel limite del contesto progettuale corrente, Cohere Command R sia un modello più performate di Llama3.2 a livello generativo, ma meno a livello computazionale.

Le valutazioni quantitative trovano riscontro con i giudizi degli esperti di dominio sulle generazioni proposte da entrambi modelli: Cohere tende a produrre risposte complessivamente più articolate e coerenti con il materiale di riferimento, mentre Llama3.2 restituisce risposte più concise e leggermente meno aderenti al *golden standard*.

I tempi di risposta di Cohere, però, non risultano accettabili in un contesto di produzione aziendali, in quanto non in linea con le aspettative dei potenziali clienti e utilizzatori. Queste differenze derivano in parte dalle rispettive architetture e strategie di generazione: Llama, ottimizzato per l'uso locale, sembra meglio integrato con la base di dati e per questo più veloce,

mentre Cohere mostra maggiore stabilità linguistica relativamente al contesto ma costo computazionale più elevato.

L'introduzione di OCI GenAI consente di ridurre i tempi di latenza: trattandosi di una piattaforma *enterprise*, mette a disposizione una quantità dedicata di risorse, permettendo così di contenere il tempo di generazione a una media di meno di cinque secondi per risposta, paragonabile alle performance di un modello locale.

Si sottolinea, inoltre, che i casi analizzati in precedenza rappresentano scenari limite: in condizioni d'uso reali, si prevede che gli utenti rimangano ben al di sotto delle 40 domande per conversazione, il che permette sia di contenere i costi, sia di mantenere un certo standard in relazione ai tempi di attesa.

Le considerazioni emerse confermano quindi la scelta di adottare Cohere Command R come Large Language Model principale all'interno del progetto, decisione supportata anche dalle opinioni professionali degli esperti di dominio.

7.2 Macchina Linux

In una prima fase, ogni componente del progetto veniva salvato ed eseguito direttamente sulla macchina Windows, utilizzata sia per lo sviluppo sia per i test. Questo approccio garantiva un controllo completo e immediato sulle diverse istanze in esecuzione, ma presentava alcuni limiti: l'introduzione di aggiornamenti richiedeva interventi più complessi e il carico computazionale risultava elevato. Per ovviare a queste criticità, è stata introdotta una macchina virtuale, con sistema operativo Linux e priva di GPU, dedicata esclusivamente ai test e alle prove degli esperti di dominio. In questo modo, un'istanza dell'ultima versione stabile del chatbot rimane sempre attiva e

pronta a ricevere query, mentre la macchina principale può essere utilizzata liberamente per lo sviluppo, aumentando il parallelismo delle attività.

La macchina Linux funge unicamente da host del prodotto finito: operazioni intensive, come la creazione della base di conoscenza, continuano ad essere eseguite sulla macchina con GPU, senza la quale non sarebbero possibili. Il database contenente i *chunk* estratti ed *embeddati* viene trasferito già popolato dalla macchina di sviluppo alla macchina host, mantenendo inalterata la struttura e le modalità di accesso.

Il modello linguistico di grandi dimensioni, accessibile in remoto tramite la piattaforma Oracle, non richiede elevate risorse hardware; pertanto, si prevedono dei tempi di risposta confrontabili a quelli osservati in precedenza.

Lo stesso test è stato eseguito anche sulla macchina virtuale Linux. Come previsto, i tempi di risposta sono risultati in linea con le aspettative, mantenendosi stabilmente attorno a una media di cinque secondi per domanda, comprensivi sia del tempo necessario al modulo di *retrieval* per recuperare i *chunk*, sia dei tempi di invio e ricezione dai server Oracle.

Complessivamente, i risultati finali sono stati ritenuti soddisfacenti da parte degli esperti di dominio, portando all'adozione della macchina virtuale come *host* principale del sistema.

Capitolo 8

Limitazioni Funzionali

Avendo concluso la panoramica sulla prima versione funzionante del chatbot, risulta corretto discutere le limitazioni progettuali e implementative del sistema, al fine di comprendere pienamente il contesto del progetto e individuare possibili aree di sviluppo future.

8.1 Metriche Automatiche

La valutazione delle risposte generate da un sistema RAG pone sfide significative, specialmente quando ci si affida a metriche automatiche tradizionali come BLEU e ROUGE Score, o simili misure basate sull'*overlapping* sintattico tra testo generato e testo di riferimento.

Sebbene queste metriche siano ampiamente utilizzate, non riescono a catturare adeguatamente la *qualità semantica* delle risposte prodotte da un LLM all'interno di una pipeline RAG. Il principale limite risiede nel fatto che esse misurano la sovrapposizione di *n-grammi* tra l'output generato e il *golden standard*, ignorando aspetti più profondi come la correttezza concettuale, la coerenza logica e l'utilità informativa: una risposte semanticamente equivalente al *golden standard* può ottenere un punteggio molto basso semplicemente perché espressa con lessico o struttura frasale diversa; al contrario, un te-

sto sintatticamente simile può ottenere un punteggio elevato pur contenendo imprecisioni o omissioni rilevanti dal punto di vista semantico.

Inoltre, nel contesto del RAG, la qualità dell'output non dipende soltanto dalla forma linguistica della risposta, e quindi dalle capacità generative di un modello, ma anche dalla qualità del *retrieval*, dalla pertinenza dei *chunk* recuperati e dalla loro integrazione nel ragionamento del modello.

Questi aspetti non vengono minimamente considerati da metriche basate sul confronto superficiale tra stringhe, e di conseguenza l'utilizzo di BLEU o ROUGE come criteri primari di valutazione rischia di fornire una rappresentazione distorta delle prestazioni reali del sistema. Per una valutazione più attendibile diventa spesso necessario affiancare tali metriche con giudizi umani, metriche semantiche, progettate per misurare la similarità del significato anziché la sola corrispondenza letterale, o analisi qualitative più profonde, in grado di cogliere la reale utilità e correttezza delle risposte generate.

8.2 Documenti

L'efficacia complessiva della pipeline RAG dipende in modo diretto e sostanziale dalla qualità e dalla struttura dei documenti di origine.

L'utilizzo del *chunking* semantico, pur migliorando la coesione interna rispetto a metodi basati su semplici soglie di caratteri, rende i risultati fortemente dipendenti dall'organizzazione originale del materiale: documenti con sezioni molto lunghe generano *chunk* estesi, talvolta troppo ricchi di informazioni eterogenee per essere utili, mentre sezioni brevi o frammentarie danno luogo a *chunk* molto corti, potenzialmente privi di sufficiente contesto.

Questa variabilità dimensionale influisce tanto sulla fase di *retrieval*, dove *chunk* troppo lunghi possono risultare solo parzialmente rilevant, e *chunk* troppo brevi possono non contenere l'informazione desiderata, quanto sulla

generazione della risposta, poiché modelli LLM tendono a performare meglio quando ricevono input coerenti, compatti e ben strutturati.

Oltre alla struttura, la qualità intrinseca dei manuali rappresenta un fattore determinante. La pipeline RAG, per sua natura, non può compensare errori concettuali, ambiguità, testi poco chiari o informazioni obsolete: difetti presenti nei documenti si riflettono inevitabilmente nelle risposte del chatbot. In più di un'occasione, alcune risposte scorrette generate dal sistema hanno permesso di individuare errori preesistenti nei manuali stessi, evidenziando come il modello non stesse allucinando, ma riproducendo fedelmente informazioni errate provenienti dalle fonti di origine.

Il confronto tra diversi gruppi di manuali ha mostrato che la qualità delle risposte varia anche in relazione allo stile espositivo: i documenti relativi a ERP, per loro natura più schematici, strutturati come liste a punti, hanno prodotto recuperi e generazioni di qualità mediamente superiore rispetto ai manuali di Affari Generali, tipicamente più discorsivi e meno formalizzati.

Questo comportamento conferma una tendenza nota: gli LLM e le pipeline RAG traggono beneficio da dati chiaramente strutturati, mentre le performance degradano quando i contenuti sono verbosi o poco organizzati. Di conseguenza, la qualità della documentazione di partenza non è solo un requisito auspicabile, ma un elemento critico che influisce direttamente sulle prestazioni e sull'affidabilità del sistema. Tra le possibili migliorie, si evidenziano una strutturazione più rigorosa del contenuto tramite una suddivisione chiara in sezioni, sottosezioni e paragrafi tematici, che facilitarebbe sia il *chunking*, sia la comprensione contestuale; inoltre, sarebbe utile l'adozione di un glossario dei termini tecnici, completo di esempi pratici e casi d'uso reali, con l'obiettivo di ridurre ambiguità semantiche e ad agevolare il ragionamento dell'LLM. Scelte più estreme, come la riscrittura dei documenti in un formato più *machine-friendly* (ad esempio, Markdown), risultano poco applicabili a livello aziendale a causa della grande quantità di lavoro

necessario a fronte di miglioramenti non così certi.

8.3 Retrieval

Il metodo concreto di *retrieval* utilizzato dal modulo *retriever* non è stato implementato internamente, ma è fornito direttamente da LlamaIndex attraverso l'oggetto *PGVectorStore*. Questo implica che l'intera fase di ricerca vettoriale, dal calcolo delle distanze alla selezione dei k elementi più rilevanti, è delegata a un componente esterno, il cui comportamento è in larga parte determinato dalle scelte progettuali del framework.

Ciò limita fortemente il controllo sul processo di *retrieval*, in quanto l'implementazione del componente risulta opaco e non modificabile dall'esterno: non è possibile intervenire direttamente sugli algoritmi di ricerca o sulla strategia di *ranking*, nè introdurre filtri personalizzati o logiche avanzate di *re-ranking*; eventuali anomalie nel recupero dei *chunk* dipendono dal comportamento interno del framework e non sono immediatamente comprensibili o correggibili dal lato applicativo.

Ne derivano conseguenze dirette sulla qualità dei *chunk* recuperati, che può variare in base al modello di *embedding*, al contenuto dei documenti, e alle politiche di ricerca implementate da LlamaIndex. Pur potendo controllare parametri esterni come il numero di *chunk* restituiti o il modello di *embedding*, non si ha accesso completo alla pipeline interna che determina la rilevanza finale dei risultati.

Nonostante questo aspetto rappresenti un limite intrinseco dell'approccio basato su un framework, la progettazione e implementazione di una funzione di *retrieval* personalizzata esula dagli obiettivi di questo progetto e risulterebbe troppo costosa in termini di tempo e di risorse; la scelta di affidarsi a una funzione di *retrieval* esterna rimane l'unica possibile, anche se, in ottica di sviluppi futuri, potrebbe essere utile introdurre migliorie al processo,

come livelli aggiuntivi di *re-ranking* personalizzato o sistemi ibridi di ricerca semantica.

8.4 Prompt di Riformulazione

Nonostante il prompt di riformulazione abbia migliorato la precisione del sistema, introduce a sua volta alcuni limiti strutturali in presenza di query complesse, ambigue o multiple.

La riformulazione opera sotto l'assunzione che la domanda dell'utente possa essere ricondotta a una forma standard, composta da un verbo all'infinito, un oggetto diretto e una categoria applicativa ben definita. Tuttavia, non tutte le query reali rispettano questa struttura: richieste generiche come "Ho ricevuto un errore, cosa devo fare?", oppure domande articolate che includono più azioni o condizioni, non rientrano nella casistica prevista e non sono quindi riformulabili.

Inoltre, la normalizzazione linguistica effettuata dal prompt non risolve situazioni in cui la domanda contiene riferimenti impliciti o formulazioni discorsive tipiche di un dialogo naturale: in questi casi, il modello rischia di produrre una riformulazione grammaticalmente corretta ma semanticamente povera, riducendo la capacità del sistema di cogliere l'intento effettivo dell'utente.

Infine, questo approccio implica una dipendenza intrinseca dalla qualità del modello che effettua la riformulazione: nonostante il prompt lo vieti, se il modello introduce anche solo lievi distorsioni, ad esempio modificando un termine tecnico, confondendo due funzionalità simili o interpretando in modo errato una frase ambigua, l'errore si propaga nelle fasi successive della pipeline RAG. Poiché il *retrieval* opera esclusivamente sulla query trasformata, ciò può compromettere l'intero processo, portando a contesti subottimali e a risposte incoerenti o errate.

Pur rappresentando una soluzione efficace per domande semplici o moderatamente ambigue, esso mostra limiti strutturali che diventano più evidenti con l'aumentare della complessità e della variabilità linguistica delle query, e richiederebbe ulteriori approfodimenti e migliorie per essere pienamente applicabile.

Capitolo 9

Salvataggio Conversazioni

L'accesso alle conversazioni pregresse rappresenta un requisito essenziale per ogni sistema basato su modelli linguistici: analizzare gli scambi tra utenti e chatbot consente di verificare l'efficacia delle modifiche introdotte nel tempo, monitorare la qualità delle risposte e comprendere il comportamento dell'utente medio (quali domande vengono poste più frequentemente, quali difficoltà ricorrono, qual è la durata tipica delle interazioni). Allo stesso tempo, la disponibilità delle conversazioni facilita il lavoro degli esperti di dominio, che possono valutare i test svolti da altri, individuare casi limite e proporre miglioramenti mirati.

Per soddisfare queste esigenze, è stato introdotto un componente dedicato alla memorizzazione strutturata delle conversazioni, dei feedback e dei commenti, con salvataggio diretto su database. La registrazione dei dati avviene in tempo reale, non appena una nuova informazione viene prodotta dal sistema, diventando parte integrante del normale flusso di esecuzione.

È stato inoltre progettato un servizio ausiliario per consentire l'accesso remoto ai dati raccolti: un server locale gestisce la connessione al database ed espone endpoint HTTP attraverso cui è possibile recuperare e visualizzare i messaggi tramite un'interfaccia utente dedicata. Questo approccio permette agli sviluppatori e agli esperti di dominio di consultare rapidamente lo storico

delle interazioni, anche da macchine non direttamente collegate al sistema principale.

In prospettiva futura, l'infrastruttura potrà essere estesa integrando strumenti avanzati di analisi statistica, funzioni di filtraggio più sofisticaten o esportazione dei dati per studi approfonditi. Tali evoluzioni permetterebbero non solo una migliore comprensione dell'utilizzo reale del chatbot, ma anche un processo di miglioramento continuo più strutturato e reattivo.

9.1 Conversation Manager

La classe *conversation_manager* è stata introdotta per gestire in modo centralizzato il salvataggio delle conversazioni e dei singoli messaggi. Per questa funzionalità si è scelto di riutilizzare il database PostgreSQL già impiegato per la base di conoscenza, aggiungendo nuove tabelle dedicate, senza modificare la struttura esistente.

Per ciascun software (Affari Generali ed ERP) sono state progettate due tabelle: una per le conversazioni a livello aggregato (*chats*) e una per i messaggi individuali (*messages*). Ogni conversazione è identificata tramite un UUID, utilizzato come *primary key* nella tabella *chats* e come *foreign key* nella tabella *messages*, in modo da mantenere un'associazione chiara e stabile fra la conversazione e i messaggi che la compongono. I messaggi sono ulteriormente identificati da un id incrementale generato automaticamente, che permette anche di preservare l'ordine temporale con cui vengono prodotti.

Le figure 9.1 e 9.2 mostrano la struttura completa delle tabelle. Per la tabella *chats* è prevista una possibile estensione: l'aggiunta di un campo *description*, generato automaticamente da un LLM prima del salvataggio, che consentirebbe di comprendere immediatamente il tema principale della conversazione; al momento questa funzionalità non è stata ancora implementata, ma rappresenta un'evoluzione naturale del sistema.

Chats	
PK	**chat_uuid**
	category
	model
	datetime

Figura 9.1: Struttura della tabella *chats*.

Messages	
FK	chat_uuid
PK	**id**
	role
	text
	datetime
	feedback
	comment

Figura 9.2: Struttura della tabella *messages*.

Un oggetto *conversation_manager*, istanziato sia in *Chatbot* sia in *RagExecutor*, si occupa delle operazioni di scrittura. Al momento dell'inizializzazione, la classe utilizza la libreria Python *psycopg2* per stabilire una connessione al database e verificare la presenza delle tabelle necessarie, creando automaticamente quelle mancanti.

I metodi principali sono quattro. I primi due gestiscono l'inserimento di una nuova conversazione e di un nuovo messaggio: il primo viene richiamato da *RagExecutor* ogni volta che si avvia una nuova sessione, mentre il secondo viene eseguito dopo la creazione di ciascun oggetto *ChatMessage*. A ogni interazione vengono salvati tre messaggi, nell'ordine contesto, query e risposta, così da conservare traccia dell'intero flusso.

Gli altri due metodi sono invocati direttamente dalla classe *Chatbot* quan-

do l'utente invia un feedback. Entrambe le tipologie di valutazione vengono annotate nei relativi campi dell'ultimo messaggio della conversazione (la risposta del modello), sfruttando la combinazione fra UUID della conversazione e id incrementale del messaggio per individuare con precisione il record da aggiornare.

9.2 Server

Il server dedicato alla gestione delle conversazioni è stato realizzato impiegando le stesse tecnologie del server principale, ovvero FastAPI e Uvicorn in ambiente Python, così da garantire coerenza architetturale e semplificare le attività di manutenzione. A supporto dell'applicazione è presente anche una classe ausiliaria che raccoglie i metodi di utilità necessari alla connessione con il database, alla formattazione dei risultati e alla gestione di eventuali eccezioni.

Il server espone due endpoint, entrambi accessibili tramite richieste HTTP di tipo GET. Il primo, *get-ids*, viene invocato automaticamente all'apertura dell'interfaccia utente e non richiede parametri: attraverso un metodo interno, recupera tutti i record presenti nella tabella *chats* e li restituisce al client, così da mostrare immediatamente l'elenco delle conversazioni disponibili.

Il secondo endpoint, *get-convo*, è attivato in seguito a un'interazione dell'utente, nello specifico un *click* su una conversazione specifica, e richiede come parametro lo UUID della conversazione da recuperare. Il server interroga la tabella *messages* per ottenere tutti i messaggi associati a quello stesso UUID, restituendoli già ordinati per consentire una visualizzazione completa e coerente del dialogo.

Sebbene si tratti di un servizio accessibile soltanto all'interno dell'ambiente di test, sono state adottate alcune misure di sicurezza di base: la comunicazione con il database avviene tramite credenziali dedicate e con

privilegi limitati, riducendo il rischio di modifiche accidentali o accessi indesiderati; inoltre, ogni endpoint integra controlli sui parametri ricevuti, così da evitare richieste malformate o potenzialmente dannose.

La gestione degli errori è stata progettata per garantire facilità di diagnosi: eventuali problemi di connessione al database, UUID inesistenti o formati non validi vengono intercettati e restituiti all'interfaccia tramite messaggi chiari, evitando crash del server e semplificando l'individuazione della causa del problema.

Il server, nella sua implementazione attuale, risponde pienamente ai requisiti previsti dal progetto, ma lascia spazio a diverse estensioni future, tra cui l'introduzione di filtri avanzati (per data, software, presenza di feedback, categoria della domanda), e l'esposizione di endpoint aggiuntivi per l'analisi statistica delle conversazioni o per l'esportazione dei dati in formati standard. Queste estensioni permetterebbero di trasformare il server da semplice archivio consultabile a vero e proprio strumento di monitoraggio e analisi, utile sia per gli sviluppatori sia per gli esperti di dominio.

9.3 Interfaccia Utente

L'interfaccia utente, sviluppata in HTML e JavaScript, consente agli utilizzatori di consultare in modo semplice e immediato le conversazioni archiviate, senza dover accedere direttamente al database o interagire con strumenti tecnici. La GUI funge quindi da livello di visualizzazione, offrendo un punto di accesso intuitivo e completamente separato dalla logica applicativa.

L'interfaccia è strutturata in due tabelle principali. La prima, mostrata in figura 9.3, elenca tutte le conversazioni memorizzate nel sistema e viene popolata automaticamente al caricamento della pagina grazie a una richiesta al primo endpoint del server, (*get-ids*). In questo modo, l'utente dispone fin da subito della panoramica completa delle interazioni registrate.

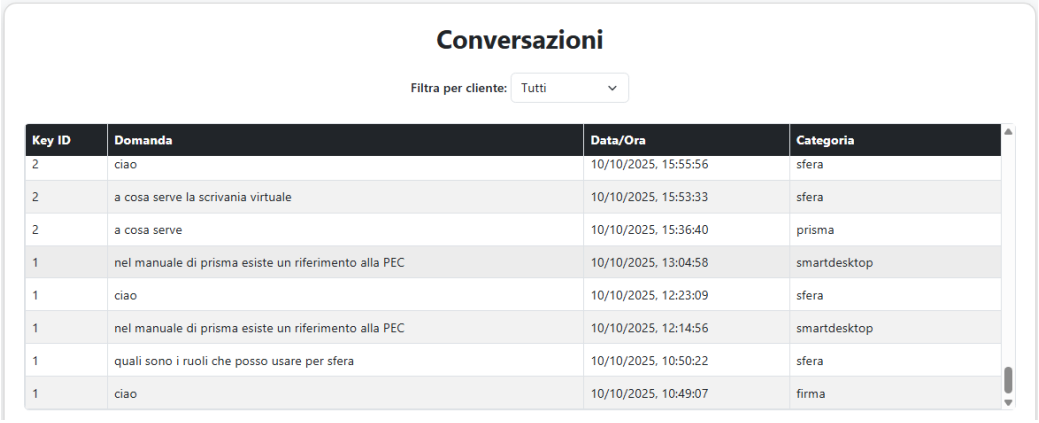

Figura 9.3: Tabella delle conversazioni.

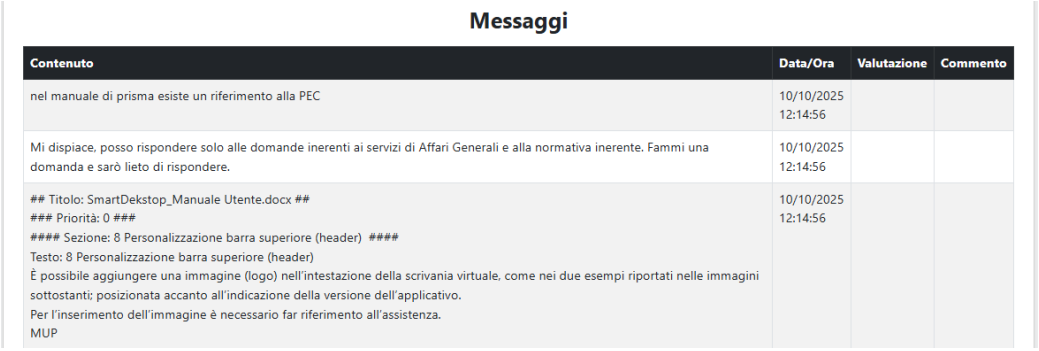

Figura 9.4: Tabella dei messaggi.

Gli identificativi delle conversazioni non sono semplici etichette, ma veri e propri pulsanti cliccabili: selezionandone uno, il browser invia una richiesta al secondo endpoint (*get-convo*), passando come parametro lo UUID corrispondente; il server recupera quindi dal database tutti i messaggi associati a quella conversazione e li restituisce all'interfaccia, che provvede a popolare la seconda tabella, inizialmente vuota, mostrata in Figura 9.4. Il processo può essere ripetuto liberamente, consentendo all'utente di navigare rapidamente tra conversazioni differenti.

Tutte le comunicazioni con il server avvengono mediante JavaScript, che gestisce l'invio delle richieste HTTP e l'aggiornamento dinamico dell'inter-

faccia. Questo approccio mantiene la GUI leggera, reattiva e facilmente estendibile, permettendo in futuro l'aggiunta di funzionalità supplementari, come filtri, ricerche avanzate o visualizzazioni più articolate.

Capitolo 10

Redis

10.1 Problematiche

Il salvataggio permanente delle conversazioni è gestito in modo efficiente, ma lo stesso non si può dire del vettore dei messaggi utilizzato internamente dal chatbot: esso non segue le *best practices* dello sviluppo software, non è scalabile né robusto e in caso di un numero elevato di utenti, come ci si aspetta in produzione, introdurrebbe latenza in tutto il processo.

Oltre a ciò, già in questa fase si presentano due criticità rilevanti, legate alla concorrenza e alla persistenza dei dati, che evidenziano i limiti strutturali dell'approccio e confermano come il vettore dei messaggi non costituisca una scelta applicabile e duratura.

10.1.1 Concorrenza

All'avvio del server, Uvicorn consente di configurare il numero di *worker*, ovvero i processi paralleli utilizzati per eseguire l'applicazione. Questa replicazione sfrutta i *core* multipli della macchina per avviare più istanze simultanee, consentendo di gestire un maggior numero di richieste in parallelo e migliorando l'efficienza del servizio [22].

Avere processi paralleli impone di gestire la concorrenza: il vettore dei messaggi passa da risorsa esclusiva a risorsa condivisa, rendendo necessaria la sincronizzazione degli accessi con particolare attenzione alle operazioni di scrittura. Per assicurare che una determinata operazione critica possa essere eseguita da un solo processo per volta, sono stati introdotti diversi *lock* all'interno del flusso di lavoro, in modo da evitare sovrascritture indesiderate, inconsistenze nei dati o comportamenti non deterministici.

Nonostante i *lock* prevengano correttamente situazioni di competizione tra i processi e il verificarsi di *deadlock*, presentano alcuni svantaggi in termini di prestazioni e scalabilità. Poiché un *lock* impedisce l'accesso concorrente a una risorsa, più processi che tentano di accedervi contemporaneamente devono attendere il rilascio del blocco, introducendo potenziali rallentamenti e colli di bottiglia. Questo comportamento diventa particolarmente evidente in scenari con un numero elevato di richieste simultanee, in cui la contesa per il *lock* può degradare sensibilmente i tempi di attesa del sistema.

10.1.2 Persistenza

Fino a questa fase dello sviluppo, la gestione della persistenza dei dati era rimasta un aspetto secondario. Il vettore dei messaggi risiedeva all'interno dell'oggetto *RagExecutor*, ereditandone interamente il ciclo di vita; ciò significava che le conversazioni rimanevano disponibili solo finché l'applicazione era in esecuzione. In caso di guasti, riavvii del server o semplici aggiornamenti dell'ambiente, tutte le interazioni in corso venivano inevitabilmente perdute, rendendo impossibile recuperarle.

A questo limite si affiancava un problema di natura opposta. Poiché i messaggi venivano mantenuti in memoria fino allo spegnimento del server, restavano presenti nel vettore anche quando non erano più necessari: un utente poteva aver chiuso o riavviato l'interfaccia grafica, oppure abbandonato la conversazione, ma i dati associati rimanevano comunque caricati. Nel

lungo periodo, ciò comportava un consumo di memoria crescente e non giustificato, con conseguente degrado delle prestazioni e aumento dei tempi di elaborazione, soprattutto in presenza di un numero elevato di conversazioni parallele.

Queste criticità hanno evidenziato la necessità di introdurre un sistema di persistenza più robusto, in grado di separare il ciclo di vita delle conversazioni da quello dell'applicazione e di evitare l'accumulo incontrollato di dati in memoria.

10.2 Redis

Come alternativa, si è deciso di applicare una soluzione incentrata sul concetto di memoria *cache*.

La memoria *cache* è una memoria veloce, relativamente piccola, che memorizza i dati usati più di recente dalla memoria principale del sistema, in modo da velocizzare gli accessi e aumentare le prestazioni del sistema. Nell'ambito del progetto, non è stata utilizzata una memoria fisica in senso stretto, ma è stata implementata una soluzione a livello virtuale tramite Redis.

Redis è un database NoSQL chiave-valore *in-memory* con persistenza facoltativa, ampiamente utilizzato come *cache* distribuita e broker di messaggi. Salvando tutti i dati in memoria centrale, permette di eseguire le operazioni di lettura e scrittura con latenza minima, dimostrandosi particolarmente performante. Supporta tutti i principali tipi di strutture dati, come stringe, liste, *hash*, *sets* e oggetti JSON [57].

Redis sostituisce completamente il vettore dei messaggi, delegando al database la gestione dell'archiviazione e dell'accesso ai dati: i processi interagiscono direttamente con esso per eseguire operazioni di lettura e scrittura, eliminando la necessità di gestire manualmente la concorrenza. Le perfor-

mance complessive risultano migliorate, grazie all'elevata velocità di Redis nel gestire strutture dati in memoria.

Essendo un database NoSQL, in Redis non si utilizzano tabelle, e quindi non è possibile replicare direttamente quelle già utilizzare su PostgreSQL per la permanenza dei dati. Le informazioni sono organizzate tramite strutture dati indirizzate da chiavi composte, che seguono una convenzione a *name-space* per rappresentare in modo logico la relazione tra gli elementi. Ogni entità è memorizzata separatamente e identificata da una chiave univoca.

Tre diverse gerarchie logiche sono state impiegate per modellare le entità del progetto e la relazione presente tra esse: la prima contiene l'id e i metadati della conversazione, la seconda i singoli messaggi, mentre la terza contiene la lista ordinata degli id dei messaggi di una conversazione. Quest'ultima relazione è stata introdotta per sostituire il vincolo di *foreign key*, non applicabile direttamente in Redis a causa dell'univocità delle chiavi.

Redis offre una soluzione *built-in* anche al problema della persistenza dei dati. Attraverso il meccanismo di persistenza facoltativa, è possibile assegnare a una chiave un tempo di vita predefinito, trasformandola in *chiave volatile*; al termine del periodo specificato, essa viene automaticamente cancellata dal database, senza necessità di interventi manuali. Rimane sempre possibile aggiornare o resettare il tempo di vita tramite comando manuale.

Ogni chiave viene impostata come volatile, con un determinato TTL (*Time To Live)* specificato nel file di configurazione; ogni volta che l'utente compie un'azione per indicare che l'interazione è ancora in corso, come l'invio di una nuova domanda o di un commento, il TTL di tutte le chiavi associate ad essa è riportato al valore iniziale.

La classe *redis_manager* incapsula tutta la logica di interazione con Redis. Inizializzato all'interno di *RagExecutor*, l'oggetto consente di salvare una nuova conversazione, accedere ai messaggi di una conversazione già esistente, e aggiungere i messaggi di contesto, la domanda dell'utente e la risposta nella

conversazione corrente.

10.2.1 Conclusioni

La sostituzione del vettore dei messaggi con un sistema di archiviazione centralizzato ha infatti risolto in modo strutturale le criticità precedentemente riscontrate, migliorando stabilità, efficienza e scalabilità dell'intero sistema.

Il primo vantaggio è la eliminazione totale dei problemi di concorrenza. L'inserimento di Redis come intermediario ha reso superflua la gestione manuale dei *lock*: l'accesso ai dati è ora affidato a un componente esterno progettato per gestire operazioni concorrenti con meccanismi nativi di atomicità, portanto a un codice più semplice, più sicuro e meno soggetto a errori difficili da tracciare.

Parallelamente, la migrazione su Redis ha introdotto un miglioramento rilevante nella gestione della persistenza temporale dei dati. Grazie al meccanismo delle chiavi volatili e del TTL, le conversazioni presenti nel database vengono automaticamente eliminate dopo un periodo configurabile di inattività; ciò garantisce una forma di pulizia automatica delle conversazioni obsolete, riducendo l'uso di memoria e mantenendo il database leggero e reattivo. Inoltre, il TTL può essere aggiornato a ogni interazione, permettendo di conservare soltanto le conversazioni ancora attive.

Un ulteriore miglioramento riguarda la scalabilità e la modularità del sistema: essendo Redis un database NoSQL *in-memory*, consente tempi di accesso estremamente rapidi, indipendentemente dal numero di conversazioni presenti, permettendo di gestire un carico crescente di richieste senza impattare significativamente sulle performance.

La scelta di strutture dati separate e indirizzate tramite chiavi composte rende inoltre il modello facilmente espandibile: nuove tipologie di metada-

ti o relazioni tra entità possono essere integrate senza modificare strutture preesistenti o migrare database.

Infine, l'adozione di un componente ampiamente utilizzato nel settore introduce anche una maggiore affidabilità e compatibilità con strumenti e librerie esterne.

Capitolo 11

Docker

Al fine di rendere il servizio più facilmente distribuibile, sia su architetture diverse, sia in previsione del deployment vero e proprio, si è deciso di containerizzare l'intera applicazione tramite Docker.

Docker è una piattaforma per lo sviluppo, la distribuzione e l'esecuzione di applicazioni; le separa dall'infrastruttura sottostante, consentendo una rapida e semplice distribuzione del software [15]. Docker offre la possibilità di impacchettare ed eseguire l'applicazione in un ambiente leggero e isolato, il *container*, il quale contiene tutti i componenti e le dipendenze necessarie per la corretta funzione del servizio [16]. Ogni container è isolato e indipendente sia dagli altri, sia dalla macchina host, il che li rende altamente portabili e condivisibili, ideali per il deployment sia su macchine virtuali che su cloud.

Per implementare la comunicazione tra i diversi container è stato utilizzato Docker Compose, uno strumento che consente di definire ed eseguire applicazioni multi-container in modo semplice e strutturato [14]. Attraverso un unico file di configurazione in formato YAML è possibile gestire l'intero *stack* applicativo, specificando servizi, reti e volumi esterni.

Questo approccio permette di avviare, configurare e collegare automaticamente i container, mantenendo un ambiente di esecuzione riproducibile e facilmente scalabile; inoltre, consente di testare in modo integrato l'interazio-

ne tra i servizi, ricreando localmente un ambiente del tutto analogo a quello di produzione [13].

11.1 Immagini

Per il progetto sono stati definiti due container distinti, ciascuno con uno scopo preciso.

Il primo container ospita Redis; poiché la società stessa distribuisce e mantiene un'immagine Docker ufficiale dell'ultima versione disponibile, è stato sufficiente importare tale immagine nello spazio di lavoro, senza ulteriori personalizzazioni.

Il secondo container contiene invece la logica di business del progetto, ovvero il server Python e tutte le classi descritte in precedenza; per la costruzione dell'immagine è stata scelta come base *python:slim*, in modo da ottenere un ambiente leggero ma completo. Durante la fase di build vengono copiati nel container i file sorgente, le configurazioni e il file *requirements.txt*, da cui vengono installate tutte le librerie necessarie all'esecuzione.

Il file di configurazione, necessario anche in questo contesto, presenta alcune differenze rispetto alla versione utilizzata in ambiente locale: vengono aggiornate le specifiche di connessione a PostgreSQL, che da servizio locale diventa un servizio remoto, e vengono aggiunti i riferimenti per l'accesso al container di Redis. Per evitare di ricostruire l'immagine a ogni modifica delle configurazioni, esso viene montato a runtime come volume, anziché essere incluso direttamente nell'immagine, così da consentire aggiornamenti dinamici senza ripetere il processo di build; il volume viene montato in una directory dedicata, dalla quale il server Python legge il file all'avvio. In questo modo, eventuali modifiche alle configurazioni vengono applicate semplicemente riavviando il container, senza alcun impatto sull'immagine o sul processo di deployment.

Un aspetto rilevante riguarda la gestione delle configurazioni di OCI all'interno del container, in particolare del file *config* e della chiave privata utilizzata per l'autenticazione. Secondo le specifiche di Oracle, il file *config* deve risiedere in un percorso fisso per poter essere correttamente individuato durante la fase di autenticazione, motivo per cui esso viene copiato direttamente all'interno dell'immagine durante la fase di build; la chiave privata, invece, è collocata nella stessa directory dei file di configurazione locali e viene montata a runtime, garantendo maggiore flessibilità e sicurezza. All'interno del file *config* è presente un riferimento esplicito al percorso della chiave, necessario per completare il processo di autenticazione.

11.2 Docker Compose

Una volta predisposti i container necessari, l'interazione tra essi è stata definita tramite Docker Compose, che funge da orchestratore dell'intero ambiente di esecuzione.

All'interno del file di orchestrazione *docker-compose.yaml*, per ciascun servizio sono specificate l'immagine Docker da utilizzare e le porte da esporre verso l'esterno, tramite cui è possibile stabilire canali di comunicazione sia tra i container stessi, sia tra i container e servizi esterni, come il database PostgreSQL, garantendo così un'integrazione coerente e facilmente gestibile dell'intero sistema.

Il servizio dedicato al server Python richiede alcune configurazioni aggiuntive, come un volume montato a runtime, necessario per caricare dinamicamente le configurazioni, e la dipendenza dal servizio di Redis, che, tramite il meccanismo di *service dependency*, ne assicura l'avvio automatico prima dell'inizializzazione del server stesso; questa sincronizzazione evita errori dovuti a tentativi di connessione prematuri e garantisce la corretta disponibilità della *cache* sin dai primi istanti di esecuzione.

Non essendoci necessità di preservare i dati presenti su Redis tra una sessione e l'altra, in quanto usato esclusivamente come memoria volatile, non sono stati configurati volumi dedicati alla persistenza dei dati, e tutte le informazioni vengono automaticamente eliminate all'arresto del container, riducendo la complessità del sistema e assicurando un comportamento pulito a ogni riavvio.

11.3 Macchina Linux

L'impiego combinato di immagini Docker e di Docker Compose consente di trasferire il servizio in modo rapido e affidabile sulla macchina Linux dedicata. Il processo di migrazione risulta notevolmente semplificato: è sufficiente copiare l'immagine del server Python e il file *docker-compose.yaml* dall'ambiente di sviluppo, quindi scaricare l'immagine ufficiale di Redis direttamente dal registro Docker. Grazie all'isolamento e alla standardizzazione garantiti dai container, il servizio può essere eseguito in maniera del tutto speculare all'ambiente Windows, assicurando la portabilità, la consistenza e la riproducibilità del deployment su qualsiasi sistema compatibile con Docker.

Capitolo 12

Autenticazione

Nel corso dello sviluppo è emersa la necessità di introdurre un nuovo requisito, l'autenticazione degli utenti. Infatti, una volta terminato, il chatbot non sarà erogato come servizio separato, bensì verrà integrato direttamente all'interno degli applicativi aziendali. Di conseguenza, è necessario implementare un meccanismo di verifica dell'identità che lo renda disponibile solo a un sottogruppo dei clienti esistenti, per poterlo vendere come estensione facoltativa e separata.

Sono stati considerati diversi meccanismi per implementare un livello di autenticazione, tra cui Basic Authentication, OAuth2.0, e autenticazione via API key.

Basic Authentication è lo schema base per l'autenticazione via HTTP: le credenziali consistono in una coppia id/password, codificate utilizzando la codifica base64. Poiché la codifica base64 è facilmente reversibile, trasmettere le credenziali in rete non è sicuro, in quanto potrebbero essere intercettate da terzi: per garantire la riservatezza delle informazioni, è fondamentale utilizzare anche un protocollo crittografato, come TLS o HTTPS. Inoltre, Basic Authentication è particolarmente vulnerabile agli attacchi *Cross-Site Request Forgery* (CSRF), poiché le credenziali utente vengono inviate in tutte le richieste, indipendentemente dall'origine [11].

OAuth2.0, acronimo di *Open Authorization*, è un meccanismo di autenticazione *token-based*, progettato per consentire a un'applicazione di accedere alle risorse ospitate da altre web app per conto di un utente. Fornisce l'accesso con consenso e limita le azioni che l'utente può eseguire sulle risorse, senza mai condividerne le credenziali, e rappresenta lo standard dell'autenticazione online.

OAuth2.0 incorpora *Access Token* nel suo funzionamento, ossia un dato che rappresenta l'autorizzazione ad accedere a una data risorsa da parte di un dato utente. Quando un utente ha necessità di accedere a una risorsa protetta, invia una richiesta al suo gestore, includendo anche il proprio *Access Token*; il gestore verifica sua validità e, in caso positivo, permette l'accesso alla risorsa [5, 55].

Una API key è un identificatore univoco assegnato da un fornitore di servizi API a un utente, con lo scopo di autenticare e monitorare le richieste effettuate. Deve essere inclusa in ogni chiamata API; se la chiave è valida, l'accesso al servizio viene autorizzato. Nonostante non sia lo standard più sicuro sul mercato, è molto diffuso per la sua semplicità di implementazione [55].

La scelta finale è ricaduta sull'autenticazione tramite API key: oltre ad essere conforme con le tecnologie e le politiche aziendali, richiede cambiamenti minimi nell'architettura e nelle API già implementati. Dato che il chatbot sarà integrato in software preesistenti, già provvisti di meccanismo di autenticazione via credenziali, i rischi posti alla sicurezza sono minimi.

Ad ogni cliente pagante verrà assegnata la propria API key. Con cliente pagante si intende l'intero ente che usufruisce di un applicativo dell'azienda; ogni singolo utente all'interno dello stesso ente utilizzerà la stessa chiave di accesso.

12.1 Implementazione

La versione descritta di seguito non rappresenta quella definitiva destinata alla produzione, in quanto non pienamente in linea con standard aziendali. Si tratta di una versione provvisoria, concepita come *placeholder* per fornire uno scheletro di base al sistema e verificarne il corretto funzionamento.

12.1.1 API Key

È stata utilizzata una singola API key, composta da due campi: un id pubblico, breve e non segreto, che identifica il cliente, e la chiave segreta vera e propria, un token a 64-bit generato in maniera casuale.

Per gestire queste credenziali è stata predisposta una tabella dedicata su PostgreSQL. Salvare una chiave in chiaro rappresenterebbe una pratica estremamente rischiosa per la sicurezza in quanto, se il database fosse compromesso, gli attaccanti avrebbero libero accesso a tutte le informazioni dei clienti; per questo motivo, ad essere memorizzata non la chiave stessa, bensì una sua *rappresentazione* sicura, generata applicando una funzione di *hashing* al segreto originale.

L'algoritmo scelto è Argon2, sviluppato da un team dell'Università del Lussemburgo e vincitore della Password Hashing Competition del 2015. In particolare, per generare la rappresentazione sicura della chiave è stata utilizzata la sua variante *Argon2id*, che offre un'elevata protezione sia contro attacchi basati sulla memoria che contro attacchi paralleli [3].

Argon2id non è invertibile: l'algoritmo contiene un elemento di randomizzazione, il *salt*, che assume un valore diverso ad ogni utilizzo; di conseguenza, l'*hash* di una stessa parola cambia ad ogni applicazione della funzione, rendendo di fatto impossibile risalire al valore originale e aumentando significativamente la robustezza contro attacchi tramite *rainbow tables* o *brute force*.

All'interno della tabella dedicata, l'id pubblico funge da *primary key* e permette di recuperare l'*hash* corrispondente per confrontarlo con la API key ricevuta e verificarne la validità. Altri campi contengono dati di utilità, come data di creazione, data di scadenza, e stato attuale della chiave.

12.1.2 API Key Manager

La chiave segreta viene inserita in un campo dedicato dell'*header* delle richieste HTTP, mentre la firma delle tre API è stata aggiornata per includere la gestione delle chiavi, completamente integrata in FastAPI.

La logica di validazione è stata incapsulata nella classe *api_key_manager*, che verifica la validità delle chiavi ricevute avvalendosi della libreria Python ufficiale *argon2*: ad ogni richiesta, il sistema recupera dal database, tramite l'ID pubblico, l'*hash* associato alla chiave e la relativa data di scadenza.

La chiave è considerata valida solo se soddisfa due condizioni: c'è corrispondenza tra l'*hash* della chiave fornita e l'*hash* memorizzato, e la data corrente precede la data di scadenza. La verifica della corrispondenza avviene tramite la funzione di libreria *verify*, che confronta in modo sicuro la chiave ricevuta con l'*hash* memorizzato, utilizzando i parametri e il *salt* originari; la validità temporale viene invece controllata tramite un semplice confronto tra *timestamp*.

Se entrambe le condizioni risultano soddisfatte, l'utente è autorizzato all'accesso e la sua richiesta prosegue attraverso il normale flusso di elaborazione del sistema. In caso contrario, l'endpoint interrompe l'esecuzione restituendo un messaggio di errore, garantendo così un livello di sicurezza adeguato e conforme alle buone pratiche nella gestione delle API key.

L'immagine 12.1 mostra il diagramma di sequenza aggiornato del chatbot come servizio, contenente anche la nuova classe *api_key_manager*, nel caso in cui l'autenticazione abbia esito positivo.

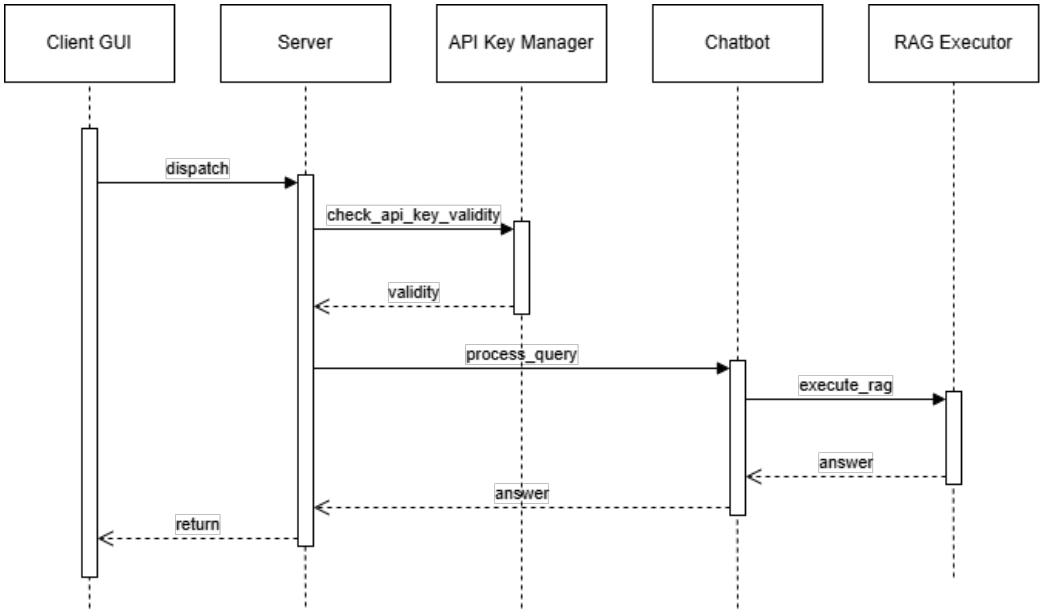

Figura 12.1: Diagramma di sequenza con autenticazione positiva.

Capitolo 13

Function Calling

Come già menzionato nella sezione *Normativa* del Capitolo 4, l'aggiunta dei documenti di normativa in tabelle distinte impone l'implementazione di un meccanismo dinamico di classificazione delle query, in modo che il modulo di *retrieval* possa essere eseguito di volta in volta sulla tabella più appropriata alla domanda ricevuta.

Oltre a ciò, il chatbot mostra alcune difficoltà nel gestire le richieste di approfondimento relative alle proprie risposte precedenti. In particolare, quando il modello fornisce una risposta articolata in punti e l'utente richiede ulteriori chiarimenti su uno di essi, il sistema non riesce a stabilire correttamente il riferimento contestuale; tende invece a restituire un contenuto generico associato al numero indicato, attingendo alla propria conoscenza di base piuttosto che al contenuto specifico della risposta precedente. Questo aspetto compromette la correttezza delle risposte e la qualità delle interazioni con gli utenti, necessitando un intervento mirato di correzione e mitigazione.

Il costrutto che permette di risolvere queste complicazioni è il *function calling*, una tecnica alternativa alle query tradizionali che permette al modello di interagire con il sistema e con il mondo esterno in maniera strutturata [19].

Il *function calling* fornisce al modello una serie di funzioni esterne, i *tool*, ed è il modello stesso a decide quali di esse applicare, in base alla

situazione corrente e all'input ricevuto, restituendo i parametri necessari per l'invocazione della funzione prescelta [23].

Tale capacità estende significativamente le funzionalità degli LLM, superando la semplice generazione di testo e consentendo loro di interagire attivamente con il mondo reale. Grazie alla possibilità di effettuare chiamate a funzioni, gli LLM oltrepassano il limite del linguaggio naturale, e diventano in grado di eseguire azioni, controllare dispositivi, accedere a database per il recupero di informazioni e svolgere un'ampia gamma di attività mediante l'utilizzo di strumenti e servizi esterni [65, 2].

Tipicamente, le chiamate di funzione prevedono un'interazione strutturata tra il modello e le funzioni esterne, articolata in passaggi distinti e ben definiti. In primo luogo, è necessario specificare le dichiarazioni delle funzioni che si intende rendere disponibili al modello. Successivamente, esso analizza l'input fornito dall'utente e i tool accessibili, individuando il più appropriato in relazione al contesto operativo; questa fase richiede l'impiego di un prompt di sistema specifico, finalizzato a orientare il processo decisionale del modello. Infine, vengono restituiti il nome e i parametri della funzione selezionata; il modello, tuttavia, non esegue direttamente la funzione, la quale deve essere richiamata in un momento successivo all'interno del flusso di lavoro, applicando i parametri forniti [2].

Non tutti gli LLM supportano nativamente il *function calling*; riconoscere quando un prompt richiede l'invocazione di una funzione, così come la scelta della funzione da chiamare, richiede un addestramento specifico [65]. Il modello Cohere Command R impiegato come motore di generazione all'interno del sistema non supporta tale meccanismo. Questa limitazione ha reso necessaria l'introduzione di un secondo LLM nel sistema, e ha portato allo sviluppo progressivo di tre diverse implementazioni, di seguito illustrate, ciascuna delle quali ha introdotto miglioramenti rispetto alla precedente, fino al raggiungimento della versione finale.

In tutti i casi, l'integrazione del concetto di *function calling* ha comportato modifiche sostanziali alla logica applicativa preesistente. È stato introdotto un secondo prompt di sistema, dedicato alla gestione e alla selezione dei tool, che fornisce al secondo modello le linee guida necessarie per orientare il processo decisionale. Parallelamente, un secondo prompt utente consente di specificare in modo più dettagliato le istruzioni operative, indicando i parametri da restituire in associazione alla funzione.

Inoltre, si è reso necessario modificare la classe *Chatbot*, che non richiama più direttamente la funzione di RAG, ma gestisce la creazione del secondo modello e l'invocazione del tool selezionato, coordinando l'intero flusso di interazione tra i due modelli e i relativi strumenti.

13.1 Tool

Per poter essere correttamente elaborate dal modello, le funzioni devono essere definite tramite uno schema preciso, al fine di specificarne tutte le caratteristiche fondamentali: nome, descrizione, casi d'uso, e parametri di ingresso con rispettivi nomi, tipi e descrizioni; più sono dettagliati e precisi gli schemi, più informazioni sono fornite al modello. Lo schema esatto da utilizzare dipende dal framework e dal modello in uso, e prevede una formattazione o l'utilizzo di classi specifiche.

Nel contesto del progetto sono stati sviluppati tre tool distinti, progettati per soddisfare le esigenze delineate in precedenza. Ciascuno di essi implementa un'operazione di RAG coerente con le modalità descritte nei capitoli precedenti; le differenze principali risiedono nelle procedure di pre-processamento applicate ai dati di input. I tool impiegati sono:

- **Manuals_rag**: tool per l'esecuzione del RAG classico, applicato alle domande di manualistica. È il tool di default del sistema, e non prevede alcuna modifica dei dati di input.

- **Regulations_rag**: tool per l'esecuzione del RAG classico, applicato alle domande di normativa. Al nome della categoria ricevuto via API è posposto il suffisso normativa; in questo modo, il modulo di *retrieval* recupera le informazioni a partire dalla tabella di normativa associata categoria specificata.

- **Deep_dive_rag**: tool per l'approfondimento, esegue il RAG prendendo come input il punto specificato della risposta precedente. Il modello inferisce il numero del punto da approfondire a partire dal messaggio utente; una specifica funzione di *parsing* recupera il testo della domanda precedente, lo suddivide nei singoli punti, estrae quello corrispondente e lo utilizza per riformulare la nuova query. L'ambito corrente (manualistica o normativa) viene mantenuto uguale a quello della domanda originaria, poiché si tratta di una richiesta di approfondimento riferita al medesimo contesto.

La classe *ToolManager* si occupa della gestione centralizzata dei tool nelle diverse versioni implementate. Essa include le descrizioni delle funzioni nei relativi formati specifici, le definizioni delle funzioni stesse e le logiche necessarie alla creazione e restituzione degli oggetti tool, istanziati come oggetti appartenenti a classi differenti in base alla versione utilizzata.

All'interno della classe *Chatbot* viene istanziato un oggetto della classe *ToolManager*, ereditato da tutte le implementazioni specifiche del chatbot, garantendo un accesso uniforme alle funzionalità di gestione e un comportamento coerente tra le diverse versioni del sistema.

13.2 Prima Versione

La prima versione del sistema è stata sviluppata interamente in appoggio al framework LlamaIndex, già adottato nelle altre sezioni del progetto. Il

framework rende disponibili un insieme di classi che incapsulano la logica necessaria all'implementazione del *function calling*, offrendo sia vantaggi sia alcune limitazioni. Da un lato, l'utilizzo di componenti già sviluppati e collaudati consente di semplificare il processo di sviluppo e di mantenere una maggiore coerenza architetturale con il resto del sistema. Dall'altro, il livello di personalizzazione risulta limitato e la comprensione approfondita del comportamento interno di tali componenti può risultare complessa, rendendo difficile ottimizzarne l'utilizzo in contesti specifici.

LlamaIndex struttura l'implementazione delle chiamate di funzione attorno al concetto di *Agent*, un'entità logica che integra un modello linguistico, una memoria conversazionale e un insieme di tool per la gestione dell'input esterno. Nello specifico, è stata impiegata un'istanza della classe *FunctionCallingAgent*, che richiede come argomenti una lista dei tool disponibili, il modello linguistico di grandi dimensioni, il prompt di sistema, e la memoria della conversazione. Come modello secondario per il *function calling* è stato utilizzato Llama3.2 remoto, in esecuzione remota sulla piattaforma Oracle OCI.

Una peculiarità degli *Agent* risiede nel fatto che, a differenza delle chiamate di funzione classiche, essi non si limitano alla selezione del tool appropriato, ma ne eseguono automaticamente la funzione associata, rielaborando il risultato prima di restituirlo all'utente finale. Inoltre, gli *Agent* richiedono che la memoria delle interazioni precedenti venga passata esplicitamente come argomento, al fine di mantenere la coerenza contestuale durante l'elaborazione.

I tool sono implementati come istanze della classe *FunctionTool*, la quale, nella sua versione base, accetta come argomenti la funzione da eseguire, il nome identificativo, e una descrizione contenente i relativi casi d'uso. La classe ToolManager è responsabile della creazione dei *FuntionTool* e, tramite chiamata di funzione, li restituisce a *Chatbot*, che li utilizza successivamente

per l'inizializzazione dell'*Agent*.

Nonostante l'elevato grado di automazione offerto da questa architettura, la versione sviluppata non soddisfa pienamente le aspettative. L'esecuzione automatica e immediata dei tool, priva di controllo esterno, limita la possibilità di gestire in modo autonomo le diverse fasi del processo e gli output intermedi. Inoltre, l'output della fase di RAG viene rielaborato integralmente dal modello secondario, con un conseguente peggioramento della qualità delle risposte finali; a ciò si aggiunge un incremento significativo dei tempi di risposta, dovuto alla rielaborazione testuale aggiuntiva, che compromette la fluidità dell'interazione e non risulta conforme alle aspettative dell'utente.

Alla luce di queste considerazioni, si è deciso di accantonare questa prima implementazione e di procedere con lo sviluppo di una versione migliorata, capace di garantire maggiore controllo sul flusso operativo e prestazioni più adeguate agli obiettivi del sistema.

13.3 Seconda Versione

Al fine di ottenere un maggiore controllo sul flusso operativo e più personalizzazione, si è deciso di adottare una seconda soluzione, più flessibile e indipendente da classi esterne predefinite. La logica di base della prima implementazione è stata mantenuta, così come la struttura dei prompt di sistema e utente e i tool già definiti in precedenza.

In questa versione, il modello linguistico secondario è Gemini 2.5 Flash, sviluppato e addestrato da Google, che offre pieno supporto nativo alle funzionalità di *function calling*. Il modello è accessibile tramite API mediante una API key, la cui versione gratuita consente un utilizzo limitato in termini di numero di richieste e di token elaborabili.

Per agevolare l'integrazione dei propri modelli all'interno di applicazioni Python, Google ha rilasciato la libreria *google.generativeai* (*genai*), che forni-

sce un'interfaccia semplificata per la creazione di istanze locali di LLM, la gestione delle conversazioni e l'invio di query ai modelli disponibili. Tale libreria è stata utilizzata come base per l'interazione diretta con il modello Gemini, sostituendo così la precedente logica basata sugli *Agent* di LlamaIndex.

Dal punto di vista architetturale, la richiesta dell'utente viene innanzitutto processata dal modello secondario, che ne effettua l'analisi semantica e determina quale dei tool disponibili chiamare, restituendo i parametri. A differenza della versione precedente, Gemini non esegue direttamente la funzione associata al tool selezionato, ma si limita a restituire il nome del tool da invocare e i relativi argomenti. L'esecuzione effettiva della funzione avviene in un secondo momento, all'interno del flusso applicativo principale, mentre l'output prodotto dal modello primario viene restituito direttamente all'utente, senza ulteriori rielaborazioni testuali.

Questa scelta architetturale ha consentito di ridurre significativamente i tempi di risposta e di preservare un elevato standard di qualità nel testo generato, rappresentando un miglioramento sostanziale rispetto alla versione precedente in termini di efficienza, chiarezza e controllo dei risultati.

Nonostante le prestazioni tecniche soddisfacenti, questa soluzione presenta alcune criticità di natura gestionale. Al momento dello sviluppo, il modello Gemini non risultava disponibile tra quelli integrabili tramite la piattaforma Oracle, già utilizzata per l'hosting del modello primario. L'uso della versione gratuita, limitata nelle risorse, non era compatibile con un ambiente di produzione, mentre l'adozione del piano *enterprise* avrebbe comportato la stipula di un nuovo contratto e un incremento dei costi operativi per l'azienda.

Alla luce di tali considerazioni, si è quindi deciso di proseguire con lo sviluppo di una terza implementazione, finalizzata a mantenere i vantaggi introdotti da questa soluzione, eliminandone al contempo le limitazioni infrastrutturali.

13.4 Terza Versione

La terza e ultima versione riprende i principi progettuali definiti nelle implementazioni precedenti, migliorandone l'architettura e garantendo al contempo una piena integrazione con l'infrastruttura già in uso nel progetto. Dal punto di vista logico, questa versione risulta sostanzialmente speculare a quella basata su Gemini 2.5 Flash, ma ne adatta l'approccio per renderlo pienamente compatibile con l'ecosistema OCI.

Come già menzionato, il modello generativo Cohere Command R disponibile su Oracle non supporta direttamente il meccanismo di *function calling*. Tuttavia, la piattaforma consente l'accesso a una variante dello stesso modello, dotata di tale funzionalità e utilizzabile tramite l'interfaccia di inferenza generativa.

La differenza principale rispetto alle versioni precedenti riguarda la modalità di istanziazione del modello. Invece di utilizzare un'istanza diretta, come nel caso base, il modello viene richiamato attraverso un oggetto *GenerativeAiInferenceClient*, configurato con le stesse credenziali OCI impiegate dal resto del sistema.

Prima di poter essere passati come argomenti al modello, i tool devono essere incapsulati nel formato specifico richiesto. A tal fine, vengono creati tre oggetti *CohereTool*, che contengono le stesse informazioni strutturali definite nelle versioni precedenti, organizzate secondo lo schema previsto dal modello Cohere per il corretto riconoscimento e utilizzo.

L'interazione con il modello non avviene più tramite i metodi forniti dal framework LlamaIndex, bensì attraverso le API native messe a disposizione da OCI, specificando il modello da utilizzare tramite la classe *OnDemandServingMode*, sfruttando direttamente le funzionalità di inferenza offerte da Oracle e riducendo la complessità complessiva del codice.

Grazie a queste ottimizzazioni, la terza implementazione offre un servizio

più rapido, stabile e coerente con il resto dell'architettura software, eliminando al contempo la necessità di ricorrere a soluzioni esterne e mantenendo nulli i costi aggiuntivi a livello aziendale. L'integrazione diretta con Oracle garantisce inoltre una maggiore affidabilità operativa e una scalabilità nativa, rendendo questa versione la più adatta per un impiego in ambiente di produzione.

La figura 13.1 illustra il flusso completo del chatbot con *function calling* nella sua versione finale, con le nuove classi che determinano quale tool utilizzare sulla base dell'input utente.

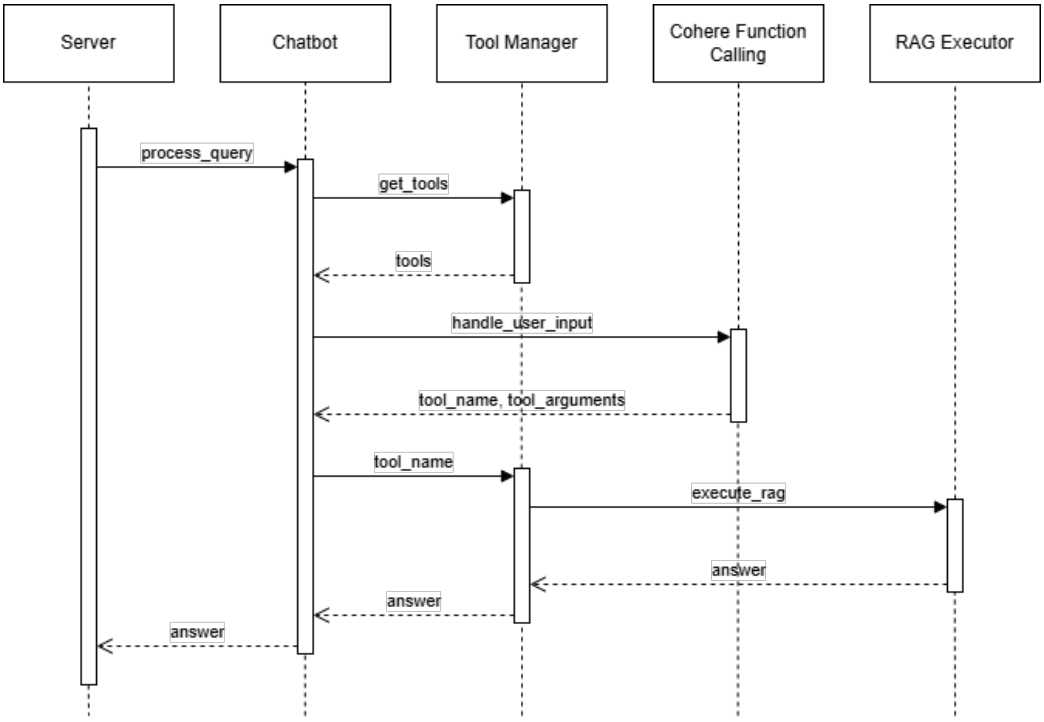

Figura 13.1: Diagramma di sequenza con *function calling*.

Capitolo 14

Deployment

L'adozione dei container Docker consente di isolare l'applicativo e le sue dipendenze, ma non offre un meccanismo di gestione automatizzata dell'esecuzione, della scalabilità o della continuità operativa. Per questo motivo si è reso necessario introdurre un sistema di orchestrazione, individuato in Kubernetes al fine di mantenere coerenza con gli standard infrastrutturali aziendali. Esso, infatti, è stato adottato come piattaforma di orchestrazione predefinita per l'intero ecosistema di progetti, favorendo l'impiego dello stesso ambiente anche in questo contesto.

Le sezioni seguenti illustrano più in dettaglio le scelte implementative prese, i tool utilizzati, e i risultati raggiunti a livello progettuale e implementativo.

14.1 Kubernetes

Kubernetes è una piattaforma open-source, portabile ed estensibile per la gestione di carichi di lavoro e servizi containerizzati. Supporta nativamente sia la configurazione dichiarativa sia l'orchestrazione automatizzata, consentendo di delegare al sistema operazioni fondamentali quali replica, scalabilità, monitoraggio e ripristino dei container. In questo modo, Kubernetes gesti-

sce in maniera autonoma l'intero ciclo di vita dell'applicazione, garantendo stabilità e continuità operativa [37].

Tra le principali astrazioni offerte dal sistema figurano i *pod*, ovvero le unità minime di esecuzione distribuibili dal cluster. Ogni *pod* può contenere uno o più container strettamente accoppiati, che condividono lo stesso spazio di rete e le medesime risorse di storage; esso rappresenta un *host* logico specifico per un componente applicativo, in una modalità analoga al livello di astrazione introdotto da *docker-compose* rispetto ai container individuali [39].

I *pod* vengono eseguiti all'interno dei *nodi di calcolo*, ovvero macchine fisiche o virtuali dedicate all'esecuzione delle unità applicative. Ogni nodo include i servizi necessari per l'esecuzione dei pod ed è gestito da un *control plane*, il livello di orchestrazione che espone l'API del cluster e coordina tutte le attività di scheduling, gestione dello stato desiderato e controllo del ciclo di vita dei container. L'insieme del piano di controllo e dei nodi di calcolo costituisce un *cluster* Kubernetes, ovvero l'ambiente distribuito nel quale vengono eseguite e gestite le applicazioni containerizzate [38].

14.2 Helm

Per gestire in modo strutturato il rilascio di un'applicazione si utilizza la risorsa *deployment*, che specifica il template dei *pod* e il numero di repliche richieste. Tuttavia, i file YAML dei deployment sono statici: ogni modifica richiede una nuova definizione o la generazione dinamica dei file tramite strumenti più flessibili.

A questo scopo viene introdotto Helm, il *package manager* di Kubernetes, che permette di raccogliere in pacchetti tutte le risorse necessarie all'installazione e di parametrizzare i file YAML attraverso un sistema di *template*. Helm

consente quindi di gestire configurazioni complesse, assegnare una versione ai vari rilasci ed effettuare aggiornamenti in modo controllato.

In Helm, l'unità fondamentale di distribuzione è il *chart*, un archivio compresso che contiene tutti i file necessari per il deployment dell'applicativo e delle risorse ad esso associate. Un *chart* gestisce congiuntamente sia la componente applicativa sia l'infrastruttura Kubernetes sottostante [28].

Per rendere l'applicativo accessibile dall'esterno, Helm può includere nel *chart* la definizione di un *service*, ovvero l'astrazione Kubernetes che espone uno o più *pod* verso l'esterno del *cluster*. Essendo soggetti a continui cicli di creazione e distruzione, gli indirizzi IP dei *pod* sono effimeri e non è possibile indirizzare direttamente il traffico verso di essi; è il *service* a garantire un endpoint stabile, mantenendo l'associazione tra le richieste in ingresso e i *pod* disponibili attraverso un meccanismo basato su label, anzichú su indirizzi IP [28].

Helm supporta inoltre la definizione di dipendenze tra chart, permettendo di comporre applicazioni complesse e riutilizzare i *chart* ufficiali memorizzati in appositi repository, concettualmente analoghi ai *registry* Docker, dai quali recuperare componenti standard come database, broker o sistemi di monitoraggio. In questo modo, l'intero processo di deployment risulta modulare, riutilizzabile e facilmente estendibile [27].

La creazione iniziale del chart ha prodotto diversi file fondamentali, tra cui *Chart.yaml*, *values.yaml*, *deployment.yaml* e *service.yaml*. A partire da tale struttura di base sono state introdotte una serie di modifiche per adattare il template alle esigenze del sistema. In particolare, dopo la pubblicazione dell'immagine Docker del server nel *registry* aziendale, essa è stata impostata come immagine principale del *cluster* Kubernetes; inoltre, il numero di repliche del *pod* è stato incrementato da una a tre per garantire maggiore disponibilità e resilienza.

Poiché il chatbot dipende da un'istanza Redis per il corretto funzionamen-

to, è stato necessario dichiarare una dipendenza verso il suo *chart* ufficiale all'interno del file *Chart.yaml* del progetto; Helm provvede automaticamente a scaricare il *chart* dal repository indicato e a ricompilarne la configurazione.

L'immagine Docker dell'applicativo richiede inoltre la presenza di diversi file di configurazione, nello specifico un file YAML, uno script Python e una chiave privata, che devono essere montati come volumi all'interno del container. L'operazione, più articolata rispetto a una configurazione Docker tradizionale, è stata gestita interamente tramite Helm: in primo luogo, i file di configurazione sono stati inseriti nella directory *config* del *chart*, al fine di unirli all'ambiente di deployment; successivamente, per ciascun file è stato definito un parametro dedicato in *values.yaml*, contenente per i primi due elementi il percorso dei file, per il terzo il valore della chiave privata.

Sulla base di tali parametri sono state create altrettante *ConfigMap*, che rappresentano l'astrazione Kubernetes utilizzata per associare coppie chiave-valore destinate ai container. Durante la loro generazione, i due file di configurazione vengono letti direttamente dai percorsi specificati dentro *chart*, mentre la chiave privata viene recuperata come valore statico. Infine, nella definizione di *deployment* sono stati configurati i volumi da montare all'interno del container, indicando sia i percorsi di destinazione all'interno dell'immagine, sia le associazioni tra ciascun volume e la relativa *ConfigMap*.

In questo modo, il file *deployment.yaml* integra sia le mappature tra volumi e *ConfigMap*, sia i punti di montaggio all'interno del container, garantendo un trasferimento coerente e sicuro dei file di configurazione necessari all'esecuzione dell'applicativo.

14.3 Istio

Per rendere l'applicativo accessibile dall'esterno del cluster è stato introdotto Istio, un service mesh che permette di gestire in maniera avanzata il

traffico in ingresso e uscita. Istio garantisce la resilienza dei sistemi distribuiti e cloud-native, abilitando controlli di sicurezza, gestione delle policy e controllo degli accessi, e ripristino in caso di errori [33].

L'adozione di Istio ha richiesto alcune modifiche sia lato applicazione, sia lato configurazione Kubernetes. Sul server FastAPI del chatbot è stato definito un nuovo contesto applicativo, identificato dal *path chatbotai*, utilizzato come prefisso per tutte le API del servizio. Tale contesto è stato integrato tramite un apposito *APIRouter*, e le firme delle API sono state aggiornate affinché includessero il nuovo *namespace*, rendendo le chiamate disponibili all'interno del *pod* tramite l'indirizzo *chatbotai/nome-api*.

Lo stesso contesto è stato poi propagato nella configurazione Kubernetes: è stato definito come parametro nel file *values.yaml* e successivamente incorporato nel file *istio.yaml*, in cui Istio viene configurato per esporre il servizio verso l'esterno. Una volta specificati il contesto e la porta su cui il *pod* è in ascolto, il tool provvede a mappare il processo applicativo interno con l'host pubblico generato al momento del deployment, operando in sinergia con *service.yaml* per l'instradamento delle richieste in ingresso. Istio si occupa di ricevere le chiamate esterne e reindirizzarle correttamente verso i *pod* corrispondenti, applicando il contesto applicativo come regola di *routing*.

L'introduzione di questo livello ha comportato anche una modifica alla modalità di collegamento con Redis: non è più previsto un riferimento diretto tra due immagini Docker, bensì una comunicazione tra i due *pod* all'interno del *cluster*. La porta rimane invariata, ma il nome del servizio viene aggiornato al nome del *pod* Redis generato da Kubernetes, in modo che l'applicativo possa risolvere correttamente l'indirizzo attraverso il DNS interno del *cluster*.

In conclusione, l'introduzione di Istio e l'integrazione del contesto applicativo all'interno dell'infrastruttura Kubernetes hanno permesso di strutturare un sistema di deployment completo e pienamente conforme agli standard aziendali. Queste implementazioni hanno contribuito a una maggiore coe-

renza architetturale, consentendo al sistema di operare in maniera scalabile, modulare e allineata alle moderne pratiche di deployment di servizi.

Conclusioni

In questo elaborato sono state descritte in modo sistematico le fasi di progettazione, sviluppo e validazione di un chatbot destinato all'assistenza aziendale. A partire dalla documentazione interna, relativa ai software già commercializzati, è stata costruita la base di conoscenza del sistema mediante una pipeline RAG, analizzando e confrontando differenti tecniche di estrazione e codifica delle informazioni, con l'obiettivo di individuare quelle più adeguate al contesto applicativo.

Sono stati valutati diversi *Large Language Model* come motore generativo del sistema, sottoponendoli a test strutturati e riproducibili al fine di determinare il modello con le prestazioni più elevate; il modello selezionato è risultato essere Cohere Command R. Per massimizzarne l'efficacia generativa, sono stati progettati un prompt di sistema e un prompt utente dedicati, in grado di vincolare il comportamento del chatbot alle specifiche funzionali richieste.

Per garantirne l'effettiva utilizzabilità in produzione, il chatbot è stato integrato in un'architettura erogata come servizio e accessibile tramite API. Sono stati implementati tutti i moduli necessari alla pipeline RAG e alla corretta restituzione dell'output generato; è stato inoltre introdotto un database Redis come memoria *cache* delle conversazioni. Al fine di limitare l'accesso al servizio a un gruppo selezionato di clienti, le API sono state estese con un sistema di autenticazione e monitoraggio degli accessi tramite API key.

L'integrazione dei documenti di normativa nella base di conoscenza ha

richiesto l'adozione del meccanismo di *function calling*, il quale consente al chatbot di interagire con i servizi esterni e adattare dinamicamente il proprio comportamento in base all'input ricevuto.

Infine, è stata inoltre progettata la distribuzione dell'intero sistema, prima tramite la creazione di un'immagine Docker, e successivamente tramite il deployment all'interno del *cluster* Kubernetes dell'azienda.

Parallelamente, è stato sviluppato un servizio per la memorizzazione e consultazione dello storico delle conversazioni, finalizzato ad attività di monitoraggio e analisi nel tempo. Mediante un servizio secondario, i dati raccolti sono visualizzabili su un'interfaccia dedicata, facilitandone l'accesso e l'interpretazione.

Per quanto riguarda gli sviluppi futuri, si pianifica di integrare il servizio all'interno dei software aziendali esistenti, con conseguenti modifiche ai meccanismi di autenticazione e alla struttura della base di conoscenza. Inoltre, si prevede l'introduzione di un'interfaccia utente rinnovata, coerente con il marchio aziendale e sviluppata in collaborazione con il reparto marketing. In una prima fase, il sistema verrà rilasciato a un gruppo ristretto di utenti *beta*, selezionati tra i clienti aziendali. I dati e i feedback raccolti durante questa fase pilota, come grado di utilizzo e livello di soddisfazione generale, saranno analizzati con l'obiettivo di stimare i costi effettivi di servizio, definire eventuali limiti di utilizzo e formulare una proposta di prezzo sostenibile e competitiva per il rilascio commerciale del servizio.

Considerato il maggior numero di LLM divenuto disponibile sulla piattaforma OCI tra la conclusione dello sviluppo e la stesura del presente elaborato, si ritiene opportuno valutare le prestazioni dei nuovi modelli, reiterando i test già condotti, al fine di individuare la soluzione più efficace all'interno di un panorama tecnologico più ampio. Essendo gli LLM molto sensibili alle istruzioni ricevute, un'eventuale sostituzione del modello adottato implicherebbe modifiche sostanziali a livello di prompt; redatti in riferimento diretto

a Cohere Command R, essi potrebbero non essere altrettanto efficaci con un modello differente, a causa delle loro peculiarità.

Oltre a ciò, il prompt di riformulazione, nella sua versione finale, non garantiva risultati accettabili in presenza di domande particolarmente complesse o che differivano significativamente dallo standard designato; in futuro, sarebbe utile introdurre nuovi meccanismi di arricchimento della query, per passare da una semplice conversione del verbo all'infinito a una trasformazione più profonda, che includa l'aggiunta di elementi utili al *retrieval*. Inoltre, il prompt potrebbe essere ottimizzato mediante l'inserimento di un numero più ampio e variegato di esempi di riformulazione corretti: oltre ai casi base, sarebbe utile includere scenari più complessi, come query che contengono più domande simultanee o situazioni in cui non è presente alcuna domanda esplicita. L'esposizione a una casistica più ricca consentirebbe di rafforzare la capacità di generalizzazione del modello e di rendere il processo robusto rispetto alle abitudini degli utenti reali.

Infine, sarebbe conveniente introdurre metriche di valutazione più avanzate rispetto agli indicatori automatici tradizionali, come le metriche semantiche (*BERTScore*, *BLEURT*), progettate per misurare la similarità del significato anziché la sola corrispondenza letterale, e le metriche specifiche per sistemi RAG (*precision/recall*, *context relevance*, *faithfulness*), utili a valutare non solo l'output finale ma anche l'efficacia dell'intero processo di recupero e utilizzo della conoscenza; entrambi gli approcci, non ancora implementati, hanno il potenziale di migliorare sia la qualità delle valutazioni interne sia la trasparenza complessiva del sistema.

Bibliografia

[1] Sahin Ahmed. *LLM Prompt Engineering for Beginners: What It Is and How to Get Started*. Consultato il 25 settembre 2025. Apr. 2024. URL: https://medium.com/%40sahin.samia/llm-prompt-engineering-for-beginners-what-it-is-and-how-to-get-started-0c1b483d5d4f.

[2] Google AI. *Function Calling â Gemini API Documentation*. Accesso: 13 novembre 2025. 2025. URL: https://ai.google.dev/gemini-api/docs/function-calling?hl=it&example=meeting.

[3] Dmitry Khovratovich Alex Biryukov Daniel Dinu. *Argon2*. Consultato il 28 settembre 2025. URL: https://www.argon2.com/.

[4] Amazon Web Services. *What Are the Benefits of RetrievalâAugmented Generation?* URL: https://aws.amazon.com/what-is/retrieval-augmented-generation/.

[5] Auth0. *What is OAuth 2.0*. Consultato il 26 settembre 2025. URL: https://auth0.com/intro-to-iam/what-is-oauth-2.

[6] Conor Bronsdon. *What is the Cost of Training LLM Models?* Accesso: 28 agosto 2025. Mar. 2025. URL: https://galileo.ai/blog/llm-model-training-cost.

[7] Katharina Buchholz. *The Extreme Cost Of Training AI Models*. Accesso: 28 agosto 2025. Ago. 2024. URL: https://www.forbes.com/

sites/katharinabuchholz/2024/08/23/the-extreme-cost-of-training-ai-models/.

[8] Chroma Docs. *Introduzione a Chroma*. Consultato il 6 settembre 2025. 2025. URL: https://docs.trychroma.com/docs/overview/introduction.

[9] Cohere. *Cohere Command R Documentation*. Consultato il 24 settembre 2025. URL: https://docs.cohere.com/v2/docs/command-r.

[10] Cohere. *Security â Cohere*. Consultato il 24 settembre 2025. URL: https://cohere.io/security.

[11] Mozilla Contributors. *HTTP Authentication*. Consultato il 26 settembre 2025. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Authentication.

[12] DeepLearning.AI. *Natural Language Processing*. Accesso: 29 agosto 2025. 2023. URL: https://www.deeplearning.ai/resources/natural-language-processing/.

[13] Inc. Docker. *Compose application model*. Consultato il 4 novembre 2025. 2025. URL: https://docs.docker.com/compose/intro/compose-application-model/.

[14] Inc. Docker. *Docker Compose*. Consultato il 4 novembre 2025. 2025. URL: https://docs.docker.com/compose/.

[15] Inc. Docker. *Docker overview*. Consultato il 4 novembre 2025. 2025. URL: https://docs.docker.com/get-started/docker-overview/.

[16] Inc. Docker. *What is a container?* Consultato il 4 novembre 2025. 2025. URL: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/.

[17] Jacob Eisenstein. *Natural Language Processing (Online Draft)*. Accesso: 29 agosto 2025. 2025. URL: https://web.stanford.edu/~jurafsky/slp3/.

[18] Omar Espejel. *Getting started with embeddings*. Lug. 2025. URL: https://huggingface.co/blog/getting-started-with-embeddings.

[19] Hugging Face. *Function Calling*. Accesso: 13 novembre 2025. 2025. URL: https://huggingface.co/docs/hugs/en/guides/function-calling.

[20] Hugging Face. *Mistral â Transformers documentation*. Consultato il 24 settembre 2025. URL: https://huggingface.co/docs/transformers/main/model_doc/mistral.

[21] MistralAI / Hugging Face. *Mistralâ7BâInstructâv0.2 Model Card*. Consultato il 24 settembre 2025. URL: https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2.

[22] FastAPI. *Server Workers – Uvicorn with Workers*. Consultato il 25 settembre 2025. URL: https://fastapi.tiangolo.com/deployment/server-workers/.

[23] Martin Fowler. *Function calling using LLMs*. Accesso: 13 novembre 2025. 2025. URL: https://martinfowler.com/articles/function-call-LLM.html.

[24] GeeksforGeeks. *Cosine Similarity*. Accesso: 13 novembre 2025. 2025. URL: https://www.geeksforgeeks.org/cosine-similarity/.

[25] GeeksforGeeks. *Understanding BLEU and ROUGE score for NLP evaluation*. Accesso: 13 novembre 2025. 2025. URL: https://www.geeksforgeeks.org/understanding-bleu-and-rouge-score-for-nlp-evaluation/.

[26] GeeksforGeeks. *What is LlamaIndex*. Accesso: 18 novembre 2025. Set. 2025. URL: https://www.geeksforgeeks.org/machine-learning/what-is-llamaindex/.

[27] Helm. *Chart Repository*. Accesso: 15 novembre 2025. 2025. URL: https://helm.sh/docs/topics/chart_repository.

[28] Helm. *Charts*. Accesso: 15 novembre 2025. 2025. URL: https://helm.sh/docs/topics/charts.

[29] Jiarun Huang. *A Deep Dive into AI: Large Models, Parameters, Tokens, Context Windows, Context Length and â|..* Consultato il 24 settembre 2025. Apr. 2024. URL: https://medium.com/@jiarunhuang/a-deep-dive-into-ai-large-models-parameters-tokens-context-windows-context-length-and-8187fb393acb.

[30] IBM. *Che cos'Ã¨ LangChain?* Accesso: 3 settembre 2025. Ott. 2023. URL: https://www.ibm.com/it-it/think/topics/langchain.

[31] IBM. *Che cosa sono i modelli linguistici di grandi dimensioni (LLM)?* Accesso: 28 agosto 2025. 2023. URL: https://www.ibm.com/it-it/think/topics/large-language-models.

[32] IMD. *LLMs explained: how Large Language Models in AI work*. Accesso: 31 agosto 2025. 2025. URL: https://www.imd.org/blog/digital-transformation/large-language-models-llms/.

[33] Istio. *The Istio service mesh*. Accesso: 15 novembre 2025. 2025. URL: https://istio.io/latest/about/service-mesh/.

[34] jwitsoe. *Explainer: What Is RetrievalâAugmented Generation?* Apr. 2024. URL: https://developer.nvidia.com/blog/explainer-what-is-retrieval-augmented-generation/.

[35] Rashmik Kabiraj. *How to Make LLMs Actually Listen â Prompt Engineering Guide*. Consultato il 25 settembre 2025. Giu. 2025. URL: https://community.ibm.com/community/user/blogs/rashmik-kabiraj/2025/06/03/how-to-make-llm-actually-listen-prompt-engineering.

[36] Divyanshu Khurana et al. «Natural language processing: state of the art, current trends and challenges». In: *Multimedia Tools and Applications* 82.3 (2023), pp. 3713–3744. DOI: 10.1007/s11042-022-13428-4. URL: https://doi.org/10.1007/s11042-022-13428-4.

[37] Kubernetes. *Cosè Kubernetes?* Accesso: 15 novembre 2025. 2025. URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/.

[38] Kubernetes. *Nodes*. Accesso: 15 novembre 2025. 2025. URL: https://kubernetes.io/docs/concepts/architecture/nodes/.

[39] Kubernetes. *Pods*. Accesso: 15 novembre 2025. 2025. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[40] *LangChain*. Accesso: 3 settembre 2025. URL: https://www.langchain.com/langchain.

[41] *LlamaIndex â Introduction*. Accesso: 3 settembre 2025. URL: https://docs.llamaindex.ai/en/stable/#introduction.

[42] Kim Martineau. *What is retrievalâaugmented generation (RAG)?* Ago. 2023. URL: https://research.ibm.com/blog/retrieval-augmented-generation-RAG.

[43] McKinsey & Company. *What is retrievalâaugmented generation (RAG)?* Ott. 2024. URL: https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-retrieval-augmented-generation-rag.

[44] Microsoft Azure. *What are Large Language Models (LLMs)?* Accesso: 31 agosto 2025. URL: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-are-large-language-models-llms%5C#How-it-works.

[45] Microsoft Learn Contributors. *RAG chunking phase*. Gen. 2025. URL: https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/rag/rag-chunking-phase.

[46] Milvus. *Che cosâÃ¨ la dimensionalitÃ degli embedding e come sceglierla?* Consultato il 5 settembre 2025. 2025. URL: %5Curl%7Bhttps://milvus.io/ai-quick-reference/what-is-embedding-dimensionality-and-how-do-you-choose-it%7D.

[47] Milad Moradi et al. «A Critical Review of Methods and Challenges in Large Language Models». In: *Computers, Materials and Continua* 82.2 (2025), pp. 1681–1698. ISSN: 1546-2218. DOI: https://doi.org/10.32604/cmc.2025.061263. URL: https://www.sciencedirect.com/science/article/pii/S1546221825000992.

[48] Nebuly. *LLM System Prompt vs. User Prompt*. Consultato il 25 settembre 2025. Ago. 2024. URL: https://www.nebuly.com/blog/llm-system-prompt-vs-user-prompt.

[49] nlmatics. *LLM Sherpa*. Accesso: 25 settembre 2025. 2025. URL: https://github.com/nlmatics/llmsherpa.

[50] Joshua Noble. *What is LLM Temperature?* Consultato il 24 settembre 2025. Dic. 2024. URL: https://www.ibm.com/think/topics/llm-temperature.

[51] Ollama. *Llama3.2:3b (blob dde5aa3fc5ff)*. Consultato il 24 settembre 2025. URL: https://ollama.com/library/llama3.2:3b/blobs/dde5aa3fc5ff.

[52] Oracle. *Generative AI Home â Oracle Cloud Infrastructure*. Consultato il 24 settembre 2025. URL: https://docs.oracle.com/en-us/iaas/Content/generative-ai/home.htm.

[53] pgvector (GitHub). *README di pgvector*. Consultato il 6 settembre 2025. 2025. URL: https://github.com/pgvector/pgvector/blob/master/README.md.

[54] PostgreSQL Global Development Group. *Informazioni su PostgreSQL*. Consultato il 6 settembre 2025. 2025. URL: https://www.postgresql.org/about/.

[55] Inc. Postman. *What Is API Authentication?* Consultato il 26 settembre 2025. URL: https://www.postman.com/api-platform/api-authentication/.

[56] Mohaimenul Azam Khan Raiaan et al. «A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges». In: *IEEE Access* 12 (2024), pp. 26839–26874. DOI: 10.1109/ACCESS.2024.3365742.

[57] Redis. *About Redis*. Consultato il 25 settembre 2025. URL: https://redis.io/about/.

[58] Sebastian Van Rooyen. *Understanding Vector Databases: A Beginner's Guide*. Gen. 2025. URL: https://dev.to/sebastiandevelops/understanding-vector-databases-a-beginners-guide-20nj.

[59] ADS S.p.A. *Soluzioni software per la Pubblica Amministrazione, la SanitÃ e le Aziende Private*. Accesso: 28 agosto 2025. 2025. URL: https://www.ads.it/soluzioni-software/.

[60] Vikram Singh. *Meet BLIP: The Vision-Language Model Powering Image Captioning*. Consultato il 12 settembre 2025. Ago. 2025. URL: https:

//pyimagesearch.com/2025/08/25/meet-blip-the-vision-language-model-powering-image-captioning/.

[61] Cole Stryker e Jim Holdsworth. *Cos'è l'NLP (elaborazione del linguaggio naturale)?* Accesso: 28 agosto 2025. 2024. URL: https://www.ibm.com/it-it/think/topics/natural-language-processing.

[62] Cole Stryker e Mark Scapicchio. *Che cos'è l'AI generativa?* Consultato il 28 agosto 2025. Mar. 2024. URL: https://www.ibm.com/it-it/think/topics/generative-ai.

[63] WiFi Talents. *AI in the Virtual Assistant Industry Statistics*. Accesso: 28 agosto 2025. Giu. 2025. URL: https://wifitalents.com/ai-in-the-virtual-assistant-industry-statistics/.

[64] Unstructured-IO. *Unstructured*. Accesso: 25 settembre 2025. 2025. URL: https://github.com/Unstructured-IO/unstructured.

[65] Analytics Vidhya. *Top 6 LLMs that Support Function Callings*. Accesso: 13 novembre 2025. 2025. URL: https://www.analyticsvidhya.com/blog/2024/10/function-calling-llms/#h-llms-that-support-function-callings.