



**Politecnico
di Torino**

Politecnico di Torino

Ingegneria Informatica

A.a. 2023/2024

Sessione di laurea Ottobre 2024

**Un approccio basato sul Prompt
Engineering e sulla valutazione
automatizzata per ottimizzare
l'interazione con chatbot aziendali**

Relatori:

Luigi De Russis

Luca Antonelli

Candidato:

Filippo Martino

Ringraziamenti

Il progetto di tesi rappresentato da questo elaborato rappresenta l'ultimo passo di un percorso molto lungo, modesto, non sempre ben vissuto.

Senza l'aiuto di diverse persone, che in modo diverso hanno saputo essermi vicine, non sarei riuscito a portarlo a termine.

Ringrazio il mio relatore, il professor De Russis, per la disponibilità, la gentilezza e la vicinanza durante tutto il percorso di tesi, e per avermi restituito la fiducia della quale avevo bisogno per andare avanti.

Ringrazio allo stesso modo il mio relatore aziendale, il dottor Antonelli, per tutto il supporto datomi in azienda, tentando sempre di supportarmi, nonostante la mole di lavoro.

Un ringraziamento speciale alla mia famiglia: mio padre, mia madre e mia sorella, che, ognuno a suo modo, hanno vissuto insieme a me questi anni standomi sempre vicino, facendo tutti i sacrifici possibili per non farmi mancare mai nulla, cercando quanto possibile di alleggerire tutti i momenti più difficili, a volte soffrendoci, sempre lontano dagli occhi. Senza di voi nulla sarebbe stato possibile. Siatene consapevoli.

Un grande grazie va ai miei coinquilini. Luca, per essermi stato vicino in interminabili serate, per aver condiviso gioie e dolori, dentro e fuori l'università, e Francesco, per aver arricchito le nostre giornate, ed anche lui per avermi sostenuto specialmente nei momenti più bui. Siete stati come una seconda famiglia, grazie.

Per tutti coloro che in questi cinque anni mi abbiano aiutato, contribuendo ad alleggerire il percorso con momenti di serenità: un sentito grazie. Dal profondo del cuore.

Indice

Elenco delle figure	VIII
1 Introduzione	1
1.1 I chatbot	2
1.1.1 Utilizzo della documentazione	2
1.1.2 Gli utenti	3
1.2 Contesto aziendale	4
1.3 Obiettivi	4
1.4 Struttura della tesi	5
2 Contesto	6
2.1 Language Models	6
2.1.1 Bidirectional Encoder Representations from Transformers (BERT)	8
2.1.2 Generative Pre-trained Transformer (GPT)	9
2.2 LLMs e Prompt Engineering	9
2.2.1 Token e costi	11
2.2.2 Parametri di configurazione	12
2.2.3 Tecniche di prompting	13
2.3 La tecnologia	13
2.3.1 Retrieval-Augmented Generation (RAG)	13
2.4 Modello di partenza	15
3 Analisi dei requisiti	16
3.1 Bisogni degli utenti	16
3.1.1 L'analisi	17
3.2 Requisiti funzionali	18
3.3 Requisiti non funzionali	20
4 Progettazione	23
4.1 Design della pipeline	23

4.2	Conversation manager	26
4.3	Documents manager	29
4.4	Question builder	30
4.4.1	Una struttura per tracciare il contesto: lo <i>Snippet della conversazione</i>	30
4.4.2	Riscrittura della richiesta dell'utente: gli esempi	31
4.4.3	Aggiornamento della domanda: struttura del prompt	34
4.5	Question scanner	36
4.5.1	Distanza media	38
4.5.2	Entropia	39
4.5.3	Qualità strutturale	40
4.5.4	Determinare il punteggio	41
4.6	Answer builder	43
4.6.1	Le risposte	43
4.6.2	Streaming	47
4.7	Funzionalità di traduzione	47
4.8	Configurazione	48
4.9	Funzioni d'utilità generale	48
4.10	Virtualizzazione	49
4.11	Interfaccia grafica	49
4.11.1	Interfacce grafiche attualmente esistenti	50
4.11.2	Euristiche	51
4.11.3	Formato delle risposte	53
4.11.4	Funzionalità aggiuntive dell'interfaccia grafica	55
5	Implementazione chatbot	58
5.1	LangChain e la generative AI	59
5.2	Docker e la virtualizzazione	60
5.3	Gestione delle conversazioni: ConversationManager	61
5.3.1	Struttura ed organizzazione dei dati	61
5.3.2	Organizzazione delle classi	63
5.3.3	Architettura del modulo	66
5.4	Gestione della base di conoscenza: DocumentsManager	67
5.4.1	Caricamento della collezione vettoriale	67
5.4.2	Ricerca documenti rilevanti	70
5.4.3	Modalità alternativa di aggiornamento	71
5.4.4	Funzionamento del modulo	71
5.5	Prompt engineering: QuestionBuilder	73
5.5.1	Caricamento e ricerca degli esempi	73
5.5.2	Costruzione del prompt e richiesta al modello	74
5.5.3	Aggiunta degli esempi	76

5.5.4	Analisi del flusso dati	76
5.6	Analisi delle richieste: QuestionScanner	77
5.6.1	Ricerca delle informazioni	78
5.6.2	Calcolo dei parametri	78
5.6.3	Assegnazione punteggio	78
5.6.4	Interfaccia del modulo	79
5.6.5	Analisi del flusso dati	79
5.7	Ottimizzazione delle risposte: AnswerBuilder	80
5.7.1	Inizializzazione del modulo	80
5.7.2	Il processo di costruzione della risposta	81
5.7.3	Generazione in tempo reale	86
5.7.4	Opzioni di contesto	87
5.8	Gestione delle traduzioni: Translator	88
5.8.1	Glossario: struttura ed inizializzazione	88
5.8.2	Processo di traduzione	89
5.8.3	Logica di traduzione	90
5.9	Punto comune per le configurazioni: Config	91
5.10	Funzioni di utilità: Utils	92
5.11	Unione dei moduli: Pipeline	93
5.11.1	La struttura	93
5.12	Messa a disposizione delle funzionalità: API	98
5.13	Predisposizioni per la distribuzione	100
6	Sviluppo interfaccia grafica	102
6.1	Tecnologie utilizzate	102
6.1.1	React	102
6.1.2	API	103
6.1.3	Bootstrap	103
6.1.4	Virtualizzazione	103
6.2	Panoramica generale dell'interfaccia	104
6.3	Colonna delle conversazioni	105
6.3.1	Il Conversation Peek	106
6.3.2	L'ordinamento	108
6.3.3	Caricamento dei messaggi	108
6.4	La chat	110
6.4.1	Barra di inserimento	110
6.4.2	I messaggi	111
6.5	La modalità avanzata	115
6.6	Funzionalità aggiuntive	117

7	Test e risultati	120
7.1	Test dell'usabilità	120
7.1.1	Pianificazione	121
7.1.2	Modalità di svolgimento	122
7.1.3	I task	124
7.1.4	Esecuzione	125
7.2	Analisi dei risultati	125
7.2.1	Riscontri pratici dei moduli introdotti	125
7.2.2	Visualizzazione dei risultati	129
8	Conclusioni e possibili sviluppi futuri	134
8.1	Possibili sviluppi futuri	135
8.2	Sviluppi futuri	135
8.2.1	Considerazioni sul fine-tuning	136
8.3	Considerazioni finali	137
	Bibliografia	138

Elenco delle figure

2.1	Calcolo del numero di token di una frase	11
2.2	RAG, schema di funzionamento [16]	14
4.1	L'architettura del chatbot	25
4.2	Schema base dati, conversazioni	27
4.3	Diagramma di flusso: processo di assegnazione punteggio alla domanda	42
4.4	Schermata iniziale - ChatGPT	50
4.5	Schermata iniziale - Gemini	51
4.6	Rappresentazione dei messaggi - chatGPT	54
4.7	Rappresentazione dei messaggi - Gemini	54
5.1	Schema base di dati - gestione delle conversazioni	62
5.2	UML - classi di gestione conversazioni	64
5.3	UML - classi di gestione messaggio	65
5.4	Architettura semplificata del ConversationManager	67
5.5	Diagramma di flusso - caricamento della base di dati vettoriale . . .	70
5.6	DocumentsManager - Diagramma di flusso	72
5.7	Struttura degli esempi, rappresentazione mediante Document e confronto con Snippet	74
5.8	Diagramma di flusso del metodo di riscrittura della domanda	75
5.9	QuestionBuilder - Diagramma di flusso	77
5.10	QuestionScanner - Diagramma di flusso	80
5.11	Diagramma di flusso - processo di costruzione della risposta	85
5.12	Translator - Diagramma di flusso	91
5.13	Sequenza di generazione risposta	96
5.14	Sequenza di generazione risposta, seconda parte	97
6.1	Panoramica interfaccia grafica	104
6.2	Panoramica dell'interfaccia in modalità scura, con una conversazione aperta.	105
6.3	Dettaglio cancellazione conversazione	106

6.4	Modifica del nome di una conversazione	107
6.5	Dettaglio - barra laterale delle conversazioni	108
6.6	Diagramma di flusso - aggiornamento dei messaggi	109
6.7	Dettaglio barra di inserimento	110
6.8	Dettaglio fonte in formato pdf	113
6.9	Dettaglio fonte in formato video	113
6.10	Dettaglio - modalità avanzata ed impostazioni di interfaccia	115
6.11	Aggiunta esempio di domanda	116
6.12	Dettaglio - Visualizzazione riscrittura della domanda nella modalità avanzata	116
6.13	Dettaglio - Prompt utilizzato per generare la risposta.	117
6.14	Messaggio introduttivo della GUI	119
7.1	Esempio di riscrittura ad opera del QuestionBuilder	126
7.2	Un altro esempio di riscrittura di una domanda mal strutturata	126
7.3	Elenco di possibili soluzioni da esplorare dopo domanda generica	127
7.4	Intervento dello scanner su una domanda senza contesto	127
7.5	Elenco di possibili soluzioni da esplorare dopo domanda generica	128
7.6	Tempo per task - utenti	129
7.7	Tempo per task - utente esperto	129
7.8	Livelli di successo dei test condotti utilizzando il chatbot e la documentazione.	130
7.9	Tempo medio dei test sull'interfaccia.	131
7.10	Distribuzione dei punteggi del questionario SUS	132

Capitolo 1

Introduzione

Tra le molteplici sfide che ogni giorno le nostre realtà aziendali affrontano, c'è sicuramente quella dell'accesso alle informazioni interne. Queste informazioni esistono solitamente sotto forma di *documentazione* aziendale della più varia tipologia: manuali di funzionamento di un particolare prodotto, linee guida, regolamenti, normative, manuali tecnici e molto altro ancora.

Un utilizzo appropriato di questo tipo di informazioni è fondamentale per il corretto svolgimento della maggioranza delle operazioni: è quindi importante mettere a disposizione degli strumenti ai dipendenti in modo da agevolarne il più possibile la fruizione.

Fino a pochi anni fa, la tecnologia disponibile per interfacciarsi con questa numerosa tipologia di informazioni era limitata a programmi di visualizzazione *standard*, che permettono di consultare le informazioni presenti nella documentazione nella modalità *classica* alla quale siamo tutti abituati. Questi programmi svolgono appieno il loro ruolo di “visualizzatori”, limitandosi a mostrare passivamente le informazioni, a volte permettendo semplici ricerche per parole chiave.

L'emergere di una nuova tecnologia nel campo dell'intelligenza artificiale, specificamente i **Large Language Model (LLM)**, ha permesso di rivoluzionare le modalità con le quali l'utente si può interfacciare con tutta la classe di informazioni sopra citate.

Questi nuovi modelli rendono possibile la creazione di assistenti virtuali, i cosiddetti **chatbot**, capaci, con la giusta programmazione, di interpretare correttamente le richieste di un utente e di guidarlo nel processo di ricerca delle informazioni.

Di fondamentale importanza, in questo contesto, sono le informazioni di partenza che vengono messe a disposizione di questi assistenti, e il **prompting**, ovvero la modalità di interrogazione con la quale si interagisce.

Questo progetto, svolto in collaborazione con un'azienda, si propone di esplorare le possibilità offerte da questa nuova tecnologia, progettando un assistente virtuale

ottimizzato nelle interazioni con l'utente, che sempre deve essere al centro dello sviluppo di sistemi di questa tipologia.

1.1 I chatbot

I chatbot sono programmi informatici progettati per simulare conversazioni umane tramite interfacce testuali o vocali. Utilizzano algoritmi di intelligenza artificiale, ed i più moderni l'elaborazione del linguaggio naturale, che permette un'interazione completa e comprensibile con gli utenti: i più avanzati hanno la possibilità di riconoscere e gestire vari tipi di linguaggio, errori grammaticali ed abbreviazioni. Fondamentale la capacità di adattare il tono delle risposte in base alle necessità degli utenti, spaziando dall'essere più formali, informali, diretti, indiretti, didattici, ecc.. In base alla necessità.

Un ambito nel quale la potenzialità di questi strumenti si rivela evidente è quello dell'**integrazione con Sistemi Aziendali**: mediante le tecnologie che verranno discusse nei prossimi capitoli è infatti possibile "istruire" il chatbot, fornendogli tutte le informazioni necessarie per trasformarlo in un assistente esperto circa prodotti e tecnologie per le quali sia necessaria assistenza.

I chatbot si rivelano ottimali nelle aree di assistenza che vanno dal *servizio clienti* al *supporto IT*, dal *marketing* al potenziamento dell'*accessibilità*, fino all'*educazione e formazione*, ambito del chatbot in esame in questa tesi.

1.1.1 Utilizzo della documentazione

La documentazione aziendale, come accennato sopra, è l'insieme strutturato di informazioni, dati, procedure e linee guida che descrivono in modo dettagliato il funzionamento, i processi, le politiche e le operazioni di un'organizzazione: possono includere procedure tecniche, linee guida per l'uso di software e protocolli di sicurezza.

Nel caso in esame la documentazione di riferimento è quella di **formazione**: sono compresi manuali per i dipendenti che descrivono il funzionamento di un determinato prodotto o processo, programmi di sviluppo professionale, indicazioni e tutto quello che può essere necessario per sviluppare la conoscenza del dipendente in uno o più ambiti specifici.

L'integrazione della documentazione aziendale con i chatbot può potenziare enormemente l'efficacia di entrambi, fornendo un accesso rapido e preciso alle informazioni critiche, migliorando l'efficienza operativa: gli assistenti virtuali possono essere configurati per accedere e recuperare informazioni specifiche dalla documentazione aziendale su richiesta, evitando la ricerca manuale delle informazioni, oppure possono direttamente rispondere con le informazioni contenute nei documenti stessi, riproponendo i concetti in modo diverso.

1.1.2 Gli utenti

Gli utenti dei chatbot sono una categoria estremamente ampia, ma nell'ambito di tesi ci si concentra sugli utilizzatori dei chatbot aziendali, nello specifico su utenti poco esperti, che utilizzano l'assistente per formarsi.

Le richieste che possono essere fatte da questa tipologia di utente spaziano dalle informazioni generali su un prodotto, un processo od una sigla, fino alle informazioni specifiche su procedure di installazione, o sul quali siano i passaggi per realizzare determinate operazioni.

La natura delle domande richiede una certa elasticità da parte dell'assistente, garantita principalmente dall'evoluzione dei nuovi modelli linguistici sui quali i chatbot moderni si basano, dalla qualità della documentazione di formazione e dalla progettazione.

Le funzionalità appena elencate sono fondamentali, ma non bastano per garantire un'esperienza completa ed efficace, a causa di un punto fondamentale: l'**utente** stesso.

L'efficacia di questi strumenti infatti, per la loro natura, è strettamente legata alla qualità delle interazioni con gli utenti che gli utilizzano, dai quali dipende in larga parte: non integrare nella progettazione elementi che vadano a tenere conto delle possibili *difficoltà* e *mancanze* (inesperienza o confusione rispetto ad un argomento o prodotto che non si conosce, ad esempio) degli utenti stessi ha come conseguenza degli assistenti tecnicamente efficaci, ma che poi nell'utilizzo quotidiano vengono rapidamente lasciati cadere in disuso.

Per capire meglio quanto sia fondamentale tenere conto delle caratteristiche degli utenti si propone il seguente esempio: un dipendente appena assunto sta cercando di capire il funzionamento di un determinato processo legato ad uno dei prodotti che l'azienda produce, ma questo prodotto è a sua volta suddiviso in diversi sottoprodotti composti da un certo numero di moduli. Un chatbot tecnicamente ben progettato, che faccia affidamento su un modello di intelligenza artificiale avanzato nella comprensione delle richieste dell'utente, e con una documentazione ben scritta, non potrà comunque dare una risposta soddisfacente se l'utente non specificherà a quale sottoprodotto o modulo è interessato: nessuna tecnologia attualmente a disposizione implementa la telepatia, e anche lo si facesse sarebbe comunque poco utile, in quanto spesso un utente che sta imparando non ha tutte le informazioni chiare in mente.

La richiesta stessa, poi, andrebbe posta all'assistente secondo delle regole di struttura, a comporre un **prompt** (2.2): un chatbot ben progettato non dovrebbe aspettarsi di comunicare con utenti consapevoli di tutte queste regole, spesso anche perché questo tipo di tecnologia viene presentato come capace di emulare in modo ottimale le conversazioni umane, nelle quali però una richiesta segue regole e segnali diversi rispetto a quelle di un assistente virtuale.

1.2 Contesto aziendale

Il progetto di tesi è stato svolto in ambito aziendale, specificatamente presso l'azienda *Tesisquare S.p.A.*, nella sede di Roreto di Cherasco. L'azienda sviluppa software in ambito Supply Chain. Fondata a Bra (provincia di Cuneo) da un piccolo gruppo di soci iniziali, raggiunge una notevole crescita in termini economici, di personale e di clienti fino a contare, ad oggi, circa 500 dipendenti.

Il lavoro è stato svolto nella *divisione di prodotto*, nel contesto dei progetti più innovativi che l'azienda ogni anno stimola e promuove.

Sono stato affiancato durante il progetto dal responsabile dell'innovazione aziendale, **Luca Antonelli**, mio tutor durante tutto il periodo passato in azienda, che sempre ha saputo aiutarmi lasciandomi però piena autonomia (e preziosa fiducia) nella progettazione del mio modello.

Esisteva già un assistente in fase "embrionale", che comprendeva le interazioni base con il modello LLM dal quale sono partito per sviluppare il progetto di tesi.

1.3 Obiettivi

La principale motivazione che ha portato l'azienda a sviluppare una prima versione del chatbot è stata la necessità di sviluppare uno strumento di supporto all'Academy aziendale: un percorso pensato per iniziare i nuovi assunti al nuovo ambiente e alle future mansioni.

Le tecnologie e gli argomenti (legati al mondo della Supply Chain), oltre ad esser presentati ai neo-assunti mediante lezioni frontali, sono disponibili all'interno di apposita documentazione sotto forma di manuali pdf e video lezioni.

Spesso, e l'ho potuto constatare io stesso durante il lavoro di tesi, quando non si ha familiarità con i prodotti che si incontrano e gli ambienti nei quali si applicano, non è banale trovare le informazioni quando queste sono concentrate insieme a molte altre: uno strumento che metta a disposizione in modo rapido ed efficace le informazioni necessarie in modo pedepedeutico si rende necessario.

L'obiettivo è quello di fornire una risposta alle mancanze evidenziate, sia in questo paragrafo che in quelli precedenti, in modo particolare:

- Garantire una buona **gestione delle richieste** degli utenti, senza che questi ne siano consapevoli: necessaria una strategia che consenta di correggerne le domande, adattandole agli standard strutturali dei prompt, mantenendone la personalità ed il tono.
- Identificare l'**incertezza** delle domande, **segnalarla** e **guidare** l'utente nella riformulazione, proponendo *possibili spunti* tra i quali possa trovarsi quello che cercano.

- **Valutare** le richieste in modo *automatico, semplice e veloce*, in modo da permettere successive discriminazioni nella risposta.
- **Ottimizzare e personalizzare** le risposte dell'assistente per aumentare l'efficacia dell'apprendimento.

Indipendentemente dall'implementazione che verrà discussa nel lavoro di tesi (5), che ne rappresenta una possibile realizzazione, verranno proposte le **logiche** attraverso le quali raggiungere questi **obiettivi**, frutto degli studi e delle analisi che hanno caratterizzato tutta la fase di individuazione delle necessità degli utenti e successiva progettazione.

Le soluzioni proposte andranno progettate in modo da poter essere integrate nei sistemi già presenti, senza alterarne le logiche di funzionamento: si vuole, per riassumere, sviluppare una **strategia strutturata in passaggi che potenzino l'esperienza dell'utente**, senza impatti negativi sulla struttura dell'assistente virtuale potenzialmente esistente.

1.4 Struttura della tesi

L'elaborato si compone dei seguenti capitoli:

- Il capitolo 2 si propone di analizzare l'attuale stato dell'arte, le tecnologie disponibili ed il punto di partenza del progetto.
- Il capitolo 3 va a fondo nei requisiti tecnici dell'assistente e dell'interfaccia
- Il capitolo 4 si occupa della fase di progettazione, descrivendo le fasi di ricerca, di sperimentazione e le considerazioni tecniche.
- Il capitolo 5 descrive i punti più articolati dell'assistente, le scelte di implementazione ed i framework utilizzati.
- Il capitolo 6 esplora le tecniche di implementazione utilizzate nello sviluppo dell'interfaccia grafica, in linea con le specifiche tecniche definite in precedenza, rispettando le più moderne euristiche dedicate agli assistenti virtuali.
- Il capitolo 7 espone le modalità di Test del progetto, la scelta degli utenti, la composizione dei task ai quali sono stati sottoposti, ed infine i risultati.
- Il capitolo 8 si conclude l'elaborato, traendo le conclusioni e proponendo possibili sviluppi futuri.

Capitolo 2

Contesto

Gli assistenti virtuali attualmente a disposizione sono il frutto dei più recenti avanzamenti nel campo del **Natural Language Processing (NLP)**.

Il **Natural Language**, traducibile in *linguaggio naturale*, è la categoria che comprende tutte le classi di linguaggio (scritto e parlato) che gli esseri umani utilizzano per comunicare. Il *natural language* si distingue dal *formal language*, che in questo contesto indica il linguaggio scientifico, come le formule matematiche.

Lo studio del linguaggio naturale ed i tentativi di costruire software capaci di generarlo autonomamente costituisce una nuova frontiera nell'ambito dell'intelligenza artificiale, dal quale sono nati diversi **language models (LM)** e, successivamente, gli **LLM**.

Questo capitolo si propone di esplorare queste evoluzioni, le proprietà di questi modelli, le tecnologie che vi ruotano attorno e le sfide che generano.

2.1 Language Models

Un modello, nel contesto dell'intelligenza artificiale, è il risultato di un processo di addestramento risultante da un certo set di dati (*training dataset*) specifici ad un certo campo. Quanto migliore è la qualità del set dati e la sua dimensione, quanto più il modello performerà bene nel compito per il quale è stato addestrato, sviluppando quella che potremmo definire un'*intelligenza* specifica al campo di addestramento.

Utilizzando la conoscenza acquisita dal *dataset* di partenza, alcuni modelli, quelli *predittivi*, sono capaci di generare **predizioni** su nuovi dati: ad esempio, un modello che sia stato addestrato nel riconoscere immagini di cani e gatti, sarà capace di stimare a quale gruppo appartenga una nuova immagine, non presente nel dataset di addestramento.

I Language Models [1] sono un sottoinsieme di modelli capace di generare, capire e manipolare il linguaggio naturale. Questi modelli costituiscono la base dei più recenti sviluppi in ambito **NLP**, da BERT (2.1.1) a GPT (2.1.2)

I modelli linguistici sono di fondamentale importanza per le applicazioni che ne derivano e di grande impatto sulla nostra vita, ad esempio il riconoscimento del linguaggio parlato, la traduzione da una lingua all'altra e soprattutto la *generazione* di linguaggio naturale.

Di seguito si propone una sintesi dei modelli principali, per contestualizzare l'analisi di quelli più recenti:

- **Statistical models** (n-grams): classe di modelli tra i più importanti, sono tutti *foundational language models* [2], ossia modelli utilizzati in più campi in funzione del linguaggio naturale. Utilizzano come contesto le $n - 1$ parole per stimare la successiva, sulla base del dataset di addestramento. La loro semplicità li rende efficienti, in quanto non tengono conto dell'intero contesto della frase, della semantica globale. Per lo stesso motivo, sono poco efficaci per la generazione di frasi più lunghe e complesse.
- **Neural network-based models**: i modelli che si basano su reti neurali sono ampiamente utilizzati in molteplici ambiti, e non potevano quindi mancare nell'NLP [3]. Il testo viene convertito in formato numerico, in forma vettoriale, in modo da poter poi esser processato dalla rete neurale. Esistono differenti tipologie di modelli che discendono da questa categoria, dai più semplici **Feedforward neural networks** (dove ogni layer di neuroni esegue dei calcoli specifici ed inoltra il segnale calcolato al livello successivo, ed ogni livello è indipendente), fino ai più complessi **Recurrent neural networks** (RNNs), che a differenza dei Feedforward condividono i parametri tra i vari livelli, mantenendo una sorta di memoria. Questi modelli, specialmente l'ultimo, sono ampiamente utilizzati nelle applicazioni per le quali non solo è importante il significato del singolo valore, ma anche quello della composizione di più valori, come le parole e di conseguenza il linguaggio.
- **Transformer Models**: Il Transformer [4] è un modello introdotto dai ricercatori di **Google** [5] nel 2017 che ha **rivoluzionato** il campo della manipolazione del linguaggio naturale. La sua caratteristica principale è l'uso del *meccanismo di attenzione*, che permette al modello di concentrarsi sulle parti rilevanti di una sequenza di parole senza elaborarle in ordine sequenziale, come fanno i modelli tradizionali. Questo approccio consente una maggiore efficienza e parallelizzazione nel processo di training, rendendo il Transformer molto **potente** e **veloce**. È composto da due parti principali: l'*encoder*, che analizza la sequenza di input, e il *decoder*, che genera la sequenza di output.

Grazie alla sua architettura innovativa e l'efficacia, il Transformer ha posto le basi per modelli avanzati come **BERT** e **GPT**, che hanno ulteriormente migliorato le performance nei vari compiti di elaborazione del linguaggio naturale. Proporzionalmente al raggiungimento di risultati via via sempre migliori sono aumentati anche i volumi dei dataset di partenza, rendendo necessaria una nuova terminologia per definire questi modelli: non più Language Models, bensì **Large Language Models (LLMs)** [6].

2.1.1 Bidirectional Encoder Representations from Transformers (BERT)

BERT, acronimo di Bidirectional Encoder Representations from Transformers, è un modello di linguaggio sviluppato da Google e presentato nel 2018 [7].

Questo modello ha contribuito a significativi miglioramenti nella comprensione del linguaggio naturale (NLP) grazie alla sua capacità di considerare il **contesto bidirezionale** nelle frasi, ossia alla capacità di tenere in considerazione non soltanto la parola in analisi sulla base delle parole precedenti, ma anche di quelle successive.

Da questa proprietà ne consegue come sia ottimale nell'analisi e nella classificazione di testi, ma **non** viene utilizzato per la generazione di nuove informazioni.

Il modello è basato sull'architettura dei Transformers, utilizzandone però solo la parte di *encoder* (da qui la mancanza nella parte generativa). Ecco alcune delle sue caratteristiche principali:

- **Bidirezionalità:** A differenza dei modelli unidirezionali, BERT considera il contesto sia a sinistra che a destra di una parola nella sequenza di testo, permettendo una comprensione più profonda del significato.
- **Pre-training:** BERT viene pre-addestrato su grandi quantità di dati di testo per mezzo di processi non supervisionati (il modello autonomamente procede ad etichettare i nuovi dati componenti il dataset):
 - *Masked Language Model* (MLM): si maschera, ovvero oscura, una determinata percentuale di parole nel testo di addestramento ed il modello deve completarle, mediante previsioni.
 - *Next Sentence Prediction* (NSP): il modello riceve coppie di frasi e deve determinare se la seconda frase segue la prima nel testo originale.
- **Fine-tuning:** Dopo il pre-training, BERT può essere finemente adattato a compiti specifici

BERT ha **contribuito** enormemente allo sviluppo di nuovi LLM, e viene tuttora utilizzato per l'**analisi** di grandi quantità di testo, specialmente per tutti quei compiti nei quali il contesto delle frasi sia rilevante.

Un esempio di questo contributo: Google, verso la fine del 2019, ha **integrato** BERT nel proprio motore di ricerca, per capire meglio le richieste dei propri utenti [8].

2.1.2 Generative Pre-trained Transformer (GPT)

Questi modelli sono quelli più conosciuti, entrati nell'immaginario comune grazie ai risultati rivoluzionari di **chatGPT** [9], un assistente virtuale prodotto dell'azienda **OpenAI** [10], basato su differenti versioni.

GPT rappresenta una classe di *foundation models*, tra i quali vale la pena citare quelli più famosi, sviluppati da OpenAI, che vanno da GPT-1 al più recente GPT-4o.

OpenAI non è naturalmente l'unica azienda ad operare in questo campo; vale la pena citare **Google**, con la sua famiglia di Transformer: Gemini [11], successore di paLM, sviluppato dalla partecipata di Google, **DeepMind** [12] (dal modello Gemini deriva l'omonimo chatbot).

La loro caratteristica fondamentale è la capacità di generare del testo basandosi su uno specifico *prompt*, ovvero la specifica del task che l'assistente deve svolgere, espressa in linguaggio naturale. L'importanza del *prompting* è fondamentale: si pensi ad una vera e propria programmazione non in linguaggio formale (come tradizionalmente vengono fornite le istruzioni che compongono i software), bensì ad una **programmazione mediante linguaggio naturale**. Per far sì che il modello restituisca un risultato soddisfacente rispetto al compito assegnatogli, è di estrema importanza la programmazione della richiesta. Questo aspetto fondamentale ha contribuito alla nascita di una nuova disciplina, il **prompt engineering**.

Infine, prima di analizzare a fondo il prompting e tutto quello che ne consegue, è importante analizzare il concetto di **fine-tuning** per i modelli GPT. Questa tecnica permette infatti di *specializzare* un dato modello in uno specifico e determinato contesto: chatGPT stesso è un'applicazione risultante da alcune versioni di GPT per le quali è stato svolto un processo di fine tuning per specializzarli nelle conversazioni.

Al contrario dell'addestramento iniziale dei modelli, che è *unsupervised* (ovvero non necessita di un supervisore che se ne occupi in modo diretto), il fine tuning deve per larga parte essere svolto e programmato da un **utente**, che **fornisce** al modello un **set mirato** di esempi di prompt e di risposte ottimali che devono essere prodotte. Questo rende il fine tuning una tecnica **potente** ma allo stesso tempo **costosa**, non sempre adatta a tutti i contesti.

2.2 LLM e Prompt Engineering

Il prompt engineering, come già accennato, può essere considerata come una vera e propria **disciplina** emersa con l'ampia diffusione degli LLM avvenuta negli ultimi

anni. L'obiettivo è focalizzarsi sulla creazione e ottimizzazione delle istruzioni fornite all'LLM, con l'obiettivo di migliorare la *qualità* e la *consistenza* dei risultati.

I modelli vengono addestrati su moli considerevoli di dati in modo da definire dei pattern, delle strutture di linguaggio e contesto, ma la **manca** di **istruzioni precise** da parte dell'utente può causare risposte irrilevanti, poco chiare, incoerenti e parziali. La necessità di ottenere delle buone risposte ha portato alla nascita di tecniche di prompting, specifiche al campo di applicazione dei risultati del modello (2.2.3).

La conoscenza di questa nuova disciplina, sebbene fondamentale per il corretto funzionamento di tutte quelle applicazioni che basano parte del proprio funzionamento sugli LLMs, *non può in nessun modo* essere sottesa negli utenti: si rende necessario predisporre sistemi capaci di tener conto di possibili mancanze da parte degli utilizzatori finali, con la possibilità di **riprogettare** il **prompt** potenzialmente incompleto dell'utente in modo da ottimizzarlo.

Un buon esempio di cosa può succedere all'interno di un sistema che preveda l'utilizzo di LLM e prompting viene fatto nella pagina informativa del prompt engineering proposta da Amazon AWS [13]:

Ad esempio, prendiamo in considerazione i chatbot basati sull'intelligenza artificiale. Un utente può inserire una dichiarazione di problema incompleta, ad esempio "Dove acquistare una maglietta". Internamente, il codice dell'applicazione utilizza un prompt ingegnerizzato che dice: "Sei un assistente alle vendite per un'azienda di abbigliamento. Un utente, con sede in Alabama, Stati Uniti, ti chiede dove acquistare una maglietta. Rispondi con i tre punti vendita più vicini che attualmente vendono una maglietta". Il chatbot genera quindi informazioni più pertinenti e accurate.

Agire con efficacia sul prompting può migliorare l'**esperienza dell'utente** senza che questo debba specializzarsi, e nel caso del progetto oggetto di questa tesi, il prompting è fondamentale per almeno tre aspetti:

- **Riduzione delle allucinazioni:** le allucinazioni, aspetto intrinseco dei modelli linguistici, sono una classe di risposte considerate non soddisfacenti [14]. Rappresentano uno dei maggiori problemi legati all'utilizzo degli LLM in ambienti di produzione: si consideri, ad esempio, un assistente virtuale. Gli episodi legati ad allucinazione che possono verificarsi portano l'assistente (se non viene contenuto) a dare risposte errate, ma con una **confidenza** tale da sembrare corrette ad un utente non esperto nel campo della risposta. Per mitigare questo problema è necessario circoscrivere il più possibile i **confini del contesto**, fornendo al modello tutte le informazioni necessarie, in modo

specifico, per produrre la risposta più **puntuale** possibile. Per questo compito il prompting è fondamentale.

- **Arricchimento del contesto:** legato al punto precedente, l'arricchimento del contesto che si può ottenere producendo dei buoni prompt serve sia a ridurre le allucinazioni, sia ad assicurare una risposta più ricca, migliorando l'esperienza dell'utente.
- **Maggiore controllo sulle richieste:** un aspetto importante del prompting che permette di gestire la tipologia di risposte che vengono date agli utenti, sulla base delle domande fatte. Un buon prompt può evitare risposte giuste ma non in linea con le linee guida di chi ha progettato l'applicazione (si pensi alle dinamiche aziendali).

2.2.1 Token e costi

I modelli non processano il testo in parole, bensì in blocchi definiti **tokens**. I tokens rappresentano sequenze di caratteri che compaiono frequentemente, e sono diversi dalle parole.

Le aziende che producono i modelli generalmente mettono anche a disposizione strumenti per mostrare come una frase viene divisa in token. Di seguito un esempio ottenuto con lo strumento messo a disposizione da openAI (*Tokenizer*):

La frase:

Suddivisione di una frase in token

Che contiene 6 parole, viene suddivisa in 9 token, come mostrato nell'immagine di seguito.



Figura 2.1: Calcolo del numero di token di una frase

Conoscere il numero di token dei propri prompt è fondamentale per due motivi:

1. **Limiti tecnici del modello:** ogni modello ha una diversa quantità di token che riesce a processare, suddivisi tra **context window** (ossia il numero di token che possono essere gestiti dal modello nella domanda), e **max output tokens** (numero di token che possono essere generati in risposta). La context window fissa il limite superiore alla dimensione dei prompt, che quindi devono essere completi ma allo stesso tempo concisi.
2. **Costi:** quando si realizza un'applicazione che utilizzi LLM, si sfruttano le *API* (Application programming interface [15]) messe a disposizione dai creatori del modello. Le API permettono di effettuare diversi tipi di chiamate, tutte a **consumo**. I costi vengono ripartiti tra i token che compongono il contesto e quelli che compongono la risposta, che possono avere un costo *diverso* a seconda del modello.

2.2.2 Parametri di configurazione

Gli sviluppatori mettono a disposizione delle API per consentire la configurazione di diversi parametri che influenzano il comportamento dei modelli, la cui **valorizzazione** è conseguentemente di primaria importanza.

Di seguito si propone una **panoramica** riassuntiva dei parametri più importanti:

- **Temperature** (la temperatura): specifica il *grado di variabilità* della risposta. Un grado più alto renderà le risposte più variabili, mentre uno più basso le renderà più deterministiche. Mantenere questo parametro basso riduce il rischio di allucinazioni, smorzandone però la creatività. Generalmente il range di temperature è $0 < T < 1$.
- **Top-p** (top parameter): si basa su una soglia di probabilità cumulativa, selezionando il minimo numero di token la cui somma delle probabilità supera un valore di soglia predefinito, p . Tra questi token, uno viene scelto casualmente (intervallo: $0 < \text{top-p} < 1$). L'adattamento della selezione è quindi dinamico, in funzione della probabilità cumulativa. Valori più alti portano a risposte più puntuali e meno creative, ma se si supera una certa soglia si rischia di limitare troppo l'insieme di parole a disposizione per l'espressione.
- **Top-k**: simile a top-p, non sfrutta la probabilità cumulativa nella scelta dei token, bensì controlla direttamente il numero di opzioni tra cui il modello può scegliere, selezionando i prossimi token solo dai k più probabili secondo il modello. Tra questi k , viene scelto un token casualmente, con una probabilità proporzionale alla sua probabilità originaria. Gli effetti sono gli stessi del top-p, mentre il range dipende dalle caratteristiche del modello.
- **Max tokens**: indica il numero massimo di token che verranno generati dal modello.

2.2.3 Tecniche di prompting

Le tecniche di prompting a disposizione sono in costante aumento, la sfida è applicare la migliore al contesto in esame. Vista questa premessa si trattano due tra le più semplici ed efficaci tecniche di prompting, utilizzate nella progettazione dell'elaborato.

- **Zero-shot**: il prompt *classico*, ovvero un'istruzione che contiene soltanto il task, **il compito**, senza specificare come risolverlo, senza fornire nessun esempio. Questa tipologia di prompt, nonostante la semplicità, è anche la più **veloce**. Considerate queste proprietà, il zero-shot prompting viene utilizzato per risolvere task semplici, impliciti (ad esempio riassumere in poche parole un testo), dove la velocità di risposta del modello è fondamentale.
- **Few-shot**: questa tecnica si propone di aumentare la precisione delle risposte specificando nel prompt, insieme ad una query di richiesta, anche un certo numero di **esempi** (il numero specifico viene definito in base all'applicazione). Il prompt prodotto (posto che la qualità degli esempi sia ottimale) comporta una risposta più guidata, **riducendo** le allucinazioni.

2.3 La tecnologia

Fino a questo punto si sono contestualizzati gli strumenti che verranno utilizzati come base per l'elaborato, ma manca ancora una tecnologia molto importante che, insieme ad altre tecniche, contribuisce a definire una **distinzione** tra una classe di chatbot più semplici che si limitano a rispondere a singole domande cercando di indovinare la risposta migliore tra le innumerevoli informazioni a disposizione del modello, ed una più complessa, che ha una **conoscenza** molto **specificata** del contesto nel quale opera.

2.3.1 Retrieval-Augmented Generation (RAG)

Il **RAG** è una tecnica di **estrazione delle informazioni** che potenzia la generazione di risposte da parte di un LLM mettendole a disposizione del modello, parallelamente alla domanda dell'utente. Le informazioni vengono estratte mediante una tecnica che prende il nome di **retrieving**, estrazione solitamente di tipo **vettoriale** da una collezione di dati **inerenti** al campo della domanda.

Questa tecnica è molto utilizzata per superare i **limiti di memoria** del modello, e per fornire informazioni **non presenti** nella sua base di addestramento, informazioni che il modello non può avere.

Implementando un chatbot che utilizzi prompt arricchiti con il RAG, è possibile sfruttare le abilità linguistiche dei modelli per estrarre e elaborare le informazioni

allegate, fornendo una risposta senza necessità di addestrare l'LLM con informazioni che possono essere riservate o che possono evolvere in futuro. Utilizzare il RAG è anche una valida alternativa ad alcuni dei vantaggi portati dal *fine tuning* (tipicamente la **specializzazione** e la **caratterizzazione** del modello), più **scalabile**. Non si rende necessario, una volta predisposto un *embedder*, inventarsi esempi per ogni possibile informazione contenuta, ad esempio, in una documentazione aziendale.

Da un punto di vista più tecnico, la ricerca delle informazioni che sta alla base della tecnica è resa possibile dai **semantic search algorithms** (algoritmi di ricerca semantica), che prevedono l'indicizzazione delle informazioni in **embeddings vettoriali**, rappresentazioni **numeriche** delle informazioni.

L'*embedding*, in questo contesto, è la rappresentazione numerica di un'informazione sotto forma di un **vettore**, in un formato che sia interpretabile da un algoritmo. Le informazioni che si vogliono indicizzare vengono convertite in vettori ed indicizzate in **database vettoriali**.

Questi database vengono interrogati *per similarità*: si vogliono trovare i dati presenti nella base di dati più vicini ad una specifica informazione, ad esempio una specifica frase. Questa viene convertita utilizzando lo stesso algoritmo con il quale sono state indicizzate le altre informazioni, poi si procede ad una ricerca per distanza: i vettori più vicini a quello appena convertito corrispondono alle informazioni semanticamente più simili a quella utilizzata per la ricerca.

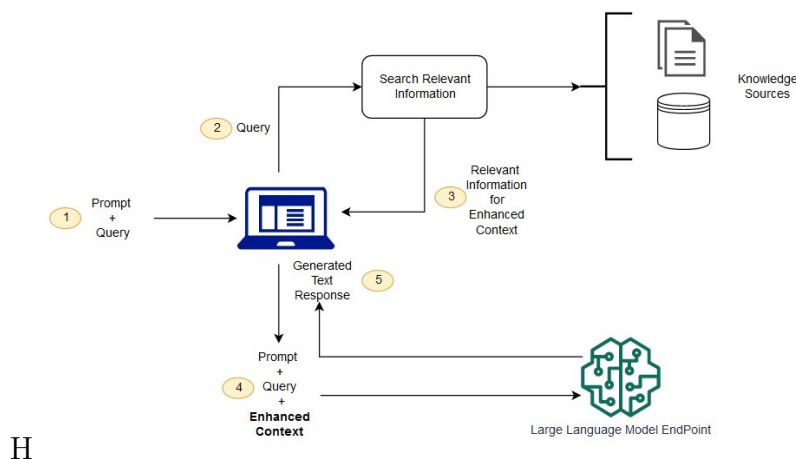


Figura 2.2: RAG, schema di funzionamento [16]

La RAG si compone quindi dei seguenti passaggi:

1. L'utente fa la propria richiesta (*query*), che verrà incapsulata nel prompt.

2. La richiesta dell'utente viene convertita in formato numerico (*embedding*), e viene fatta la ricerca.
3. Le informazioni semantiche più vicine vengono restituite.
4. Si prepara ed invia il prompt con la query dell'utente e le informazioni risultanti dalla ricerca vettoriale.
5. LLM genera e restituisce una risposta.

2.4 Modello di partenza

Prima del progetto di tesi l'azienda aveva già a disposizione un chatbot embrionale, *stateless* (non aveva un sistema di gestione dei dati per salvare le informazioni relative alle conversazioni), facente affidamento ad un'interfaccia di chat molto elementare, un plug-in di Moodle. Tutta l'infrastruttura era già virtualizzata sotto forma di *container* docker [17].

Di seguito le funzionalità più importanti presenti:

- Utilizzo di un sistema RAG elementare che sfruttava un file per l'indicizzazione delle informazioni (nessun db).
- Già a disposizione i dati per il sistema RAG, in forma di oggetti JSON.
- Framework LangChain [18], che verrà trattato meglio in seguito, utilizzato per le chiamate API all'LLM e per la composizione di prompt statici.
- Abbozzo di traduzione mediante i servizi cloud di Google.
- Formattazione delle risposte nelle quali venivano gestiti i link alle fonti utilizzate dal modello per dare la risposta.

Da questo chatbot iniziale sono partite le ricerche e gli sviluppi che hanno contribuito alla sua evoluzione.

Capitolo 3

Analisi dei requisiti

Prima di procedere alla progettazione è bene definire le necessità degli utenti (*user needs*), ed i requisiti che l'applicazione dovrà avere di conseguenza.

3.1 Bisogni degli utenti

L'ultima generazione di chatbot, basata sull'utilizzo degli LLM, ha cambiato la percezione che gli utenti hanno quando utilizzando questo tipo di assistenti.

La capacità di costruire conversazioni complesse e comporre risposte simili a quelle formulate da un essere umano rappresentano un'evoluzione importante nel miglioramento dell'usabilità di questi sistemi, ma comportano anche *nuove sfide*.

Nella conversazione che avviene tra due esseri umani, è naturale passare da un contesto all'altro senza necessità di specificarlo, grazie alla capacità implicita di percepire questi passaggi in modo naturale.

Quando un utente utilizza un assistente virtuale che sembra consapevole della conversazione, in modo quasi umano, è naturale una **traslazione** di ciò che avviene nelle conversazioni tra persone, senza specificare quindi i cambi di contesto, o esprimendo **richieste generali** aspettandosi però **risposte specifiche**.

Quando avviene questo tipo di traslazione, può succedere che il modello non riesca a produrre una buona risposta, spesso dando luogo ad **allucinazioni** (il modello tenta, se non limitato, di dare sempre una risposta con un alto grado di confidenza).

Questa necessità di produrre richieste **chiare e specifiche** può essere considerata, fino ad un certo grado, un **limite** del sistema: l'utente deve aver la possibilità di esprimersi nel modo **più naturale** possibile, senza preoccuparsi in partenza della struttura delle domande. Questo non implica necessariamente un sistema capace di gestire **qualsiasi** grado di incertezza nelle richieste: nemmeno un essere

umano sarebbe capace di conversare con un individuo che si esprimesse in modo estremamente confuso.

Utenti diversi hanno requisiti diversi.

Nel caso trattato in questa tesi l'utente necessita di uno **strumento didattico**, per esplorare argomenti poco conosciuti. Un utente di questo tipo non può produrre domande troppo specifiche, in quanto non ha una conoscenza così profonda degli argomenti trattati dall'assistente. In aggiunta, un'altra necessità è sicuramente il grado di dettaglio ed il **formato** delle risposte: ciò che può sembrare ridondante ad un utente esperto spesso è **fondamentale** per un utente alle prime armi.

Da questa necessità ne deriva un'altra: l'importanza di **non essere limitati** ad un solo strumento di apprendimento (come potrebbe essere il solo chatbot), ma di integrare questo in un set di strumenti più ampio.

Per concludere, si osserva come gli utenti che interagiscono con questi assistenti tendano ad aspettarsi un certo grado di **fluidità** nelle risposte, considerando garantita la possibilità di gestire e **continuare** tutte le conversazioni avute con l'assistente, mediante interfacce **user-friendly** e **minimali**.

3.1.1 L'analisi

Queste necessità sono il frutto di un'analisi svolta nel primo periodo dell'esperienza aziendale: l'**eterogeneità** dei dipendenti che componevano la divisione di sviluppo nella quale è stato svolto il lavoro di tesi si è rivelata utile per raccogliere informazioni diverse.

I dipendenti *più giovani* hanno rappresentato la base di studio, in quanto direttamente interessati dal processo di apprendimento. Questi utenti avevano terminato l'*Academy* (così l'azienda chiama il processo di apprendimento delle dinamiche, dei processi e delle tecnologie utilizzate internamente) da non più di un anno, si erano cimentati nell'utilizzo dei **documenti** e delle loro limitazioni e provato il chatbot **embrionale**. In aggiunta, durante lo svolgimento dei compiti quotidiani, alcuni utilizzavano abitualmente assistenti virtuali come ChatGPT.

A questa tipologia di utenti è stato chiesto come si trovassero con la documentazione, quale approccio avessero con i chatbot basati sull'intelligenza artificiale, quanto spesso li utilizzassero durante il loro lavoro e quali limitazioni si sentivano di trovare e se fosse possibile osservare, quando capitava, le loro interazioni.

Da queste conversazioni sono emerse alcuni punti interessanti, che hanno portato alla definizione delle necessità citate in precedenza: quella in primo luogo di **formulare** bene le richieste dovuta alla confusione su un determinato argomento, che fosse la mancanza di comprensione di un processo aziendale o di un dettaglio del software che si stesse utilizzando.

Un'altra quella della **limitazione** degli assistenti virtuali a disposizione in commercio, come chatGPT o GitHub Copilot [19] quando si tratta di analizzare

software scritto in linguaggi poco supportati (nel caso specifico dell'azienda si trattava di ColdFusion [20], piattaforma di sviluppo web per la creazione di applicazioni dinamiche e interattive, utilizzando un linguaggio di scripting specifico). Si è osservato anche che, durante l'utilizzo del chatbot a disposizione dell'azienda, fosse dato per "scontato" un grado di comprensione dell'assistente che non si sarebbe avuto durante una conversazione tra colleghi. In questa fase sono anche state analizzate le domande realizzate da alcuni esperti dei prodotti dell'azienda, utilizzate per effettuare i test dell'assistente in dotazione: queste evidenziavano come anche gli utenti più esperti si aspettassero risposte **specifiche** a fronte di domande piuttosto **generiche** (le domande di test erano affiancate dalle risposte ottimali che ci si aspettava di ricevere). Questo ha portato alla definizione della necessità, per l'assistente, di **intercettare l'incertezza** e fare ulteriori richieste all'utente.

Parallelamente, l'importanza della documentazione, della sua **struttura** (per poterla poi incorporare a livello tecnico nell'architettura RAG) e **qualità** si è rivelata **fondamentale**: dalla sua analisi e successive domande con gli utenti che avevano frequentato l'Academy è emerso come la documentazione fosse, a volte, troppo sintetica e poco esaustiva per gli utenti non esperti nello specifico ambito trattato; da qui l'esigenza di avere un assistente capace di rispondere in modo esaustivo, elaborando quanto presente nella base di conoscenza per **potenziarne** la didattica.

Queste osservazioni sono state confermate ed ampliate durante una serie di riunioni alle quali si è avuto modo di partecipare, che l'azienda ha organizzato come parte del **programma di innovazione aziendale**, definite "tavoli AI", che avevano come obiettivo la discussione delle nuove tecnologie in ambito di Intelligenza Artificiale (specialmente di modelli linguistici), la loro possibile applicazione in dinamiche aziendali ed i possibili problemi: qui il contributo degli utenti più esperti è stato utile per definire parte delle necessità espresse in precedenza, come l'**integrazione** dell'assistente insieme alla documentazione (e non la sostituzione), l'importanza della riduzione della **latenza** e quella di una buona **interfaccia**.

3.2 Requisiti funzionali

Sulla base delle necessità degli utenti elencate sopra, si propongono una serie di requisiti funzionali e non funzionali.

Si elencano inizialmente i **requisiti funzionali**, che rappresentano requisiti di più alto livello, specificatamente sul *come* il servizio debba comportarsi. I requisiti **non funzionali**, trattati in seguito, affronteranno aspetti più tecnici, legati al *cosa* l'applicazione debba fare per soddisfare le necessità espresse dagli utenti.

- RQ1) Generazione automatica dei prompt:** in ottica di *prompt engineering*, il sistema prevede la **costruzione automatica** del prompt a partire dall'input, che deve essere arricchito per migliorare contesto e specificità. L'applicazione sarà in grado di **analizzare** la richiesta dell'utente, trovare dei prompt ottimizzati nella stessa categoria di domanda, e richiedere una riscrittura al modello stesso.
- RQ2) Analisi dell'incertezza per *follow-up*:** si prevede un sistema di analisi della domanda, capace di stimare l'indice di incertezza e bontà della stessa (utilizzando una serie di parametri), che **intercetti** la richiesta prima di inoltrarla al modello. Questa analisi permetterà l'integrazione di domande di *follow-up*, per richiedere all'utente di specificare meglio l'oggetto della domanda e, guidandolo, fornirgli le informazioni per la formulazione di una richiesta più specifica e quindi una **risposta più puntuale**.
- RQ3) Gestione delle domande non pertinenti:** un modo per gestire gli errori dovuti a domande non pertinenti. Il sistema, sempre mediante analisi della domanda, **filtrerà** tutte quelle non pertinenti al contesto aziendale, evitando in questo modo risposte incoerenti, erranee o potenzialmente spiacevoli.
- RQ4) Gestione delle fonti:** Previsto il mantenimento del **sistema di citazione** delle fonti già previsto nella versione iniziale dell'assistente, che verrà evoluto. L'utente potrà in questo modo approfondire il concetto riassunto dal chatbot andando direttamente all'origine dell'informazione.
- RQ5) Recupero del contesto:** L'assistente sarà capace di rispondere ad una richiesta basandosi, quando necessario, sul contesto della conversazione avuta fino a quel punto. Sarà, di conseguenza, capace di riprendere vecchie conversazioni dal punto in cui erano state interrotte.
- RQ6) Gestione delle conversazioni:** l'interfaccia darà la possibilità di gestire tutte le conversazioni avute con l'assistente, permettendo di cambiarne le proprietà ed eventualmente di eliminarle.

Tabella 3.1: Tabella riassuntiva dei requisiti funzionali

RQ #	Requisito	Descrizione
1	<i>Generazione automatica dei prompt</i>	Costruzione automatica dei prompt per migliorare contesto e specificità.
2	<i>Analisi dell'incertezza per follow-up</i>	Stima dell'incertezza per guidare l'utente con domande di follow-up.
3	<i>Gestione delle domande non pertinenti</i>	Filtraggio delle domande non pertinenti per evitare risposte incoerenti.
4	<i>Gestione delle fonti</i>	Evoluzione del sistema di citazione per approfondire le informazioni.
5	<i>Recupero del contesto</i>	Capacità di riprendere conversazioni interrotte basandosi sul contesto.
6	<i>Gestione delle conversazioni</i>	Gestione e modifica delle conversazioni con possibilità di cancellazione.

3.3 Requisiti non funzionali

Similmente a quanto fatto per i *requisiti funzionali*, una panoramica di quelli *non funzionali*.

RQ1) Usabilità ed accessibilità: l'interfaccia di chat dev'essere intuitiva e rispettare le **euristiche** indicate per gli assistenti virtuali. Deve rispettare gli **standard grafici** ai quali gli utenti sono abituati per questa tipologia di applicazioni, mantenendo allo stesso tempo una personalità caratterizzante.

RQ2) UI/UX efficace: la *User Experience* e la *User Interface*, ovvero l'esperienza utente e l'aspetto visivo dell'applicazione, devono essere garantiti da un **design pulito, ordinato e minimalista**, che consenta la massima semplicità di utilizzo, e nel quale le azioni che l'utente possa svolgere siano chiare.

RQ3) Ottimizzazione della latenza: lavorare con i modelli LLM comporta dover affrontare problemi di latenza. Il sistema sarà ottimizzato sia nel *back-end*, limitando al massimo le chiamate al modello e la dimensione delle stesse, sia nel **front-end**, proponendo come possibilità la generazione delle risposte in formato *streamed* (generazione standard per le chat più avanzate, a.e. ChatGPT), per limitare l'attesa dell'utente.

RQ4) Formattazione delle risposte: l'applicazione garantirà la generazione delle risposte formattate in html, in modo da renderle più **espressive** possibile. Quando previsto, il modello sarà istruito per garantire la produzione di risposte facenti uso di paragrafi, elenchi puntati, tabelle e quanto si renda necessario per rendere le risposte più **leggibili**.

- RQ5) Virtualizzazione:** Sia il chatbot che l'interfaccia saranno eseguiti in ambienti virtualizzati, per garantire l'**isolamento** e la **compatibilità** tra diversi sistemi, così come la facilità di **distribuzione**.
- RQ6) Persistenza:** Saranno previsti sistemi di archiviazione sia per i dati riguardanti le conversazioni, sia per la gestione degli *embeddings*, i dati in formato numerico per il sistema *RAG*.
- RQ7) Gestione automatica delle nuove informazioni:** L'applicativo gestirà in autonomia qualsiasi nuovo dato fornito dall'esterno (da nuove informazioni per il *RAG*, ai nuovi esempi per la composizione dei prompt), senza necessità di intervenire in nessun punto del codice.
- RQ8) Traduzione:** Il sistema prevederà la possibilità di eseguire richieste in qualsiasi lingua, indipendentemente dal linguaggio delle informazioni sul quale basa le risposte. La traduzione sarà possibile per le richieste in formato *non-streamed*.
- RQ9) Modularità:** L'architettura del chatbot farà della modularità un caposaldo. I moduli che implementano i requisiti principali saranno in buona parte indipendenti, acquisendo dati in ingresso, processandoli, e producendo dati in uscita. Si limiteranno il più possibile vincoli di implementazione, che rendano ad esempio obbligato l'uso di più di un modulo anche quando le funzionalità implementate non siano necessarie.
- RQ10) Facilità di configurazione:** L'ambiente del chatbot deve essere facilmente configurabile, tutti i parametri rilevanti per il funzionamento dell'assistente devono essere facilmente accessibili e modificabili. Più in generale tutte le variabili di configurazione (anche quelle non modificabili dall'utente) devono essere in una sezione dedicata, ben organizzate.

Tabella 3.2: Tabella riassuntiva dei requisiti non funzionali

RQ #	Requisito	Descrizione
1	<i>Usabilità ed accessibilità</i>	L'interfaccia di chat deve essere intuitiva e rispettare le euristiche per gli assistenti, mantenendo una personalità caratterizzante.
2	<i>UI/UX efficace</i>	La User Experience e la User Interface devono garantire un design pulito, ordinato, minimalista e di facile utilizzo.
3	<i>Ottimizzazione della latenza</i>	Ottimizzazione della latenza sia nel back-end che nel front-end, con possibilità di generazione delle risposte in formato streamed.
4	<i>Formattazione delle risposte</i>	Generazione delle risposte formattate in HTML per garantire una migliore leggibilità e presentazione.
5	<i>Virtualizzazione</i>	Esecuzione in ambienti virtualizzati per garantire isolamento, compatibilità e facilità di distribuzione.
6	<i>Persistenza</i>	Archiviazione dei dati delle conversazioni e gestione degli embeddings per il sistema RAG.
7	<i>Gestione automatica delle nuove informazioni</i>	Gestione autonoma delle nuove informazioni senza necessità di intervento sul codice.
8	<i>Traduzione</i>	Possibilità di eseguire richieste in qualsiasi lingua, con traduzione per le richieste in formato non-streamed.
9	<i>Modularità</i>	Architettura modulare, con moduli indipendenti che limitano i vincoli di implementazione.
10	<i>Facilità di configurazione</i>	Ambiente facilmente configurabile.

Capitolo 4

Progettazione

La progettazione dell'assistente ha avuto, come già descritto nel capitolo 3, la modularità come principio base. Sulla base dei requisiti elencati sono state fatte diverse considerazioni, concentrando le funzionalità comuni all'interno di moduli specializzati. Il sistema è quindi concepito come una *pipeline* (già presente nel software iniziale) di dati elaborati *in sequenza* da moduli altamente **specializzati**. Si è cercato di rispettare il principio di modularità anche per quanto riguarda aspetti più tecnici, raggruppando tutte le funzionalità comuni in sezioni ben diversificate. La struttura iniziale del chatbot è stata in larga parte rivoluzionata, passando da un progetto sperimentale ad uno candidabile alla fase di **produzione**.

Dal prompt engineering alla formattazione dei risultati, fino al design dell'interfaccia utente, si sono proposte una serie di **soluzioni indipendenti**, applicabili anche ad altri sistemi che sfruttino le potenzialità degli LLM. Questo capitolo si propone di esplorare questi aspetti, partendo dallo schema generale ed andando in seguito ad analizzare i singoli moduli, i dati in ingresso e quelli, prodotti, in uscita.

4.1 Design della pipeline

Progettare un certo numero di moduli il più possibile indipendenti richiede come primo passaggio l'individuazione dei compiti principali che l'assistente deve svolgere, in *accordo* con le necessità degli utenti.

Analogamente, tutte le funzionalità più tecniche devono essere anch'esse definite e classificate in modo da convergere all'interno di *moduli d'appoggio*. Si rivela quindi importante, prima di definire i moduli principali, stilare la panoramica di quelle che sono le esigenze *di base*, dalle quali costituire un set di strumenti rilevante.

Quanto viene elencato in questo paragrafo si propone di soddisfare il requisito non funzionale **RQ9** (3.2 Modularità).

Esigenza fondamentale primaria è rappresentata dall'elemento di base dell'assistente: le conversazioni. La persistenza delle informazioni costituenti domande e risposte, potenzialmente integrabili con altri dati, rende necessario uno strumento che incorpori queste funzionalità, fornendo le opportune interfacce che il resto dei moduli possa utilizzare. Questo modulo prende il nome di **conversation manager**, il gestore delle conversazioni (4.2).

Rimanendo nell'ambito della gestione dei dati, resta l'esigenza di gestire l'altra categoria di informazioni componente l'assistente: quelle che ne costituiscono la base di conoscenza, indicizzate in forma numerica. L'oggetto che si occupa della gestione di questa tipologia di dato deve operare con una tecnologia diversa rispetto alle basi di dati tradizionali, garantendo le **stesse funzionalità** di inizializzazione, salvataggio, ricerca e aggiornamento: il **documents manager** (4.3).

Infine, vengono definiti alcuni moduli per la gestione delle **funzionalità aggiuntive**, che vanno dall'esigenza di fornire strumenti di traduzione a quelli di configurazione dell'assistente (mantenendo tutto in un unico luogo), la gestione della modalità di generazione delle risposte, la definizione delle API per utilizzare il chatbot, ed il concentrare le funzionalità troppo piccole per giustificare la creazione di un intero modulo all'interno di un oggetto "*d'appoggio*".

Poste le fondamenta del progetto, è possibile dedicarsi ad una panoramica dei tre moduli principali, nati dall'unione delle necessità degli utenti (3.2).

- **Question builder** (4.4): risponde alla necessità di rielaborare, quando si renda necessario, la richiesta dell'utente per migliorare il prompt associato. Questo modulo è protagonista del **prompt engineering**: una volta intercettata la richiesta dell'utente elabora una **potenziale riscrittura** sulla base di come sono state riscritte in passato richieste simili, utilizzando degli esempi. Sfrutta un modello LLM per la riscrittura dell'input.
- **Question scanner** (4.5): permette l'**analisi** della domanda, per le necessità di controllo dell'*incertezza* e gestione delle domande non pertinenti. Il modulo riceve la richiesta dell'utente e calcola, in base a diversi parametri, quanto questa sia inerente alla base di conoscenza dell'assistente, e quanto sia precisa. I diversi indicatori permettono di associare alla domanda uno specifico punteggio, sulla base del quale sia potenzialmente possibile gestire in modo discrezionale il processo di risposta.
- **Answer builder** (4.6): garantisce una chiara e corretta formattazione della risposta, per rispondere alla necessità di fornire richieste leggibili ed informative. Con un funzionamento logico simile al *question scanner*, costruisce un prompt che va ad istruire il modello LLM per un efficace formattazione della risposta. Il prompt viene generato dinamicamente utilizzando esempi che indichino la formattazione ottimale per esprimere i concetti in risposta a diverse tipologie

di domande. Sfruttando il punteggio fornito dal *question scanner*, permette di gestire domande non pertinenti, fuori contesto, oppure poco chiare.

Si propone, di seguito, un'analisi del funzionamento della pipeline, aiutata dallo schema in figura 4.1 (nello schema il modello LLM è stato rappresentato utilizzando l'icona di riferimento di OpenAI, questo *non* comporta l'imposizione di vincoli, piuttosto una rappresentazione basata su un'immagine *iconica*).

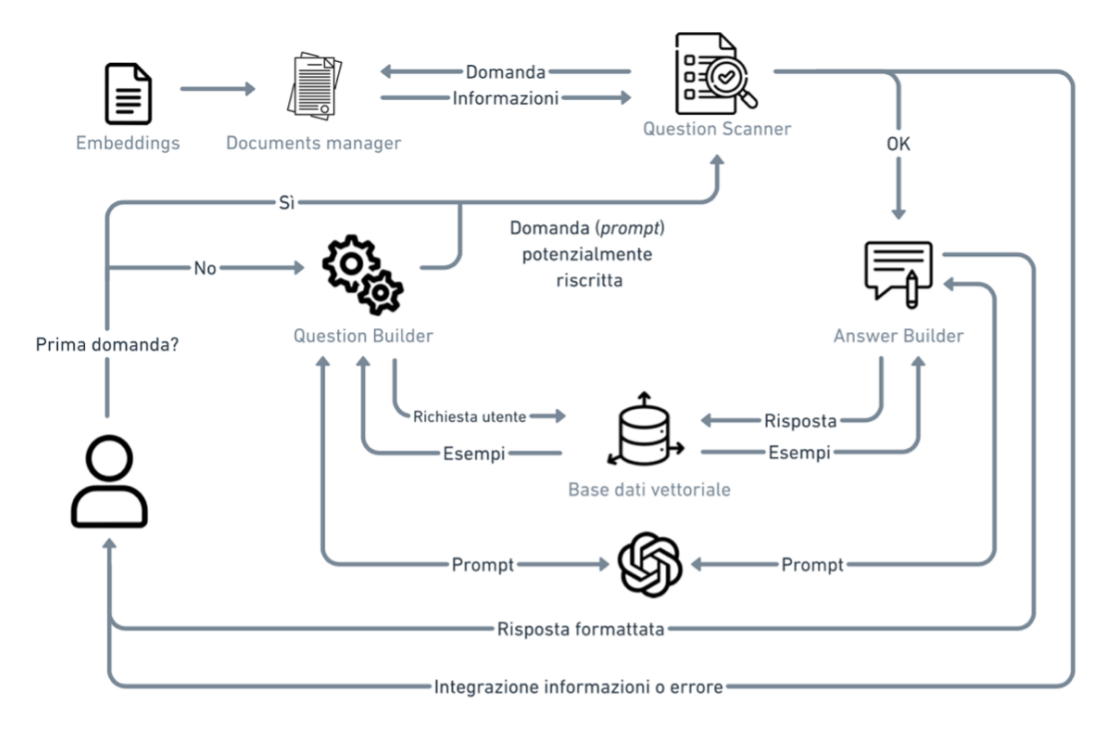


Figura 4.1: L'architettura del chatbot

Il processo ha come punto di partenza la richiesta dell'utente, che viene analizzata per capire se sia la prima o meno della conversazione. Nel caso sia la prima non si effettuano modifiche, in quanto non si hanno informazioni di contesto per renderle efficaci, mentre in caso contrario si fa intervenire il **question builder**: il modulo controlla se esistono degli esempi per la domanda in questione, compone un *few-shot prompt* e chiede al modello di migliorare la domanda sulla base delle indicazioni fornite dagli esempi. All'uscita del modulo si ha la richiesta dell'utente, *opzionalmente*, migliorata.

La domanda, modificata o meno, viene successivamente analizzata dal **question scanner**, che come primo passaggio richiede al *documents manager* un certo numero (*configurabile*, come molti altri parametri) di informazioni inerenti alla richiesta. Sulla base delle informazioni trovate e alla struttura della domanda fornisce un

punteggio, e salva al proprio interno i riferimenti alle informazioni recuperate per la domanda in analisi.

Terminato il processo di analisi si passa alla costruzione della risposta, in carico all'**answer builder**: per mantenere l'indipendenza tra i moduli è possibile procedere alla costruzione della risposta sia considerando il punteggio fornito dal question scanner, sia procedendo senza. Nell'architettura elaborata per soddisfare le esigenze degli utenti, naturalmente, il punteggio viene elaborato. La valutazione assegnata dal *question scanner* può avere all'atto pratico due conseguenze (analizzate meglio nel paragrafo 4.6):

- **KO**: la domanda *non* è ben definita, possibile che sia troppo generica o mal posta. Non si rende necessario l'utilizzo del modello LLM (con conseguente risparmio in latenza), si procede a rispondere all'utente con messaggi diversi in base alla tipologia di problematica riscontrata.
- **OK**: la domanda è stata valutata come buona, per la quale può valer la pena chiedere la generazione della risposta al modello. Sulla base del punteggio assegnato sono possibili diverse integrazioni.

Nel caso ci si trovi nella situazione definita *OK*, il modulo procede alla generazione del *few-shot prompt* cercando esempi di formattazione, basandosi sulla domanda. Il prompt viene arricchito (opzionalmente) con il contesto, selezionando parte della conversazione precedente. Una volta generata la risposta del modello, questa viene fornita all'utente (si vedranno in seguito due diverse modalità di *output*).

Quanto è stato descritto rappresenta l'architettura modulare risultante dalle esigenze valutate nel capitolo 3: i moduli sono stati realizzati tenendo in ogni caso in considerazione la possibilità di un loro utilizzo che trascenda da quanto richiesto. La struttura permette infatti un approccio diverso, più semplice, per la realizzazione di un'assistente più leggero ma meno avanzato, potenzialmente utile in altri contesti nei quali l'azienda possa necessitare di un chatbot. In aggiunta, questo tipo di divisione per "*ruoli e compiti*" rende più chiaro il funzionamento dell'intera applicazione e ne facilita, di conseguenza, la **mantenibilità** e gli eventuali sviluppi e **aggiornamenti**.

Si procede, di seguito, ad un'analisi approfondita dei singoli moduli, primari e secondari.

4.2 Conversation manager

Questo modulo si occupa dell'organizzazione delle conversazioni e di tutti i loro dati, lo schema gerarchico è quello in figura 4.2

Il modulo realizza il *requisito funzionale* **RQ5** (3.1 Recupero del contesto), l'**RQ6** (3.1 Gestione delle conversazioni), e quello *non funzionale* **RQ6** (3.2 Persistenza).

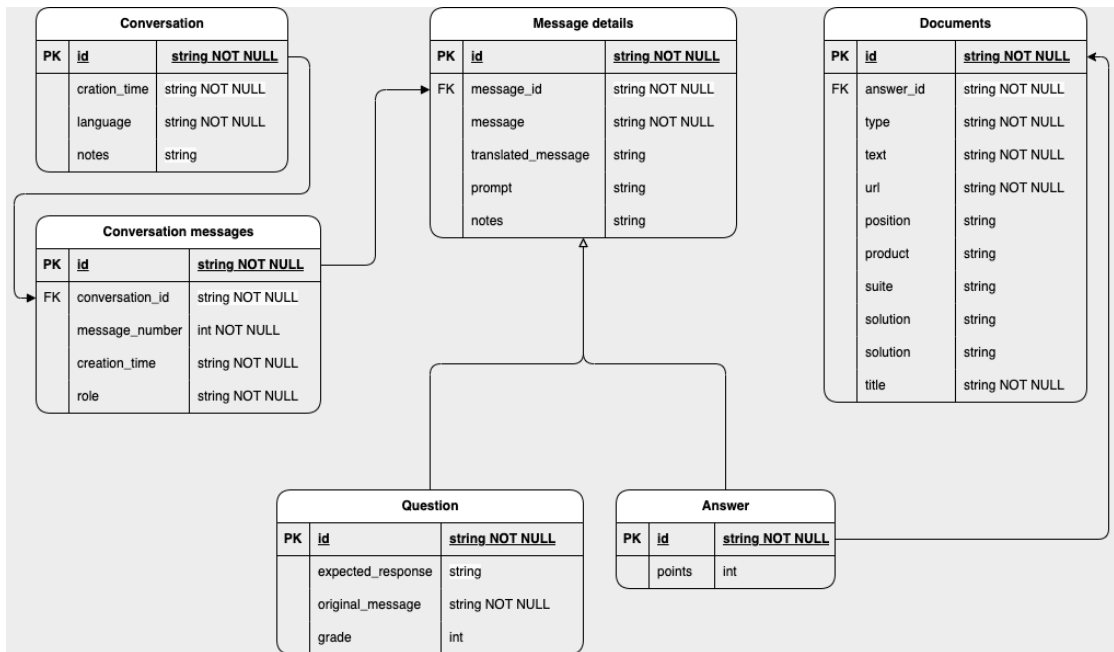


Figura 4.2: Schema base dati, conversazioni

La conversazione, l'entità gerarchicamente più alta, è associata ad un *tempo di creazione* ed un *linguaggio*: il primo dato permette l'ordinamento e la localizzazione temporale, il secondo di gestire la traduzione. Viene predisposto, in aggiunta, un campo *note*, che permette di associare eventuali stringhe di testo alla conversazione. Questo campo, come altri che verranno trattati in seguito, è stato introdotto in modo da permettere la realizzazione di test sull'efficacia del chatbot. Durante le fasi di sviluppo questa speciale tipologia di campi è stata utilizzata per tener conto dell'efficacia dei prompt generati, permettendo di **salvare valutazioni** consultabili in un secondo momento.

Associati ad ogni conversazione vi sono i messaggi, caratterizzati da una *numerazione automatica* (per una rapida visualizzazione dell'ordine nel dialogo), il *tempo di creazione* ed il *ruolo*. Quest'ultimo campo specifica se il messaggio è relativo all'utente o all'assistente, permettendo quindi di diversificarne le proprietà: i dettagli del messaggio avranno, rispettivamente, le caratteristiche di una domanda, oppure, di una risposta.

Le informazioni riguardanti i messaggi vengono estese dai dettagli, separati per maggior semplicità logica. Queste informazioni riguardano il *testo* stesso del messaggio, la sua *traduzione* (opzionale), il *prompt* utilizzato per generarlo (opzionale nel caso della domanda) e le *note* (anch'esse opzionali). Gli ultimi due campi hanno, come le note associate alla conversazione, funzionalità di valutazione, di test.

In base al ruolo del messaggio, i dettagli possono essere di due tipologie:

- **Domande**, associate al ruolo *utente*, consentono di archiviare il *messaggio originale*, ossia quello che l'utente ha espresso all'ingresso della pipeline (e che viene opzionalmente aggiornato dal *question builder*), il *voto* (grade) associato e la *risposta corretta* (l'expected response). Come è possibile intuire, gli ultimi due parametri sono di test: in particolare, l'expected response ha lo scopo di permettere di impostare domande con risposte corrette in partenza, per poi analizzare le risposte generate dal chatbot e fare un confronto. Le funzionalità di test sono messe a disposizione (verrà trattato in seguito) da API specifiche, che possono essere sfruttate per la generazione di un'interfaccia specifica di test.
- **Risposte**: Contengono il solo parametro di test *punteggio*, con le stesse funzionalità ormai ampiamente discusse.

L'ultima tabella, *Documents*, conserva le informazioni trovate nella base di dati vettoriale con le quali le risposte vengono prodotte. Gli elementi di queste speciali basi di dati possono essere organizzati secondo diversi formati; quello scelto per le necessità del progetto viene definito **Document**: rappresenta un contenuto al quale sia stata aggiunta una serie di parametri che contribuiscano a ricavare ulteriori informazioni, i **metadati**.

Ogni elemento della tabella è associato alla risposta in relazione alla quale è stato trovato e contiene, come parametri obbligatori:

- *type*, la tipologia: indica il formato dell'informazione. Nel caso del progetto in esame i "documenti" possono essere relativi a video (il cui contenuto è in forma di *sottotitoli*) oppure a documenti *pdf*.
- *text*, il contenuto.
- *url*, la posizione, l'indirizzo del documento.
- *title*, il titolo della sorgente (video o pdf) alla quale il contenuto fa capo.

Sono poi presenti una serie di parametri facoltativi, tipici della documentazione aziendale ed utilizzati per migliorare la visualizzazione delle fonti nelle risposte.

Il **conversation manager** è strutturato per essere utilizzato nel modo più semplice possibile, sempre chiarificando se si stia operando su una domanda, una risposta, e quale conversazione vi sia associata: i dettagli di implementazione verranno trattati estesamente nel capitolo dedicato.

4.3 Documents manager

Introdotta nel paragrafo precedente, si occupa di gestire l'indicizzazione e l'interrogazione delle informazioni che costituiscono la base di competenze dell'assistente: insieme al **conversation manager** completa il *requisito non funzionale* **RQ6** (3.2 Persistenza). La gestione automatizzata di nuove informazioni costituenti la base di conoscenza dell'assistente soddisfa inoltre un altro *requisito non funzionale*, l'**RQ7** (3.2 Gestione automatica delle nuove informazioni).

Queste informazioni vengono fornite all'applicativo in un formato standard, sotto forma di oggetti JSON. Il modulo, all'avvio, deve controllare in autonomia se sia o meno presente la **collezione** di dati indicizzati.

Una collezione è un “**contenitore logico**” per vettori ed i loro metadati associati, contiene:

- **Vettori**: in questo caso rappresentazione numeriche di dati testuali.
- **Metadati**: descrivono i dati testuali, possono essere sfruttati per filtrare e migliorare le interrogazioni.
- **Schema**: come una base dati tradizionale, definisce la struttura dei dati che contiene, includendo la dimensione dei vettori ed informazioni sui metadati.

Come anticipato nel paragrafo precedente, questo assistente organizza i dati sotto forma di *Documents* ossia informazioni testuali alle quali sono associati dei metadati, informazioni aggiuntive. Quando, da ora in avanti, si utilizza il sostantivo **Documents** (e derivati), si indica l'insieme di queste informazioni.

Il modulo dopo aver eventualmente creato una nuova collezione nel caso non sia presente, procede alla ricerca di nuove informazioni (processo configurabile) da aggiungervi. Un processo fondamentale quando si inseriscono nuovi dati all'interno di una collezione è il controllo preventivo della loro eventuale presenza: la natura vettoriale della collezione non impedisce l'inserimento di duplicati, che però, com'è intuibile, vanno a sporcare le ricerche.

All'atto di inserimento di una nuova informazione si deve quindi controllare se questa non sia già presente: per realizzarlo si fa un'interrogazione, utilizzando come *query* il contenuto che si sta provando ad aggiungere. Le interrogazioni possono restituire, insieme al contenuto trovato, anche un punteggio di distanza tra questo e la *query* di interrogazione. Sulla base del punteggio è possibile valutare quanto i due vettori siano vicini, quindi selezionare una soglia molto alta oltre la quale non inserire nulla, in quanto il dato è troppo simile a quello già presente (se non identico).

Una volta definito questo approccio il modulo può inicializzarsi in autonomia, mettendo poi a disposizione le seguenti funzionalità:

- **Aggiunta di nuove informazioni:** fornisce un'interfaccia per arricchire la collezione con nuovi *Documents*, aggiunti rispettando le stesse logiche citate in precedenza.
- **Ricerca:** interfaccia che permette di effettuare la ricerca nella collezione, basandosi su contenuto testuale. La ricerca deve essere ottimizzata sfruttando le parole chiave possibilmente presenti all'interno della richiesta, utili per filtrare mediante i metadati, i risultati. Il numero di documenti trovati deve essere configurabile.
- **Ricerca con punteggio:** gemella della ricerca al punto precedente, offre anche il punteggio di distanza rispetto alla chiave di ricerca.

4.4 Question builder

Il modulo che realizza il **prompt engineering**, e quindi anche il *requisito funzionale RQ1* (3.1 Generazione automatica dei prompt).

Prima di andare a fondo nelle dinamiche di generazione di prompt, è opportuna una panoramica di background.

Similmente al **Documents manager**, anche questo modulo (analogamente, si vedrà in seguito, l'*Answer builder* (4.6)) sfrutta le funzionalità dell'indicizzazione vettoriale: il modulo impiega una chiamata al modello LLM di riferimento per il miglioramento della richiesta dell'utente, chiamata che avviene mediante un prompt del tipo *few-shot*. Questo prompt richiede degli esempi, che devono di conseguenza essere trovati basandosi sulla similarità con quella che è la domanda dell'utente. La tecnologia da utilizzare è quindi la stessa impiegata per la documentazione.

Come per il modulo di gestione dei documenti, anche questo deve, una volta attivato, inizializzare una collezione contenente degli esempi, con le stesse modalità descritte nel paragrafo precedente.

Deve essere anche predisposta la funzionalità di aggiunta di nuovi esempi, strutturata in modo da impedire l'aggiunta di dati già presenti all'interno della collezione.

4.4.1 Una struttura per tracciare il contesto: lo *Snippet* della conversazione

Prima di procedere all'analisi degli esempi, è bene introdurre una struttura che farà la sua comparsa tutto dove sia necessario indicare una sezione di conversazione. Questa struttura viene utilizzata nella composizione dei prompt che devono poi essere utilizzati da modello, ed è stata quindi progettata con l'obiettivo di mantenere abbastanza informazioni di contesto ma non appesantire l'LLM con

troppa cronologia della conversazione (che sortisce l'effetto opposto, aumentando la confusione del modello): lo **snippet** rappresenta logicamente l'ultimo "pezzetto" della conversazione tra l'utente e l'assistente, ed è strutturato come segue:

1. Domanda dell'utente.
2. Risposta dell'assistente.
3. Ultima domanda dell'utente (per la quale non è ancora stata formulata una risposta).

Questi tre elementi, si vedrà successivamente, contengono i riferimenti alle informazioni presenti nella base dati delle conversazioni, in modo da poter risalire velocemente a tutte le informazioni correlate.

4.4.2 Riscrittura della richiesta dell'utente: gli esempi

Gli esempi devono aiutare il modello nella produzione, eventuale, di un nuovo *prompt utente*: si distingue il prompt utente da quello generale nel quale viene incorporato.

Si propone, di seguito, un'analisi degli obiettivi degli esempi.

Mantenimento del contesto

Nello scambio di informazioni che avviene tra un assistente e l'utente è naturale che quest'ultimo, se il chatbot è avanzato, non abbia la necessità di specificare ogni volta un contesto.

Questo punto, inoltre contribuisce al rispetto del *requisito funzionale RQ5* (3.1 Recupero del contesto). Per garantire questa funzionalità, deve intervenire un'aggiunta di informazioni che integri la domanda.

Per capire meglio questa funzione è utile considerare il seguente *snippet* di conversazione:

Utente: Come funziona il motore di un'automobile a benzina?
Assistente: Nel caso di un'automobile, un motore a benzina...
Utente: E a diesel?

Si pensi alla struttura della pipeline descritta in figura 4.1: se l'ultima domanda non venisse modificata non solo il modello avrebbe difficoltà nel risalire al contesto della domanda, problema che potrebbe venir arginato integrando lo snippet della conversazione nel prompt più generale, ma se ne figurerebbe uno più serio: l'ultima domanda rappresenta la query di ricerca nella base di dati vettoriale contenente le informazioni sulla base delle quali il modello deve rispondere all'utente. Facendo

sempre riferimento all'esempio si pensi ad un LLM che non abbia informazioni sui motori: si deve quindi fare affidamento ai dati che gli vengono passati mediante RAG. Una domanda così generica potrebbe anche intercettare tra i vettori informazioni riguardanti motori, ma il modello riceverebbe sicuramente informazioni confuse, magari di motori diversi (navali, generatori, automobili, ecc..) e la risposta prodotta sarebbe confusa come la ricerca.

Evidente la necessità di riscrivere l'ultima domanda; una possibilità, tra le molte, è la seguente:

Utente: Come funziona il motore di un'automobile a diesel?

L'esempio prodotto, quindi, sarà l'unione dello **snippet** con l'**ultima domanda riscritta**.

Un altro obiettivo raggiungibile impiegando gli esempi opportuni è la riscrittura mirata di alcuni input, che può portare a risultati più efficaci da parte dell'LLM. Questa tecnica abilita due funzionalità, che possono anche essere complementari.

Riscrittura mirata: impostare il tono della risposta

È possibile abilitare diverse risposte sulla base degli utenti che utilizzano l'assistente. Nelle considerazioni fatte nel capitolo 2 si è descritta la necessità di un assistente "*didattico*": questa tipologia di chatbot deve tenere conto della scarsa conoscenza degli utenti nell'ambito di conversazione, adeguandone quindi il tono. Questo obiettivo può essere raggiunto impostando un prefisso statico nei prompt globali, richiedendo un tono più "scolastico" nelle risposte generate, ma è possibile fare di più. Si consideri il seguente esempio:

Utente: Quali differenze ci sono tra un motore a diesel ed un motore a benzina?

Questa domanda implicitamente chiede al modello di fare un confronto tra due elementi diversi: in base all'LLM utilizzato saranno possibili diverse risposte, alcune migliori di altre, ma la presentazione delle informazioni sarà tutta a discrezione del modello.

Sfruttando gli esempi è invece possibile fornire delle istruzioni specifiche, senza che l'utente debba intervenire, per guidare la generazione della risposta. Prendiamo sempre in analisi l'esempio appena fatto: in questo caso ci si trova davanti ad un confronto, si potrebbe procedere in questo modo con la riscrittura:

Utente: Elenca le differenze presenti tra un motore diesel ed un motore a benzina, utilizzando una tabella. Esponi le informazioni in modo chiaro e semplice. Utilizza un tono didattico.

_____ *oppure* _____

Utente: Fai un elenco puntato delle principali differenze tra un motore a benzina ed un motore diesel. Esponi le informazioni in modo chiaro e semplice. Utilizza un tono didattico.

La sfida nell'applicare questo approccio è fornire nella fase di riscrittura degli esempi simili a quanto l'utente abbia chiesto: è possibile predisporre un set di esempi abbastanza mirati, almeno inizialmente (ad esempio in questi casi di confronto), in quanto tendenzialmente si conosce la documentazione sulla quale il chatbot basa la sua conoscenza e di conseguenza le domande più frequenti che gli utenti possono trovarsi a fare. Nel caso di questo progetto sono state analizzate le domande fatte dagli utenti per cercare le risposte necessarie a risolvere una serie di task (in passato, prima che si iniziasse a lavorare a questa generazione dell'assistente): da queste emergono dei pattern, attorno ai quali è possibile cucire degli esempi.

Riscrittura mirata: integrare informazioni specifiche

Questo caso richiede un'ottima conoscenza della base documentale. Spesso accade che la domanda generica di un utente sia in realtà relativa ad un aspetto specifico presente nella base di conoscenza dell'assistente, ma non venga espressa nella richiesta. L'integrazione in questo caso deve fare ampio affidamento allo *snippet* di conversazione per capire quale sia l'argomento specifico al quale l'utente vuole arrivare (o che, nel caso non ne sia nemmeno consapevole, sia necessario trovare, per poter rispondere nel modo più corretto possibile). Per spiegare meglio questo concetto si propone nuovamente un esempio, in linea con quelli precedenti. Si supponga quindi di aver un'assistente che basi la propria base di conoscenza sui motori. Si sa che la casa automobilistica X produce un modello di auto che può montare un motore a benzina o un motore diesel, rispettivamente Y_B e Y_D . I due modelli di auto hanno informazioni diverse, quindi specificare quale sia il modello per il quale si desidera avere informazioni è importante. Si supponga la seguente conversazione:

Utente: Quali motori offrono le prestazioni maggiori tra quelli prodotti dall'azienda X ?

Assistente: I motori a benzina offrono le prestazioni migliori, per i seguenti motivi ...

Utente: Mi piace il modello Y , dammi più informazioni

Si può intuire come la ricerca delle informazioni in carico al *document manager* possa migliorare nel caso l'ultima domanda dell'utente venga riscritta come:

Utente: Fammi una panoramica professionale del modello Y_B prodotto dall'azienda X , sottolineando specialmente gli aspetti legati alle prestazioni

Evitare riscritture non necessarie

Essendo il compito di riscrivere la domanda dell'utente affidato ad un modello, è fondamentale che questo non stravolga il significato della richiesta. Dopo numerose sperimentazioni si è giunti alla conclusione che il modo migliore per evitare che questo cambi il significato delle richieste (specialmente di quelle leggermente ambigue) sia anche qui l'utilizzo di esempi.

In questo caso si propone uno *snippet* con, al posto dell'indicazione di riscrittura, un carattere di default, in questo caso ‘-’.

In questo modo si segnala all'LLM di non cambiare nulla (eventualmente, si è visto, la richiesta dell'utente viene riscritta per correggere un errore ortografico).

4.4.3 Aggiornamento della domanda: struttura del prompt

Dopo aver introdotto il concetto di *snippet* e fatto una panoramica degli esempi, descrivendone le possibilità, è possibile affrontare la struttura del prompt globale, quello che si occupa di richiedere al modello la riscrittura della domanda dell'utente. La struttura si divide in tre sezioni:

- **Prefisso:** include l'introduzione del ruolo che avrà il modello e la descrizione di quanto riceverà in input. Segue un primo riassunto di istruzioni e l'introduzione alla sezione successiva.
- **Sezione esempi:** elenco degli esempi, strutturati come snippet più la risposta riscritta.
- **Suffisso:** chiude il prompt l'istruzione finale, che chiarisce come l'unico risultato debba essere la risposta riscritta. Viene specificato infine lo snippet legato al quale è richiesta l'operazione

Di seguito si propone la versione tradotta (nel chatbot operativo viene gestito tutto in lingua inglese) di un prompt completo; la parte centrale, quella degli esempi, contiene il *template* sulla base del quale avviene, in esecuzione, la selezione dinamica (in numero degli esempi è, come anticipato, configurabile). Si noti l'utilizzo della divisione in paragrafi, così come quella in sezioni mediante i caratteri ‘‘---’’.

Tutto ciò che compare tra le parentesi graffe ({...}) indica un'etichetta nella quale verranno inseriti i dati.

Sei un assistente virtuale che aiuta a riscrivere l'ultima domanda in una chat per mantenere il contesto delle domande precedenti.

Ti verrà fornita l'ultima interazione tra l'utente e l'assistente, più l'ultima domanda dell'utente.

Devi riscrivere la domanda solo ed esclusivamente se essa non si riferisce a un'altra specifica: ogni volta che la domanda riguarda qualcosa di specifico, deve essere lasciata invariata.

Ecco alcuni esempi, se la risposta riscritta è ‘-’ significa che in quel caso la risposta non viene riscritta:

```
---
utente: {utente}
assistente: {assistente}
ULTIMA_DOMANDA_UTENTE: {ultima_domanda_utente}
DOMANDA_RISCRITTA: {ultima_domanda_riscritta}
---
```

DEVI rispondere solo con la risposta (riscritta o meno) basata sulla seguente conversazione.

```
utente: {utente}
assistente: {assistente}
ULTIMA_DOMANDA_UTENTE: {ultima_domanda_riscritta}
```

Non viene riportato qui, ma è stato predisposto un prompt di generazione più generale nel quale viene chiesto al modello di riscrivere la domanda senza l'utilizzo di esempi: è stato introdotto per gestire i casi nei quali non esistono

esempi abbastanza buoni per la richiesta dell'utente ma si vuole comunque utilizzare appieno il modulo per il *mantenimento del contesto*.

Per concludere la trattazione del modulo, viene predisposta un'interfaccia che consenta l'**aggiunta di esempi** (inerenti al *question builder*) alla base di dati vettoriali.

4.5 Question scanner

Questo modulo si occupa dell'analisi delle richieste dell'utente, e nasce dalla necessità di rispondere ad una sfida complessa: capire se la domanda dell'utente sia pronta per essere processata dal modello LLM oppure no.

Le funzionalità implementate vanno a soddisfare il *requisito funzionale RQ2* (3.1 Analisi dell'incertezza per follow-up), e parte del *requisito non funzionale RQ3* (3.2 Ottimizzazione della latenza).

Il *question scanner* è fondamentale per migliorare l'esperienza dell'utente, attraverso le funzionalità analizzate di seguito.

Evitare allucinazioni da parte dell'LLM

Intercettare domande troppo vaghe o mal formulate permette di non porle al modello, che altrimenti tenderebbe a formulare risposte allucinate, inutili e fuorvianti.

Permettere domande di follow-up

Punto fondamentale, riuscire a produrre una domanda al posto di una risposta. Nel linguaggio convenzionale tra due persone, quando la prima pone un quesito all'altra, ma alla seconda non è chiaro, è naturale che quest'ultima, invece di rispondere, chieda ulteriori informazioni per capire meglio l'oggetto della conversazione. Nella normale implementazione di un assistente virtuale questo non accade, l'assistente *tenta* comunque una risposta, anche quando la base di informazioni sulla quale può costruirla è scarsa e debole.

Ne consegue che, se si riesce a individuare quelle che siano classi di domande non ben poste, sia possibile rispondere all'utente con la richiesta di integrazione di ulteriori informazioni, possibilmente fornendo un'elenco dei possibili contesti.

Per capire meglio questo concetto si propone un esempio. Si supponga che l'utente ponga la seguente domanda (stesso contesto degli esempi fatti in precedenza):

Utente: Come funziona un motore?

Dopo una scansione della domanda è possibile intercettare un certo grado di inconsistenza, dovuta alla mancata specificità della tipologia di motore, e del campo di applicazione.

Si ricordi che il contesto nel quale opera il chatbot di riferimento è quello di assistenza mirata per argomenti specifici: certo, in questo caso sarebbe possibile rispondere alla domanda dell'esempio mediante una descrizione generica di cosa sia un motore, ma nell'applicazione reale una domanda simile, su un argomento specifico nel contesto aziendale, non può considerarsi ottimale (gli esempi cercano di rendere chiaro il concetto, senza utilizzare argomenti poco conosciuti).

A seguito della scansione è possibile prevedere una risposta statica, senza passare per un'interrogazione al modello LLM, che potrebbe in questo caso essere:

Assistente: Ti chiedo scusa, ma per rispondere al meglio a questa tua domanda avrei bisogno di informazioni un po' più specifiche, a cosa ti stai riferendo? Di seguito trovi una lista di possibili argomenti:

- Motori benzina, categoria automobili.
- Motori diesel, categoria automobili.
- Motori diesel, categoria autocarri.
- ...

Risulta evidente come questo approccio possa potenziare notevolmente l'esperienza utente, specialmente nei confronti di tutti quegli utenti che non abbiano (come nel caso di riferimento di questo progetto) una conoscenza approfondita degli argomenti per l'approfondimento dei quali stanno interrogando l'assistente.

Arricchimento delle risposte

A volte, sebbene la domanda dell'utente sia buona, è possibile che la richiesta trovi possibili sviluppi in contesti diversi (o *leggermente* diversi) da quelli espressi.

Analizzandola è possibile richiedere una risposta al modello aggiungendo anche una lista di spunti, che l'utente avrà la possibilità di esplorare in libertà nel caso ne sia incuriosito, o nel caso capisca che l'argomento desiderato si esprima in un diverso panorama contestuale.

Nella progettazione di queste funzionalità è stata presa inizialmente in considerazione l'ipotesi di affidare l'operazione nuovamente ad un modello LLM. Sono state fatte delle prove, con diversi prompt, ma questa strategia è stata esclusa per due motivi:

- **Latenza:** prima su tutte vi è il ritardo aggiuntivo causato dall'introduzione anche in questo modulo di una chiamata al modello. Questo è problematico anche per i costi, che aumenterebbero di conseguenza.
- **Efficacia:** Dopo numerosi tentativi si è visto come la discrezionalità dei principali modelli LLM provati non sia ottimale nel determinare la qualità di una richiesta: è possibile che in futuro, con l'evoluzione degli LLM, si evolvano anche le funzionalità di interpretazione delle domande (e quindi i modelli diventino capaci, in autonomia, di determinare quando rispondere con una richiesta di chiarimento).
Al momento della progettazione di questo assistente i modelli più avanzati testati (*Gpt-4o* e *Gemini 1.5 pro*) non si sono rivelati efficaci.

Passiamo ora all'analisi dei parametri sulla base dei quali il modello offre la propria valutazione.

4.5.1 Distanza media

Il parametro più semplice da calcolare. Nel paragrafo 4.1, nella trattazione di come la pipeline funzioni, si è descritto come il *question scanner* si occupi, nella sua analisi, anche di richiedere al modulo *documents manager* un certo numero (configurabile) di documenti coerenti alla domanda dell'utente. La ricerca delle informazioni messa a disposizione dal modulo di gestione dei documenti può essere effettuata ricevendo anche, come risultato, il punteggio di distanza tra il singolo documento trovato e la chiave di ricerca, la domanda. Andiamo ora nello specifico di questo valore.

Le tecnologie di basi dati vettoriali permettono diverse modalità di configurazione per il calcolo della distanza [21]: quando vengono inizializzati ne va specificata una precisa, in base alle diverse esigenze.

Nel contesto del progetto in analisi, la misura di distanza che si è rivelata migliore tra quelle messe a disposizione della tecnologia utilizzata (ChromaDB, trattato in seguito) è stata la **Cosine similarity** [22]. Questa metrica misura la **similarità** tra due vettori utilizzando come criterio l'**angolo** tra i due, calcolandone il coseno.

La scelta di questa misura, sebbene non fosse la predefinita, è stata fatta in merito all'intervallo di valori che produce come risultato: la metrica generale prevede, come sappiamo, che il coseno assuma valori nell'intervallo $[-1, 1]$. Nell'applicazione all'estrazione di testo, come nel caso trattato, non sono possibili valorizzazioni negative di vettori, di conseguenza il range di valori possibili di similarità si restringe a $[0, 1]$: questo è importante in quanto tutte le metriche utilizzate da questo modulo sono state per conformità e semplicità definite nel medesimo intervallo.

Una valore di *Cosine similarity* pari a 0 corrisponde a due vettori coincidenti, identici, mentre al valore 1 si hanno due vettori completamente diversi, ortogonali.

Applicando questo concetto alla ricerca nella base di dati vettoriale è immediato capirne l'utilità: tanto maggiore sarà il punteggio di *Cosine similarity* quanto più il documento (“pezzo” di informazione) trovato sarà distante dalla richiesta dell'utente.

Dal momento che esiste la possibilità di misurare, con un valore compreso nell'intervallo $[0,1]$, quanto ad una richiesta si avvicinino le informazioni, è logica la seguente deduzione:

A partire da una richiesta, dato un certo numero di Documents da estrarre dalla base di dati vettoriale, quanto più la *Cosine similarity* media sarà alta, quanto meno la domanda dell'utente sarà inerente alle informazioni presenti.

Un valore alto di *Cosine similarity* non implica necessariamente che la domanda sia posta in modo errato, in quanto l'utente potrebbe star chiedendo informazioni relative ad un argomento non presente nella base di conoscenza dell'assistente.

Nel caso in esame, però, si è deciso di restringere le possibilità di interazione con il chatbot al solo ambito professionale delimitato dalla base dati: ne segue come questa metrica sia fondamentale per smussare, sin dal primo momento, le richieste.

4.5.2 Entropia

L'entropia è utilizzata in questo contesto come **misura del disordine**. In fase di progettazione si è fatto il seguente ragionamento: sarebbe utile poter dedurre se la domanda dell'utente sia vaga oppure specifica, ma come si può fare in modo generale, senza analizzarne la struttura?

Si potrebbe misurare, complessivamente, il **risultato** che questa produce rispetto all'operazione di ricerca di informazioni: vanno di conseguenza considerati i documenti estratti nel loro insieme, non singolarmente.

Le informazioni estratte dalla base di dati vettoriali possono appartenere ad un'unica sorgente o a sorgenti diverse. Le sorgenti sono i file originali dai quali le informazioni vengono estratte: come analizzato in precedenza, i dati estratti contengono, oltre al testo, una serie di metadati che identificano, tra le altre cose, il documento (il file) di origine, ad esempio un *pdf*.

Da proprietà, un altro ragionamento logico:

Quanto più le informazioni estratte siano derivanti da sorgenti diverse, quanto più la domanda è generica. Le informazioni relative ad una domanda specifica, infatti, sono di norma presenti in una sola sorgente (ad esempio un documento PDF con le informazioni su uno specifico modello di auto a benzina, per richiamare gli esempi fatti in precedenza)

Misurando quindi il grado di disordine, di **entropia**, relativo alle sorgenti delle informazioni, è possibile determinare il grado di generalità della domanda: questo apre alla possibilità di definire quando dover richiedere ulteriori informazioni.

L'entropia viene **calcolata** utilizzando il **Teachman index** [23], *normalizzato*, in modo da ottenere esclusivamente valori nel solito intervallo $[0,1]$.

L'indice viene calcolato come il prodotto tra la somma negativa delle frequenze ed il logaritmo della frequenza per ogni informazione estratta. Il risultato viene poi diviso per il logaritmo del numero totale di informazioni per normalizzarlo.

$$\text{Teachman Index} = - \sum_{i=1}^k p_i \cdot \log(p_i)$$

$$\text{Entropy} = \frac{\text{Teachman Index}}{\log(N)}$$

- N è il numero totale di documenti.
- p_i è la frequenza relativa del documento i (ovvero il numero di occorrenze del documento i diviso per N).
- La somma $\sum_{i=1}^k$ viene effettuata su tutti i documenti unici k .

Agli estremi dell'intervallo di possibili valori corrispondono, quindi:

- 0.00: le informazioni estratte appartengono tutte allo stesso documento. Questo è il caso migliore.
- 1.00: le informazioni estratte appartengono tutte a documenti diversi. Caso peggiore.

4.5.3 Qualità strutturale

La qualità strutturale rappresenta una metrica di fine, non impattante come le due descritte nei paragrafi precedenti.

Lo scopo è quello di calcolare la qualità strutturale della domanda fornita dall'utente, ossia una stima di quanto bene sia stata formulata, focalizzandosi sulla **proporzione** tra le parole significative, rispetto al totale.

Dalla domanda vengono rimosse le cosiddette "*stop words*", ossia le parole comuni che non aggiungono significato alla frase (come "il, lo, la,..."): la qualità strutturale viene semplicemente calcolata come il **rapporto** tra le parole risultanti da questa operazione di filtraggio con il totale. Anche qui l'intervallo di valori che risultano dall'operazione è $[0,1]$.

Il limite inferiore coincide con frasi senza significato.

Questa metrica, come analizzato nel paragrafo successivo, è di ausilio alle altre due come discriminante nei casi limite.

4.5.4 Determinare il punteggio

Il punteggio complessivo della domanda viene determinato seguendo un diagramma di flusso, che assegna un punteggio da 0 a 5, dove ad ogni numero corrisponde uno stato:

- 0 - OK: la domanda viene considerata buona, nessun operazione necessaria.
- 1 - OK_PLUS_SOLUTIONS: la richiesta è considerata buona, ma risulta in parte generica, si consiglia quindi di aggiungere le fonti per offrire più informazioni all'utente.
- 2 - KO_SOFT_INFO_SUBJECT: non si inoltra nulla al modello, la domanda non ha un soggetto ben formulato. Si potrebbe rispondere con la richiesta di più informazioni, specificamente riguardo al soggetto, fornendo anche le sorgenti che si sono trovate.
- 3 - KO_SOFT_INFO_CONTEXT: simile al punto precedente, ma qui non è il soggetto a mancare, bensì il contesto.
- 4 - KO_HARD_INFO_CONTEXT: I due punti precedenti sono considerati *soft*, ossia è possibile, chiedendo chiarimenti all'utente, trovare le informazioni per produrre una buona risposta. In questo caso invece la domanda è troppo generica, ma si può provare ancora a dare una lista di possibili contesti tra i quali cercare, avvisando però che per quella domanda, formulata in quel modo, non si garantisce una risposta completa.
- 5 - KO: Caso limite, la domanda è molto mal posta e nemmeno si riescono a trovare le sorgenti, si consiglia di rispondere con un messaggio che esprima l'incapacità di rispondere.

Il diagramma di flusso in figura 4.3 è il risultato di numerose combinazioni dei tre parametri. Dopo una serie di test è emersa questa combinazione.

Per ognuno dei parametri di entropia e distanza media sono stati trovati due valori che permettessero di definire delle soglie di confronto. Per il parametro di qualità strutturale si è deciso di definire solamente un limite inferiore. Di seguito i parametri numerici, che nel progetto sono comunque configurabili come variabili di ambiente:

- Per la *cosine distance* media, identificata con AVG_D :
 - D_1 : 0.65
 - D_2 : 0.75
- Per l'entropia, identificata con E :

– E_1 : 0.55

– E_2 : 0.8

- La qualità strutturale, identificata come SQ , ha soglia minima $SQ_{min} = 0.4$

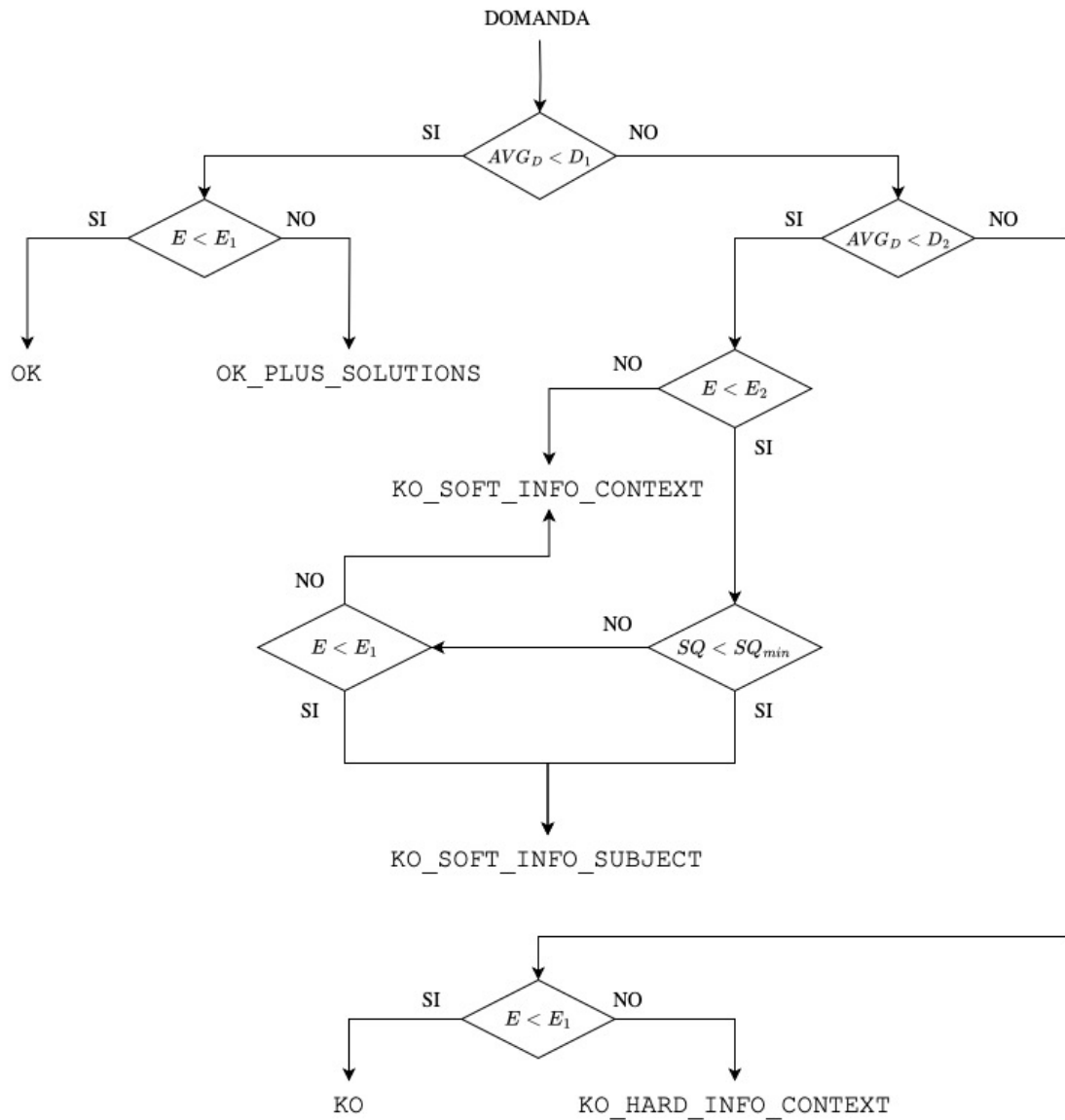


Figura 4.3: Diagramma di flusso: processo di assegnazione punteggio alla domanda

4.6 Answer builder

L'ultimo dei moduli principali, ottimizza la formattazione delle risposte all'utente.

Realizza, sulla base delle informazioni ricavate mediante il *question scanner*, il *requisito funzionale RQ3* (3.1 Gestione delle domande non pertinenti). Sfruttando le informazioni recuperate dal *document manager* rispetta poi il *requisito funzionale RQ4* (3.1 Gestione delle fonti), e mediante l'utilizzo dello *snippet* l'*RQ5* (3.1 Recupero del contesto).

Per quanto riguarda invece i *requisiti non funzionali*, il modulo implementa l'*RQ3* (3.2 Ottimizzazione della latenza) mediante la *streamed mode*, così come l'*RQ4* (3.2 Formattazione delle risposte).

Vengono condivise logicamente con il *question builder* le funzionalità di organizzazione degli esempi e la conseguente gestione di una collezione di dati indicizzati sotto forma di vettori: come si è visto in precedenza per guidare la riscrittura delle domande, anche nel caso dell'*answer builder* si adotta un approccio "ad esempi". La struttura del prompt è quindi organizzata in modo simile, con prefissi e suffissi, sfruttando la selezione di esempi dinamica sulla base della domanda.

Sempre nel paragrafo dedicato al *question builder* si è analizzato come, sulla base della richiesta dell'utente, fosse possibile far sì che la domanda venisse riscritta richiedendo una formattazione specifica sulla base del contesto.

L'*answer builder* si propone nello specifico di estendere questo concetto, ampliandolo: riscrivere le domande dell'utente richiedendo una formattazione specifica aiuta il modello, ma spesso lascia un grado di libertà troppo ampio, che porta a formattazioni spesso inconsistenti. Applicare però lo stesso principio, ossia quello di fornire un certo numero di esempi guida, incanala in modo efficace il risultato. Sulla base di necessità tecniche, si è deciso di fornire tutti gli esempi in formato HTML: è fondamentale far sì che il modello risponda nella modalità ottimizzata per la tipologia di interfaccia sulla quale si prevede la visualizzazione, vista la difficoltà di personalizzarla in un secondo momento, essendo il testo generato imprevedibile.

4.6.1 Punteggi e costruzione delle risposte

Il modulo prevede due modalità di funzionamento: una che faccia affidamento sul punteggio fornito dal *question scanner*, l'altra statica.

La modalità statica è simile nella costruzione a quella vista per il *question builder*: vi è un prompt standard con un prefisso ed un suffisso non variabili, con una serie di esempi dinamici, che vengono ricavati dalla specifica collezione vettoriale contenente domande con relative risposte dedicate. Questa versione è stata prevista per garantire il funzionamento del modulo senza che questo preveda forzatamente un accoppiamento con il *question scanner*, ma si preferisce di seguito

un'analisi della modalità che sfrutti il punteggio, che in ogni caso comprende anche il funzionamento della modalità "standard".

La modalità che sfrutta il punteggio dato dallo scanner ne completa il discorso iniziato nel paragrafo dedicato: non sono solo gli esempi a cambiare nella generazione della risposta, ma anche il prompt stesso.

Si propone un riassunto delle casistiche con relativa descrizione.

Caso OK

Il prompt è quello standard, la richiesta dell'utente è ottimale. Di seguito il prompt utilizzato:

```
Sei un chatbot che aiuta un utente che ha studiato diversi documenti sul funzionamento di prodotti aziendali, aiutandoli rispondendo a domande relative alla documentazione.
```

```
Crea una risposta concisa e informativa per una data domanda basandoti esclusivamente sui documenti forniti.
```

```
La risposta deve seguire le linee guida di leggibilità ed accessibilità integrando i principi dell'HCI
```

```
Fondamentale produrre le risposte in formato HTML.
```

```
Alla fine della risposta, DEVI citare i documenti di origine (SOLO E SOLO SE ne hai utilizzate le informazioni).
```

```
Qui puoi trovare alcuni esempi di domande e risposte: usa lo STESSO formato HTML degli esempi, adattato al tuo caso specifico.
```

```
===
```

```
{domanda}  
{risposta}
```

```
....
```

```
===
```

```
Di seguito trovi le informazioni sulla base della quali fornire la risposta, che DEVE essere in formato HTML:
```

```
{documenti}
```

```
Domanda: {input_utente}
```

```
Risposta:
```

Caso OK con soluzioni correlate

Si richiede la lista dei documenti trovati dallo scanner e si prepara un prompt nel quale, oltre a richiedere la risposta al modello, si indica anche di riportare una serie di possibili sorgenti tra le quali è possibile che si trovi l'argomento della domanda.

In questo caso, rispetto al prompt del punto precedente, cambia il suffisso, che diventa:

```
Di seguito trovi le informazioni sulla base della quali
fornire la risposta, che DEVE essere in formato HTML:
```

```
{documenti}
```

```
Perfavore, costruisci la risposta e specifica, alla fine
la possibilità per te di fornirne un'altra potenzialmente
migliore se l'utente lo desidera, specificando un contesto
più preciso: elenca la lista delle opzioni tra le quali
l'utente può cercare, nella la lista che segue.
```

```
Soluzioni: {soluzioni}
```

```
Domanda: {input_utente}
```

```
Risposta:
```

Caso KO a causa del soggetto, con possibilità di recupero

Si prevede un messaggio di default, senza passare per il modello.

```
Sono davvero dispiaciuto. Sarebbe possibile ricevere
maggiori informazioni circa il soggetto della tua domanda?
```

Esiste la possibilità di includere nella risposta pre-generata, per aiutare l'utente, una lista di possibili argomenti (in questo caso documenti) che potrebbero essere correlati alla domanda: sono le informazioni più vicine a quanto richiesto, ma si avvisa l'utente che non si ha la certezza che quanto chiesto sia li disponibile.

Nel prossimo caso è presente un esempio in questo senso.

Caso KO a causa del contesto, con possibilità di recupero

Come per il punto precedente, cambia però il messaggio, con integrazione di informazioni che vengono aggiunte successivamente, in forma di elenco puntato:

```
Sono davvero dispiaciuto.  
Sarebbe possibile ricevere maggiori informazioni  
circa il contesto della tua domanda?<br/>  
Basandomi su quanto hai chiesto, è possibile che  
l'argomento che cerchi ricada in una di queste soluzioni:
```

KO con poche possibilità di recupero, a causa del contesto

Simile ai due punti appena trattati, ma il messaggio ha carattere differente, improbabile trovare una risposta.

```
Sono davvero dispiaciuto, ma non sono stato capace di  
trovare le informazioni che stai cercando.<br/>  
Se vuoi, posso tentare una ricerca migliore nel caso tu  
riesca a darmi qualche informazione in più sul contesto,  
ma non posso comunque garantire di poter rispondere  
in modo migliore alla domanda, mi scuso ancora per  
l'inconveniente.<br/> Nel caso tu voglia ritentare, trovi  
di seguito alcune soluzioni che potrebbero aiutarti:  
le ho dedotte basandomi sulla rilevanza che sono  
riuscito a dedurre.
```

KO senza possibilità di recupero

Messaggio di errore, impossibile trovare una risposta.

```
Sono mortificato, ma con la mia conoscenza attuale non sono  
capace di rispondere alla tua domanda.<br/>  
Mi scuso nuovamente per l'inconveniente.
```

Si noti come i punti che prevedono risposte dirette per l'utente siano già in formato HTML.

4.6.2 Streaming

Lavorare con i modelli LLM richiede un'attenzione particolare ad una serie di funzionalità legate all'aspetto UI/UX, prima tra tutte la **gestione della latenza**.

Aspettare la risposta dal modello richiede infatti che questo abbia ricevuto la richiesta ed elaborato completamente una soluzione. Nel contesto della pipeline in esame, poi, il flusso ottimale prevede una **doppia** interrogazione: una per riscrivere la domanda dell'utente ed una per la risposta.

Per ottimizzare il tempo di risposta nella prima interazione è necessario comporre prompt efficienti ed il più corti possibili, trovando un buon compromesso per gli esempi (lunghezza e numero); l'utente non è consapevole di questa fase, e questo compone la prima parte del ritardo.

A questa si aggiunge una certa latenza dovuta alle operazioni del modulo di scanner, dipendenti dalle capacità di calcolo dell'HW: anche queste non sono visibili all'utente, ed il tempo si somma a quelle di riscrittura.

Il discorso cambia per quanto riguarda la generazione dell'ultima risposta, quella del modulo attualmente in esame in questo paragrafo. Per ridurre la sensazione di attesa di una risposta, infatti, i principali fornitori di modelli mettono a disposizione delle interfacce che permettono di aprire un canale di comunicazione, sul quale fornire, segmento per segmento, parola per parola, la risposta **durante** la sua generazione.

L'*answer builder* predispone di entrambe le modalità, in modo da permetterne l'utilizzo più adatto a seconda del contesto, **senza imporre** in questo modo **vincoli**. Durante lo sviluppo dell'assistente è stato scelto di utilizzare la modalità chiamata informalmente **streamed**, ossia "*a flusso*", che permette all'utente di ricevere un riscontro più veloce della risposta dell'assistente: questa operazione è infatti la più consistente, quella che richiede più tempo da parte del modello.

4.7 Funzionalità di traduzione

Nel caso si escluda l'utilizzo della pipeline in modalità *streamed*, ovvero si utilizzi la modalità di risposta in *blocco*, è possibile prevedere un servizio di traduzione, che va a soddisfare il *requisito non funzionale* **RQ8** (3.2 Traduzione).

Questo è già stato in parte accennato nel paragrafo 4.2, nel quale si è trattata la predisposizione di un campo dedicato per salvare la traduzione.

Il servizio di traduzione si basa sulla lingua della base di informazioni dell'assistente. Quando il chatbot riceve una richiesta ed il servizio di traduzione è attivo sarà necessario fornire anche la lingua della domanda: in questo modo sarà immediato determinare se vi è la necessità di una traduzione oppure no (la lingua "interna" dell'assistente è infatti, come detto, quella della sua base di conoscenza).

Viene predisposto, nel servizio, il riconoscimento delle parole chiave attraverso un glossario, in modo da evitare traduzioni incorrette.

Tutti questi servizi vengono concentrati in un modulo di traduzione al quale tutti gli altri possano fare affidamento nel caso la funzionalità venga richiesta.

La scelta di predisporre campi diversi, nel *conversation manager*, per le richieste (e risposte) tradotte e quelle originali è stata presa per garantire la possibilità, in fase di test, di verificare come la traduzione impatti sul funzionamento ottimale dell'assistente.

4.8 Configurazione

Durante la fase di progettazione è emersa la necessità di gestire un numero elevato di parametri di configurazione dell'assistente: dai parametri del modello a quelli delle basi di dati, dai nomi dei file a quelli dei prompt, e molto altro. Concentrare questo servizio è necessario per soddisfare il *requisito non funzionale RQ10* (3.2 Facilità di configurazione)

A seguito di queste considerazioni è stato scelto di dividere i parametri in *variabili d'ambiente* e *variabili di manutenzione*:

- **variabili di ambiente:** questi parametri sono quelli più legati al funzionamento del modello e della pipeline, come i percorsi dei file che possono cambiare base alla configurazione dell'ambiente.
- **variabili di manutenzione:** parametri interni, difficilmente cambiano durante la vita del programma, ma si è comunque scelto di raggrupparli in una sezione dedicata in modo da rendere più facile la manutenzione del codice per chi debba svolgerla, in futuro.

4.9 Funzioni d'utilità generale

Tutte le funzionalità che non possono essere coagulate in un singolo modulo vengono raggruppate tutte insieme:

- Caricamento dei prompt.
- Controlli sulle collezioni di dati vettoriali, comuni al *question builder*, *answer builder* e *documents manager*.
- Gestione dello *snippet*, utilizzato sia dal *question builder* che dall'*answer builder*.
- Gestione delle *labels*, ossia di tutte le etichette che possono venir utilizzate come filtri di ricerca.

4.10 Virtualizzazione

Tutte le funzionalità descritte fino a questo punto andranno a comporre il chatbot nella sua interezza, ma è necessario, in accordo con il *requisito non funzionale RQ10* (3.2 Virtualizzazione), predisporre fin da subito l'applicazione in un'ottica di virtualizzazione, i cui dettagli implementativi verranno discussi nel capitolo specifico.

La scelta di dare importanza alla virtualizzazione è stata fatta dall'azienda prima del lavoro di tesi, che si è limitato ad accettarne la rilevanza e ad adattare di conseguenza tutto il lavoro.

Un applicativo virtualizzato offre diversi vantaggi significativi, soprattutto in termini di **efficienza**, **scalabilità** e **gestione**. In primo luogo, la virtualizzazione garantisce un maggiore isolamento e sicurezza, poiché consente di eseguire un'applicazione in un ambiente isolato, riducendo il rischio che eventuali problemi di una singola applicazione possano influenzarne altre.

In aggiunta, la virtualizzazione favorisce flessibilità e scalabilità, permettendo di distribuire le risorse o creare istanze aggiuntive in modo rapido e semplice in risposta a un aumento della domanda, senza dover modificare l'hardware fisico sottostante. Questo approccio rende la gestione della capacità molto più dinamica e reattiva ai cambiamenti. Sul piano dell'efficienza delle risorse, la virtualizzazione consente un utilizzo più efficiente dell'hardware, poiché diversi applicativi possono condividere le stesse risorse fisiche, come CPU, memoria e storage, riducendo così i costi e migliorando l'efficienza complessiva.

4.11 Interfaccia grafica

La progettazione dell'interfaccia grafica che permetta l'interazione con l'assistente ha richiesto, in primis, lo studio delle principali soluzioni esistenti e delle rispettive funzionalità: si è cercato di dare, in ogni caso, importanza alla creatività e all'originalità, cercando di sfruttare le interfacce già presenti solo per definire una struttura corretta.

Un'altro aspetto sul quale è stata fatta un'attenta analisi è stato quello delle **euristiche** [24]: principi guida che contribuiscono a creare esperienze utente **efficaci** e **intuitive** quando si interagisce con gli assistenti virtuali.

Le euristiche utilizzante nel contesto dell'elaborato si ispirano a quelle definite da Jakob Nielsen [25] (uno dei più importanti ricercatori al mondo in ambito Human Computer Interaction) per la progettazione dell'interazione uomo-computer, adattati alle specificità dei chatbot.

Lo studio e la progettazione si sono concentrati su un ambiente *desktop*, secondo le necessità espresse dall'azienda. Di conseguenza, si è fatto riferimento alle

applicazioni web accessibili mediante browser, nel contesto degli assistenti virtuali, più famose.

Nello sviluppo dell'interfaccia si è garantito il rispetto del *requisito funzionale* **RQ4** (3.1 Gestione delle fonti), e di quelli *non funzionali* **RQ1** (3.2 Usabilità ed accessibilità) ed **RQ2** (3.2 UI/UX efficace).

4.11.1 Interfacce grafiche attualmente esistenti

Si sono analizzate le GUI (Graphical User Interface) dei due chatbot attualmente più popolari: **ChatGPT** di OpenAI e **Gemini** di Google.

Prima di proporre un'analisi dei punti in comune e delle differenze, si propongono le schermate iniziali di entrambe le soluzioni.

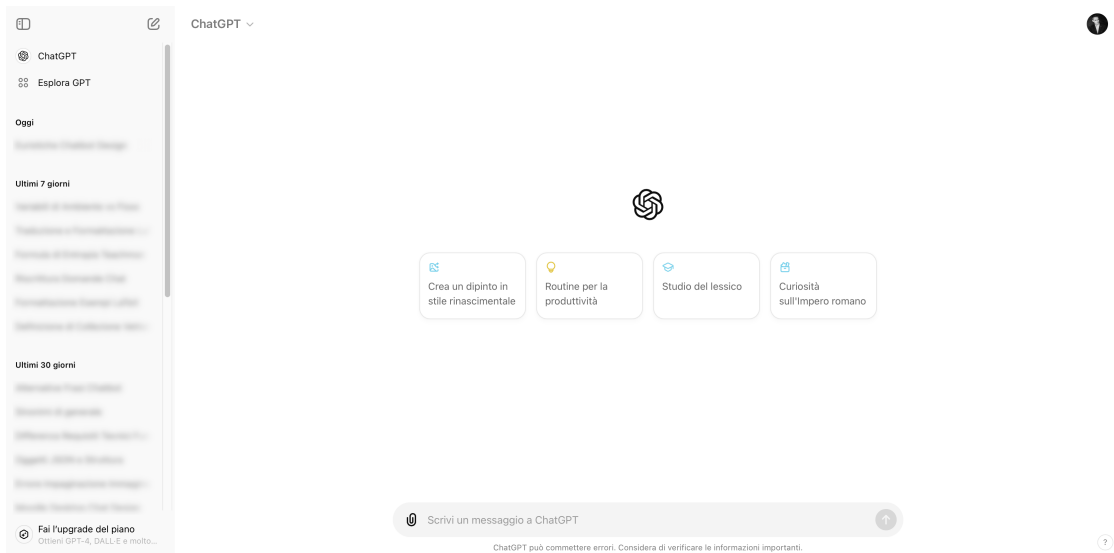


Figura 4.4: Schermata iniziale - ChatGPT

Sebbene gli stili siano ragionevolmente diversi, entrambe le GUI propongono funzionalità simili: l'interfaccia è principalmente costituita da una sezione centrale fissa dedicata alla conversazione, che si avvia utilizzando la barra di inserimento, “*convenzionale*” in tutti i sistemi di dialogo ai quali gli utenti sono abituati (come le numerose applicazioni di messaggistica). Notiamo poi la barra laterale, per convenzione posta a sinistra, il cui principale scopo è la gestione delle conversazioni: viene permesso l'edit del nome (che viene generato in modo indipendente in base all'argomento della prima domanda) e la cancellazione dell'intera conversazione. Selezionandone una qualsiasi si ripristina il dialogo nella parte centrale dell'interfaccia, che può essere così continuato. Sempre nella barra si trovano, in forma diversa,

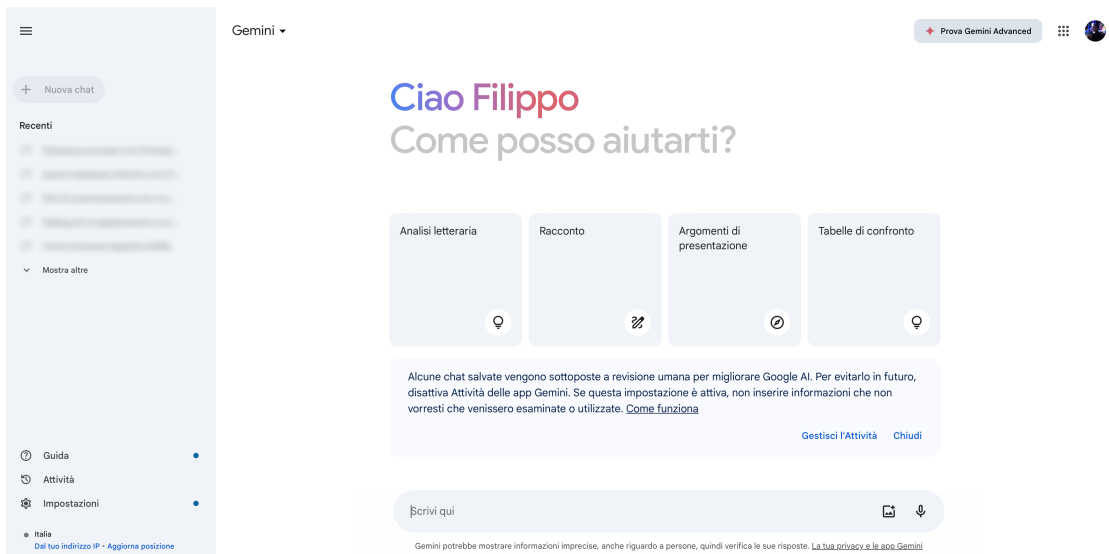


Figura 4.5: Schermata iniziale - Gemini

i pulsanti per avviare una conversazione nuova, fondamentale quando la sezione centrale ne contenga una già avviata.

Altro aspetto fondamentale sono i messaggi che si trovano sotto la barra che permette di scrivere i propri messaggi e, nel caso dell'interfaccia di Google, anche nella sezione centrale: questi contribuiscono alla consapevolezza dell'utente, chiarendo i limiti dei modelli che costituiscono il nucleo dell'applicazione.

Nel caso in esame in questa tesi, l'assistente virtuale che si va a progettare non necessita (allo stadio di sviluppo corrente al lavoro di tesi) delle molte funzionalità aggiuntive accessibili dalle due interfacce analizzate: dalla gestione dell'account ad una pagina di impostazioni, dalla selezione del modello LLM alla possibilità di allegare contenuti diversi dal testo nella barra di input, e per concludere di consigli sul cosa cercare.

4.11.2 Euristiche

Le euristiche applicate coinvolgono 11 punti e sono state ricavate dall'adattamento ad opera dell'esperto di **Human Centered Design**, **Gary Hsieh** [26] delle euristiche più generali di Nielsen:

H1) Visibilità dello stato del sistema: l'interfaccia deve segnalare il proprio status, dalla generazione dei messaggi ad uno stato di attesa, fino ai caricamenti. Per soddisfare questo punto si utilizzano i *loader* (elementi grafici che rappresentano uno stato di caricamento) ed altri elementi nell'interfaccia (ad

esempio un cerchio animato che segnali la generazione della risposta da parte dell'assistente), integrati in modo naturale.

- H2) Corrispondenza tra il sistema ed il mondo reale:** l'assistente deve comunicare mediante il linguaggio più adatto alle esigenze dell'utente, utilizzando la terminologia più appropriata al contesto ed alle casistiche specifiche. Nel caso in esame gli utenti in analisi si aspettano un linguaggio didattico, che sappia esporre i concetti nel modo più chiaro e semplice possibile.
- H3) Libertà e possibilità di controllo:** Il sistema non deve vincolare l'utente in uno stato particolare, va lasciata piena libertà di "movimento". Tutto quanto visibile ed accessibile non deve bloccare l'utilizzatore.
- H4) Consistenza e rispetto degli standard:** l'interfaccia deve essere consistente e rispettare gli standard grafici, ad esempio si deve mantenere la stessa tipologia di colori e forme nei pulsanti con le stesse funzionalità, oppure la stessa tipologia di carattere del testo per i titoli o i sottotitoli. Questo vale anche e soprattutto nelle risposte dell'assistente, che devono essere processate in modo da renderle il più possibile uniformi all'interfaccia (formattando elenchi puntati, tabelle, titoli, paragrafi, et cetera...).
- H5) Prevenzione degli errori:** In tutti gli stati nei quali l'utente possa inavvertitamente causare un errore o un comportamento indesiderato devono essere attuati meccanismi di prevenzione. Un esempio semplice è la richiesta di conferma in caso di cancellazione di una conversazione, oppure l'icona di invio del messaggio presente solo nel caso in cui nella barra di inserimento sia presente del testo.
- H6) Possibilità di aiuto e guida:** Le funzionalità devono essere chiare, mentre quelle non necessarie in una determinata situazione non devono essere presenti. Per aiutare gli utenti ed aumentare l'accessibilità si predispongono, ad esempio, il *tooltip* su ogni elemento che preveda una selezione: questo prevede la comparsa di una breve descrizione dell'azione che viene eseguita (o la proprietà) selezionando l'elemento sul quale ci si posiziona con un cursore.
- H7) Flessibilità ed efficienza d'utilizzo:** l'utilizzo del sistema deve poter essere agevole anche per gli utenti più esperti, ad esempio prevedendo combinazioni di pulsanti per ottimizzare alcune operazioni. Un esempio è la predisposizione di una combinazione per inviare un messaggio all'assistente senza dover utilizzare il mouse.
- H8) Design piacevole, accattivante e minimalista:** un'interfaccia grafica pulita ed esteticamente ben fatta è fondamentale per rendere più efficace e piacevole l'esperienza di utilizzo.

- H9) Riconoscimento degli errori, diagnostica e risoluzione:** I messaggi di errore che il sistema può produrre devono essere *gestiti*. Quando è possibile non vanno mostrati codici all'utente che utilizza l'interfaccia, piuttosto è importante fornire istruzioni su come procedere ed eventualmente chi contattare se la natura del problema lo richiede.
- H10) Preservare il contesto:** il contesto della conversazione va mantenuto durante la stessa, l'utente non deve far riferimento ogni volta alle sue richieste precedenti. Questo punto viene rispettato non tanto dall'interfaccia grafica in se, bensì dall'implementazione del chatbot inteso come sistema.
- H11) Fiducia:** il sistema deve predisporre tutti gli accorgimenti per poter essere considerato degno di fiducia, mediante la trasparenza sull'utilizzo della tecnologia sulla quale l'assistente si basa e soprattutto alle sue **limitazioni**. Fondamentale anche citare le **fonti** sulla base delle quali le risposte che vengono costruite, fornendo informazioni aggiuntive all'utente quando non riesce a rispondere alle sue domande.

4.11.3 Formato delle risposte

L'obiettivo secondario nella progettazione dell'interfaccia, dopo il rispetto delle euristiche elencate sopra ed il mantenimento della struttura standard di mercato è la realizzazione di un'interfaccia creativa e quanto più possibile **originale**: la sezione nella quale è possibile questo lavoro creativo è la parte di conversazione, come vengono formattati i messaggi, in quanto la barra laterale, oltre ad una differenza data dai colori e dalla scelta del carattere (che però sono generali dell'interfaccia nel suo insieme), non presenta troppe differenze tra le interfacce analizzate.

Per quanto riguarda la rappresentazione dell'interfaccia l'approccio che viene seguito normalmente nelle applicazioni di messaggistica è quello di dividere i vari messaggi in “**bolle**”, e disponendoli generalmente in modo speculare alla sezione, per evidenziare la differenza sia tra i messaggi in generale, sia tra i due lati della conversazione. Nel caso delle interfacce analizzate la rappresentazione dei messaggi è in parte semplificata dalla natura del rapporto 1 : 1 dei segmenti di conversazione: l'utente fa la propria richiesta, il chatbot procede con la propria risposta, non vengono generati più messaggi né da una parte, né dall'altra (come avviene più in generale nelle interazioni tra utenti umani).

Si è quindi analizzato come le GUI dei due diversi assistenti presi in considerazione, *chatGPT* e *Gemini*, rappresentino i messaggi.

Come si può notare nell'immagine 4.6, l'approccio di openAI è di tipo *classico*: il messaggio dell'utente è rappresentato in una bolla, disposto a destra come solitamente accade nelle applicazioni di messaggistica, mentre la risposta è disposta a sinistra, senza bolla per evidenziare la diversa natura dell'assistente e per rendere

più leggibili risposte molto lunghe e strutturate; è presente un'icona che aiuta a definire il ruolo di chi ha generato la risposta (gpt) e rappresentarne in modo più chiaro l'inizio. Vi sono poi una serie di funzionalità che chatGPT mette a disposizione per ogni risposta, che non sono richieste nel progetto esame di questa tesi.

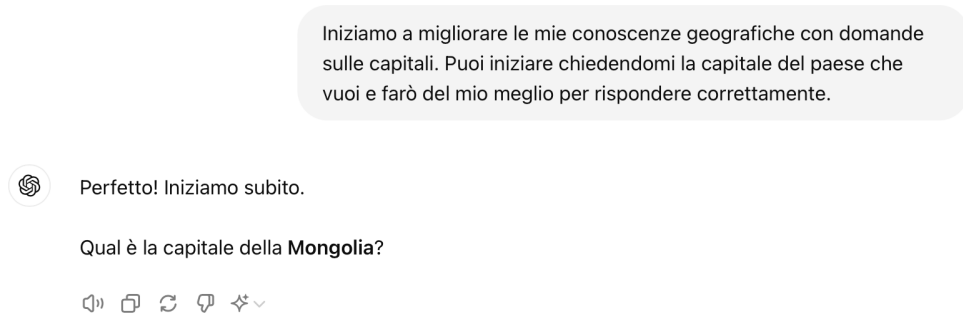


Figura 4.6: Rappresentazione dei messaggi - chatGPT

Questa rappresentazione **non è stata scelta** perché meno originale rispetto a quella proposta da Google in Gemini: l'interfaccia di conversazione del secondo assistente, infatti, non prevede bolle o disposizioni parallele, bensì una separazione a *strati* tra le richieste dell'utente e le risposte dell'assistente, come visibile in figura 4.7

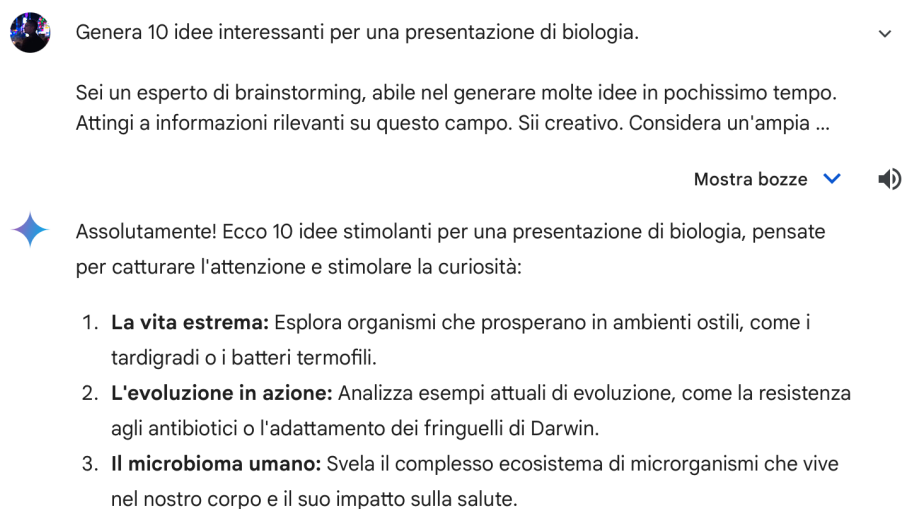


Figura 4.7: Rappresentazione dei messaggi - Gemini

Anche qui vi è l'utilizzo delle icone per una distinzione migliore, ma la disposizione è più originale e aumenta la separazione grafica tra ciò che si è abituati a considerare una chat tra esseri umani e quello che invece è una conversazione con un'assistente virtuale.

La sfida più grande di questa rappresentazione è render chiaro quale sia la domanda dell'utente e quale sia la risposta dell'assistente: è stato scelto di progettare questa sezione dell'interfaccia utilizzando lo stile generale di Gemini, introducendo alcune differenze:

- Icona solo al fianco delle richieste dell'utente; l'assistente non necessita di personificazione.
- Utilizzo di *caratteri* diversi per la rappresentazione della domanda dell'utente e per la risposta dell'assistente: questa differenza deve essere, nel rispetto dell'integrità dell'interfaccia, il più possibile marcata.
- Introduzione di un separatore grafico tra domanda e risposta ed incapsulamento delle due in un componente che preveda una certa distanza tra quello che è sopra e quello che è sotto, in modo da distinguere tra loro i diversi segmenti della conversazione.
- Rappresentazione delle fonti, che nel caso dell'assistente aziendale possono essere file *pdf* o *video*: fondamentale un formato chiaro e completo, ma allo stesso tempo coerente con lo spirito minimalista dell'interfaccia.

Per mantenere quanta più coerenza ed integrità possibile con l'interfaccia, si prevede la definizione di stili generali che adattino le risposte HTML prodotte dall'assistente allo stile grafico predefinito.

4.11.4 Funzionalità aggiuntive dell'interfaccia grafica

Infine, si sono volute predisporre alcune funzionalità aggiuntive utili per migliorare in generale l'esperienza dell'utente.

Modalità scura

Funzionalità prettamente grafica, permette di cambiare i colori dell'interfaccia passando da una *palette* (gamma di colori) chiara ad una scura, che utilizzi colori dedicati, selezionati per lo scopo. Questa funzione è stata introdotta per rispettare una tendenza che segue il desiderio di una categoria di utenti con una spiccata preferenza per contenuti rappresentati in colore più scuro. Questa modalità è meglio conosciuta come **Dark Mode**, sempre più prevista nelle interfacce moderne [27].

Supporto multilingue

Si predispose la possibilità di definire diverse versioni per i testi che compongono l'interfaccia. Questo permette di cambiare mediante un semplice pulsante la lingua generale dell'applicazione: la funzionalità si limita all'interfaccia, non alle risposte dell'assistente, la cui lingua deve essere impostata seguendo considerazioni diverse.

Modalità avanzata

Si è voluto prevedere anche la possibilità di una modalità avanzata che consenta agli utenti di inserire degli esempi, sia per quanto riguarda le domande, sia per le risposte. Questi esempi sono quelli trattati nei paragrafi dedicati al *question builder* (4.4) ed all'*answer builder* (4.6).

Si è pensato di prevedere questa funzione per immediatezza e semplicità: quando l'utente non è soddisfatto della risposta ricevuta o del modo con il quale il *question builder* ha riscritto la domanda, può immediatamente andare a fornire un esempio migliore, legato alle esigenze del momento.

Questa funzionalità introduce alcuni nuovi aspetti (visibili in caso la modalità avanzata sia attivata):

- **Prompt di modifica della domanda:** nel caso la domanda dell'utente sia stata riscritta viene visualizzata un'icona, premuta la quale viene mostrata la nuova domanda sulla cui base è stata costruita la risposta. Nell'interfaccia normale non viene mai visualizzata la domanda riscritta, l'utente deve visualizzare, com'è naturale che sia, la sua richiesta.
- **Prompt di generazione della domanda:** sempre nel caso in cui la domanda sia stata riscritta è possibile, sempre mediante la selezione di un'icona, visualizzare il prompt utilizzato per la riscrittura. Questo può essere utile specialmente per visualizzare quali esempi sono stati utilizzati per la generazione.
- **Prompt di generazione della risposta:** come per la domanda, anche la risposta è stata generata (in questo caso sempre) da un prompt. Mediante l'icona corrispondente sarà possibile visionarlo, con gli stessi vantaggi descritti al punto precedente.
- **Aggiunta di un esempio di domanda:** come anticipato sopra, azionando uno specifico pulsante si proporrà una finestra in cui sarà possibile aggiungere un nuovo esempio di riscrittura della domanda, che rispetti i requisiti definiti dagli esempi.
- **Aggiunta di un esempio di risposta:** come al punto precedente, ma per la risposta.

Quando appena descritto può essere, in un secondo momento, abilitato solo per una determinata categoria di *utenti avanzati* che abbiano la possibilità di modificare il comportamento dell'assistente.

Capitolo 5

Implementazione chatbot

La progettazione del chatbot, come delineato nei capitoli precedenti, ha posto le basi per la sua realizzazione. Questo capitolo si propone di approfondire gli aspetti più tecnici dell'implementazione, esplorando le scelte tecnologiche che hanno guidato lo sviluppo del progetto.

Il chatbot è stato implementato sfruttando le funzionalità del linguaggio **Python**, scelto per la sua flessibilità, la vasta gamma di librerie disponibili e la sua popolarità nell'ambito del machine learning e dell'elaborazione del linguaggio naturale.

L'intero sistema è stato progettato per operare in un ambiente virtualizzato, sfruttando **Docker** e Docker Compose per garantire portabilità, scalabilità e facilità di deployment. Questa scelta ha permesso di isolare le dipendenze del progetto e semplificare la sua distribuzione su diverse piattaforme.

Per quanto riguarda l'interazione con i Large Language Models e la gestione delle catene di elaborazione è stato utilizzato **LangChain**, un framework potente e versatile che ha facilitato la costruzione della pipeline di dati e l'integrazione con i modelli linguistici. La gestione della base di conoscenza vettoriale, cuore del sistema **RAG**, è stata affidata a **ChromaDB**, un database vettoriale open-source scelto per la sua efficienza e la sua integrazione nativa con LangChain.

Infine, per garantire la persistenza dei dati delle conversazioni e la gestione degli esempi per il prompt engineering, è stato utilizzato **SQLAlchemy**, un ORM (Object-Relational Mapping) che ha permesso di interagire con il database relazionale in modo astratto e flessibile. L'esposizione delle funzionalità del chatbot è stata realizzata mediante un'API *RESTful*, sviluppata con il micro framework **Flask**, garantendo all'interfaccia utente e ad eventuali altre applicazioni l'accesso a tutte le funzionalità messe a disposizione.

Nel corso di questo capitolo, si propone di esplorare nel dettaglio l'implementazione di ciascun modulo della pipeline, analizzando le scelte tecniche, le sfide affrontate e le soluzioni adottate per garantire il raggiungimento degli obiettivi prefissati.

5.1 LangChain e la generative AI

Il *framework* che ha permesso un'interazione intuitiva con i modelli LLM in questo progetto nasce nel 2022, opera di *Harrison Chase*. LangChain si propone di aiutare gli sviluppatori mediante un approccio modulare, fornendo un ampio set di componenti che permettono la connessione dei modelli LLM con servizi e fonti di dati esterni.

Dallo sviluppo iniziale il fondo al quale il framework fa riferimento è riuscito ad attrarre una quantità ingente di investimenti da gruppi come Sequoia Capital e Benchmark.

LangChain semplifica lo sviluppo delle applicazioni che fanno utilizzo di LLM fornendo componenti e soprattutto le cosiddette **catene**: sequenze di operazioni che hanno l'obiettivo di fornire l'elaborazione di un qualche contenuto, sfruttando uno o più modelli. Un esempio semplice è una catena che si occupi di recuperare dei dati, costruire un prompt allegato e far processare la richiesta al modello: tutti questi passaggi sono supportati su un'unica catena. Le *maglie*, ossia i blocchi (processuali) di cui è composta la catena possono essere combinati fino a crearne di complesse; il limite è quello tecnico o creativo degli sviluppatori. Gli aspetti chiave del framework possono essere condensati come segue:

- **Modularità e flessibilità**: come anticipato, il comportamento intrinseco delle catene garantisce un approccio modulare e adattabile agli scenari più disparati.
- **Integrazione con Modelli Linguistici**: permette di lavorare con la maggior parte dei modelli linguistici a disposizione sul mercato, garantendo un alto grado di indipendenza dell'applicazione sviluppata. Questa funzionalità permette infatti di cambiare senza difficoltà il modello con il quale si svolgono le operazioni (tutte o in parte) senza alterare il codice o la logica.
- **Supporto**: con una comunità attiva ed in crescita, l'utilizzo del framework è supportato da manuali, guide e forum che ne semplificano l'apprendimento e la risoluzione dei problemi.
- **Funzionalità**: la *gestione della memoria* (con la possibilità di affidare la gestione di una o più basi di dati direttamente al framework senza occuparsi del loro design), la possibilità di *accesso a dati esterni* il meccanismo delle *catene* ed il supporto per *agenti* (ovvero processi che eseguono azioni in un determinato ambiente a seconda delle situazioni) rendono LangChain uno strumento fondamentale nella realizzazione di applicazioni LLM based.

Come sottolineato nell'ultimo punto, il framework ha un gran numero di funzionalità: nel caso del chatbot in esame non sono naturalmente state utilizzate tutte,

bensì quelle necessarie per la costruzione dei prompt e l'interrogazione del modello LLM, così come l'utilizzo della base di dati vettoriale gestita da *ChromaDB*. Tutto questo, ancora una volta, è stato reso possibile grazie all'approccio modulare.

5.2 Docker e la virtualizzazione

L'applicazione fa ampio utilizzo della piattaforma di virtualizzazione **Docker**, mediante la quale viene installata, configurata ed eseguita.

La piattaforma consente di automatizzare il processo di creazione, distribuzione ed esecuzione delle applicazioni all'interno di container. I container possono essere considerati come dei piccoli "contenitori" leggeri ed isolati che racchiudono tutto ciò di cui l'applicazione (nel caso in esame il chatbot) ha necessità per funzionare: codice, librerie ed impostazioni specifiche dell'ambiente di esecuzione sono alcuni esempi.

A differenza delle altrettanto diffuse **macchine virtuali**, che simulano estensivamente gli aspetti di un intero sistema operativo, i container **condividono** il kernel del sistema operativo "reale", della macchina che gli sta eseguendo. Questa caratteristica gli rende più veloci rispetto alla virtualizzazione "*classica*", a parità di risorse a disposizione.

Ogni container è costituito da un'immagine Docker, che è un pacchetto leggero e portabile contenente le specifiche necessarie per far funzionare l'applicazione: comprende le **istruzioni** per l'installazione di tutte queste risorse, che quando messa in atto si risolverà, appunto, in un container (che è l'insieme delle risorse installate e dell'applicazione). Le immagini sono solitamente definite mediante un **dockerfile**, che contiene tutte le istruzioni su come costruirle, come installare dipendenze, configurare l'ambiente e così via. In alternativa, e quando un applicativo è costituito da più **elementi interdipendenti** che collaborano tra di loro (e quindi più container), è utile utilizzare per la configurazione un **DockerCompose**, che ne semplifica i requisiti: nel caso di questa tesi è stata utilizzata questa tipologia di file di configurazione, in quanto inizialmente il progetto comprendeva più servizi. La configurazione è stata mantenuta anche a lavoro finito (con un solo container) per permettere in futuro una facile configurazione di altri container, contenenti, ad esempio, l'interfaccia grafica che più si preferisce.

L'implementazione di logiche di virtualizzazione, soprattutto mediante docker, porta a svariati vantaggi, tra i quali il principale è sicuramente la portabilità: un'applicazione definita in un'immagine Docker può essere eseguita su qualsiasi macchina che abbia Docker installato, indipendentemente dalle differenze tra i sistemi operativi o le configurazioni hardware. Un altro vantaggio è l'efficienza nell'utilizzo delle risorse: poiché i container condividono il kernel del sistema operativo e non ne richiedono uno intero per ogni applicazione, l'overhead è inferiore

rispetto alle macchine virtuali. Ciò consente di eseguire molte più applicazioni sulla stessa macchina fisica, **ottimizzando** l'uso delle risorse.

In aggiunta Docker fornisce un livello di **isolamento** tra le applicazioni, il che significa che un container non può influenzare direttamente gli altri o l'host sul quale viene eseguito. Questo isolamento riduce il rischio di conflitti tra dipendenze e migliora la sicurezza, poiché le applicazioni possono essere eseguite in ambienti separati e controllati.

5.3 Gestione delle conversazioni: Conversation-Manager

Si propone di iniziare la trattazione dell'implementazione dell'applicativo seguendo un approccio **bottom up**, partendo dal modulo che gestisce le conversazioni, il `ConversationManager`.

La struttura del modulo `ConversationManager` adotta un approccio orientato agli oggetti, **gerarchico**, basato sull'ereditarietà, promuovendo così il **riutilizzo del codice** e la specializzazione nella gestione di diverse tipologie di messaggi. La classe `Message` costituisce la base per la rappresentazione di un messaggio generico, incorporando attributi e metodi comuni a tutti i tipi di messaggi. Le classi `Question` e `Answer`, derivate da `Message`, estendono questa struttura aggiungendo funzionalità specifiche per la gestione di domande e risposte, rispettivamente. A livello superiore, le classi `ConversationManager` e `Conversation` offrono un'interfaccia per la manipolazione e l'interrogazione delle conversazioni e dei relativi messaggi, garantendo una gestione strutturata e modulare delle interazioni utente all'interno del sistema di conversazioni.

5.3.1 Struttura ed organizzazione dei dati

L'argomento è stato trattato nella sezione 4.2, nel quale si è discusso anche della struttura dati sulla quale il modulo opera: per comodità si riporta di seguito.

La struttura della base di dati è stata ideata con l'obiettivo di massimizzare sia la semplicità e conseguente leggibilità, sia pensando alle logiche di utilizzo del modulo, che verranno analizzate nei successivi paragrafi.

La base di dati e le tabelle che la compongono è stata realizzata utilizzando il toolkit `SQLAlchemy`, un *Object-Relational Mapping (ORM)*. Gli **ORM** forniscono un *ponte* tra il mondo orientato agli oggetti (che in questo caso viene implementato utilizzando l'ambiente di *python*), ed il mondo relazionale delle basi di dati: questo permette di non dover strutturare il db secondo le normali tecniche relazionali, bensì utilizzando le logiche della programmazione ad oggetti, prime su tutte le classi. Questo consente l'astrazione del db, introducendo l'interazione mediante

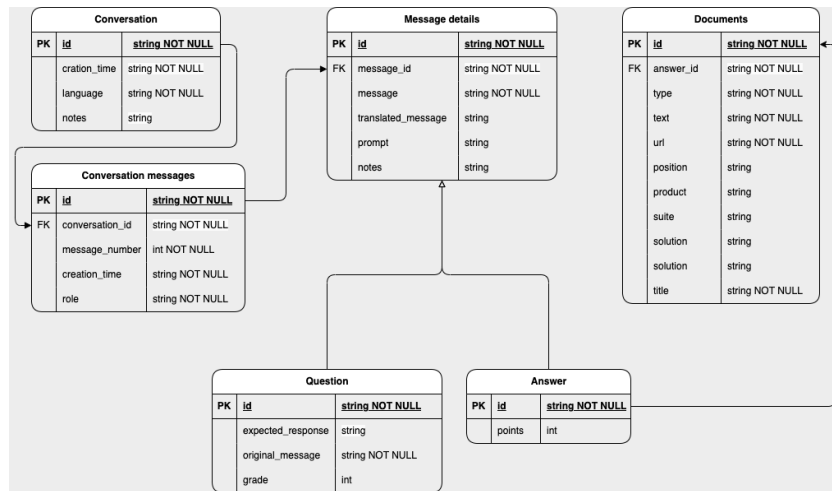


Figura 5.1: Schema base di dati - gestione delle conversazioni

oggetti e classi, anziché quella mediante query in SQL, migliorandone la leggibilità, la mantenibilità e la portabilità tra più sistemi di gestione di basi di dati (DBMS), slegando le logiche dichiarative da quelle implementative.

La base di dati viene creata da un istanza della classe SQLAlchemy, punto di accesso principale per tutte le operazioni: da quest'ultimo vengono create le tabelle, in forma di classi. Di seguito si riporta, ad esempio, la dichiarazione della tabella `conversations`, quella gerarchicamente più in alto.

```

1
2 class ConversationData(db.Model):
3     __tablename__ = 'conversations'
4     id             = Column(String,
5                          primary_key = True,
6                          default = str(uuid4()))
7     name          = Column(String, nullable = True)
8     creation_time = Column(String,
9                          default = datetime.now()
10                             .strftime(
11                                 '%Y-%m-%d %H:%M:%S'
12                             ),
13                          nullable = False)
14     language      = Column(String, default = "en",
15                          nullable = False)
16     notes         = Column(String, nullable = True)
    
```

Listing 5.1: dichiarazione della tabella "conversations"

Tutte le operazioni sulla tabella `conversations` potranno essere realizzate intervenendo sull'istanza di `ConversationData`

Come si può notare anche nell'esempio, gli identificatori vengono dichiarati come stringhe che rispettino la condizione di unicità: per garantire questa proprietà si introduce il concetto di **UUID** (*Universally Unique Identifiers*), ovvero stringhe alfanumeriche generate in modo da essere uniche in qualsiasi contesto, anche senza la necessità di un coordinamento centrale o di un database.

Nel codice presentato, la funzione `uuid4()` viene utilizzata per generare UUID versione 4, che si basano su numeri pseudo-casuali per garantire un'alta probabilità di unicità. L'utilizzo di UUID come chiavi primarie nelle tabelle del database assicura l'identificazione univoca di ogni record, anche in scenari distribuiti o in caso di fusione di dati provenienti da fonti diverse. In questo modo, si evita la necessità di ricorrere a contatori o altre strategie di generazione di chiavi che potrebbero portare a collisioni in ambienti complessi.

In tutta l'applicazione, ogni volta che si renda necessario generare un identificativo univoco, questo appena presentato sarà l'approccio utilizzato.

5.3.2 Organizzazione delle classi

Come anticipato sopra, il funzionamento del modulo è legato ad una logica gerarchica, che è possibile affrontare su due livelli.

Livello di conversazione

Il diagramma UML illustra la relazione e le funzionalità principali tra le due classi: `ConversationManager` e `Conversation`. Queste costituiscono i componenti chiave nella gestione delle interazioni utente all'interno della logica conversazionale.

La classe `ConversationManager` funge da punto di ingresso per la gestione delle conversazioni. Attraverso metodi come `get_or_create_conversation` e `get_conversation`, permette di creare nuove conversazioni o recuperare quelle esistenti, identificate da un ID *univoco*. Vengono offerte tutte le funzionalità per ottenere informazioni su tutti quelli che possono essere i dettagli legati ad una o più conversazioni, nonché fornire i metodi per l'eliminazioni.

La classe `Conversation` rappresenta una singola conversazione e fornisce i metodi per interagire con essa. Per aggiungere nuovi messaggi vi sono i metodi dedicati `add_question` e `add_answer`, differenziati in base alla tipologia di messaggio, che vanno a restituire la rispettiva istanza della classe specifica; sono a disposizione i metodi per ricavare le istanze del singolo messaggio avendo a disposizione l'identificativo (si è deciso di separare sempre i riferimenti a domande e risposte, anche quando le funzionalità, come in questo caso, sono medesime: questo per aumentare la leggibilità del codice).

La classe offre anche metodi per gestire i metadati della conversazione, come il nome (`update_name`) e le note (`set_notes`, `append_notes`), e per ottenere

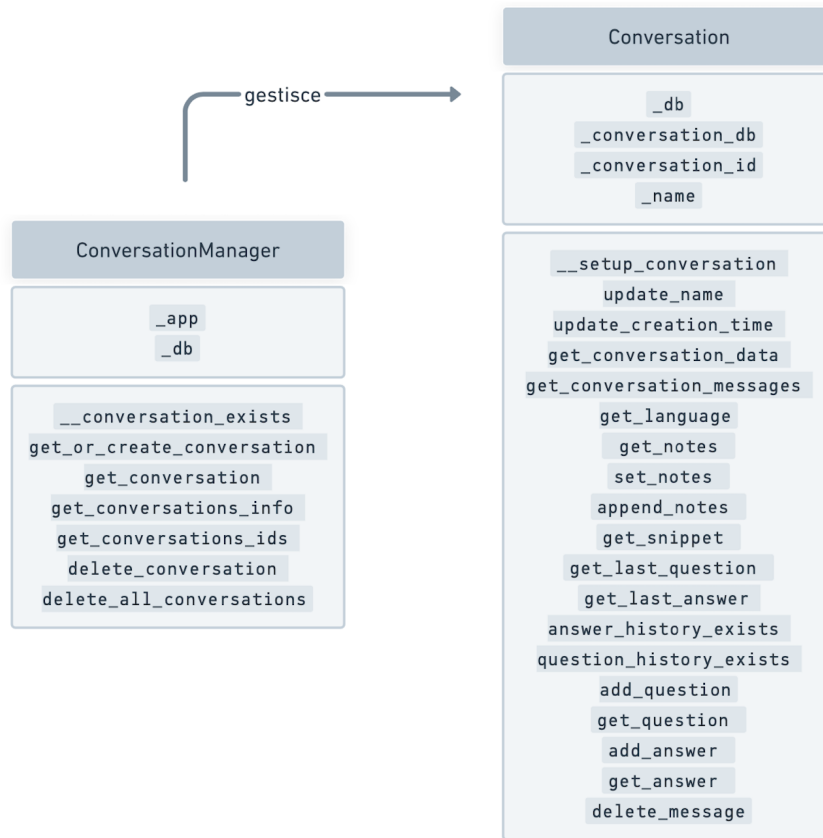


Figura 5.2: UML - classi di gestione conversazioni

informazioni sulla conversazione stessa, come i suoi dati (`get_conversation_data`), i messaggi (`get_conversation_messages`) e la lingua (`get_language`).

Con `get_snippet` è possibile ricavare l'oggetto `Snippet` che verrà trattato in seguito, utile per presentare un contesto all'utente.

Infine, `delete_message` permette di eliminare un messaggio specifico dalla conversazione.

Le due classi sono legate: `ConversationManager` gestisce le conversazioni a livello globale, mentre `Conversation` ne rappresenta una singola ed offre i metodi per interagirvi in modo specifico. Questa organizzazione consente una chiara separazione delle responsabilità e una gestione efficiente delle conversazioni all'interno del sistema.

Livello di messaggio

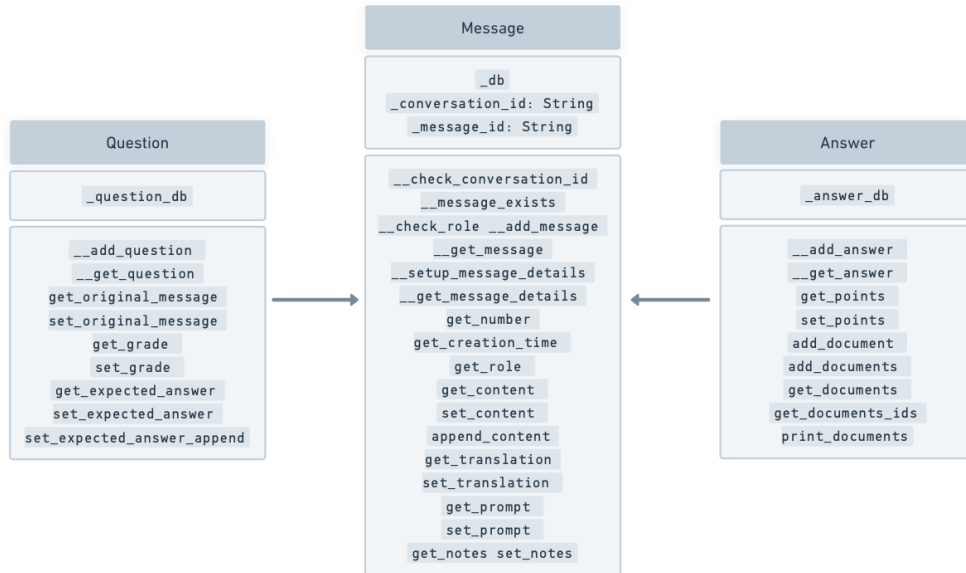


Figura 5.3: UML - classi di gestione messaggio

Nel secondo diagramma sono rappresentate la gerarchia e le funzionalità delle classi `Message`, `Question` ed `Answer`. Queste classi sono fondamentali per la rappresentazione e la gestione dei messaggi all'interno della conversazione.

La classe `Message` è la base per tutti i tipi di messaggi, definendo attributi e metodi comuni. Include proprietà per l'identificativo univoco del messaggio (`message_id`) e della conversazione a cui appartiene (`conversation_id`), oltre al ruolo del mittente (sono disponibili due valori, uno per l'assistente e uno per l'utente, che vengono determinati nel file di configurazione: vengono implementati dei controlli per garantire che non vi siano inseriti altri valori) e al contenuto del messaggio stesso. Metodi come `get_content`, `set_content` e `append_content` consentono di accedere e modificare il contenuto del messaggio, mentre `get_translation` e `set_translation` gestiscono eventuali traduzioni. `get_prompt` e `set_prompt` sono utili per memorizzare il prompt che ha generato il messaggio, se applicabile. Infine, `get_notes` e `set_notes` permettono di associare note aggiuntive al messaggio: questo campo rientra nella logica di test analizzata nel capitolo precedente.

Le classi `Question` e `Answer` ereditano da `Message`, specializzandosi per rappresentare domande e risposte. `Question` aggiunge la possibilità di memorizzare il messaggio originale dell'utente (`get_original_message`, `set_original_message`), un voto (`get_grade`, `set_grade`) e la risposta attesa (`get_expected_answer`, `set_expected_answer`, `set_expected_answer_append`).

`Answer`, invece, include la gestione dei punti associati alla risposta (`get_points`, `set_points`) e l'associazione dei documenti utilizzati per la sua costruzione (`add_document`, `add_documents`, `get_documents`, `get_documents_ids` e `print_documents`).

La relazione di ereditarietà tra `Message`, `Question` e `Answer` indica che le classi figlie ereditano tutti gli attributi e i metodi della classe padre, aggiungendo funzionalità specifiche per il loro ruolo nella conversazione. Questa struttura favorisce il riutilizzo del codice e una chiara organizzazione delle responsabilità, consentendo di gestire in modo efficiente e flessibile diversi tipi di messaggi all'interno del sistema conversazionale. Inoltre, come visto nella sezione precedente, il modulo è predisposto per fornire solo ed esclusivamente istanze di `Question` ed `Answer`. La classe `Message`, da sola, non ha un'applicazione diretta, dal punto di vista tecnico è una classe **astratta**:

si definiscono classi astratte quelle classi che definiscono una struttura comune ed un comportamento base per tutte quelle che le estendono.

Anche se in python non sono propriamente supportate, si è deciso di procedere in questo modo per evitare duplicazioni del codice ed organizzare la logica nel modo più modulare possibile.

5.3.3 Architettura del modulo

Il modulo mette a disposizione, come si è descritto nei paragrafi precedenti, oggetti dedicati in base alle operazioni necessarie, in modo che sia sempre chiaro su cosa si stia operando durante l'utilizzo delle funzionalità.

Tutto parte dal `ConversationManager`, classe dalla quale vengono generate tutte le altre: da qui si ha la possibilità di generare un nuovo oggetto `Conversation`, oppure di ottenerne uno con già predisposte tutte le configurazioni di una conversazione esistente, fornendone l'identificativo.

L'oggetto conversazione, a sua volta, oltre a poter esser utilizzato per modificare gli attributi generali della stessa, permette di aggiungere nuove domande e risposte, fornendo i rispettivi oggetti.

Sempre seguendo la stessa logica, in base alla tipologia del messaggio, tutti gli aggiornamenti come il contenuto, l'eventuale traduzione, le proprietà come i prompt o le valutazioni vanno operati sull'oggetto specifico, che sia esso `Question` o `Answer`.

Di seguito si propone uno schema molto semplice per illustrarne la logica.

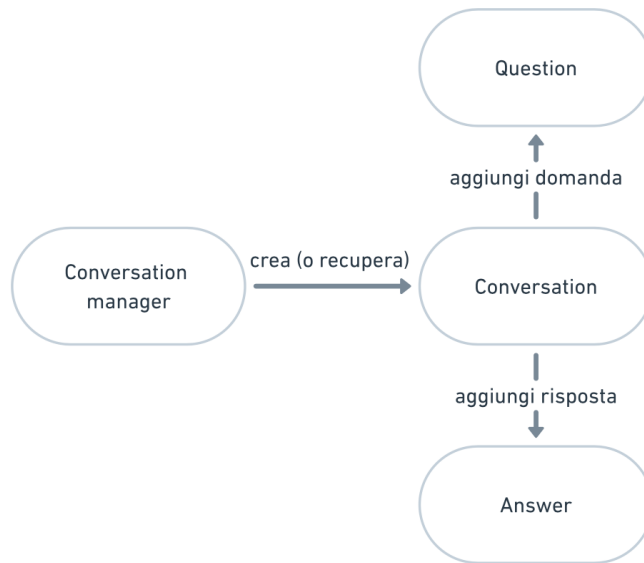


Figura 5.4: Architettura semplificata del ConversationManager

5.4 Gestione della base di conoscenza: DocumentsManager

La classe `DocumentsManager` mette a disposizione quanto necessario per gestire tutte le informazioni sulla base delle quali l'assistente basa la propria conoscenza specializzata.

Si propone l'analisi dei metodi che costituiscono la classe, con particolare attenzione al primo, quello che si occupa del caricamento della collezione vettoriale, in quanto sarà lo stesso metodo, da un punto di vista logico, con le stesse funzionalità, che utilizzeranno anche il *question builder* e l'*answer builder*.

5.4.1 Caricamento della collezione vettoriale

Il metodo `_load_vecstore` gestisce il caricamento della base di conoscenza vettoriale (vector store), specificatamente per una collezione, che è fondamentale per il funzionamento del sistema RAG. Questo vector store permette di memorizzare e recuperare informazioni in modo efficiente, basandosi sulla similarità semantica tra le query e i documenti archiviati.

La base di dati vettoriale componente questo progetto è divisa in **collezioni**, ognuna specifica per il modulo che ne faccia uso: il caricamento di cui si discute in questo paragrafo ne riguarda una specifica, quella appunto della documentazione

aziendale, ma ne sono disponibili altre due per gli esempi (di domande e risposte). Per discriminare tra una collezione e l'altra vengono utilizzati dei nomi, definiti nel file di configurazione, che permettono in questo modo di identificare i gruppi di informazioni e non dover, ad ogni avvio dell'applicazione, ricompiere tutte le operazioni di codifica.

Il metodo viene eseguito automaticamente ogni volta che un'istanza viene generata. Inizia l'esecuzione verificando se la variabile di configurazione `relatiav` è impostata sul valore Boolean `True` oppure `False`: questa è una variabile di ambiente, configurabile esternamente da chi installa l'applicazione, e consente di specificare all'applicativo se sono state aggiunte o meno nuove informazioni all'interno delle cartelle predisposte. Queste informazioni devono essere in **formato JSON** e rispettare i criteri indicati nella documentazione dell'applicazione.

Per chiarire meglio cosa si sta trattando, si propone di seguito un esempio di quello che viene definito in questa tesi "documento JSON di partenza":

```
1
2 {
3   "type": "pdf",
4   "text": "TESI - CONTROL TOWER is a software suite...",
5   "id": {
6     "document_id": "...",
7     "heading": "TESISQUARE Platform ..."
8   },
9   "content": {
10    "product": "Platform",
11    "suite": "Control Tower",
12    "solution": "SRM Procurement",
13    "title": "Control Tower Srm Procurement Handbook"
14  },
15  "roles": {
16    "implementator": true,
17    "presales": true,
18    "referral": true
19  }
20 }
```

Listing 5.2: Rappresentazione JSON di una piccola parte di un documento aziendale in formato pdf

Come si nota dall'esempio, la documentazione in forma di **oggetto JSON** presenta tutta una serie di parametri, in primis il testo che contiene parte del documento, ma anche tutta una serie di informazioni in più che vanno dalla tipologia di documento al suo `url` di origine, dal titolo del manuale alla specifica categoria di soluzione di prodotto che sta trattando.

Tutta la documentazione va caricata seguendo questa struttura, che è stata definita dall'azienda in precedenza al lavoro di tesi, che si è limitato ad adattarne

la struttura ed implementare i controlli.

Nel caso sia stata impostata la configurazione che segnali la presenza di nuovi documenti il modulo provvede alla loro ricerca ed alla codifica in forma di **Document**, rappresentazione già discussa, permessa dal framework *LangChain*, che permette di strutturare logicamente il contenuto come una parte *principale*, costituita dal **testo**, ed una parte di **metadati**, che comprendono tutte le **informazioni accessorie** (che si possono, ad esempio, vedere nel caso illustrato poco sopra).

Una volta predisposti i **Document**, si cerca se la collezione esista oppure no: in caso non esista ne viene generata una nuova, nella quale gli oggetti vengono convertiti in **embeddings**, ovvero nella loro rappresentazione numerica con una funzione di embedding messa a disposizione sempre dal framework, e la funzione termina restituendo un riferimento alla collezione vettoriale. In caso invece la collezione (alla quale, si ricorda, viene fatto riferimento con un nome pre-configurato) esista già, è necessario fare un controllo su ogni elemento **Document**, per accertarsi che non esista già all'interno della collezione. Passati i controlli si procede nello stesso modo elencato per l'altro caso.

Nella situazione, invece, nella quale l'ambiente non sia stato configurato per il caricamento di nuove informazioni, si procede semplicemente con il caricamento della collezione vettoriale, gestendo eventuali errori.

Si è deciso di adottare un approccio che prevedesse una variabile di configurazione per segnalare la necessità di aggiornare (o inizializzare se la collezione non esiste) la base di dati vettoriale a causa del tempo di calcolo necessario sia nella fase di controllo presenza di documenti già esistenti, sia in quella di codifica di questi ultimi in formato vettoriale.

Quando l'applicativo viene eseguito con risorse limitate, ed a fronte di una mole significativa di informazioni, questo processo di inizializzazione può richiedere svariati minuti, a seconda delle capacità del calcolatore: la logica predisposta mette a disposizione dell'amministratore del sistema una **scelta**, sulla base delle sue esigenze. Nel caso vi sia una buona disponibilità di risorse, ed una frequenza di aggiornamento delle informazioni significativa, sarà sufficiente avviare l'ambiente con la variabile di configurazione impostata sul valore **True**, in caso contrario, in uno scenario che propenda più al risparmio di risorse, sarà necessaria più attenzione ed il corretto passaggio dallo stato di **True** a quello di **False**, come verrà descritto nella sezione dedicata alla configurazione ed inizializzazione dell'applicativo (5.13).

Di seguito si propone un diagramma di flusso che schematizza quanto descritto. Si è deciso di proporre una rappresentazione quanto più accurata possibile (mantenendo alto anche il grado di semplicità) delle operazioni di inizializzazione in quanto sono le stesse che vengono effettuate dal *question builder* ed *answer builder*, nei quali cambia soltanto il contenuto degli elementi **Document**, ma non la logica.

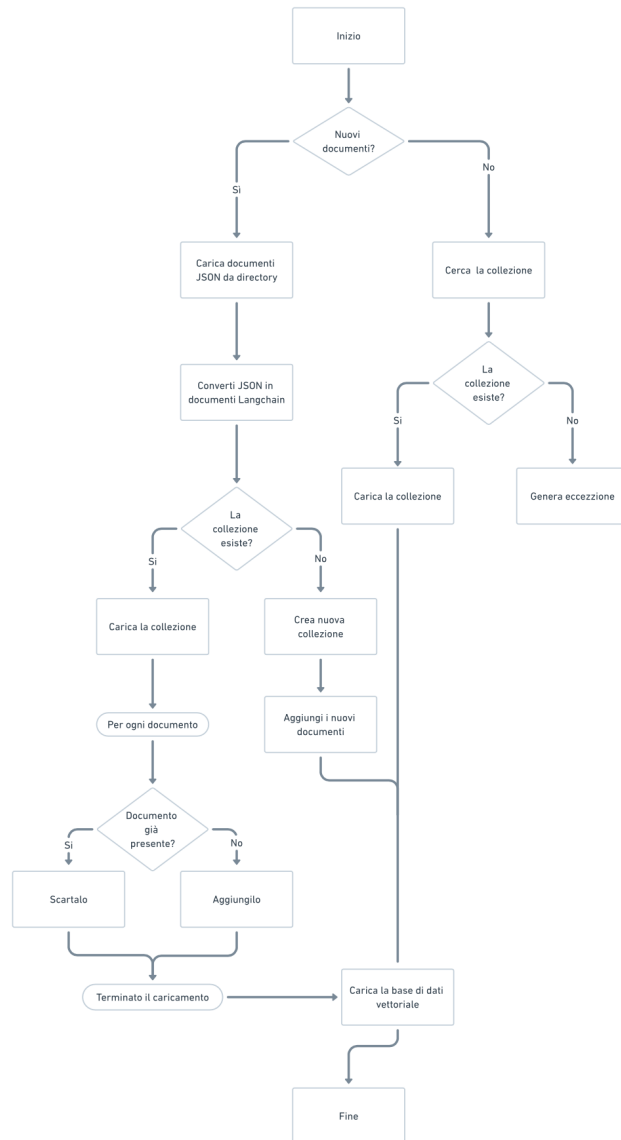


Figura 5.5: Diagramma di flusso - caricamento della base di dati vettoriale

5.4.2 Ricerca documenti rilevanti

Il modulo mette a disposizione **due** metodi che permettono la ricerca dei *documenti* rilevanti: `retrieve` e `retrieve_with_score`.

Entrambi i metodi hanno lo scopo di recuperare i `Document` più rilevanti in relazione ad una *query* data, sfruttando delle **labels**, ovvero delle parole chiave, per

ottimizzare la ricerca: l'utilizzo di queste *etichette* ottimizza il processo di ricerca affinandola non solo al contenuto della documentazione aziendale, bensì anche alla tipologia: se in una richiesta viene specificato un qualche prodotto e soluzione presenti nei metadati e classificati come *etichette*, la ricerca si concentrerà **anche** su quelle informazioni, aiutando a filtrarle.

Il risultato di ogni ricerca può essere integrato dal rispettivo punteggio di distanza: ogni valore estratto dalla base di dati vettoriale può quindi essere un `Document` singolo oppure una coppia (`Document`, `score`), dove `score` rappresenta la distanza tra il contenuto del `Document` trovato e la *query* di ricerca. La possibilità di scegliere se ricevere solo il risultato singolo o la coppia è garantita dalla presenza dei due metodi: `retrieve` e `retrieve_with_score`. Entrambi restituiscono un certo numero di risultati (che possono essere una lista di `Document` singoli o di coppie come descritto poco sopra) configurabile come il numero di `Document` che vengono estratti (indipendentemente dalla distanza), modificando la rispettiva variabile d'ambiente (`N_DOCUMENTS_RETRIEVED`).

5.4.3 Modalità alternativa di aggiornamento

Si è affrontato nel paragrafo 5.4.1 l'aggiornamento (o l'inizializzazione) delle informazioni in forma di file `JSON` contenenti, segmentate, le informazioni componenti i documenti aziendali.

Per praticità si è deciso di fornire anche un metodo che consentisse di aggiornare la collezione vettoriale utilizzando l'oggetto `JSON` fornito direttamente come parametro e non in forma di *file* (che normalmente ne contengono una lista, la cui somma di elementi rappresenta la codifica di, ad esempio, un intero `pdf`).

La logica di funzionamento è la stessa descritta in precedenza: vengono fatti i controlli di struttura, convertito il `JSON` in `Document`, controllata l'eventuale presenza nella collezione ed eventualmente aggiunto.

5.4.4 Funzionamento del modulo

Per concludere la descrizione del gestore dei documenti, si propone di seguito un **diagramma di sequenza** del flusso completo di utilizzo del modulo.

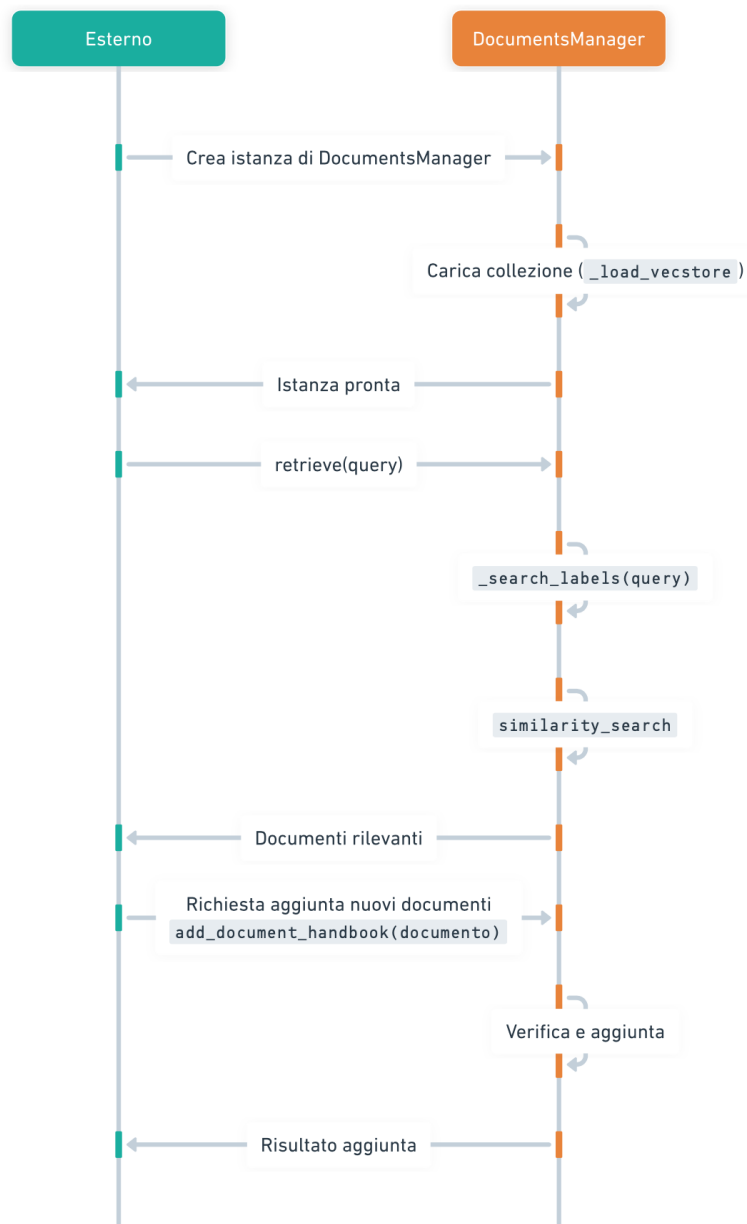


Figura 5.6: DocumentsManager - Diagramma di flusso

5.5 Prompt engineering: QuestionBuilder

Il `QuestionBuilder` è il modulo incaricato di realizzare il **prompt engineering** per ottimizzare le richieste dell'utente, facendole riscrivere in modo istruito mediante un l'uso dell'LLM selezionato.

Viene implementato mediante una classe che mette a disposizione i metodi per la gestione della collezione vettoriale di esempi, per la costruzione del prompt e rispettiva interazione con il modello linguistico e l'aggiunta di ulteriori esempi.

Il **prompt** costruito dalla classe viene messo a disposizione esternamente come un **attributo** della stessa, in modo da essere facilmente accessibile: questa proprietà conterrà sempre il prompt aggiornato relativo all'ultima richiesta di costruzione della domanda.

5.5.1 Caricamento e ricerca degli esempi

La logica è la medesima analizzata nel metodo `_load_vecstore()` visto per il `DocumentsManager`, schematizzato nella figura 5.6: si utilizza una variabile d'ambiente per segnalare la presenza di nuovi esempi (o per la creazione della collezione al primo avvio dell'applicativo), differente da quella utilizzata per segnalare la presenza di nuova documentazione aziendale.

Il metodo si occupa quindi di caricare o creare la collezione in base alla casistica, eventualmente aggiornandola mediante nuovi esempi: questi sono strutturati sempre utilizzando l'elemento `Document` che è già stato trattato. Per capire il perché della scelta di utilizzare sempre questo elemento è utile riportare la rappresentazione JSON degli esempi:

```

1 {
2   "user": "Posso registrare una spedizione relativa a...",
3   "assistant": "Si certo, puoi registrare la spedizione...",
4   "user_last_question": "come?",
5   "rewritten_last_question": "Parlami ora del COME possa
6     registrare una spedizione relativa a [...],
7     rispondimi in modo esteso fornendo l'elenco
8     dei passaggi per eseguire l'operazione"
9 }

```

Listing 5.3: Rappresentazione JSON dell'esempio di riscrittura di una domanda

Da questo esempio, oltre a capire la logica di riscrittura, emerge anche la chiave di ricerca, ovvero lo `Snippet`. In precedenza era stato anticipato come, visto il suo utilizzo esteso, lo `Snippet` fosse stato implementato come un vero e proprio elemento, che permettesse un facile accesso all'ultima parte del dialogo.

Per capire meglio come lo snippet costituisca la chiave di ricerca degli esempi e come questi siano organizzati nella base vettoriale da un punto di vista logico si faccia riferimento al seguente schema.

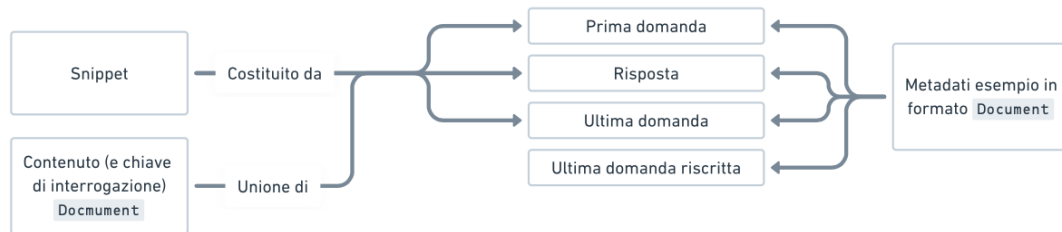


Figura 5.7: Struttura degli esempi, rappresentazione mediante Document e confronto con Snippet

Nel Document adibito ad esempio il contenuto è rappresentato dall’unione (per mezzo di caratteri di escape `\n`) degli elementi componenti lo Snippet: questo è stato fatto per aumentare la precisione della ricerca degli esempi. La parte necessaria infatti al modello linguistico per riscrivere l’ultima richiesta dell’utente è rappresentata dall’informazione “ultima domanda riscritta”, o `rewritten_last_question` (come da nome della variabile): per trovare gli esempi che contengano la riscrittura ottimale per la richiesta in esame si è deciso di utilizzare l’ultimo pezzo di conversazione per aumentare il contesto dell’esempio, anziché esclusivamente la richiesta dell’utente nel suo singolo.

L’utilità (quasi obbligatorietà) di questa strategia è evidente dall’esempio illustrato sopra, per il quale se si cercassero gli esempi che più si avvicinano solo utilizzando l’ultima domande dell’utente si utilizzerebbe “*come?*” in chiave di ricerca, ottenendo molto probabilmente consigli di riscrittura sbagliati.

L’elemento Document incaricato di **incapsulare** l’esempio è quindi costituito da un contenuto rappresentato dall’unione degli elementi dello Snippet, e dai metadati nei quali, per facilità di accesso si riprendono i singoli componenti appena condensati nel contenuto, aggiungendovi la domanda come deve essere riscritta.

All’avvio del `QuestionBuilder` il metodo `_load_vecstore()` si occuperà quindi, come per il `DocumentsManager`, di mettere a disposizione (come parametro interno della classe) un riferimento alla collezione vettoriale relativa agli esempi di riscrittura della domanda.

5.5.2 Costruzione del prompt e richiesta al modello

Il tentativo di **miglioramento** della domanda dell’utente avviene mediante il metodo di `build(Snippet)`, che richiede **obbligatoriamente** il parametro `Snippet`,

senza il quale è impossibile risalire al contesto della domanda e quindi alla ricerca degli esempi più appropriati di riscrittura, come spiegato nel paragrafo precedente.

Si è usato il sostantivo **tentativo** in quanto non è sempre detto che il modulo riscriva la domanda dell'utente, anzi. Fondamentale, vista la possibile integrazione di questo modulo all'interno di una pipeline, il filtraggio delle richieste dell'utente: il modulo non può essere infatti "sfilato" dalla pipeline a piacere, ma deve capire in autonomia quando sia effettivamente necessario riscrivere l'input dell'utente e quando invece no.

Se fino ad un determinato momento, in una conversazione, ci si è concentrati su un soggetto, e in un secondo momento si vuole passare al successivo, il modulo **non** deve riscrivere le richieste dell'utente per riportarlo al soggetto precedente: questo è un comportamento causato dall'LLM che deve essere gestito. La strategia più semplice che si propone di evitare riscritture inopportune è la stessa che si occupa delle scritture: **gli esempi**. Ne viene fornita una serie nei quali l'ultimo elemento, la frase riscritta, è costituita da una barra orizzontale: "-". Ogni volta che il modello riceve questi esempi (o un mix tra questi ed altri) viene guidato nella procedura di valutazione, se riscrivere o no la richiesta.

Tornando al metodo di build, se ne propone di seguito l'analisi, fornendo anche un diagramma di flusso per aiutare il lettore visivamente.

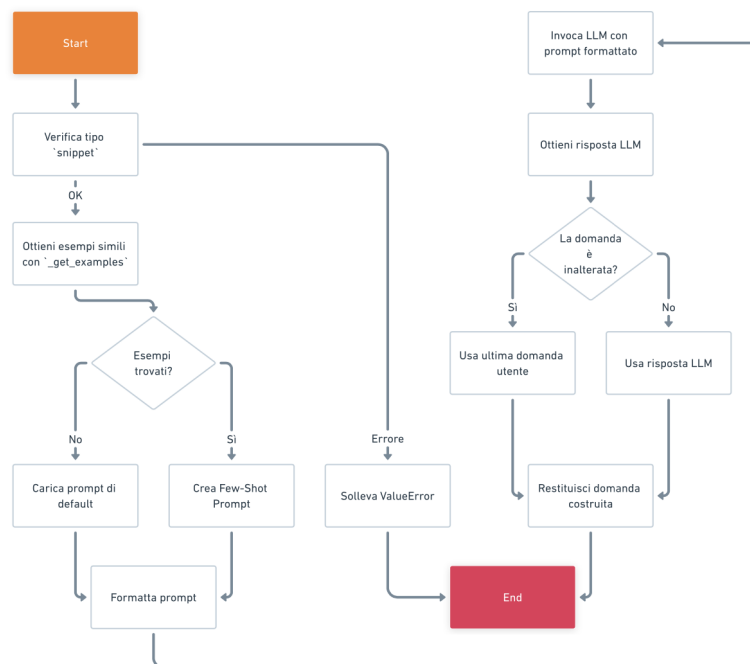


Figura 5.8: Diagramma di flusso del metodo di riscrittura della domanda

Dopo le verifiche allo `Snippet` si procede alla ricerca di esempi, per mezzo del metodo dedicato `get_examples()`, che possono essere sia di riscrittura che di *non* riscrittura: nel caso non venga trovato nulla (viene posto un limite di distanza oltre al quale gli esempi trovati vengono automaticamente scartati in quanto troppo diversi dalla chiave di ricerca) si procede all'utilizzo di un *prompt di default* che non necessita di esempi, in caso contrario si utilizza un *few-shot prompt*.

Struttura del prompt

I prompt vengono costruiti mediante i **template** messi a disposizione da **LangChain**, trattati nello specifico paragrafo nella parte di progettazione (4.4.3). Un template è un modello predefinito che specifica come organizzare l'input per il modello di linguaggio (LLM). Nel caso del few-shot, il template è composto da una struttura che include variabili (`user`, `assistant`, `user_last_question`, `rewritten_last_question`) e sezioni testuali fisse (prefisso e suffisso) che forniscono un contesto coerente per il modello. Il template è un prompt senza dati **dinamici**, bensì con dei **segnaposto** dove in un secondo momento vanno inseriti i dati che variano ad ogni utilizzo: questi ultimi vengono inseriti nella chiamata all'LLM.

Valore di ritorno

Una volta invocato il modello linguistico si analizza il risultato: se questo è variato rispetto a quello di partenza si restituisce quello modificato, in caso contrario si restituisce lo stesso valore in entrata (ovvero l'ultimo elemento dello `Snippet`).

5.5.3 Aggiunta degli esempi

Il modulo mette a disposizione il metodo `add_sample()` per aggiungere esempi in formato JSON alternativamente all'utilizzo di un file: ogni volta che un esempio viene aggiunto utilizzando questa modalità l'esempio viene controllato, nel caso non ancora presente nella collezione aggiunto e, se viene utilizzato un parametro specifico nella chiamata al metodo (`store_json`), l'esempio appena aggiunto alla collezione viene anche inserito all'interno del file presente al percorso standard (configurabile).

5.5.4 Analisi del flusso dati

Infine, per fornire una panoramica generale del funzionamento del modulo, si propone il seguente diagramma di flusso (figura 5.9) che si propone di illustrarne il comportamento completo.

Si parte dall'istanziamento del modulo e conseguente caricamento della collezione e gestione degli esempi. Segue la richiesta di `build` di una richiesta utente incapsulata all'interno di uno `Snippet`: il modulo recupera gli esempi mediante il metodo `_get_examples`, inserisce le informazioni trovate all'interno del template più adatto (quello di `default` nel caso non vi siano esempi, quello `few-shot` in caso di ricerca avvenuta con successo) ed esegue la chiamata all'LLM, che restituisce la domanda modificata (o meno, a seconda del caso).

Viene poi anche considerata la casistica di aggiunta di un esempio, che viene prontamente controllato dal modulo e, se il controllo di presenza da esito negativo, aggiunto alla collezione ed, opzionalmente, al file JSON.

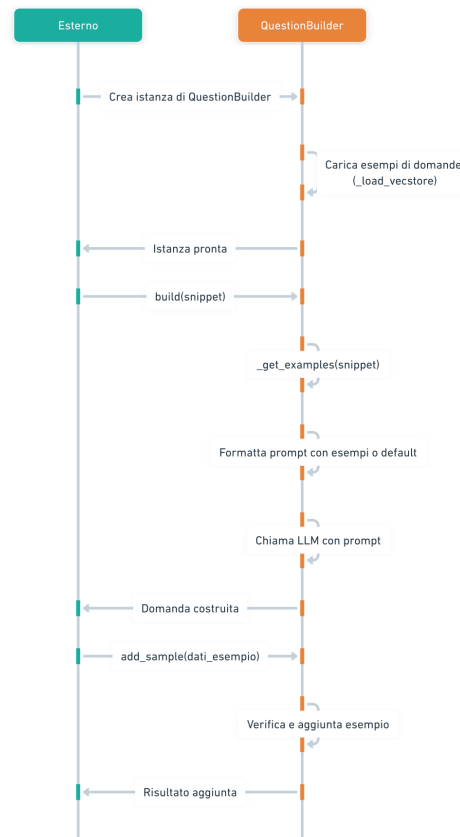


Figura 5.9: QuestionBuilder - Diagramma di flusso

5.6 Analisi delle richieste: QuestionScanner

Anche il modulo che si occupa di analizzare la bontà delle domande è stato organizzato come una classe che metta a disposizione un semplice metodo di scansione, `scan(...)`, e gestisca tutte le operazioni internamente.

Il modulo, durante le operazioni di analisi, fa affidamento alle funzionalità del `DocumentManager`, al quale richiede le sezioni di documentazione aziendale più vicine alla richiesta: i documenti ricavati, insieme ai parametri che compongono l'analisi ed il voto della domanda vengono salvati nello spazio di memoria dell'istanza

e messi a disposizione. Questa tipologia di informazioni sarà sempre relativa all'ultima scansione effettuata: sarà il soggetto che ha istanziato lo scanner ad occuparsi, eventualmente (come accadrà) della loro persistenza.

5.6.1 Ricerca delle informazioni

Quando il modulo riceve la richiesta di analisi della domanda, la prima operazione è il recupero dei `Document` correlati, delegando l'operazione al `DocumentsManager`. Essendo qui coinvolto un processo di valutazione, il metodo che si andrà ad utilizzare per la richiesta sarà il `retrieve_with_score(question)`, in quanto la misura della distanza tra l'informazione recuperata e la richiesta è fondamentale per il calcolo della distanza media.

Aver predisposto un modulo dedicato permette di gestire con facilità questa operazione, garantendo anche una miglior logica del codice, in quanto la classe `QuestionScanner` sarà l'unica ad avere un'istanza di `DocumentsManager`, evitando ridondanze.

5.6.2 Calcolo dei parametri

Il calcolo dei parametri è stato estesamente discusso nel paragrafo 4.5 del capitolo precedente. Da un punto di vista implementativo, sono disponibili metodi dedicati ad ogni metrica:

- `_get_entropy()`: si occupa di calcolare l'entropia sulla base della differenza tra i documenti recuperati dal `DocumentsManager` (4.5.2).
- `_get_average()`: calcola la media della distanza tra i `Document` recuperati e la richiesta di partenza (4.5.1).
- `_get_structural_quality()`: calcola la qualità strutturale come analizzato nel paragrafo dedicato nella parte di implementazione (4.5.3).

Tutti i parametri vengono arrotondati a due cifre decimali.

5.6.3 Assegnazione punteggio

Il punteggio viene assegnato seguendo la logica descritta nel diagramma di flusso 4.3, implementato seguendo una normale logica condizionale.

Per motivi di sviluppo e, in un secondo momento, di controllo, si è deciso di mettere a disposizione due metodi, rispettivamente `_grade()` e `_grade_with_message()`: entrambi restituiscono il valore di punteggio, ma `_grade_with_message()` stampa anche a console un messaggio pre-configurato che avvisa circa la qualità della

domanda. Questo si è reso necessario in quanto il punteggio è un valore che va da 0 a 5: per rendere più leggibile il codice si è deciso di “mascherare” questi valori in forma di **costanti** (sempre definite nel file con tutte le variabili di configurazione, ma non modificabili come variabili di ambiente, in quanto, appunto, *costanti*), che richiamano, come già trattato nel paragrafo 4.5.4, il significato del punteggio.

Per semplificare le operazioni di analisi di funzionamento dell’assistente si è quindi predisposta questa opzione `_with_message`, che consente di vedere immediatamente a console in quale caso ricade l’analisi della domanda (sicuramente migliore, da un punto di vista di immediatezza e comprensibilità, del valore numerico).

5.6.4 Interfaccia del modulo

L’interfaccia del modulo è il metodo `scan(...)`, che richiede, come parametri:

- `print_metrics`: Bool: nel caso sia positivo, vengono stampate a console anche le metriche assegnate alla domanda. Utile soprattutto in contesti di test.
- `return_message`: Bool: nel caso sia positivo, viene stampato a video il messaggio corrispondente al punteggio, discusso nel paragrafo precedente.

`scan()` si limita a richiamare la versione di `grade()` sulla base del parametro `return_message`, e a stampare le metriche se necessario.

5.6.5 Analisi del flusso dati

Si propone anche per questo modulo un diagramma che aiuti a visualizzare il flusso dati e la sequenza delle operazioni, riassumendo quanto descritto. Viene proposto spezzato in orizzontale, per ottimizzarne la presentazione.

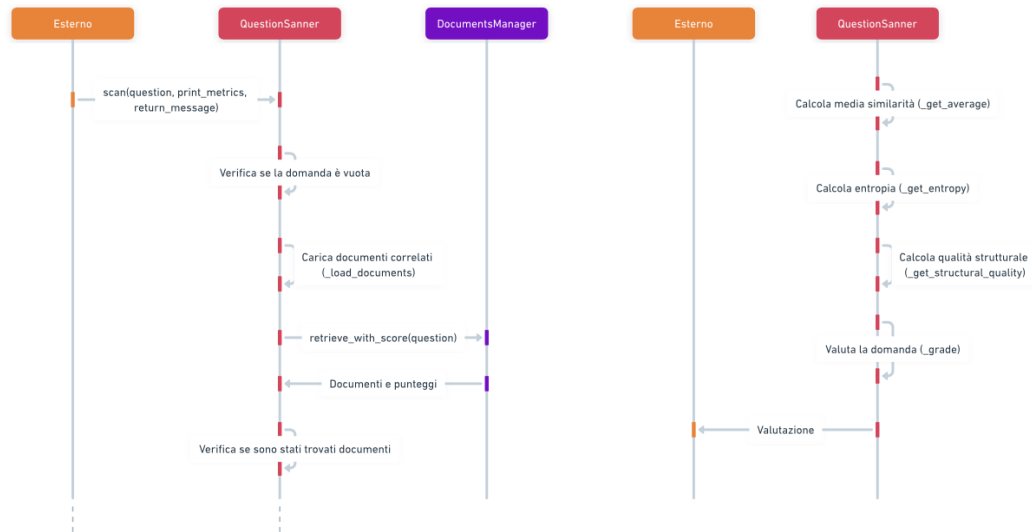


Figura 5.10: QuestionScanner - Diagramma di flusso

5.7 Ottimizzazione delle risposte: AnswerBuilder

L'**AnswerBuilder** conclude la trattazione dei tre moduli principali che compongono l'assistente: è quello in caricato di generare le risposte alle richieste degli utenti, migliorate sulla base delle valutazioni espresse dal **QuestionScanner** e dalle indicazioni di formattazione. Condivide con il **QuestionBuilder** la logica dell'utilizzo degli esempi e la gestione della terza e ultima collezione vettoriale. Infine, come opzioni rilevanti già anticipate nel capitolo precedente, vi sono la possibilità di avere una generazione in tempo reale della risposta e la denominazione delle conversazioni.

5.7.1 Inizializzazione del modulo

Il modulo fa ampio uso della collezione vettoriale di esempi di domande che deve essere conseguentemente gestita. Anche qui vi è il metodo `_load_vecstore()` dedicato per la particolare tipologia di eventi, che genera o recupera la collezione, automaticamente, ad ogni avvio. La logica di funzionamento è la stessa già trattata nel **QuestionBuilder** e soprattutto nel **DocumentManager** (5.5).

Si vogliono trattare, comunque, gli esempi, che sono strutturalmente più semplici di quelli del **QuestionBuilder**: contengono infatti soltanto l'esempio di una domanda e la rispettiva risposta formattata nel modo ottimale.

Per le necessità dell'azienda si è deciso di **guidare la formattazione** dell'assistente in forma di codice **HTML**, considerato che la visualizzazione delle risposte avviene in ambiente *web*.

```

1 {
2   "question" : "Quali sono le differenze tra il modulo
3               X ed il modulo Y?",
4   "answer"   : "<h3>Incompatibilità tra X ed Y</h3>
5               {...panoramica incompatibilità...}
6               <br/>
7               Di seguito, per una panoramica completa delle
8               incompatibilità tra i due moduli e le varie
9               opzioni che possono essere gestite,
10              si propone una tabella:
11              <table>...</table>",
12 }

```

Listing 5.4: Rappresentazione JSON dell'esempio di riscrittura di una risposta

Come si può dedurre dall'esempio riportato, si introduce la possibilità di **indicare** al modello il comportamento da adottare nella formattazione di tutta una classe di richieste in modo dinamico e senza un numero eccessivo di esempi.

Per riprendere l'esempio della generazione di una tabella, si noti come questa sia stata indicata come risposta ad un quesito che prevede il confronto tra due moduli: quando si cercherà un'esempio per una domanda che contenga un confronto differente ma semanticamente simile (ad esempio sempre tra moduli, a volte anche tra soluzioni o processi, in base alla varietà ed il numero di esempi disponibili) questo verrà selezionato e il modello sarà guidato a formattare la risposta secondo le indicazioni.

La gestione della collezione è automatizzata ed avviene al momento di creazione del modulo, così come il controllo di nuovi esempi (che sempre segue la regola già discussa dell'utilizzo della variabile di ambiente, in questo caso `Config.LOAD_ANSWERS_SAMPLES_FROM_DIR`). Anche per l'`AnswerBuilder`, poi, si è deciso di mettere a disposizione un metodo per l'aggiunta di esempi codificati mediante **JSON**, logicamente simile a quello discusso per il `QuestionBuilder` (5.5.3), con sempre la possibilità di doppio salvataggio tra collezione vettoriale e file **JSON** di *default*.

5.7.2 Il processo di costruzione della risposta

Il processo di generazione della risposta inizia con la chiamata ad uno dei quattro metodi messi a disposizione dal modulo, che differiscono per la modalità di generazione e per il contesto del prompt. In questo paragrafo si discute la struttura del metodo “di base”, ovvero `build()`, mentre nei paragrafi successivi si discute

circa le differenti opzioni, che ad ogni modo condividono la struttura e la logica di quanto verrà trattato di seguito.

I parametri

Sono uguali per tutti e quattro i metodi:

- **question**: stringa rappresentante la domanda dell'utente.
- **documents**: lista dei `Document` contenenti le informazioni recuperate dalla base di conoscenza specifica dell'assistente più vicine alla richiesta.
- **verbose**: opzione per stampare o meno informazioni aggiuntive sul processo di costruzione della risposta a console. Il valore è Boolean.
- **citation**: aggiunge un semplice elenco con la citazione delle fonti dopo la domanda. Il parametro esiste in quanto il modulo ha come comportamento di default l'estrazione della fonti dalla domanda ed il salvataggio nella memoria dello stesso: nel caso in esame il sistema nel suo complesso andrà a gestire le fonti prelevandole dalla memoria del modulo e salvandole nella base di dati, ma si voleva comunque offrire la possibilità di avere una visualizzazione più semplice ed immediata delle fonti, senza per questo vincolare il client ad un comportamento specifico. Anche questo valore è Boolean.

Gestione del punteggio

Questo passaggio può, *opzionalmente*, essere **saltato**, per rispettare i requisiti di modularità dell'applicativo. Se infatti questo fosse un passaggio obbligato ne conseguirebbe il vincolo di costruire la pipeline includendo sempre anche l'analisi da parte del `QuestionScanner` della domanda.

Nel caso in cui il punteggio (**grade**) sia disponibile viene chiamato il metodo `_manage_grade()` passandolo come parametro insieme alla lista di `Document` recuperati.

Il metodo ritorna una tupla di valori (`bool`, `string`), in base all'elaborazione del punteggio.

Se il **grade** appartiene alle casistiche `OK` e `OK_PLUS_SOLUTIONS`, siamo nella situazione nella quale la domanda è buona e può quindi procedere nel processo di costruzione: in questo caso la coppia di valori ritornati sarà valorizzata a (`true`, `None`). In questo caso non viene utilizzato il secondo parametro di ritorno, che come vedremo è dedicato alla risposta di default.

Nel caso in cui, invece, il punteggio appartenga ad una delle altre casistiche, si prepara la stringa di risposta, formattata sempre in `HTML`. I `Document` che compongono la lista passata come parametro al metodo possono essere utilizzati

per dare un'idea all'utente di cosa si è trovato, una lista di spunti per provare a migliorare la domanda: questo avviene ovviamente nel caso il punteggio indichi un fallimento a causa del contesto. Se il punteggio ricade in questa casistica la tupla ritornata sarà nella forma (`false, pre_defined_message: str`)

Sulla base della tupla fornita da `_manage_grade()` il metodo di `build()` può interrompere il processo e restituire una risposta oppure continuare.

Costruzione del prompt

Nel caso il punteggio lo permetta, il passaggio successivo è la costruzione del **prompt** che viene delegata al metodo `_build_prompt()`, che si limita a richiedere come parametri la domanda ed il punteggio.

Gli esempi necessari, se disponibili, per la costruzione del prompt vengono recuperati all'interno del metodo, utilizzandone un altro, `_get_samples`, che si occupa di interrogare la collezione vettoriale.

I prompt generati possono essere di tre tipologie:

- **zero-shot**: questo è il caso nel quale ci si ritrova se non vengono trovati esempi di formattazione per la domanda. Il prompt richiederà al modello una risposta fornendo le informazioni necessarie e richiedendo l'utilizzo del formato HTML ed alcune informazioni di formattazione per ottimizzare l'output (come l'utilizzo di elenchi puntati, la suddivisione in paragrafi, l'utilizzo del grassetto per le parole chiave, ecc..). Nel caso il punteggio sia `OK_PLUS_SOLUTION` si richiede anche di fornire l'elenco delle soluzioni in "*consiglio*".

Con un numero sufficientemente alto di esempi (dai 10 in su, dalle prove che sono state fatte) è raro ricadere in questa casistica, che non garantisce purtroppo una buona formattazione.

- **few-shot**: situazione migliore, quando il punteggio è OK e vengono trovati alcuni esempi correlati alla domanda. Il prompt viene costruito in modo logicamente simile a quanto è stato trattato per il `QuestionBuilder()` nel caso di *few-shot*: si predispose un **prefisso**, lo spazio per gli **esempi** ed un **suffisso** nel quale si fanno le richieste. Il template che esce da questo processo ha già inseriti gli esempi al loro posto, lasciando il resto dei parametri *dinamici* da inserire (la domanda e le informazioni specifiche con le quali costruirla).

In caso non venga utilizzato il `QuestionScanner` nella pipeline, e quindi non si abbia a disposizione un punteggio, si ricade in questa casistica.

- **few-shot con consigli**: Stessa logica del punto precedente, con un **suffisso** leggermente **diverso**, nel quale si richiede al modello la generazione di una risposta con un elenco di possibili spunti per affinare, opzionalmente, l'eventuale successiva richiesta.

Gestione delle chiamate al modello

Ricevuto il prompt dal metodo `_build_prompt()` si deve passare all'interrogazione del modello: in base alla tipologia del template ed il punteggio (o la sua assenza), vengono aggiunti i parametri necessari.

Formattazione della risposta

Una volta ricevuta la risposta rimane un passaggio importante da eseguire: l'estrazione delle fonti.

per tutti i prompt prodotti, **indipendentemente** dalla casistica, viene sempre richiesto al modello di citare quali `Document` ha utilizzato per la generazione di quella particolare risposta.

Si supponga ad esempio che siano stati forniti, per la costruzione della domanda, cinque “pezzi” di informazione distinti, ognuno appartenente ad un diverso `Document`: si ipotizzi adesso che il modello utilizzi solamente il primo ed il quarto per la generazione della risposta, come si ricava questa informazione?

Sia nel prompt che negli esempi si indica all'LLM di citare le informazioni utilizzate sfruttando la seguente codifica (che riprende il caso dell'esempio):

```
Document [0] , Document [3]
```

L'output generato deve quindi essere processato da un metodo che estragga questa informazione, considerando anche che, a volte, il modello riporta le informazioni citate non esattamente come gli è stato indicato.

Per questo compito si è predisposto il metodo `_format_answer()`, che ha come parametri sempre la domanda e la lista di `Document`.

Il metodo utilizza una combinazione di **espressioni regolari**, le *regex*, strumenti potenti per la ricerca e la manipolazione di stringhe basate su schemi specifici. Da un punto di vista logico-funzionale, una regex è una sequenza di caratteri che definisce un modello di ricerca, utilizzando simboli speciali (come `*`, `+`, `?`, e `[]`) per rappresentare insiemi di caratteri o ripetizioni. Questo modello permette di identificare, estrarre, sostituire o validare parti di testo che corrispondono allo schema definito. Constatato, durante un estesa fase di test, che a volte il modello non riporta le informazioni esattamente come richiesto, sono state predisposte le regex per gestire anche questi casi sporadici (ad esempio `Document (0)` al posto di `Document [0]`).

Estratti i numeri corrispondenti ad un certo numero variabile dei `Document` componenti la lista, viene rimossa ogni loro citazione nella risposta.

Il riferimento alle fonti viene quindi salvato in una nuova lista di informazioni effettivamente utilizzate all'interno della memoria del modulo, che si rifà sempre

all'ultima domanda costruita (si discuterà successivamente come queste informazioni vengano prontamente salvate nella rispettiva tabella nella base dati).

Opzionalmente, in base al parametro specificato nel metodo di `build()` che specifica la richiesta di citare i documenti nella risposta in modo "semplice", viene aggiunta in coda alla risposta del modello un elenco in formato HTML con il richiamo ai titoli dei documenti aziendali ed i capitoli (o paragrafi) nei quali è stata trovata l'informazione sulla base della quale la risposta è stata prodotta.

Diagramma di flusso riassuntivo

Si propone di seguito un diagramma di flusso che ha come obiettivo quello di riassumere un poco quanto discusso in questo paragrafo, semplificandone la visualizzazione.

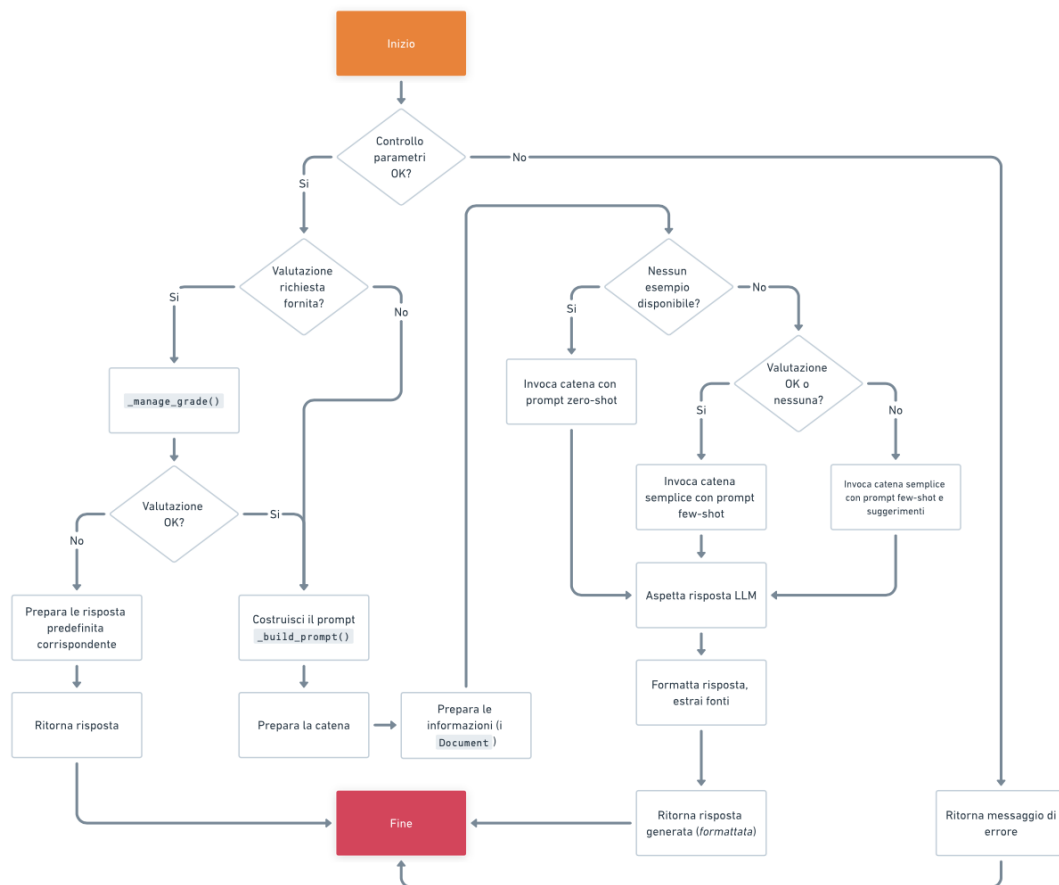


Figura 5.11: Diagramma di flusso - processo di costruzione della risposta

5.7.3 Generazione in tempo reale

Come largamente anticipato in precedenza, il modulo mette a disposizione sia la costruzione della domanda appena vista nel paragrafo precedente, sia una logicamente identica ma che avviene in tempo reale.

Il metodo di `build()` segue una strategia lineare per generare una risposta: vengono elaborati i dati necessari ed elaborati in modo sequenziale, costruendo l'intera risposta prima di restituirla, senza mostrare all'utente (o meglio all'elemento di **client**, ovvero il soggetto che fa "affidamento" sul modulo) nulla fino a quando l'elaborazione non è completamente terminata. Questo comportamento può comportare dei vantaggi, come la possibilità di manipolare la risposta prima di restituirla (abilitando, ad esempio, la funzionalità di traduzione), ma anche degli svantaggi, primo su tutti una latenza percepita dall'utente superiore, dovuta all'attesa di una generazione completa, durante la quale non si hanno feedback.

Il metodo che realizza l'altra opzione è `stream_build()`, che adotta un approccio di streaming per la generazione della risposta. Invece di aspettare che il contenuto sia completo prima di restituirlo, il metodo costruisce la risposta a piccoli blocchi (anche chiamati *chunk*) e li trasmette immediatamente. Questo comportamento permette al *client* di ricevere subito parti della risposta, man mano che questa viene generata.

Lo streaming è particolarmente utile per aumentare la reattività del sistema, offrendo un'interazione più fluida e dinamica.

Da un punto di vista tecnico, la generazione "a flusso", utilizza un generatore Python, che consente di produrre output parziali attraverso l'uso della tecnica di `yield` all'interno del metodo, in tutti i punti nei quali l'output viene generato ed è pronto per essere parzialmente restituito al chiamante. Durante l'esecuzione, la richiesta viene prima formattata e passata attraverso una catena di componenti messi a disposizione dal framework *LangChain*, che permette di richiedere una generazione in questo particolare formato. I segmenti generati in tempo reale dal modello vengono processati da un **Parser**, ovvero un elemento che agisce sui frammenti generati estraendone solo il contenuto testuale, assicurando poi coerenza e chiarezza anche nel flusso di dati parziale.

Questa modalità non permette, oltre al servizio di traduzione, nemmeno di estrarre le fonti e di conseguenza formattare in modo ottimale la domanda. Per risolvere questo problema si utilizza una combinazione di operazioni cosiddette *client* e *server*: l'estrazione delle fonti viene delegata all'utilizzatore del metodo, che potrà poi capire quali sono state effettivamente utilizzate facendo un confronto tra i **Document** relativi a quella risposta.

Per il lavoro di testi si è deciso di utilizzare questa modalità di generazione delle risposte.

5.7.4 Opzioni di contesto

Per concludere la trattazione del modulo, si cita la possibilità di utilizzare, in alternativa ai metodi `build()` e `stream_build()`, i corrispettivi `build_with_context()` e `stream_build_with_context()`.

Le due coppie di metodi condividono le stesse identiche logiche, ma la seconda aggiunge nel prompt di generazione anche lo **Snippet** della conversazione, in modo che sia ancora più chiaro al modello il contesto della conversazione per quanto riguarda l'ultimo pezzo di dialogo. Questa funzionalità è stata predisposta per garantire il mantenimento del contesto anche in quei casi nei quali il modulo `QuestionBuilder` non venga utilizzato, permettendo comunque al modello di tenere in considerazione la logica della conversazione.

La scelta di predisporre due metodi separati, senza utilizzare un parametro per segnalare la differenza, segue il principio di leggibilità del codice: in questo modo è più immediato capire quando si è scelta l'opzione con il mantenimento del contesto.

5.8 Gestione delle traduzioni: Translator

Il **Translator** è il metodo che si occupa di implementare i servizi di traduzione. Alla sua base vi è l'utilizzo del servizio di traduzione messo a disposizione dalla piattaforma **Google Cloud**, affinato dall'uso di un **glossario**.

Per utilizzare i servizi messi a disposizione di Google, come per OpenAI, è necessario predisporre delle credenziali corrette, che vanno generate in seguito alla sottoscrizione di una determinata classe di servizi. Per il funzionamento dell'assistente è necessario avere a disposizione una sottoscrizione al servizio che comprenda le funzionalità di traduzione; vanno in seguito generate le credenziali specifiche per quel servizio, che comprendano anche i permessi per la generazione ed utilizzo di un glossario. Una volta generate le credenziali queste vanno fornite all'interno della sezione dell'applicativo dedicato, in forma di file **JSON**.

5.8.1 Glossario: struttura ed inizializzazione

In principio è importante definire cos'è un glossario nel contesto di traduzione.

Un glossario è una raccolta di termini specialistici o specifici di un determinato dominio, accompagnati dalle relative definizioni o spiegazioni. Esso funge da strumento di riferimento per chiarire il significato di vocaboli che potrebbero risultare oscuri o ambigui per chi non è esperto nel campo in questione.

L'utilizzo di un glossario favorisce una comunicazione più precisa ed efficace, evitando fraintendimenti.

Nello specifico del lavoro di Tesi, si è rivelato necessario prevedere un sistema che **istruisse** i servizi di traduzione di Google sul quali fossero le parole che **non** necessitavano di traduzione, e potenzialmente specificare traduzioni specifiche per determinate parole. Se non si utilizza un glossario è facile che, per quanto riguarda la terminologia aziendale, le traduzioni siano incorrette: nel caso in esame è molto probabile che nelle risposte (o addirittura già nelle domande) vi siano riferimenti a termini che la traduzione stravolgerebbe.

A livello implementativo il glossario è rappresentato, localmente, da un file in formato **csv** (formato di file semplice utilizzato per archiviare dati tabulari, dove ogni riga rappresenta un record e i valori dei campi sono separati da caratteri predefiniti, solitamente virgole).

Per poter essere utilizzato dal servizio di traduzione, il glossario così creato deve essere caricato in uno specifico spazio nel sistema di archiviazione del progetto Google al quale il servizio di appartiene. Successivamente, è necessario specificare al servizio di traduzione la volontà di utilizzare il glossario e la sua posizione: questo permetterà la generazione di un oggetto Glossario interno al progetto di Google Cloud, che verrà identificato mediante un nome e che potrà essere utilizzato richiedendolo al momento della traduzione.

Dal punto di vista pratico, questo avviene al primo utilizzo, quando il glossario non è ancora stato creato, e all'eventuale aggiornamento: per garantire il funzionamento del modulo, si è predisposto un metodo di inizializzazione che viene eseguito ad ogni istanziazione del modulo: `create_glossary()`.

Il metodo si occupa di localizzare il glossario a livello locale, di controllare la sua eventuale presenza nel sistema Cloud e della creazione in caso il controllo di esito negativo.

Per identificarlo si utilizza un nome di glossario specificato a livello di sistema come variabile globale, utilizzata anche per il suo utilizzo.

Il file in formato `csv` è strutturato come una tabella la cui prima riga specifica le lingue supportate, nel caso in esame l'inglese e l'italiano. Seguono, riga dopo riga, le specifiche traduzioni, come è riportato nel seguente esempio. Il servizio di traduzione, quando verrà richiesta l'omonima operazione da una lingua ad un'altra presente nel glossario, si limiterà a riportare i valori specificati.

en	it
Transport Management System	Transport Management System
Transporters	Autotrasportatori

5.8.2 Processo di traduzione

Il metodo `_translate_text_with_glossary()` viene utilizzato per la chiamata al servizio indipendentemente dalla direzione di traduzione, richiede i seguenti parametri:

- `text`: la stringa contenente il testo da tradurre.
- `source_language_code`: una stringa che indichi la lingua sorgente. Questa deve necessariamente rispettare le codifiche definite dallo standard **ISO 639-1** [28] che definiscono un codice di 2 lettere per identificare una lingua.
- `target_language_code`: vale la stessa logica del punto precedente, identifica la lingua di destinazione della traduzione.

Oltre a questi tre parametri, vengono utilizzate due variabili globali per identificare il progetto ed il glossario all'interno del sistema di GoogleCloud.

Il metodo è responsabile dell'effettiva traduzione di un testo. Inizialmente, stabilisce una connessione con il servizio di traduzione tramite il client. Successivamente, recupera il percorso del glossario specifico, identificato dal suo ID univoco, all'interno del progetto Google Cloud (spiegato nel paragrafo precedente).

Successivamente, configura le impostazioni di traduzione, associando il glossario alla richiesta e specificando le lingue di origine e destinazione del testo da tradurre.

Infine, invia la richiesta di traduzione al servizio Google Cloud Translation, che utilizza il glossario per applicare le traduzioni personalizzate definite. Il metodo restituisce il testo tradotto, garantendo una maggiore accuratezza e coerenza nella traduzione di termini specialistici o specifici del dominio applicativo.

5.8.3 Logica di traduzione

La logica scelta per il servizio utilizza due metodi:

- `translate_text_to_embeddings()`
- `translate_text_from_embeddings()`

Questi gestiscono la traduzione del testo in relazione alla lingua utilizzata per gli embedding (rappresentazione in formato JSON) nel sistema.

`translate_text_to_embeddings()` traduce il testo dalla lingua di origine dell'utente alla lingua di embedding, per l'elaborazione successiva da parte del modello.

`translate_text_from_embeddings()`, invece, traduce il testo dalla lingua di embedding alla lingua dell'utente, rendendo la risposta comprensibile. Prima della traduzione, il testo viene pre-elaborato per preservare URL e parole chiave specifiche del dominio, sostituendoli con segnaposto. Dopo la traduzione, questi elementi vengono ripristinati nel testo tradotto.

In entrambi i casi, la traduzione effettiva viene eseguita tramite il metodo `_translate_text_with_glossary()`, sfruttando il glossario per garantire una traduzione più accurata e coerente dei termini specialistici.

A livello di sistema si è deciso di pre-configurazione una lingua di default per gli embeddings, per facilitare questo processo: sarà quindi immediato per l'assistente capire se sia necessario o meno attivare il servizio di traduzione semplicemente verificando se la richiesta dell'utente sia uguale oppure no alla lingua con la quale sono stati scritti i documenti aziendali, lingua di default anche degli embeddings.

Si propone, di seguito, un diagramma di sequenza riassuntivo per illustrare il funzionamento del modulo.

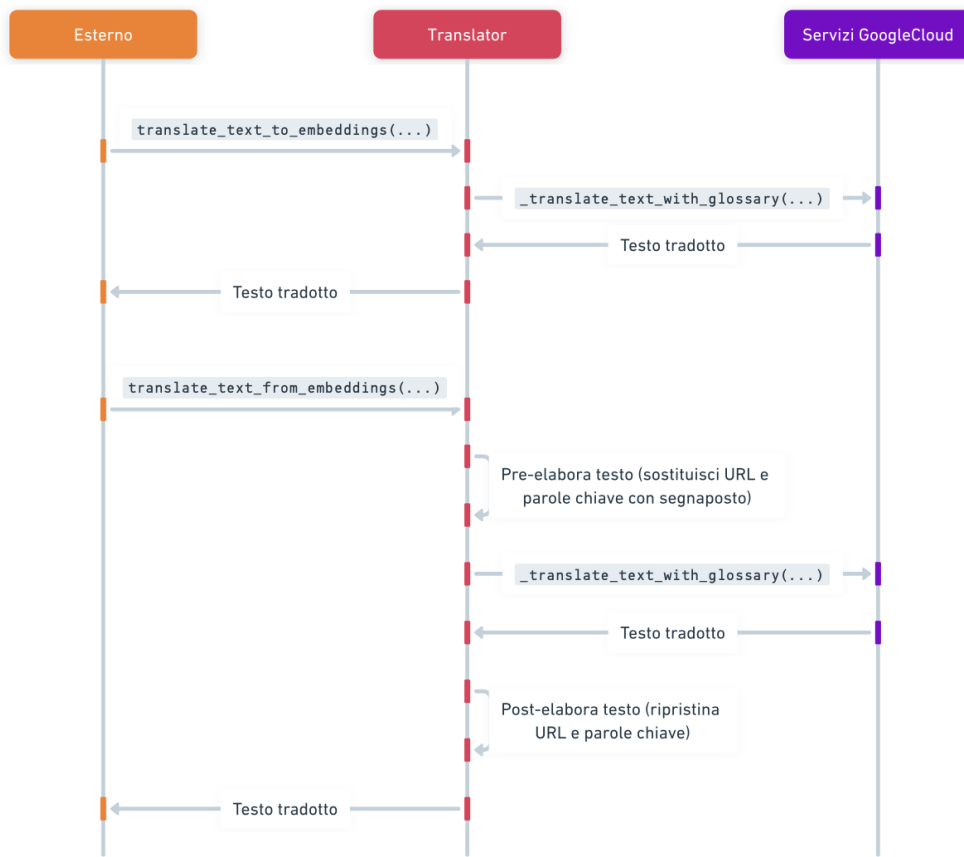


Figura 5.12: Translator - Diagramma di flusso

5.9 Punto comune per le configurazioni: Config

Come già ampiamente discusso, si è deciso di dedicare un modulo alla gestione delle variabili di configurazione, suddivise in costanti e variabili d'ambiente.

Le costanti raccolgono tutte quelle variabili che non necessitano di modifiche da parte di chi installa l'assistente, ma che è comunque buona pratica non impostare direttamente nel punto di utilizzo (in gergo tecnico *hard-coded*). Ricadono in questa categoria i **messaggi predefiniti** in caso di punteggi non buoni, i **nomi dei templates** contenenti i *prompt*, le **corrispondenze valore-costante** dei punteggi del QuestionScanner, le liste con i **campi obbligatori**, tutti i parametri delle **collezioni** (nomi, percorsi, vincoli di ricerca), quelli per il **servizio di traduzione** e i **valori di default**.

La categoria delle variabili di ambiente contiene invece tutti quei parametri che, per un motivo od un altro, possano essere di interesse per chi configura l'assistente. Tra queste si trovano, ad esempio, i **parametri di configurazione**

del **modello LLM**, quelli relativi alla **documentazione aziendale** (ad esempio quanti documenti recuperare dalla collezione vettoriale), ed ancora quelli relativi agli **esempi** (per esempio la loro quantità o la soglia di similarità di ricerca), i parametri per segnalare la presenza di **nuovi esempi o documentazione** nelle cartelle ed i **parametri di valutazione** del `QuestionScanner`.

Tutti questi valori sono definiti nell'ambiente di esecuzione dell'applicativo, dal quale sono recuperate: possono essere definite in più modi da chi installa l'assistente, ad esempio utilizzando un terminale oppure, come nel caso in esame, utilizzando uno speciale file di configurazione **Docker Compose**, che permette, tra le altre cose, di valorizzare in partenza tutte queste informazioni, mantenendo una correlazione identica tra il nome con le quali queste variabili sono definite nel file di configurazione.

5.10 Funzioni di utilità: Utils

Il modulo di `Utils` contiene le funzionalità aggiuntive non abbastanza caratterizzanti per esser organizzate in un modulo dedicato, come per tutto ciò che è stato trattato fino ad ora. Si propone, di seguito, una breve descrizione di queste funzionalità.

Funzionalità generale per il caricamento dei prompt

Permette di ricevere una lista di coppie chiave-calore, in `python` realizzata mediante `dictionary`, dove le chiavi sono i nomi dei prompt definiti come costanti e i valori il loro contenuto. La funzionalità viene utilizzata ogni volta che sia necessario il template di un determinato prompt (che viene identificato dal nome della variabile di costante).

Funzionalità generale per il controllo dell'esistenza delle collezioni

Permette di verificare, ricevuto il nome di una collezione, se questa esiste nella base di conoscenza vettoriale. Viene utilizzata dai tre moduli (`QuestionBuilder`, `AnswerBuilder` e `DocumentsManager`) durante le operazioni di inizializzazione.

L'oggetto Snippet

Lo `Snippet` è stato già ampiamente discusso; viene definito in questo modulo di utilità generale, in forma di oggetto. Per la creazione richiede sia il contenuto che gli identificativi delle domande e della risposta che lo compongono (domanda, risposta, ultima domanda), che possono essere poi velocemente ricavati come proprietà dell'oggetto. Sebbene la decisione di dichiarare un oggetto così semplice possa sembrare

esagerata, è stata presa per aumentare la leggibilità del codice e l'organizzazione logica.

Gestione delle etichette

Le etichette sono quelle *parole chiave* utilizzate per filtrare i metadati dei documenti. Si definisce una classe specifica (`Labels`) inizializzata con un'elenco di `Dictionary` contenenti, per ogni parola chiave (la chiave del dizionario), un coppia di valori che la legano alla tipologia di prodotto e alla sua categorizzazione.

5.11 Unione dei moduli: Pipeline

Quanto trattato fino a questo punto deve essere “montato” per costruire l'assistente, ed il modulo `Pipeline` si propone di realizzarlo.

Considerate le due diverse modalità di generazione della risposta, ovvero la generazione nella sua interezza oppure “a pezzi” (*streamed*), sono stati predisposti due diversi metodi con i quali il chiamante può farsi generare una risposta: il metodo di `run()` e quello di `stream_run()`.

I due metodi condividono la stessa logica di base, ma con una comprensibilmente diversa gestione delle risposte da parte del secondo.

Il modulo, dopo esser stato istanziato, richiede che venga impostata una conversazione, utilizzando lo specifico metodo `set_conversation()` che richiede un oggetto `Conversation` valido, che andrà a salvare come attributo ed utilizzerà per la generazione degli oggetti di `Question` ed `Answer`. La decisione di impostare la conversazione mediante uno specifico metodo e non nel costruttore della classe è stata adottata per permettere l'utilizzo di una singola istanza di `Pipeline` per più conversazioni, che vengono impostate dal chiamante, risparmiando in questo modo memoria.

Per evitare ripetizioni ed allungare eccessivamente il capitolo, si propone quindi la descrizione del metodo di `stream_run()`, che “monta” la pipeline allo stesso modo del primo.

5.11.1 La struttura

Il metodo riceve, come parametri, esclusivamente la **domanda** e, opzionalmente, una **risposta prevista** (casistica di test).

Si inizia controllando se la conversazione sia stata impostata o meno, e nel caso di mancata impostazione si avvisa il chiamante.

Generazione domanda ed invio informazioni

Una volta accertata la presenza dell'oggetto `Conversation`, si genera da questo una nuova domanda, `Question`, fornendone già il contenuto.

A questo punto è già necessario comunicare con il chiamante, specificatamente il **client** predisposto per il lavoro di tesi, per comunicare alcune informazioni: si chiarisce ora questo aspetto.

Nel metodo “semplice” `run()` la risposta alla domanda, è un oggetto `JSON` che contiene, oltre al suo contenuto, anche una serie di informazioni aggiuntive. Nello specifico:

- `conversation_id`: identificativo della conversazione alla quale la domanda è associata, nella base di dati.
- `question_id`: identificativo della domanda nella base dati.
- `question`: la domanda fatta.
- `answer_id`: identificativo della risposta nella base dati.
- `answer`: la risposta.

Queste informazioni sono utili al chiamante per, eventualmente, fare dei successivi riferimenti alla conversazione o a specifiche domande e risposte, utilizzando gli specifici identificativi (che si ricorda vengono generati dal `ConversationManager`, quindi dal **server**).

Questo discorso è valido per la struttura normale di `run()`, cambia per quello trattato in questo paragrafo: le informazioni vanno inviate in forma di risposte ben definite, in modo che il client sappia quali di queste rappresentano informazioni sulla conversazione e quali invece siano il contenuto della “risposta”. Si potrebbe inviare tutto alla fine della generazione, ma in questo caso, se il client (come nel lavoro di tesi) è un'interfaccia grafica, non avrebbe la possibilità di render visibile, graficamente, il messaggio, pezzo per pezzo, fino a quando questo non fosse completamente ricevuto (rendendo de-facto inutile l'utilizzo di un metodo *streamed*). Il client deve quindi avere tutte le informazioni su conversazione, domanda e risposta prima che si inizi la generazione di quest'ultima, per poter disporre, a valle, tutto quanto necessario per accogliere la sequenza di informazioni che seguirà.

Alla luce di queste considerazioni, dopo aver generato l'oggetto `Question`, si restituiscono le informazioni su conversazione e domanda, in forma di `JSON`, con il meccanismo, già visto, dello `yield`. Si propone, in esempio, l'istruzione per ritornare le informazioni circa la conversazione:

```
yield f"conversation_info: {json.dumps(conversation_info)}\n"
```


Ricostruzione ed analisi della domanda

Dopo aver inviato le informazioni al client si procede alla potenziale ricostruzione della domanda. Per prima cosa va controllata la presenza di uno **Snippet**, possibile solo se la domanda non è la prima della conversazione: lo si richiede al **ConversationManager**. Nel caso lo **Snippet** venga fornito, si tenta di riscrivere la domanda mediante il metodo di `build()` del **QuestionBuilder**.

Ricevuta la domanda, nel caso questa sia stata riscritta, si procede all'aggiornamento delle informazioni nella base dati, operate direttamente sull'oggetto **Question** con gli specifici metodi.

Successivamente si procede alla valutazione: la domanda viene fornita al metodo `scan()` del **QuestionScanner**, ed il punteggio salvato in una variabile **grade**, prontamente aggiunto al set di informazioni della domanda. Anche la lista di **Document** recuperati dal **QuestionScanner** sono prontamente salvati all'interno di una variabile.

Costruzione della risposta

Ottenuti il punteggio della domanda e la lista di informazioni associate si procede a ricavare dal **ConversationManager** una nuova risposta. Si aggiungono poi contenuto e **Document** associati e, similmente a quanto avvenuto per quelle relative alla conversazione ed alla domanda, si invia l'identificativo al client.

In base alla presenza o meno dello **Snippet**, si utilizza il metodo di costruzione della risposta con contesto oppure no. Una volta chiamato il metodo che si occupa di costruire la risposta, per ogni suo pezzetto, **definito chunk**, si controlla se questo sia uno dei messaggi predefiniti oppure una risposta generata dal modello (si ricorda come, in base al punteggio del **QuestionScanner**, potenzialmente si possa ricevere una delle domande di follow-up predefinite): nel primo caso sarà sufficiente un solo **chunk** di risposta, dopo il quale sarà possibile chiudere la trasmissione, mentre nel secondo caso bisognerà iterare su ognuno di quelli generati dall'LLM, attenendo la fine e chiudendo solo successivamente il canale (avvisando con una stringa predefinita).

Sequenza

Si propone, come fatto per altri moduli, un diagramma di sequenza per illustrare visivamente il processo di generazione della risposta. Il diagramma è composto da due diverse immagini, per aumentarne (per quanto possibile dal formato), la leggibilità.

Implementazione chatbot

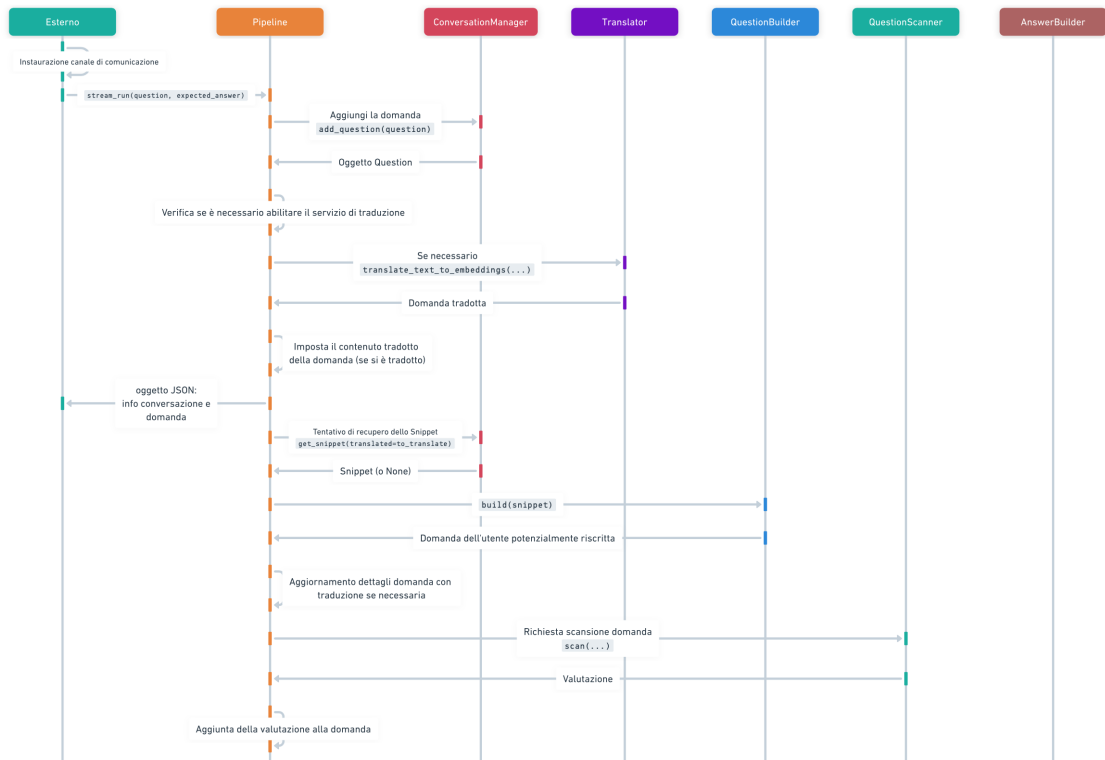


Figura 5.13: Sequenza di generazione risposta

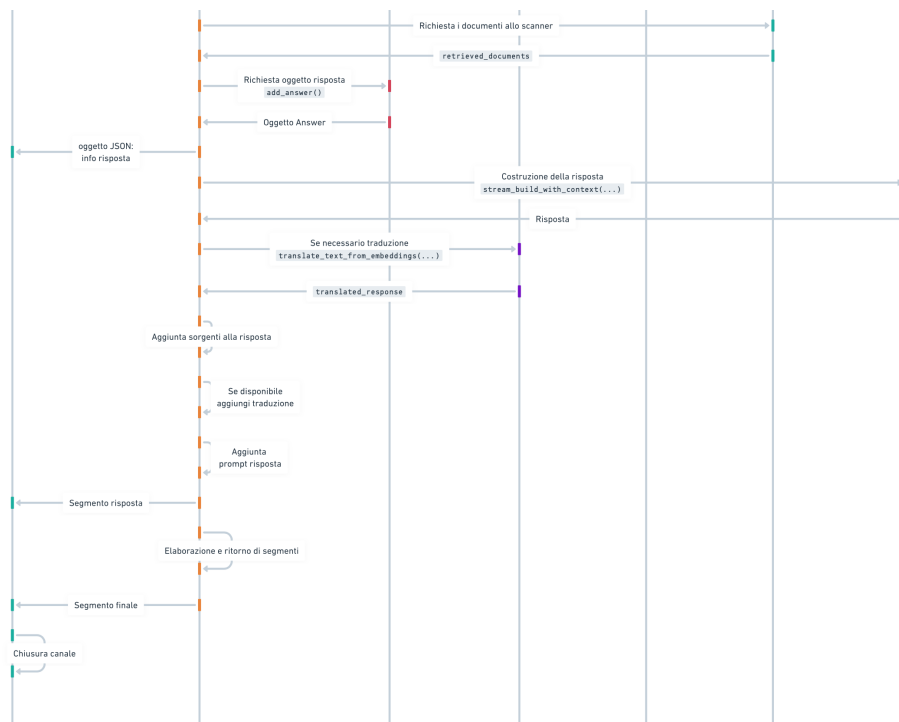


Figura 5.14: Sequenza di generazione risposta, seconda parte

5.12 Messa a disposizione delle funzionalità: API

Tutte le funzionalità dell'applicativo devono in qualche modo essere accessibili all'esterno, in modo da renderlo utilizzabile.

In *generale* le **API**, acronimo di Application Programming Interfaces, sono essenzialmente un insieme di regole e specifiche che permettono a diversi software di comunicare e interagire tra loro. Immaginale come un traduttore che consente a due persone che parlano lingue diverse di comprendersi. Nel mondo digitale, le API permettono a diverse applicazioni, anche create da sviluppatori diversi, di scambiarsi dati e funzionalità in modo sicuro e standardizzato. Questo facilita l'integrazione di servizi e la creazione di nuove applicazioni.

Nel contesto di questa applicazione, le **API** vengono messe a disposizione grazie a **Flask**, un framework web leggero che ne costituisce il cuore. Si occupa di gestire le richieste **HTTP** in arrivo, instradandole alle funzioni appropriate, ed allo stesso modo di generare le risposte da inviare al client.

Si abilita anche l'opzione **CORS** (*Cross-Origin Resource Sharing*) per consentire alle risorse messe a disposizione di essere accessibili da domini diversi da quello in cui sono ospitate, fondamentale per applicazioni (ad esempio un'interfaccia grafica) in esecuzione su domini differenti: abilita, ad esempio, la possibilità di aver il chatbot in esecuzione su un ambiente cloud separato.

L'architettura con la quale sono definite le API è l'**architettura REST** (*Representational State Transfer*), modello di progettazione per creare API web che utilizzano i metodi HTTP standard (GET, POST, PUT, DELETE) per interagire con le risorse, rappresentate da URI, in modo stateless e scalabile.

Di seguito si propone l'elenco delle API messe a disposizione dal sistema:

/api/chat (POST): Gestisce le domande, ricevendo in input insieme, opzionalmente, all'ID della conversazione, al nome, alla lingua ed alla risposta attesa. Crea o recupera conversazioni esistenti ed esegue la pipeline per generare risposte.

/api/stream_chat (POST): Simile alla precedente, ma fornisce risposte in modalità streaming, per un'esperienza utente più interattiva.

/api/conversations (GET): Restituisce informazioni su tutte le conversazioni.

/api/conversation/<conversation_id> (GET): Restituisce informazioni su una conversazione specifica.

/api/conversation/<conversation_id>/messages (GET): Restituisce tutti i messaggi di una conversazione specifica.

/api/conversation/name (POST): Imposta il nome di una conversazione.

- /api/conversation/time (POST):** Imposta l'orario di creazione di una conversazione.
- /api/conversation/notes (POST):** Imposta o aggiunge note a una conversazione.
- /api/conversation (DELETE):** Elimina una conversazione specifica.
- /api/conversations (DELETE):** Elimina tutte le conversazioni.
- /api/question/sample (POST):** Aggiunge un esempio di domanda e risposta per migliorare il modello di generazione delle domande.
- /api/answer/sample (POST):** Aggiunge un esempio di domanda e risposta per migliorare il modello di generazione delle risposte.
- /api/conversation/<...>/<...>/documents (GET):** Restituisce i documenti utilizzati per generare una risposta specifica.
- /api/conversation/<...>/answer/<...>/prompt (GET):** Restituisce il prompt utilizzato per generare una risposta specifica.
- /api/conversation/<...>/question/<...>/prompt (GET):** Se presente, restituisce il prompt utilizzato per generare una domanda specifica.
- /api/conversation/<...>/question/<...>/updated_message (GET):** Ritorna il messaggio riscritto (se presente) di una domanda specifica.
- /api/add_document_handbook (POST):** Aggiunge un documento pdf alla base di conoscenza del chatbot (alla collezione vettoriale).
- /api/question/scan (POST):** Analizza una domanda utilizzando il `QuestionScanner`.
- /api/name_dialogue (POST):** Assegna un nome ad una conversazione in base alla domanda e alla risposta.

5.13 Predisposizioni per la distribuzione

Per concludere il capitolo si propone un riassunto delle disposizioni per la configurazione dell'applicativo, per chiarire il più possibile le scelte fatte in fase di implementazione: questo paragrafo può anche essere visto come una conclusione riassuntiva, nella quale si ripercorrono i passaggi illustrati in forma di istruzioni per la installazione e configurazione.

Configurazione delle credenziali di autenticazione

Necessario assicurarsi che la cartella `/auth` contenga i file delle credenziali per i servizi LLM (OpenAI) e di traduzione (Google Cloud).

Configurazione del glossario

Specificare un glossario nella cartella `/glossaries`, con il nome del file `special_words_glossary.csv`, necessario per definire come tradurre determinati contenuti nelle lingue supportate.

Configurazione del dockercompose

Utilizzare la configurazione fornita per l'installazione dal repository (l'insieme del codice componente l'assistente). Assicurarsi che i percorsi dei volumi siano configurati correttamente. Impostare i parametri (le variabili d'ambiente di cui si è discusso) `LOAD_QUESTIONS_SAMPLES_FROM_DIR`, `LOAD_ANSWERS_SAMPLES_FROM_DIR` e `LOAD_DOCUMENTS_FROM_DIR` su `true` per la compilazione iniziale, in modo che l'applicazione popoli il database con i dati di esempio e di documentazione forniti. Si consiglia di ricompilare successivamente con i parametri impostati su `false` per evitare di popolare il database ogni volta che il contenitore viene ricaricato.

Build e test

Eseguire prima il comando:

```
docker compose build
```

ed in seguito, per testare il corretto funzionamento dell'assistente eseguire una richiesta POST a `/api/chat` con il payload specificato (...) per verificare il corretto funzionamento dell'applicazione.

Autenticazione

L'applicazione utilizza due servizi che richiedono l'autenticazione:

- **Servizi LLM di OpenAI:** Richiedono una chiave API da inserire in un file `.txt` nella cartella `/auth`.
- **Servizi di traduzione di Google Cloud:** Il chatbot utilizza l'applicazione di traduzione di Google Cloud per tradurre alcuni contenuti. È necessario generare un token di autorizzazione che consenta anche la creazione e l'utilizzo di un glossario, caricato come bucket nello spazio di archiviazione del progetto.

Dati

Il chatbot si basa su un sistema RAG (Retrieval-Augmented Generation) e le fonti di dati con cui lavora devono essere fornite nella cartella `/data`. I dati devono essere pre-elaborati in modo che il sistema possa organizzarli in un database vettoriale. Attualmente sono supportati PDF e video (il contenuto sono i sottotitoli), che devono aderire a una struttura specifica in formato JSON.

Database

L'applicazione utilizza due tipi di database:

- **Database classico (SQLite):** Memorizza conversazioni, dettagli sui prompt utilizzati, le fonti su cui si basano le domande e le eventuali valutazioni.
- **Database vettoriale:** Memorizza documenti (embedding dei dati forniti nella cartella `data`) ed esempi di domande e risposte.

Servizi di traduzione

I servizi di traduzione sono disponibili per le API che non includono la generazione in streaming di una risposta. È possibile specificare un glossario nella cartella `glossaries` per definire come tradurre determinati termini.

Prompt ed esempi

La cartella `/prompts` contiene tutti i prompt utilizzati dall'applicazione per gestire le varie richieste all'LLM, mentre nelle directory `samples/answers/` e

`samples/questions/` è possibile specificare file in formato JSON contenenti gli esempi su cui il sistema baserà la generazione delle risposte.

Capitolo 6

Sviluppo interfaccia grafica

L'interfaccia dell'assistente è stata sviluppata in modo **separato**, indipendente, dall'assistente stesso. Questo approccio permette all'azienda di scegliere se e quando utilizzare l'interfaccia progettata per il lavoro di tesi, senza vincoli.

Nel capitolo 4 si sono discussi i requisiti dell'interfaccia, che da un punto di vista dell'architettura svolge il ruolo di **client** rispetto all'assistente trattato estensivamente nel capitolo 5, che ha invece ruolo di **server**: questo significa semplicemente che il *server-chatbot* mette a disposizione dei **servizi** al *client-interfaccia* che li utilizza.

6.1 Tecnologie utilizzate

6.1.1 React

Per l'implementazione della **GUI**, considerate le caratteristiche ed i requisiti (3), si è optato per un'architettura a **pagina singola**, implementata grazie alla libreria open-source **React**, basata su *JavaScript*, che permette di realizzare, in modo modulare ed efficace, interfacce utente **interattive**.

Il concetto chiave di React sono i **componenti**, unità modulari e riutilizzabili che rappresentano parti dell'interfaccia utente: possono essere definiti come *funzioni* o *classi*, accettano input in forma di proprietà (*props*), e convenzionalmente restituiscono appunto **componenti** HTML, composti utilizzando la sintassi JavaScript nella sua versione estesa, **JSX**.

Uno dei principali punti di forza di React è l'aggiornamento automatico del DOM (*Document Object Model*, rappresentazione ad albero della struttura di un documento HTML o XML, che consente di interagire e modificare dinamicamente il contenuto e lo stile della pagina): la libreria lo realizza utilizzando il **virtual DOM**, la sua versione virtuale, aggiornando esclusivamente il documento (che in questo

caso è inteso come la pagina web, il documento HTML) nelle sue differenze con la versione virtuale.

Per la gestione del **Routing**, ovvero la navigazione tra una funzionalità e l'altra della GUI, si è utilizzata l'estensione **React Router**, una libreria standard per la gestione della navigazione.

Permette di definire rotte e associarle a componenti specifici, aumentando anche la leggibilità del codice.

6.1.2 API

Anche l'interfaccia, come l'assistente, definisce delle sue API, questa volta specifiche "lato client". La tecnologia utilizzata per la realizzazione è quella standard, **Fetch API**, integrata nei principali Browser moderni per effettuare richieste di rete.

Per la gestione delle richieste si utilizzano le **Promise**, (in questo contesto, oggetto in JavaScript che rappresenta l'eventuale completamento o fallimento di un'operazione asincrona, quali sono le richieste al server, permettendo di gestirne i risultati futuri), che semplificano la gestione del flusso di controllo ed aumentano la leggibilità del codice.

6.1.3 Bootstrap

Bootstrap è un **framework front-end** open-source, progettato per semplificare lo sviluppo di interfacce web reattive e moderne. Fornisce una vasta raccolta di componenti e classi predefinite che permettono di creare layout e design coerenti e scalabili senza dover scrivere codice personalizzato da zero. Uno dei suoi elementi distintivi è il **sistema a griglia** (*grid system*), che consente di definire layout responsivi che si adattano automaticamente a diverse dimensioni di schermo.

Offre una vasta gamma di componenti UI predefiniti, tutti personalizzabili e facili da implementare attraverso classi CSS standardizzate. Questi componenti sono progettati per essere accessibili e coerenti, garantendo che l'interfaccia sia intuitiva e funzionale su diversi browser e dispositivi.

Bootstrap si è affermato come uno degli strumenti più popolari per lo sviluppo di siti web responsive grazie alla sua facilità d'uso, alla sua ampia gamma di componenti e alla sua robustezza cross-browser: da qui la sua naturale scelta per il lavoro di tesi

6.1.4 Virtualizzazione

Si accenna qui, brevemente, che l'applicazione costituente l'interfaccia grafica rispetta gli stessi requisiti di virtualizzazione del *back-end* costituito dal chatbot.

La GUI viene predisposta per poter essere installata, eseguita e mantenuta in ambiente virtuale, potenzialmente nella stessa immagine del chatbot, come modulo separato (non attualmente realizzato, ma facilmente integrabile grazie a *Docker compose*). Si è predisposto un server **Nginx** che permette l'installazione dell'applicativo di interfaccia su un dominio differente da quello dal quale arrivano le richieste, svolgendo il ruolo di **reverse proxy**: questa configurazione permette il caso limite nel quale (con le corrette configurazioni e permessi) è possibile avere il chatbot su un dominio, la GUI su un altro e l'accesso a quest'ultima da un browser slegato dall'interfaccia; questa configurazione deve ovviamente tenere conto di tutte le considerazioni di sicurezza opportune, che non sono l'oggetto della tesi.

6.2 Panoramica generale dell'interfaccia

Prima di procedere con la trattazione più approfondita dei componenti che costituiscono la *GUI*, si propone uno sguardo di insieme all'interfaccia, in modo da familiarizzare con l'ambiente che verrà poi sezionato.



Figura 6.1: Panoramica interfaccia grafica

Dall'immagine si possono notare immediatamente due **macro aree**: l'elenco delle conversazioni a sinistra, e l'area di chat a destra.

Sempre a destra, ma in alto, sono presenti una serie di icone che costituiscono funzionalità aggiuntive della GUI.

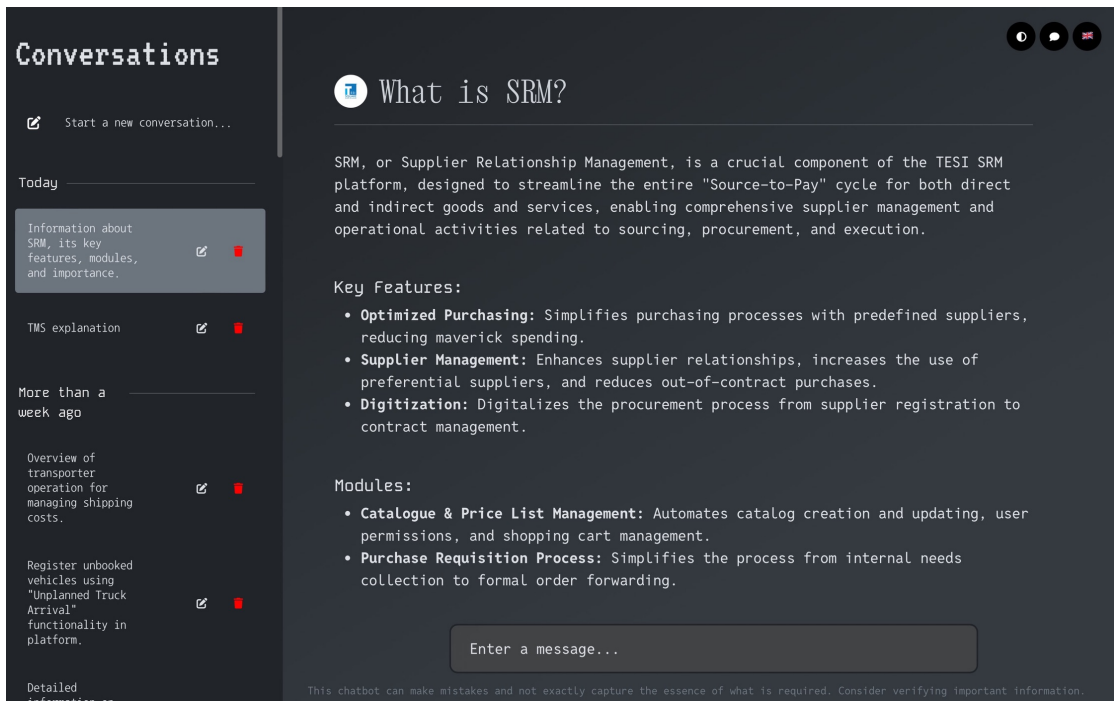


Figura 6.2: Panoramica dell'interfaccia in modalità scura, con una conversazione aperta.

Nella seconda immagine viene mostrata l'interfaccia in modalità scura, com'è stata implementata rispetto ad i requisiti: si noti la differenza di lingua (le domande all'assistente possono essere fatte, attualmente, esclusivamente in inglese per mancanza di una documentazione in italiano). Si è proposta la visualizzazione con una chat aperta. La modalità scura, insieme ad altre funzionalità aggiuntive, viene trattata nel paragrafo 6.6

L'interfaccia nella sua interezza viene, a livello implementativo, **montata** dal **componente ChatBuilder**, su richiesta dell'applicazione ed in seguito alla configurazione delle *routes*, dei percorsi.

Nei prossimi paragrafi si andranno ad analizzare in dettaglio queste aree, a partire dalla colonna delle conversazioni.

6.3 Colonna delle conversazioni

Il componente incaricato della costruzione della barra laterale delle conversazioni è **ConversationsSideBar**, il quale si occupa del **caricamento** della lista delle conversazioni e del loro **ordinamento**.

Le informazioni circa le conversazioni (tutti i dati relativi, come il nome e l'identificativo) sono ricevute dal `ChatBuilder`, che le recupera sfruttando l'API `get_conversations_data()`: tutte le informazioni globali, di interesse di tutta l'interfaccia, vengono dichiarate come stati (mediante l'utilizzo della `useState()` di React) nel livello più alto, nel `ChatBuilder`. Gli **stati** vengono passati dove necessario come proprietà ai moduli, in modo da evitare problemi di inconsistenza dovuti a mancati aggiornamenti o duplicazioni.

Con le informazioni ricevute dal livello superiore, la `ConversationSideBar` può quindi generare il numero opportuno di `ConversationPeek`, singoli componenti di conversazione.

6.3.1 Il Conversation Peek

Questo componente implementa le principali operazioni sulla conversazione: la sua cancellazione, la modifica del nome, e la sua selezione.

L'ultima funzionalità, apparentemente banale, viene implementata a livello di singolo modulo sfruttando un meccanismo che viene adoperato estensivamente nella logica d'implementazione: la conversazione attiva, infatti, è uno **stato** definito (come anticipato prima) nel livello più alto, e passato (sia lo stato che la funzione per modificarlo), dal `ChatBuilder` al `ConversationSideBar`, che a sua volta lo passa ad ogni `ConversationPeek`, che ha quindi la possibilità di utilizzarlo per verificare se sia necessario o meno cambiare *veste grafica* (un "peek" selezionato è diverso da tutti gli altri) o, nel caso l'utente selezioni la conversazione rappresentata dal componente, modificare l'identificativo di quella selezionata con il proprio.

Questa logica viene utilizzata in tutti i casi simili a quello appena trattato; per evitare ripetizioni e ridondanza, si evita di ripetere la spiegazione in seguito.

Ogni `ConversationPeek` ha, ben visibili, le icone di **edit**, per modificare il nome della conversazione, e quello di **delete**, per cancellarla.

Selezionando l'icona del cestino, l'icona di *edit* viene momentaneamente disabilitata fino a quanto non si concluda o annulla l'operazione di eliminazione (in ottica di **error prevention**).

Nel caso venga effettivamente richiesta la cancellazione della conversazione, questa viene comunicata al server utilizzando l'API `delete_conversation()` passando come parametro l'identificativo della conversazione: in caso l'operazione proceda senza errori, l'interfaccia

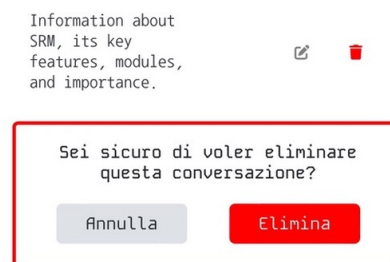


Figura 6.3: Dettaglio cancellazione conversazione

si adatta automaticamente dal momento che l'id della conversazione attiva viene impostato al valore `null`, che causa il reset dello spazio di chat a quello di avvio di una **conversazione vuota**.

Per quanto riguarda l'altra opzione, quella di **edit**, i passaggi sono più di uno:

- Selezione icona di modifica.
- Lo spazio nel quale è possibile modificare il nome della conversazione è occupato dal nome attuale, facilmente modificabile. Nel caso in cui si desideri eliminare il vecchio nome è sufficiente premere sull'icona della scopa, per pulire il campo.
- Inserire o modificare il nome.
- Salvare.



Figura 6.4: Modifica del nome di una conversazione

Anche qui, per evitare possibili errori, fino a quando non si conclude il processo di modifica del nome (o si annulla) il pulsante di eliminazione rimane disabilitato.

In conclusione del paragrafo, si vuole dare una breve descrizione di un componente simile al `ConversationPeek`: il `NewConversationPeek`.

Per mantenere la modularità tipica dell'architettura React, si è deciso infatti di differenziare la *conversazione standard* trattata fino a questo momento, dal componente di **nuova conversazione**.

Questo componente permette, una volta selezionato, di iniziare una nuova conversazione: viene utilizzato soprattutto quando si è in una conversazione e si

decide che, ad esempio per un cambio di argomento, si vuole passare ad una nuova: premendo su questo speciale **peek**, è possibile ritrovarsi nella “posizione” iniziale, quella della figura 6.1, dalla quale è possibile iniziare una nuova conversazione. Nell’immagine è visibile il componente di inizializzazione, che è **sempre** il primo della lista, con la classica icona di *scrittura* che aiuta l’utente ad identificarlo.

6.3.2 L’ordinamento

Da come è possibile vedere nella figura 6.5, le conversazioni vengono ordinate in blocchi, rispettivamente in conversazioni avvenute **oggi, ieri, l’ultima settimana** oppure **più vecchie di una settimana**.

Questa divisione si adatta agli standard dei quali si è discusso nel capitolo 4, nei quali tutti gli assistenti maggiormente diffusi implementano un qualche tipo di raggruppamento delle conversazioni.

Da un punto di vista implementativo, la suddivisione viene realizzata dal `ConversationsSideBar`, che utilizza una serie di componenti `TimeDivider` che adattano il messaggio e le dimensioni sulla base di una chiave passata come parametro. La divisione viene fatta per ogni conversazione, smistandole nello “slot” appropriato.

Conversazioni

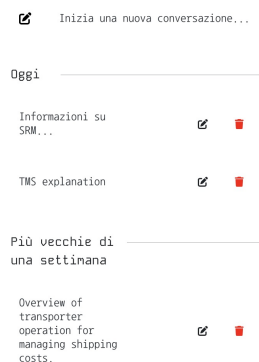


Figura 6.5: Dettaglio - barra laterale delle conversazioni

6.3.3 Caricamento dei messaggi

L’interfaccia può avere una sola conversazione **attiva** alla volta, ovvero è possibile visualizzare una sola conversazione nello spazio predefinito. Da questo segue come sia possibile selezionare una sola conversazione dalla barra alla volta.

Sebbene questa logica possa sembrare lapalissiana, è utile per capire come viene gestito il caricamento dei messaggi.

Il modulo `ConversationSideBar` possiede, come proprietà, lo stato dei messaggi e la possibilità di cambiarli (`setMessages`): nel virtual DOM, quando i messaggi vengono aggiornati, si procederà a cambiare l’interfaccia di chat; all’interno del modulo è anche presente una funzione, `handleConversationChange()` che riceve come parametro l’identificativo della conversazione attiva, ossia la nuova conversazione selezionata. Questa funzione si occupa di impostare correttamente tutte

le proprietà che dipendono dalla conversazione attiva, aggiornandole: nel caso dei messaggi viene utilizzata l'API `get_conversation_messages` che, utilizzando l'identificativo della conversazione attiva, riceve (se tutto funziona correttamente) i messaggi associati.

Per evitare di dover creare una doppia logica su due livelli diversi tra

`ConversationSideBar` e `ConversationPeek`, si è deciso di passare a quest'ultimo, come proprietà, oltre ad `activeConversationID`, `setActiveConversationID` in forma però della funzione che ne gestisce il cambiamento: ogni volta che il componente utilizzerà la funzione per impostare la nuova conversazione selezionata (se stessa), starà in realtà chiamando la funzione del componente padre che andrà a richiedere al server tutti i messaggi e gli aggiornerà automaticamente nella proprietà del, a sua volta, componente padre `ChatBuilder`.

Questa è la strategia adottata ogni volta che un componente al fondo della gerarchia opera su dati che hanno ripercussioni su tutta l'interfaccia (come in questo caso l'identificativo della conversazione): questo approccio garantisce sia buone prestazioni che una buona leggibilità del codice, rispettando le convenzioni della libreria.

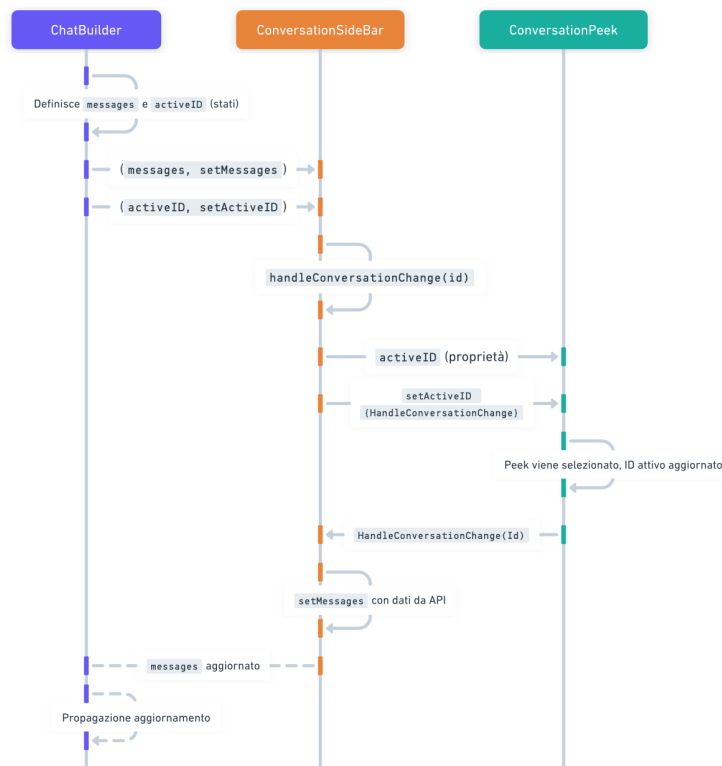


Figura 6.6: Diagramma di flusso - aggiornamento dei messaggi

Il diagramma in figura si propone di illustrare meglio quanto descritto in questo paragrafo, semplificandone la logica.

6.4 La chat

Parte centrale dell'interfaccia, la chat comprende la barra di inserimento e lo spazio in cui viene visualizzata la conversazione, composta da domande e risposte, differenziate graficamente.

Il componente che si occupa di gestire questo spazio è `ConversationBuilder`, che, similmente al componente che gestisce la barra laterale delle conversazioni, **monta** la parte di chat.

6.4.1 Barra di inserimento

Realizzata dal componente `InputBar` comprende sia la barra di input realizzata mediante `text box`, sia il pulsante per l'invio del messaggio, che appare esclusivamente quando l'utente aggiunge contenuto al messaggio, sempre in ottica di *error prevention*. Il componente implementa poi l'azione dell'invio del messaggio a seguito della pressione del **tasto invio**, in modo da permettere agli utenti più "esperti" un utilizzo più agevole della GUI. Il tasto invio è stato scelto dopo i test di usabilità dell'interfaccia: dopo averne provate altre, si è rivelata essere la più efficace, quella alla quale gli utenti rispondevano meglio.

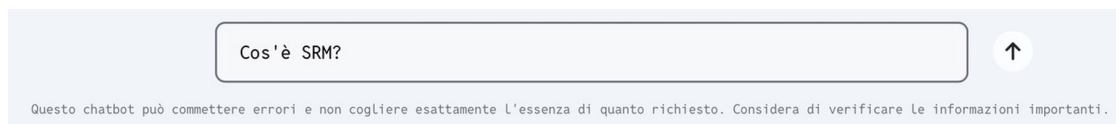


Figura 6.7: Dettaglio barra di inserimento

Anche in questo componente si è utilizzata la logica illustrata in precedenza circa il **passaggio delle funzioni** dal componente padre a quello figlio per mezzo delle proprietà: il `ConversationBuilder` implementa infatti la funzione `fetchAnswer`, che si occupa di interrogare l'assistente virtuale. La funzione viene direttamente chiamata dal componente della barra di inserimento in seguito alla richiesta di invio messaggio (per mezzo del pulsante dedicato o della pressione del tasto invio).

Infine, subito dopo la chiamata alla funzione che gestisce la richiesta, viene impostato (sempre dal componente `InputBar`, sempre con la logica delle proprietà), uno stato che indica che il sistema sta processando una richiesta.

Nel rispetto delle euristiche (4.11.2), sotto la barra di inserimento, com'è possibile notare nella figura 6.7, è stato inserito un messaggio che avvisa l'utente circa le

limitazioni dell'assistente. Si riporta in seguito il messaggio nella sua versione italiana.

Questo chatbot può commettere errori e non cogliere esattamente l'essenza di quanto richiesto. Considera di verificare le informazioni importanti.

6.4.2 I messaggi

I messaggi vengono gestiti dal **componente chiave Message**, che varia la tipologia di formattazione sulla base del tipo di messaggio che deve rappresentare.

La formattazione condizionale può generare tre diverse tipologie di messaggio:

- **Domanda:** la formattazione comprende un'icona (a rappresentare una simil "*immagine profilo*", utile per aumentare la differenza visiva tra domanda e risposta), la domanda formattata con uno specifico carattere ed una barra di divisione.
- **Risposta già generata:** ricade in questa categoria una risposta che sia stata generata in passato, della quale si conosce già il contenuto dall'inizio alla fine. Il messaggio ha una sua formattazione data da un misto di indicazioni generali (impartite naturalmente mediante **css**) su ogni possibile **TAG HTML** che il modello linguistico possa aver utilizzato, ed una parte dedicata a quella specifica riservata alla parte delle fonti, ricevute come proprietà.
- **Risposta *streamed*:** caso più complesso da gestire, è il messaggio che subisce una formattazione continua fino all'esaurimento della generazione. Strutturalmente segue la logica del punto precedente, con la differenza che tutte le operazioni di formattazione devono essere ripetute fino alla completa ricezione del messaggio.

Il componente aggiunge al messaggio anche gli elementi che verranno trattati successivamente nel paragrafo dedicato alla "**modalità avanzata**" (6.5).

Tutti i dati riguardanti il messaggio sono presenti nel parametro **message** (elemento estratto dalla lista di **messages** trattata in precedenza), che oltre al contenuto comprende la tipologia (domanda o risposta), le fonti, il prompt utilizzato per la sua generazione e altre informazioni in base alla situazione.

Si propone, di seguito, l'analisi delle tre tipologie.

Domanda

Come per i due casi che verranno trattati in seguito, si utilizza il ruolo del messaggio per selezionarne il percorso di formattazione tra domanda e risposta.

La domanda è la tipologia di **message** più semplice, composta dall'icona, dal contenuto e da un divisore.

Sono aggiunti anche il prompt con il quale la domanda è stata eventualmente riscritta e il risultato della riscrittura, che possono essere attivati nella modalità avanzata (6.5).

Risposta

Nel caso di risposta già generata, il componente deve restituire il messaggio formattato e la possibilità di vedere le fonti, anch'esse rappresentate in modo opportuno.

La ricezione del messaggio viene gestita a livello superiore dalla funzione `fetchAnswer`, che di per se ha già una sua rappresentazione in HTML, come ampiamente discusso. Prima di poter eseguire il rendering del componente è ancora necessario estrarre le fonti utilizzate, che appaiono come citazione al fondo della risposta ricevuta, come da esempio.

Document [0] , Document [3]

Per gestire l'estrazione delle fonti si è predisposta la funzione `extract_documents()`, che riceve come parametro la lista di tutti i **Document** associati alla domanda alla quale la risposta corrisponde.

La funzione si occupa anzitutto dell'estrazione, per mezzo di **regex** (espressioni regolari), delle citazioni aggiunte dal modello, che vengono prontamente eliminate. Nel caso siano state incluse in **TAG HTML** (comportamento potenziale), anche questi vengono rimossi.

Dopo l'operazione di pulizia si procede all'identificazione della tipologia fonte: nella versione finale della tesi, per esigenze aziendali, si è deciso di rimuovere il supporto dei video (lasciando solamente una base documentale composta da **pdf**), ma l'interfaccia supporta anche la formattazione dei video.

La rappresentazione delle fonti è stata progettata per essere minimalista ed intuitiva, fornendo al contempo tutte le informazioni più rilevanti:

- **fonti pdf**: riquadro con un'icona che richiama un documento, con l'aggiunta del **titolo** del documento aziendale e, se disponibile, anche del **capitolo** nel quale l'informazione con la quale è stata costruita la domanda è stato trovato. Interagendo con il riquadro (*cliccandoci sopra*) viene aperta una nuova pagina del browser che rimanda al documento.
- **fonti video**: frame che consente la riproduzione di un video (al momento della progettazione, a scopo dimostrativo, si è deciso di optare per un frame *YouTube*), con l'elenco dei **minuti** ai quali le fonti si trovano. Vi è infatti la

possibilità, come d'altronde per la documentazione pdf, che più fonti arrivino dallo stesso documento. A differenza della tipologia pdf, allo stato attuale il video viene visualizzato direttamente nella chat. La scelta di utilizzare un frame youtube a scopo di demo deriva dall'impossibilità al momento del progetto di accedere ai video utilizzati per la base di conoscenza dell'assistente, a causa di politiche aziendali che non vengono qui trattate.

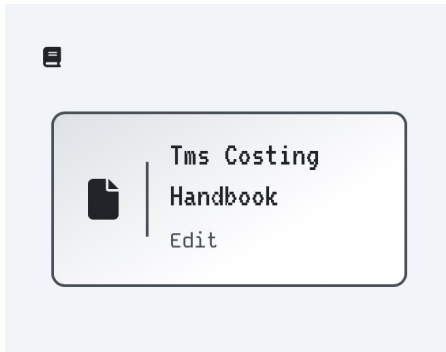


Figura 6.8: Dettaglio fonte in formato pdf

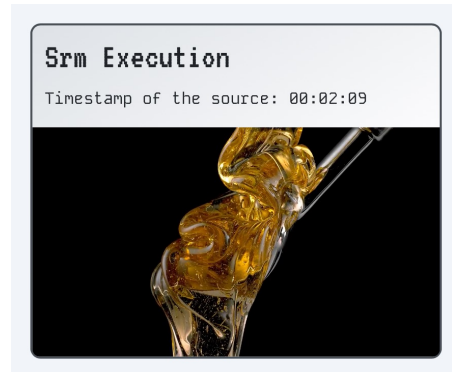


Figura 6.9: Dettaglio fonte in formato video

Per ridurre la quantità di elementi che occupano l'interfaccia, si è deciso di non mostrare tutte le fonti direttamente, bensì prevedere un'icona con il simbolo dei documenti che, una volta attivata, le mostri tutte al di sotto del messaggio. L'icona che mostra i documenti si può notare nell'immagine a sinistra.

Risposta streamed

La risposta in formato streamed è strutturalmente identica a quella vista nella sezione precedente, ma vanno presi degli accorgimenti per la gestione del contenuto generato in tempo reale.

Come anticipato, il componente `ConversationBuilder` si occupa della richiesta al server, ed è lui che gestisce il flusso all'interno dello `stato message`: ogni volta che nuovo contenuto (inteso come attributo di `message`, che nell'implementazione è una struttura con più dati) viene ricevuto, lo stato si aggiorna e, per la logica alla base di React, il componente `Message` deve aggiornarsi alle differenze.

La funzione con la quale il `ConversationBuilder` si occupa di realizzare quanto detto è `fetchAnswer()`, di cui si prova a dare una descrizione in quanto una delle funzioni più importanti del client.

Il primo passo della funzione consiste nel verificare se la domanda fornita dall'utente non è vuota. Se la domanda è presente e il bot non è già impegnato

nell'elaborazione di una richiesta, viene attivato un flag per impedire l'elaborazione simultanea di più richieste.

Successivamente, la funzione crea copie degli stati attuali della domanda e della risposta (`newQuestion` e `newAnswer`), oltre a inizializzare le variabili necessarie per tracciare e gestire l'interazione con il bot. Vengono inoltre introdotte variabili Booleane come `isAnswerGenerated` e `isConversationNamed`, che servono per monitorare l'avanzamento della risposta e la denominazione della conversazione.

Si definisce una funzione interna, `handleMessage(data)`, che si occupa di interpretare i dati ricevuti dal bot utilizzando espressioni regolari per estrarre informazioni relative alla conversazione, alla domanda e alla risposta: questa funzione verrà utilizzata dall'API `message_with_bot()`. Dopo aver aperto la connessione con il server, infatti, l'API ha bisogno di due funzioni (`onMessageCallback`, `onDoneCallback`) che incapsolino le logiche rispettivamente per gestire i "pezzetti" di messaggio in arrivo e l'operazione finale quando si chiude la connessione: `handleMessage()` è il primo parametro.

In questa funzione vengono definite, per mezzo di espressioni regolari, i filtri per intercettare le informazioni in arrivo dal server che non siano il contenuto della risposta, come l'identificativo della domanda, quello della risposta e della conversazione (trattati nel capitolo 5).

Questa funzione si occupa sia di aggiornare l'elemento di domanda, sia quello di risposta al quale aggiungere i pezzi di contenuto man a mano che questi arrivano: tutto questo viene fatto appunto filtrando in base a dei *marker*, delle parole chiave, concordati con il server (nella documentazione).

Nel caso in cui il server segnali la fine della risposta con specifici marker (`--end-of-answer--` o `--one-shot--`), la funzione aggiorna lo stato della risposta e ne registra la conclusione. Se vengono riscontrati errori nel recupero di dati o documenti, questi vengono gestiti senza interrompere il flusso principale dell'applicazione. A questo punto, la risposta viene progressivamente aggiornata e mostrata all'utente.

La funzione `handleDone(error)` (il secondo parametro dell'API discusso sopra) viene invocata al termine del processo di generazione della risposta. In questa fase, si tenta di recuperare ulteriori informazioni come i prompt associati alla domanda e alla risposta che viene qui integrata con eventuali documenti associati tramite la funzione asincrona `get_documents()`, che utilizza un omonima API per, una volta noto l'ID della risposta, recuperare quelli associati. Se la risposta è stata generata e la conversazione non ha ancora un nome, il sistema assegna automaticamente un titolo alla conversazione, aggiornando anche la lista dei messaggi, sfruttando un'altra API (`name_conversation()`).

Infine viene richiamata l'API `message_with_bot()`, che esegue tutto quanto definito fino a qui.

Da un punto di vista visivo, il componente **Message** implementa due funzionalità per aiutare l'utente a capire che la risposta sta ancora venendo generata e a rimanere visivamente connesso:

- **Dot:** durante tutta la fase di generazione della risposta è presente un cerchio pulsante in coda all'ultimo segmento di contenuto visualizzato, che aiuta a tener traccia della generazione, spostandosi. Al termine della generazione della risposta, questo elemento scompare graziosamente, segnalando la fine del processo.
- **Auto-scroll:** per mantenere visivamente contatto con la generazione del testo, si è implementato lo scorrimento automatico della risposta.

6.5 La modalità avanzata

Questa modalità è stata integrata per agevolare il test delle funzionalità principali del chatbot, ovvero l'aggiunta degli esempi e la verifica di come questi vadano ad impattare sulla costruzione di domande e risposte.

Nella figura 6.10 è possibile notare il dettaglio delle funzionalità aggiuntive dell'interfaccia, tra le quali la **modalità scura** (prima icona), il **cambio lingua** (terza icona) e la **modalità avanzata** (seconda icona, visualizzazione quando attiva).

La modalità avanzata, quando attiva, modifica alcuni elementi dell'interfaccia. Come prima cosa, visibile sempre nell'immagine, compaiono le icone per aggiungere un esempio di riscrittura di domanda e quello, simile, per la risposta.

Quando si selezionano queste opzioni compare un'interfaccia che consente di specificare nuovi esempi, nella quale sono stati implementati sia controlli a livello di invio, come il controllo del contenuto dei campi prima dell'invio della richiesta, sia riscontri a livello di ricezione, come nel caso in cui l'esempio sia già presente oppure venga aggiunto con successo.

L'altra modifica all'interfaccia è la comparsa delle icone che consentano di vedere i prompt di generazione delle domande e delle risposte, e quella per visualizzare l'eventuale riformulazione della domanda (nel caso questa non sia avvenuta, l'icona non sarà presente).

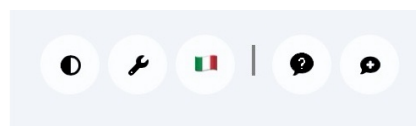


Figura 6.10: Dettaglio - modalità avanzata ed impostazioni di interfaccia



Aggiungi un nuovo esempio di domanda

Domanda:

Risposta dell'assistente:

Ultima domanda:

Ultima domanda, come la si vuole riscrivere:

Chiudi Invia l'esempio

Figura 6.11: Aggiunta esempio di domanda

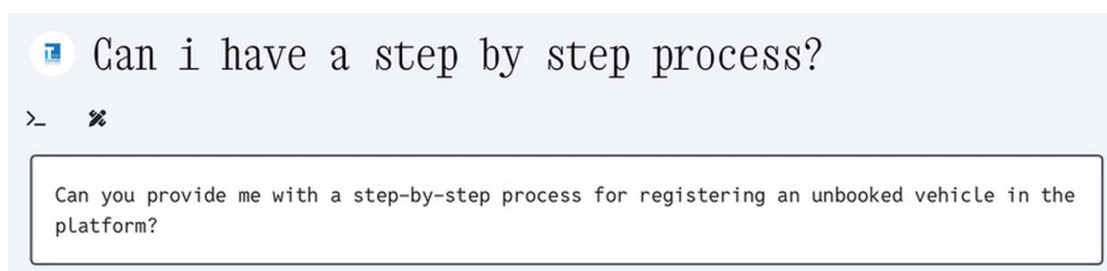


Figura 6.12: Dettaglio - Visualizzazione riscrittura della domanda nella modalità avanzata

Dall'immagine 6.12 è possibile notare sia l'icona per visualizzare la domanda riscritta (la seconda), sia quale sia stata la modifica.

La prima icona permette di visualizzare il prompt della richiesta al seguito della quale la domanda è stata riscritta.

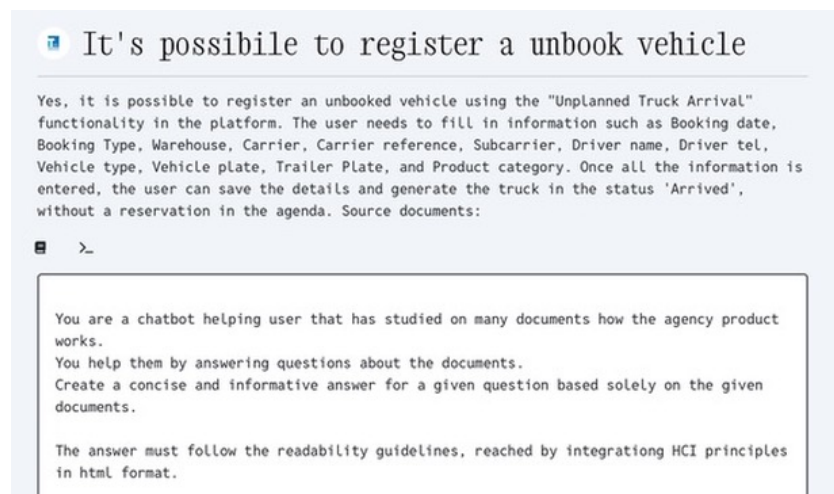


Figura 6.13: Dettaglio - Prompt utilizzato per generare la risposta.

Infine, la figura 6.13 illustra, in questo caso per una risposta, il relativo prompt di generazione. Per aumentarne la leggibilità sono state fatte delle modifiche che ne permettono la giusta formattazione e la colorazione delle parole chiave per gli esempi, quelle che identificano i ruoli.

Purtroppo le immagini, quando riportano casi d'uso reali, sono in lingua inglese, unica lingua attualmente disponibile per la comunicazione con l'assistente.

Tutte queste funzionalità e l'accesso alle fonti potrebbe essere potenzialmente limitato ad una specifica categoria di utenti in futuro, implementando un sistema di sessioni ed accessi all'accesso della GUI. Tuttavia questo trascende il progetto di tesi e non era nemmeno richiesto, in questa fase, dall'azienda.

6.6 Funzionalità aggiuntive

Si conclude il capitolo con una breve descrizione di tutte le componenti aggiuntive della GUI.

Multilinguismo: implementato con la libreria React `react-i18next`, utilizza una struttura di traduzioni organizzata in file JSON all'interno della cartella `locales/`, suddivisa nelle lingue attualmente supportate (ovvero italiano ed inglese). L'oggetto `i18n` gestisce il cambio dinamico della lingua e aggiorna automaticamente l'interfaccia in base alle traduzioni presenti nei file.

Tooltip: questi elementi aumentano l'usabilità dell'interfaccia, consentendo di ottenere la descrizione di un elemento semplicemente "passandoci sopra con il mouse". La funzionalità è stata implementata mediante la funzionalità `title`, nativa HTML.

Modalità scura: predisposta a livello di progettazione, per ogni elemento definito si sono previste due **palette** di colori differenti, una che segue i toni chiari ed una i toni scuri. Per uniformità e leggibilità, l'implementazione ha utilizzato il concetto di **contesto** (**Context**) di React. Il contesto **DarkThemeContext** viene creato con il valore predefinito `false`, che indica la modalità chiara. Successivamente, attraverso il Provider, il valore `isDarkMode` (*di contesto*) viene passato all'intera applicazione, permettendo ai componenti di accedere dinamicamente alla modalità scura o chiara in base allo stato corrente, senza necessità di passare esplicitamente le proprietà tra i componenti.

Gestione degli errori: Gli errori sono gestiti con specifici elementi che compaiono “*al di sopra*” dell'interfaccia, sfruttando allo stesso tempo un effetto di **blur** (implementato con la stessa logica di **Context** utilizzata per la modalità scura). Gli elementi possono essere facilmente chiusi con un semplice pulsante. I messaggi di errore sono filtrati in modo da mostrarne all'utente una versione semplificata: tutte le volte che il messaggio è di tipo tecnico, con codici di errore dopo una chiamata, ad esempio, questo viene stampato a **console**, mentre l'utente viene avvisato dell'errore utilizzando una terminologia non specifica ma che ne chiarisca la natura.

Sidebar personalizzata: in tutti i posti in cui sia possibile un'azione di scorrimento si è utilizzata una **Sidebar** personalizzata, adattandone movimenti, dimensioni e colori all'interfaccia.

Gli Spinner: durante tutte le operazioni che possono occupare del tempo, come il passaggio da una conversazione ad un'altra, l'intera interfaccia viene offuscata con il filtro di *blur* e viene utilizzato uno **Spinner** react per animare un componente a movimento circolare che indica all'utente il classico caricamento.

Messaggio iniziale: In fede alle euristiche trattate nel paragrafo 4.11.2, all'apertura dell'interfaccia si è prevista la visualizzazione di un messaggio che avvisa, più estesamente di quanto non venga fatto sotto la barra di inserimento, dello scopo dell'assistente e delle sue limitazioni. Nell'immagine è possibile visualizzarlo.

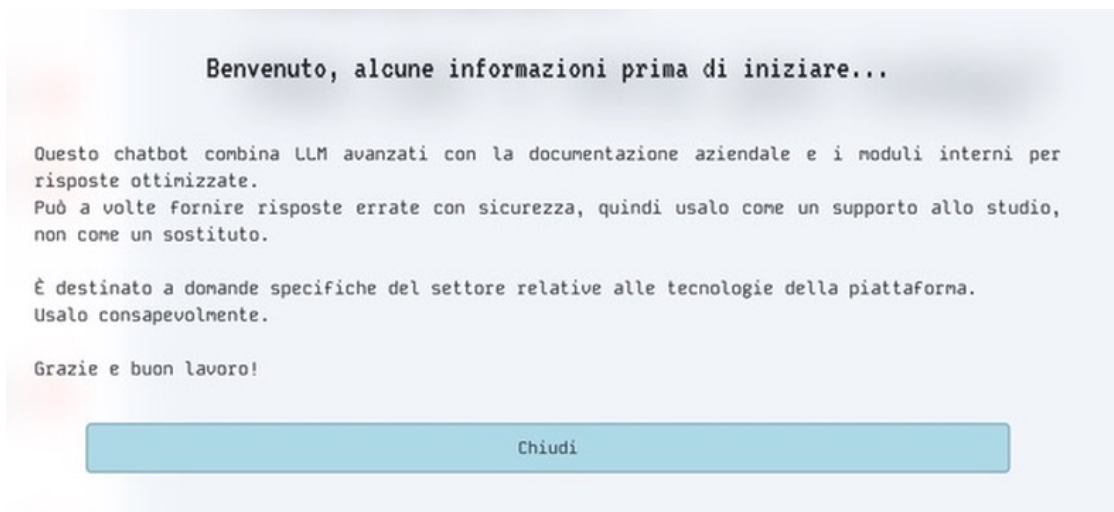


Figura 6.14: Messaggio introduttivo della GUI

Capitolo 7

Test e risultati

Questo capitolo si focalizza sull'analisi dell'effettiva efficacia di quanto discusso fino a questo punto. Il processo di valutazione si concentrerà su due aspetti differenti: valutare quanto efficace sia l'assistente in se, quanto i moduli implementati effettivamente vadano a “potenziare” l'esperienza dell'utente, e, secondo, valutare l'interfaccia.

L'obbiettivo di una valutazione di questo tipo è testare il sistema nel suo *complesso*, la sua **usabilità** e le sue **funzionalità**, in accordo con le premesse, i **requisiti**, posti in fase di progettazione. Questa tipologia di test prende il nome di **usability testing**, test dell'usabilità, di cui si propone un'analisi nel prossimo paragrafo, adattandone la descrizione a quanto fatto, nella pratica, nel lavoro di tesi. Parte di quanto descritto si basa sul materiale trattato nel corso di Human Computer Interaction, nella parte specifica di test [29].

7.1 Test dell'usabilità

Gli obbiettivi di questi test sono molteplici, focalizzati sull'identificare eventuali problemi nel *design* dell'applicazione e nella possibilità di individuare possibili miglioramenti. Osservare come gli utenti utilizzano l'applicativo aumenta la comprensione dell'utilizzo di quest'ultimo, sulla base di comportamenti ed esigenze.

La definizione di **usability testing** viene fornita dal *Nielsen-Norman Group*:

In una sessione di test di usabilità, un ricercatore (chiamato “facilitatore” o “moderatore”) chiede a un partecipante di eseguire dei compiti (**task**), di solito utilizzando una o più interfacce utente specifiche. Mentre il partecipante completa ogni compito, il ricercatore osserva il comportamento del partecipante e ascolta i suoi feedback.

I test si realizzano in tre fasi:

1. **Pianificazione:** si individuano i **partecipanti**, si definisce **cosa** dovrà essere testato e con quali **modalità**. Viene redatto uno **script** riassuntivo con una parte nella quale si discute circa le scelte prese (tipologia di utente, modalità di test, ecc..), sia si imposta una serie di **task**, ovvero di compiti, che l'utente dovrà eseguire per completare il test.
2. **Esecuzione:** si svolgono più sessioni con un **singolo** partecipante alla volta, durante le quali si collezionano dati **direttamente** (mediante feed-back) ed **indirettamente** (si appuntano eventuali comportamenti particolari, difficoltà, preferenze)
3. **Analisi:** Si estraggono informazioni dai dati colletti, **qualitative** e **quantitative**.

7.1.1 Pianificazione

Si è partiti dalla selezione del numero e della tipologia di partecipanti: l'utente di riferimento è un dipendente neo-assunto che necessita di trovare **informazioni specifiche** circa le soluzioni offerte dal principale prodotto aziendale, che non riguardano loro specifiche aree di competenza (delle quali ha, di conseguenza, una **conoscenza limitata**, di superficie).

Più in generale, fanno parte della **target population** tutti gli utenti che *non hanno delle conoscenze specifiche* circa il principale prodotto dell'azienda ed i suoi derivati, e che per un motivo piuttosto che per un altro, abbiano necessità di approfondirne gli aspetti.

Viene poi definita una seconda categoria minoritaria di utenti, definiti **esperti**, rappresentata da dipendenti con ruoli specifici su un determinato sottoprodotto: a questi utenti si propone un test speciale, con soli compiti inerenti ad un argomento non conosciuto. Sebbene si sia riusciti a coinvolgere solo un utente, all'interno dell'azienda, di questa tipologia, il suo contributo si è rilevato **fondamentale**, in quanto una maggiore conoscenza del contesto aziendale, unito ad una generale vasta esperienza lavorativa, ha fornito degli spunti interessanti su possibili **sviluppi futuri** della tecnologia.

L'età dei partecipanti è compresa tra i 18 ed i 45 anni.

Gli utenti, indipendentemente dalla tipologia, devono poi rispettare le seguenti caratteristiche:

- Elevata familiarità con l'utilizzo di strumenti informatici.
- Esperienza minima con la terminologia aziendale, con le sigle utilizzate, con il contesto nel quale l'azienda opera.

- Esperienza minima con l'utilizzo di strumenti di intelligenza artificiale, specificatamente assistenti (averne in mente il concetto).
- Esperienza nella consultazione di documentazione di tipo manualistico.
- Buona conoscenza della lingua inglese, caratteristica fondamentale per porre le domande e leggere risposte e documenti (si ricorda che l'assistente funziona, attualmente, solo il lingua inglese).

I task che verranno proposti agli utenti comprendono le due soluzioni principali offerte dall'azienda (TMS e SRM, uno per la gestione dei trasporti, una per la logistica interna). Nel caso dell'utente esperto, la selezione dei task che verranno eseguiti dovrà tenere conto di queste differenze, in modo da testare effettivamente l'utilizzo del sistema circa argomenti poco conosciuti. Per i partecipanti che non hanno competenze specialistiche verrà scelta una combinazione di task di entrambe le soluzioni.

7.1.2 Modalità di svolgimento

La prova consiste nella risoluzione di un certo numero di task (da 5 a 10, a seconda del tempo a disposizione del candidato, dei task settoriali disponibili e della velocità di svolgimento) che devono essere eseguiti utilizzando due diversi sistemi:

- Ricerca delle informazioni utilizzando ***esclusivamente*** la **documentazione aziendale**.
- Ricerca delle informazioni mediante l'**utilizzo dell'assistente virtuale** oggetto di questa tesi, sfruttando l'interfaccia grafica associata.

I task rappresentano dei gli obiettivi che gli utenti devono perseguire per completare il test: non sono delle richieste dirette, bensì indirette inserite in uno scenario (in seguito si faranno alcuni esempi), che imposta il contesto dell'azione che l'utente si presta ad eseguire (tenta).

L'utente svolgerà il test su un solo laptop, munito di mouse e con la possibilità di utilizzare un trackpad (in modo da non vincolare l'utente ad uno dei due sistemi). Avrà a disposizione tutta la documentazione in una cartella, ed il chatbot aperto in una finestra su un browser.

Verrà chiesto di eseguire il task corrente a volte prima utilizzando la documentazione (senza limitazioni di interazione, ad esempio utilizzando la barra di ricerca) a volte prima il chatbot, alternandosi in modo da non dare la precedenza ad un sistema piuttosto che un altro. Per ogni task eseguito mediante il chatbot, l'utente è libero di scegliere se suddividere il task in sotto-task e quindi fare più di una domanda o cercare di ottenere una risposta che comprenda tutte le richieste del task.

In entrambi i casi viene misurato il tempo che intercorre tra l’inizio e la fine del task. Alla fine di ogni task verrà richiesto un **feedback**, un riscontro, sulla qualità della risposta ottenuta con entrambi i sistemi.

Il test si propone quindi di misurare effettivamente le **differenze di prestazioni** tra le due soluzioni disponibili nella ricerca delle informazioni, e il **grado di dettaglio** e **soddisfazione** di quanto trovato.

Il test si svolgerà in modalità **Cooperative Evaluation**, modalità di testing dell’usabilità in cui l’utente e l’esaminatore **collaborano attivamente** durante l’intero processo di test. A differenza dei metodi tradizionali in cui il tester osserva in modo passivo, nella cooperative evaluation c’è un’interazione continua tra le due parti, favorendo il dialogo e lo scambio di informazioni.

Per misurare il successo dei due metodi, per ogni test, verrà stilata una tabella di report strutturata come segue:

Tabella 7.1: Esempio tabella di report - test dell’usabilità

Task #	Livello di successo	Tempo impiegato per il task	Osservazioni
1	2	2 : 43	Quisque ullamcorper placerat ipsum.
2	1	1 : 25	Morbi vel justo vitae lacus tincidunt ultrices.
...
8	1	0 : 46	x

Dove la metrica “**livello di successo**” può assumere 4 valori, rispettivamente:

1. **successo**: l’utente ha completato il task senza problemi, ha trovato l’informazione ed è riuscito ad utilizzare (sostanzialmente, l’ha capita, è quindi stata presentata bene).
2. **successo con problemi minori**: l’utente ha completato il task trovando l’informazione, ma è presente un leggero livello di confusione, sono state necessarie più iterazioni per capire quanto necessario per completare il task.
3. **successo con problematiche maggiori**: l’utente ha completato parzialmente il task, non riuscendo a padroneggiare l’argomento in questione.
4. **fallimento**: l’utente non è riuscito a completare il task, le informazioni non sono state trovate.

Per capire quale punteggio assegnare (specialmente tra il 2 ed il 3) verrà chiesto all'utente, dopo ogni task, di esprimere il proprio parere sul grado di confidenzialità raggiunto sull'argomento appena cercato. Eventuali osservazioni verranno annotate nella rispettiva casella.

Dopo aver completato la lista dei task verrà ancora richiesto di compilare il questionario **SUS**, per l'usabilità dell'interfaccia.

Il questionario SUS (*System Usability Scale*) è uno strumento semplice e standardizzato, utilizzato per valutare l'usabilità di un sistema, prodotto o servizio: consiste in un questionario composto da dieci affermazioni a cui gli utenti rispondono con una scala da uno a cinque punti, che varia da “*fortemente in disaccordo*” a “*fortemente d'accordo*”. Le domande sono progettate per catturare le percezioni generali dell'utente sull'**usabilità**, compresi aspetti come la **facilità d'uso**, la **complessità** e la **fiducia** nell'utilizzo del sistema. I risultati del SUS forniscono un punteggio complessivo che va da 0 a 100, con punteggi più alti indicativi di una migliore usabilità.

Si vuole anche misurare la bontà delle risposte del chatbot ed una loro eventuale “ricchezza”: ogni volta che la risposta sarà particolarmente ben costruita, ovvero quando oltre a fornire le informazioni corrette presenti nella documentazione, sia capace o meno di rielaborare e presentare le informazioni in un formato più chiaro rispetto a quelle disponibili nella documentazione, verranno appuntati i risultati.

7.1.3 I task

I task sono stati scritti su consiglio di **esperti** del prodotto, pensati per rappresentare richieste mirate, rispettivamente ai moduli disponibili.

Non viene qui proposta l'intera lista dei task, ma solo tre esempi, uno per ogni tipologia, ovvero **TMS** (la suite per la gestione dei trasporti), l'**SRM** (quella per la gestione dell'approvvigionamento delle risorse), ed infine l'interfaccia utente. Gli esempi hanno come obiettivo quello di aiutare a comprendere come sono stati condotti i test, quale sia stato il tono.

TMS: Ti potresti trovare nella circostanza di dover apportare delle personalizzazioni mirate, per un cliente, circa un modulo specifico del TMS, quello di costing. Devi capire qual'è l'obiettivo principale del modulo e quali sono le operazioni principali che questo può fare.

SRM: Potresti incontrare, in generale, dei widgets durante le operazioni sull'interfaccia. Potrebbe capitare di dover intervenire su alcuni in particolare, ad esempio su quelli cosiddetti di *call-of*, ma devi conoscere quali sono e di cosa si “occupano”.

Interfaccia: Potrebbe nascere la necessità di cambiare i colori dell'interfaccia per aumentarne la leggibilità. Agisci in questo senso.

7.1.4 Esecuzione

Prima dell'esecuzione del test è stato chiesto ad ogni utente di compilare un modulo di consenso, nel quale viene descritto l'obiettivo del test e si avvisa il partecipante della possibilità di ritirarsi in qualsiasi momento. L'utente viene anche **avvisato** come qualsiasi problema che nasca dall'utilizzo dell'applicativo **non sia** da attribuirsi ad una sua incapacità, ma da una **mancanza** da parte della tecnologia utilizzata. Questo approccio è necessario per mantenere l'utente testato il più possibile **a suo agio**, evidenziando quanto appena detto lungo tutta la valutazione, ripetendolo eventualmente nel caso l'utente abbia delle difficoltà.

I test sono stati eseguiti senza particolari problematiche, soltanto a volte si è rivelato necessario diminuire il numero di compiti a causa della mancanza di tempo, ma per la maggior parte degli utenti si sono riusciti ad eseguire, tra relativi alla ricerca di informazioni aziendali ed interfaccia, circa 10 task a persona.

7.2 Analisi dei risultati

I risultati dei test hanno sono stati molteplici. Si è trovato anzitutto riscontro dei miglioramenti introdotti dal sistema di pipeline, oggetto della tesi, ma anche dei principali vantaggi e svantaggi dell'utilizzo della documentazione e dell'assistente nel processo di ricerca delle informazioni, di risoluzione dei task.

7.2.1 Riscontri pratici dei moduli introdotti

Si inizia l'analisi dimostrando come, quando i task sono stati svolti utilizzando l'assistente, gli aggiornamenti introdotti a livello di moduli della pipeline si sono rivelati molto utili per potenziare l'esperienza utente. Si propone di seguito una serie di esempi presi dalle conversazioni di test condotte dagli utenti, nei quali sono intervenuti i moduli di cui si è largamente discusso.

QuestionBuilder

In un task nel quale l'utente doveva cercare di capire come potevano essere gestiti i costi di trasporto legati alle spedizioni, il modulo è intervenuto per riscrivere l'input: nel caso in figura 7.1 la riscrittura ha migliorato una domanda di per se non pessima ma andando ad arricchirla con il contesto (recuperato dallo **Snippet** di conversazione) e richiedendo un formato di risposta che includesse la spiegazione delle principali strategie per la gestione dei costi.

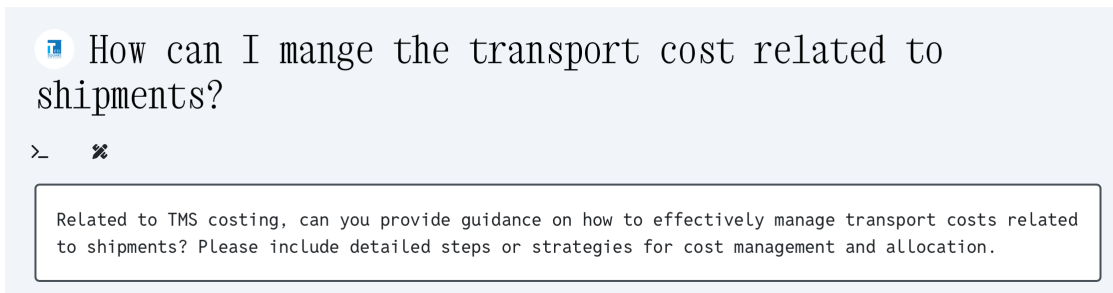


Figura 7.1: Esempio di riscrittura ad opera del QuestionBuilder

Nel secondo caso invece il modulo è intervenuto per correggere una domanda che era stata posta male, in modo poco chiaro e che avrebbe sicuramente portato ad una risposta meno efficace. Si noti come il modello, nonostante avesse a disposizione in entrambi i casi lo stesso set di esempi per la riscrittura, nel secondo caso si sia limitato ad una ristrutturazione della domanda per migliorarne il formato, ma non abbia aggiunto ulteriori richieste di “formattazione”: questo è dovuto al modello utilizzato (`gpt-3.5-turbo`), più economico ma meno efficace a gestire operazioni più complesse (nel caso dell’esempio avrebbe dovuto aggiungere anche le indicazioni di formattazione). L’utilizzo di modelli più potenti è una soluzione, trattata nel capitolo 8, ma si vuole **sottolineare** come l’intervento dei moduli (in questo caso il `QuestionBuilder`) permetta di raggiungere comunque un buon livello di ottimizzazione delle domande e, vedremo, delle risposte, utilizzando anche i modelli più economici (si consideri che, al tempo del progetto di tesi, vi è una differenza di prezzo tra `gpt-3.5-turbo` e `gpt-4o` di circa 20 volte).



Figura 7.2: Un altro esempio di riscrittura di una domanda mal strutturata

QuestionScanner

Il `QuestionScanner` è intervenuto sia per dare opzioni di “ulteriore esplorazione” nel caso di argomenti molto generali, oppure guidandoli verso i moduli corretti nel caso di domande.

Explain SRM

Supplier Relationship Management (SRM) is a crucial component of the TESI SRM platform, designed to streamline the entire "Source-to-Pay" cycle for purchasing direct and indirect goods and services. It enables comprehensive supplier management and operational activities related to sourcing, procurement, and execution.

Key Aspects of SRM:

- **Sourcing Process:** Involves activities prior to awarding contracts to suppliers.
- **Procurement Process:** Manages order placement, logistics, and administrative cycles with suppliers.
- **Supplier Management:** Enhances collaboration and visibility with trading partners.

Benefits of SRM:

- **Efficiency:** Improves supply chain efficiency and performance.
- **Compliance:** Ensures adherence to regulations and standards.
- **Cost Savings:** Optimizes processes for significant cost savings.

SRM is essential for digitalizing processes, enhancing collaboration, and effectively managing supplier relationships within the TESI SRM platform.

If you would like a more detailed explanation tailored to a specific aspect of SRM, please let me know which solution you are interested in:

- Srm Procurement Handbook
- Srm Sourcing Handbook
- Srm Execution Handbook
- Srm Replenishment Handbook

Feel free to choose a specific context for a more in-depth understanding.

Figura 7.3: Elenco di possibili soluzioni da esplorare dopo domanda generica

Nella figura 7.3 viene appunto mostrato il primo esempio quando, al seguito di una domanda descrittiva di una delle tre soluzioni principali messe a disposizione dall'azienda, vengono proposte diverse sotto-soluzioni con le quali continuare, se si vuole, il dialogo.

Some information about Transport costs related to shipments

I'm very sorry, based on my current knowledge, I can't find an answer to your question. I apologize for the inconvenience. If you provide more context or details about your question, I can try to give you a better answer. Here are some options based on the available solutions:


1. Tms Costing Handbook
2. Tms Planning Handbook

Figura 7.4: Intervento dello scanner su una domanda senza contesto

Per rimanere invece sugli stessi esempi fatti nella sezione precedente, in figura 7.4 viene mostrato come il `QuestionScanner` intervenga (in concerto con l'`AnswerBuilder`) nel caso in cui vengano richieste informazioni sui costi di trasporto relativi alle spedizioni senza specificare un contesto, ovvero il modulo *TMS Planning*: queste informazioni infatti si trovano in più soluzioni, e, a differenza dell'esempio visto nella sezione precedente, qui l'utente non aveva specificato nello `Snippet`, ossia nella parte precedente di conversazione, il modulo. Una volta ricevuta la lista con le opzioni, è stato immediato per l'utente specificare la prima, ed ottenere la risposta corretta.

AnswerBuilder

Questo modulo interviene per la formattazione di ogni risposta, com'è già stato anche mostrato nella figura 7.3. In figura 7.5 si vuole proporre un altro esempio in cui l'`AnswerBuilder` si è rivelato molto utile: uno dei task richiedeva infatti di recuperare alcune informazioni che permettessero di capire quali fossero dei widgets presenti nell'interfaccia di uno dei prodotti, e nella documentazione queste informazioni erano abbastanza disperse, in più proprio l'elenco dei due widgets non era semplice da trovare. L'assistente, opportunamente istruito sul come costruire una tabella appunto dal modulo che dà il titolo a questa sezione, ha saputo condensare tutte le informazioni in un formato che è stato apprezzato da tutti gli utenti meno esperti, proprio per la sua immediatezza.

 Can I get an overview of the call-off widgets?

Overview of Call-off Widgets

Widget	Role	Function	Redirect	Preset Filters	Notes
Number of Schedules to Confirm	Planner	Counts the number of call-off rows in Publishing status and visible to the user.	Call-off Rows Dashboard	schedules with row status = "Published"	No filters are set on date from, date to. Preset filters cannot be changed.
	Supplier	Counts the number of call-off rows in Publishing status and addressed to the supplier user.	Call-off Rows Dashboard	schedules with row status = "Published" and supplier = the vendor accessing the system	
Number of Schedules in Negotiation	Planner	Counts the number of call-off rows in Negotiation status, in charge to Internal profile and visible to the user.	Call-off Rows Dashboard	schedules with row status = "Negotiation", in charge to = "internal user"	No filters are set on date from, date to. Preset filters on status cannot be changed.
	Supplier	Counts the number of call-off rows in Negotiation status, in charge to Supplier profile and addressed to the supplier user.	Call-off Rows Dashboard	schedules with row status = "Negotiation", in charge to = "supplier" and supplier = the vendor accessing the system	

Figura 7.5: Elenco di possibili soluzioni da esplorare dopo domanda generica

7.2.2 Visualizzazione dei risultati

Da un punto di vista più generale i test evidenziano anzitutto una sostanziale differenza nel **tempo** impiegato per risolvere i task: l'assistente, tolta un iniziale fase di "apprendimento", ha permesso in generale di risolvere i task molto più velocemente. I documenti, per contro, si sono rilevati più lenti da consultare, sia da parte degli utenti target del test, senza conoscenze specifiche (per ogni task su un prodotto esistono più documenti ognuno per un sotto-prodotto), sia per l'utente esperto, che aveva però dei compiti che non comprendevano la soluzione aziendale sulla quale era solito lavorare.

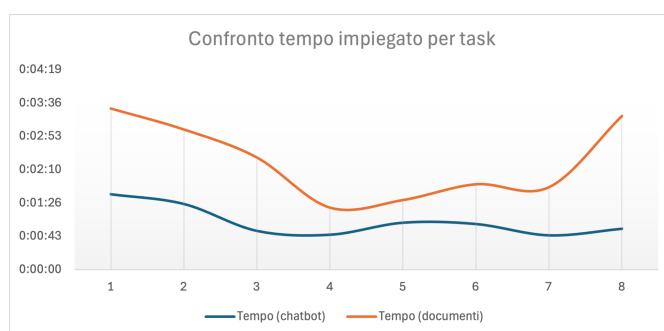


Figura 7.6: Tempo per task - utenti

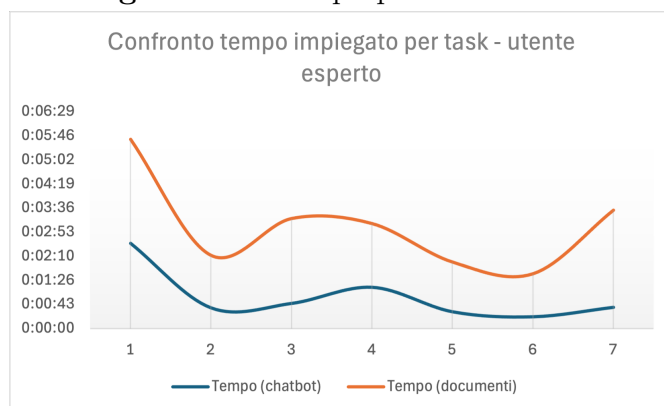


Figura 7.7: Tempo per task - utente esperto

Per quanto riguarda invece i livelli di successo delle due modalità, nessuna delle due ha prodotto valori critici, e nessuna delle due ha nettamente superato l'altra in termini assoluti, come da figura 7.8

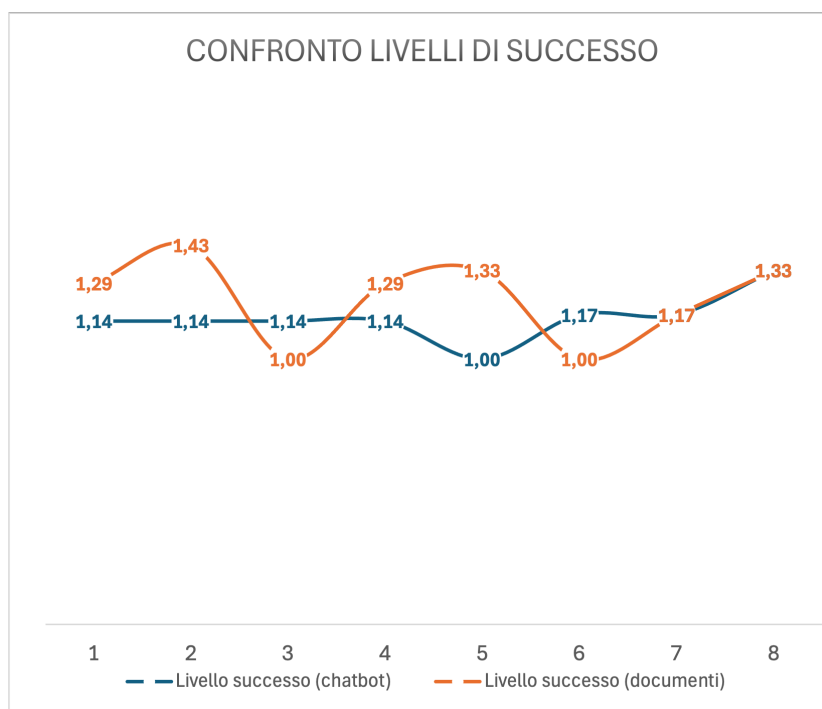


Figura 7.8: Livelli di successo dei test condotti utilizzando il chatbot e la documentazione.

La causa del risultato leggermente più negativo della documentazione è da ricercarsi nel fatto che spesso le informazioni utili si sono rilevate disperse, non immediatamente accessibili e strutturate in un modo non agevole per utenti che gli affrontano per la prima volta (molti elenchi, informazioni stringate e non ben contestualizzate, ad esempio). Per contro, nei rari casi (è successo 2 volte) nei quali l'utente per qualche motivo avesse già lavorato in un contesto nel quale aveva avuto l'esigenza di utilizzare le informazioni del compito in questione, è stata preferita la presentazione dei documenti, più immediata.

Un altro **grande vantaggio** della documentazione rispetto all'assistente è rappresentata dalla presenza di immagini, specialmente di quelle a corredo di processi: in questo caso sono molto utili in quanto danno immediatamente un riscontro se le operazioni che l'utente potrebbe trovarsi a dover compiere sono quelle corrette, fornendo quindi elementi in più.

Tra i vantaggi dell'utilizzo dell'assistente sono comunque molteplici.

- **Immediatezza nella ricerca** delle informazioni, a patto di specificare bene la richiesta.

- Una **formattazione estesa**, didattica, che contestualizza meglio i concetti per l'utente che non abbia mai affrontato la specifica informazione e da generalmente una spiegazione formulata meglio, più comprensibile.
- Ottimale la presentazione **passo passo**, spesso non presente nella documentazione, specialmente nella descrizione dei processi.
- La possibilità di **dividere i compiti in sotto-compiti**, di suddividere una domanda in più domande, che l'assistente è capace di gestire.

Per quanto riguarda l'interfaccia, di seguito si presentano i risultati dei task relativi, che a livello di successo hanno tutti dato come risultato 1: nessun utente ha avuto alcuna difficoltà a trovare il pulsante per il cambio dei colori dell'interfaccia, piuttosto che ad eliminare una conversazione.

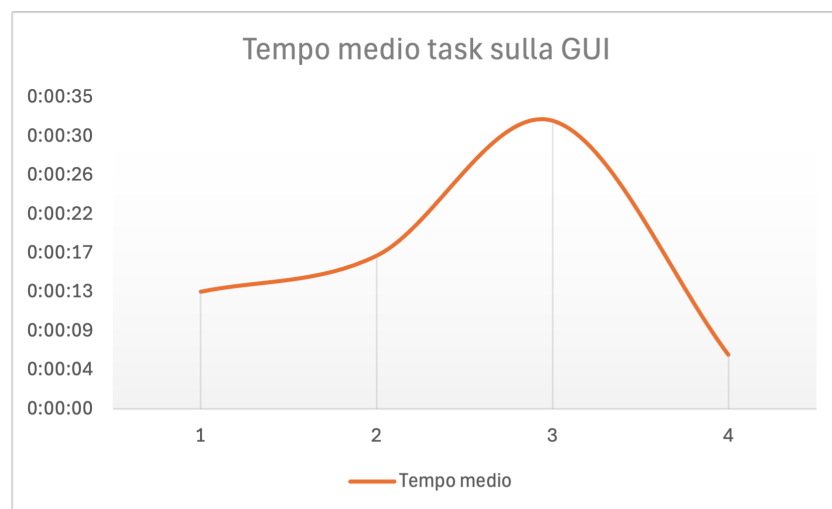


Figura 7.9: Tempo medio dei test sull'interfaccia.

In figura 7.9 è possibile vedere la distribuzione del tempo medio impiegato per ognuno dei quattro task: in generale gli utenti hanno completato molto velocemente questa parte del test.

Il questionario di fine test ha prodotto anch'esso degli ottimi risultati, evidenziando una generale soddisfazione nell'utilizzo dell'applicazione.

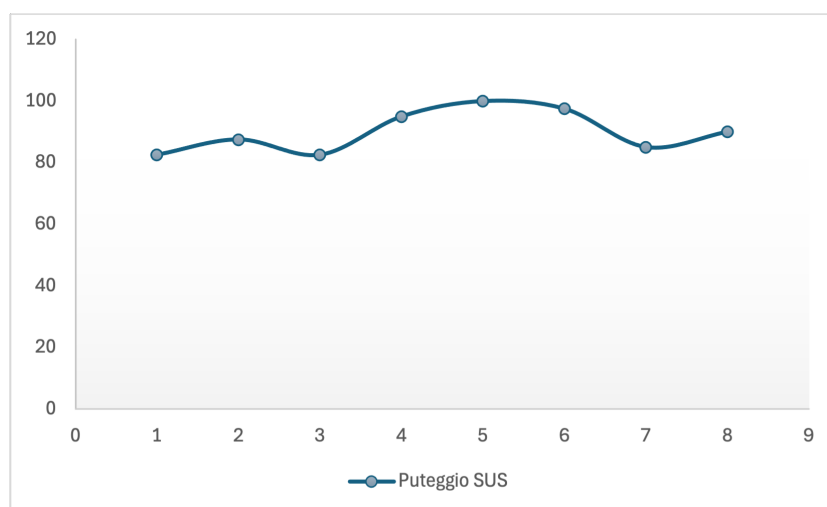


Figura 7.10: Distribuzione dei punteggi del questionario SUS

La **considerazione più importante**, generale, che emerge dopo la fase di test, è il considerare l'assistente come un **ausilio** importante allo studio ed alla **consultazione** delle documentazione, **non come un suo sostituto**. Per quanto infatti il chatbot possa dare delle risposte spesso corrette, succede anche che possa sbagliarsi o non trovare le informazioni corrette per tutte le casistiche trattate in questa tesi: fondamentale in tutti questi casi, ed in tutti quelli nei quali si renda necessario un reale **approfondimento** degli argomenti trattati, l'utilizzo della **documentazione**, per sua natura più completa. La funzionalità di citazione delle fonti è, di conseguenza, molto importante, e richiede per questo motivo particolari attenzioni. L'utilizzo consigliato di questo strumento è quindi, dopo l'analisi dei risultati dei test e degli appunti presi durante il loro svolgimento, quello di **accelerare la ricerca di informazioni**, che possono essere a volte sufficienti nella presentazione che l'assistente propone, mentre altre volte possa essere necessario approfondire nella documentazione, in modo comunque più veloce perché già contestualizzato e velocemente accessibile dalle fonti.

In conclusione è importante ancora specificare un aspetto che può sembrare banale ma non lo è per nessun motivo: una **buona documentazione**, scritta correttamente tenendo bene in mente quale sia l'utente di riferimento che dovrà utilizzarla **deve** necessariamente rappresentare la base di conoscenza ottimale per un buon assistente e, riguardo a quest'ultimo, la **semplicità e la completezza** con la quale vengono poste le richieste è un'abilità fondamentale per gli utenti che lo utilizzano, abilità che andrebbe istruita come tutte le altre che costituiscono la base di competenza di un buon dipendente: tutte le tecnologie e le strategie che questa tesi introduce sono sì utili, ma non possono sostituire interamente quelle che si possono considerare come abilità base degli utenti, che dai test si sono rivelate

presenti in tutto il campione.

Capitolo 8

Conclusioni e possibili sviluppi futuri

Il progetto descritto in questa tesi evidenzia l'**impatto positivo** dell'introduzione e del conseguente utilizzo di un assistente virtuale come strumento di ausilio per la consultazione della documentazione e più in generale in ottica di **didattica aziendale**.

La sperimentazione ha dimostrato che l'assistente **non** deve essere visto come un **sostituto** della documentazione, ma come uno strumento **complementare**, capace di velocizzare l'accesso ai dati e semplificare il processo decisionale.

Un aspetto fondamentale emerso è la capacità del chatbot di guidare l'utente verso informazioni rilevanti e specifiche, riducendo il tempo necessario per la consultazione. Tuttavia, è altrettanto chiaro che in alcune situazioni, come quelle più complesse o che richiedono una maggiore precisione, l'approfondimento attraverso la documentazione rimane imprescindibile. Questo mette in luce l'importanza della sinergia tra il chatbot e la documentazione stessa, con un sistema di citazione delle fonti che assume un ruolo cruciale.

Tutte queste funzionalità sono rese possibili dalla **collaborazione** sinergica tra il modello LLM alla base dell'assistente ed i moduli implementati per la sua **ottimizzazione**.

Un altro punto chiave è rappresentato dalla gestione **ottimale** dell'interfaccia utente e dell'**esperienza utente (UI/UX)**, che devono garantire **semplicità**, **efficienza** e una **risposta veloce** alle esigenze dell'utente.

I test hanno mostrato, in generale, un alto tasso di soddisfazione, soprattutto grazie alla chiarezza dell'interfaccia e alla velocità con cui si potevano portare a termine i compiti.

8.1 Possibili sviluppi futuri

Sia durante la fase di progettazione che in quella di test sono emerse delle potenziali funzionalità che per diversi motivi (scelte aziendali, tempo, necessità di ristrutturazione della documentazione) non sono state

8.2 Sviluppi futuri

Alla luce delle possibili future funzionalità e sviluppi emersi specialmente in fase di test si propone, in questa sezione, la discussione delle più significative.

Miglioramento della scansione delle domande: Nella versione finale del progetto di tesi, la scansione delle domande ad opera del **QuestionScanner** utilizza, come metrica tra le più importanti, l'**entropia** risultante tra le soluzioni associate alla domanda: più queste soluzioni sono diverse, più la domanda sarà vaga. Il livello al quale avviene questo calcolo è molto alto, specificatamente a livello di soluzione, di *pdf* o *video* (ad esempio *TMS Planning*, *TMS costing*): questo consente sicuramente una prima analisi su argomenti presenti in più documenti aziendali, ma non tiene conto della confusione della domanda nel caso l'argomento oggetto della stessa sia presente in parti diverse dello stesso documento, ma slegate tra loro. Posto che questo sarebbe risolvibile con una buona organizzazione e scrittura della documentazione aziendale, è bene che una potenziale versione futura dell'assistente ne tenga conto.

Per realizzare questa funzionalità sarebbe necessaria una codifica di **maggior fino** della documentazione, in cui ad ogni contenuto fosse anche associato il **paragrafo** (ad oggi vi è già una prima, embrionale, forma di localizzazione più precisa: nelle fonti viene citata la localizzazione del contenuto, ma è solitamente il capitolo generale, non il paragrafo, ed in più non sempre è disponibile): in questo modo sarebbe possibile confrontare i paragrafi associati alla domanda, e non i documenti, aumentando di molto l'efficienza del modulo.

Risposte in formato multimodale: Dopo aver sentito i punti di vista degli utenti, è emersa l'importanza della presenza delle immagini nel processo di apprendimento: presenza che ha molte volte portato a preferire l'uso della documentazione all'assistente. Un possibile sviluppo futuro, che passa nuovamente per una migliore formattazione della documentazione (della sua codifica in formato **JSON**), potrebbe essere quello di associare al contenuto non soltanto il paragrafo, ma anche le immagini: in questo modo sarebbe possibile rendere l'assistente multimodale **senza** doversi necessariamente appoggiare ad un modello che lo sia a sua volta: sarebbe sufficiente aver la possibilità di accedere alla raccolta di tutte le immagini presenti nella documentazioni, localizzare

quelle giuste in base alla soluzione utilizzata nella risposta e metterle a disposizione al client (ad esempio all'interfaccia), che a sua volta deciderà come inserirle all'interno della risposta. In questo modo si potrebbero **integrare** anche le immagini **senza aumentare i costi**.

Multimodale con modelli generativi: Nel caso l'azienda lo ritenga opportuno e necessario, è possibile sfruttare gli ultimi modelli LLM multimodali per generare, oltre al contenuto testuale della risposta, anche altre informazioni grafiche da associarci. Una possibile, utile, integrazione, sarebbe la generazione di grafici, specialmente diagrammi di flusso e di sequenza, che aiuterebbero molto gli utenti a capire il funzionamento dei processi più complessi. Questi grafici, qualora necessario, previsto ed autorizzato, potrebbero poi anche essere conservati, andando ad arricchire il patrimonio "didattico" aziendale.

Funzionalità dinamiche dell'interfaccia: possibilità di prevedere un sistema ad utenze per l'interfaccia grafica, specificatamente per la funzionalità di "chat avanzata", quella che permette l'aggiunta di esempi e la visualizzazione di prompt e riscritture. Nella versione attuale del progetto di tesi non è stata prevista sia perché non fosse oggetto del progetto, sia perché non richiesta da parte dell'azienda, ma rappresenta sicuramente un punto interessante per aumentare il controllo dell'intero applicativo e renderlo più adatto ad un ambiente di produzione scalabile.

Tuning: Il comportamento dell'assistente, la riscrittura delle domande e la composizione delle risposte, è facilmente adattabile alle esigenze di utenti diversi. Fornendo diverse classi di esempi è infatti possibile, tenendo a mente quale siano gli utenti di riferimento, personalizzare le risposte dell'assistente rendendole più o meno didattiche, più o meno riassuntive, e tutte le altre personalizzazioni del caso, a seconda delle specifiche esigenze. L'implementazione dell'applicativo in un ambiente virtualizzato facilita l'applicazione di questo punto, rendendo immediato la creazione di nuove istanze, modificabili, dell'applicazione di partenza.

8.2.1 Considerazioni sul fine-tuning

Analizzando il lavoro fatto in questa tesi chiunque, con una base pratica nell'utilizzo delle tecnologie trattate, potrebbe sollevare un interrogativo: il sistema di esempi alla base di due dei tre moduli più importanti non è sostituibile dal fine-tuning del modello utilizzando gli esempi stessi? Per quanto l'obiezione non sia errata, ed in un primo momento si è pensato di integrare il metodo nel processo di costruzione dell'assistente, questo ragionamento si è rivelato limitato: sebbene il fine-tuning sul singolo modello porterebbe a dei risultati in parte simili per quanto riguarda,

ad esempio, la produzione del messaggio finale (es. la sua formattazione) vi sono le seguenti limitazioni:

- **Legare l'assistente ad un singolo modello:** il fine-tuning, per com'è disponibile ad oggi, si deve eseguire in modo indipendente su ciascun modello utilizzato. Sebbene sia possibile in una qualche misura condividere i set di esempi tra modelli di uno stesso produttore, non è possibile farlo tra produttori diversi (ad esempio OpenAI e Google).
- **Costo:** Il set di esempi richiesti è notevole, necessario quindi dedicare una risorsa umana per un tempo non banale al compito, per ogni versione dell'assistente che si desidera implementare.

Alla luce delle considerazioni iniziali di **modularità**, **semplicità** ed **economicità**, si è deciso di implementare un sistema certo ispirato al fine-tuning, ma slegato dal singolo produttore.

8.3 Considerazioni finali

Questo lavoro rappresenta un passo importante verso l'integrazione efficiente di chatbot aziendali, offrendo un'analisi concreta delle potenzialità e dei limiti di tali strumenti. I moduli progettati ed implementati si sono rivelati, in seguito ai test effettuati, una soluzione innovativa che permette di raggiungere un buon grado di soddisfazione nell'interazione con gli assistenti virtuali: tutto questo viene raggiunto, si ricorda, sfruttando un modello LLM leggero ed economico, fornendo quindi un motivo in più per l'utilizzo nei processi aziendali di quanto discusso in questa tesi. Con i giusti sviluppi futuri, inoltre, sarà possibile rendere l'interazione tra utenti e sistemi di gestione delle informazioni più fluida e produttiva, sempre mantenendo l'utente al centro in ogni fase, dalla progettazione alla realizzazione, come è stato fatto in tutto questo progetto di tesi.

Bibliografia

- [1] Wikipedia contributors. *Language model* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 7-August-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Language_model&oldid=1234845936 (cit. a p. 7).
- [2] Competition and Markets Authority. *AI Foundation Models: Initial Report*. (2023). URL: https://assets.publishing.service.gov.uk/media/65081d3aa41cc300145612c0/Full_report_.pdf (cit. a p. 7).
- [3] Viktoria Szabo, Marianna Plesiak, Rui Yang and others. *Modern Approaches in Natural Language Processing – chapter 2 –Introduction: Deep Learning for NLP*. URL: https://slds-lmu.github.io/seminar_nlp_ss20/introduction-deep-learning-for-nlp.html (cit. a p. 7).
- [4] Vaswani Ashish, Shazeer Noam, Parmar Niki, Uszkoreit Jakob, Jones Llion, Gomez Aidan N, Kaiser Łukasz, Polosukhin Illia. *Attention Is All You Need*. URL: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf> (cit. a p. 7).
- [5] Wikipedia contributors. *Google* — *Wikipedia, The Free Encyclopedia*. 2024. URL: <https://en.wikipedia.org/w/index.php?title=Google&oldid=1239020858> (cit. a p. 7).
- [6] JAlec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever. *Language Models are Unsupervised Multitask Learners*. URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf (cit. a p. 8).
- [7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. URL: <https://arxiv.org/pdf/1810.04805v2> (cit. a p. 8).
- [8] Pandu Nayak. *Understanding searches better than ever before*. URL: <https://blog.google/products/search/search-language-understanding-bert/> (cit. a p. 9).

-
- [9] Wikipedia contributors. *ChatGPT* — *Wikipedia, The Free Encyclopedia*. 2024. URL: <https://en.wikipedia.org/w/index.php?title=ChatGPT&oldid=1238639683> (cit. a p. 9).
- [10] Wikipedia contributors. *OpenAI* — *Wikipedia, The Free Encyclopedia*. 2024. URL: <https://en.wikipedia.org/w/index.php?title=OpenAI&oldid=1239078470> (cit. a p. 9).
- [11] Wikipedia contributors. *Gemini (language model)* — *Wikipedia, The Free Encyclopedia*. 2024. URL: [https://en.wikipedia.org/w/index.php?title=Gemini_\(language_model\)&oldid=1238136852](https://en.wikipedia.org/w/index.php?title=Gemini_(language_model)&oldid=1238136852) (cit. a p. 9).
- [12] Wikipedia contributors. *Google DeepMind* — *Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Google_DeepMind&oldid=1239261599 (cit. a p. 9).
- [13] Amazon. *Cos'è la progettazione dei prompt?* URL: <https://aws.amazon.com/it/what-is/prompt-engineering/> (cit. a p. 10).
- [14] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, Ting Liu. *A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions*. URL: <https://arxiv.org/pdf/2311.05232> (cit. a p. 10).
- [15] Wikipedia contributors. *API* — *Wikipedia, The Free Encyclopedia*. 2024. URL: <https://en.wikipedia.org/w/index.php?title=API&oldid=1239085349> (cit. a p. 12).
- [16] Amazon. *RAG schema*. URL: <https://docs.aws.amazon.com/images/sagemaker/latest/dg/images/jumpstart/jumpstart-fm-rag.jpg> (cit. a p. 14).
- [17] Wikipedia contributors. *Docker (software)* — *Wikipedia, The Free Encyclopedia*. 2024. URL: [https://en.wikipedia.org/w/index.php?title=Docker_\(software\)&oldid=1232784397](https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1232784397) (cit. a p. 15).
- [18] Wikipedia contributors. *LangChain* — *Wikipedia, The Free Encyclopedia*. 2024. URL: <https://en.wikipedia.org/w/index.php?title=LangChain&oldid=1239269811> (cit. a p. 15).
- [19] Wikipedia contributors. *GitHub Copilot* — *Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=GitHub_Copilot&oldid=1242362671 (cit. a p. 17).
- [20] Wikipedia contributors. *Adobe ColdFusion* — *Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Adobe_ColdFusion&oldid=1234499492 (cit. a p. 18).

-
- [21] Neil Kanungo. *How Vector Databases Search by Similarity: A Comprehensive Primer*. 2023. URL: https://medium.com/kx-systems/how-vector-databases-search-by-similarity-a-comprehensive-primer-c4b80d13ce63?title=Human%E2%80%93computer_interaction&oldid=1238920306 (cit. a p. 38).
- [22] Wikipedia contributors. *Cosine similarity* — *Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Cosine_similarity&oldid=1238947356 (cit. a p. 38).
- [23] Solanas A, Selvam RM, Navarro J, Leiva D. *Some common indices of group diversity: upper boundaries*. 2012. URL: [Psychol%20Rep.%202012%20Dec; 111\(3\):777-96.%20doi:%2010.2466/01.09.21.PR0.111.6.777-796.%20PMID:%2023402047](https://doi.org/10.2466/01.09.21.PR0.111.6.777-796). (cit. a p. 40).
- [24] Wikipedia contributors. *Heuristic evaluation* — *Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Heuristic_evaluation&oldid=1230937430 (cit. a p. 49).
- [25] Wikipedia contributors. *Jakob Nielsen (usability consultant)* — *Wikipedia, The Free Encyclopedia*. [https://en.wikipedia.org/w/index.php?title=Jakob_Nielsen_\(usability_consultant\)&oldid=1232788379](https://en.wikipedia.org/w/index.php?title=Jakob_Nielsen_(usability_consultant)&oldid=1232788379). 2024 (cit. a p. 49).
- [26] Gary Hsieh. *How to Evaluate Chatbots (and Conversational User Interfaces) With Heuristics?* 2023. URL: <https://medium.com/@garyhsieh/how-to-evaluate-chatbots-and-conversational-user-interfaces-with-heuristics-d9c46eda14cb> (cit. a p. 51).
- [27] Henriette Einfeld, Felix Kristallovich. *The Rise of Dark Mode*. 2020. URL: <https://www.diva-portal.org/smash/get/diva2:1464394/FULLTEXT01.pdf> (cit. a p. 55).
- [28] ISO. *ISO 639-1 code list*. URL: https://www.loc.gov/standards/iso639-2/php/code_list.php (cit. a p. 89).
- [29] Polito e-Lite research group. *Usability testing*. URL: <https://github.com/polito-hci-2023/materials/blob/main/slides/12-usability-testing.pdf> (cit. a p. 120).