POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Politecnico
di Torino

Tesi di Laurea Magistrale

Documentation Chatbot : Una soluzione innovativa per l'accesso alla documentazione software

Supervisore

Prof. Riccardo COPPOLA

Candidato

Antonio MINONNE

Aprile 2025

Sommario

Questo lavoro di tesi presenta lo sviluppo e l'implementazione di un chatbot, progettato per fornire un accesso immediato e continuo alla documentazione relativa a software prodotti in contesto aziendale, essenziale per le attività di sviluppo e supporto tecnico. L'obiettivo principale è creare uno strumento che permetta ai consulenti di recuperare informazioni cruciali in qualsiasi momento in modo agevole e veloce, migliorando così l'efficienza del loro operato, la qualità del servizio offerto ai clienti e il mentoring delle nuove risorse. La capacità del chatbot di fornire rapidamente una panoramica aggiornata sulla documentazione, consente di ottimizzare anche il processo di aggiornamento dei software e della risoluzione di eventuali problematiche. Il lavoro è stato svolto iniziando dalla definizione della struttura del sistema, individuando il modello architetturale più adatto, per poi procedere con la selezione delle risorse software più adeguate per ciascun componente. L'ultimo step è stato quello di progettazione e implementazione del software, durante il quale sono state integrate le varie parti in un sistema funzionale e coerente. Il risultato del progetto di tesi è un applicativo basato su un'architettura Event-Driven a microservizi, in cui diversi componenti lavorano in sinergia per implementare il paradigma Retrieval-Augmented Generation (RAG). I vari moduli del sistema sono specializzati in diverse funzionalità, come la gestione delle interazioni con il database vettoriale, l'interfacciamento con un modello di Large Language Model (LLM) e l'elaborazione dei testi della documentazione scritti in formato Markdown. Questo approccio consente all'applicativo di recuperare e fornire informazioni aggiornate, oltre a permettere l'elaborazione continua di documenti inseriti dall'utente. La User Interface (UI) si basa su un chatbot Telegram che garantisce un'interazione fluida e immediata per gli utenti, che possono accedere rapidamente alle informazioni tramite la piattaforma di messaggistica. I risultati ottenuti dimostrano l'efficacia dell'applicazione, la quale è in grado di elaborare e recuperare informazioni dalla documentazione, presentando risposte adeguate alle domande poste dall'utente, includendo anche parti di codice e immagini. Sebbene la soluzione abbia il potenziale per essere utile in contesti reali, esistono ancora margini di miglioramento che possono essere esplorati in futuri sviluppi.

Ringraziamenti

Desidero ringraziare di cuore tutte le persone che sono state presenti durante gli anni dei miei studi universitari.
In particolar modo un ringraziamento speciale va al Prof. Riccardo Coppola che con la sua disponibilità e il suo supporto ha reso questo traguardo raggiungibile.
Inoltre non posso non esprimere la mia profonda graditudine verso la mia famiglia, che ha sempre supportato le mie scelte anche quando sembravano difficili da percorrere. Parlando di famiglia, voglio lasciare un pensiero per il me del futuro, che un giorno riprenderà in mano queste pagine ricordando il mio cane Dave, che mi è stato accanto per tutto il mio percorso universitario e che purtroppo se n'è andato poco prima della fine. Ti voglio bene, amico mio.
Ringrazio inoltre i miei ex-coinquilini di Via Tripoli 10 e tutti gli altri amici, con cui ho condiviso le infinite esperienze che hanno formato la mia persona. Tra questi sento di dover esprimere particolare gratitudine verso Paolo, Donato, Andrea, Riccardo N. e Riccardo G. che nei momenti veramente diffici e complicati, hanno permesso alla mia psiche di non cedere e che rappresentano a tutti gli effetti una seconda famiglia. Anche se a volte non si capisce, vi voglio bene ragazzi.
Per il lavoro di tesi, è doveroso ringraziare Pay Reply e tutti i colleghi con cui ho avuto a che fare, che con grande pazienza mi hanno trasmesso le conoscenze fondamentali per la creazione del progetto e per la mia formazione professionale. Nello specifico un grazie va ad Alessandro, che mi ha sempre spinto nell'andare oltre le mie aspettative.
Infine, grazie Maura. Grazie per essermi stata accanto in questi anni, per brillare ancora di più proprio nelle giornate più cupe e per avermi donato quelli che ora sono i ricordi più belli.

Indice

Elenco delle figure

Acronimi

AI
artificial intelligence

RAG
Retrieval Augumented Generation

MTTR
Mean Time to Repair

NLP
Natural Language Processing

SWEM
Static Word Embeddings Matrix

HPA
Horizontal Pod Autoscaler

VPA
Vertical Pod Autoscaler

AOF
Append-Only File

Capitolo 1

Introduzione

1.1 Contesto e Background

Questo lavoro di tesi nasce dall'opportunità che il Politecnico di Torino mi ha fornito di collaborare con Pay Reply. Pay Reply è una società di consulenza che fornisce soluzioni software competitive nell'ambito dei pagamenti digitali. Il primo incarico lavorativo assegnatomi una volta iniziato lo stage di formazione, è stato quello di supporto tecnico su un particolare componente software in ambito bancario generalmente chiamato Payment Orchestrator. Quest'ultimo è di fondamentale importanza all'interno dell'ecosistema di un'architettura software bancaria, infatti il suo compito è quello di fornire un'interfaccia unica per tutti i tipi di pagamento rendendo più semplice l'integrazione con il corebanking (altro componente che si occupa direttamente del trasferimento di denaro tra banche). Quindi, come è facile intuire, tutti i tipi di pagamenti transitano da questo modulo, rendendolo per definizione un componente di vitale importanza su cui si poggia la buisness logic complessiva dell'architettura. Considerata la sua centralità è necessario gestire un Payment Orchestrator con cura e in caso di guasti occorre intervenire nel modo piu veloce ed efficiente possibile per riportare il sistema in condizione stabili. Qualsiasi nuovo consulente in ambito informatico che si ritrovi a lavorare su un progetto di tali dimensioni può essere spiazzato dalla mole di codice, dalla struttura del progetto e dalle varie integrazioni con i provider molto particolari con cui quest'ultimo si integra.Per questo motivo è di fondamentale importanza che una nuova risorsa disponga di un aiuto in ambito tecnico per far fronte alle esigenze del business. Infatti le difficoltà che molte risorse riscontrano ad inizio percorso sono legate agli aspetti sopra citati, in particolare alla mancanza di conoscenza delle logiche di business che il componente implementa e delle logiche di integrazione con servizi di terze parti. Nonostante l'aiuto che i colleghi più esperti possono fornire nella fase iniziale, in alcune situazioni critiche vi è comunque la necessita che una nuova

risorsa operi autonomamente per hotfix su specifiche logiche di business o risoluzioni di problemi di carattere più generale non sapendo però come agire per risolverli. Ma, tralasciando per un momento la questione dal punto di vista della "nuova risorsa", anche i consulenti più esperti, all'interno delle società di consulenza, possono trovarsi in questa situazione. Infatti, per sua natura, una società di consulenza prevede che molti dipendenti vengano assegnati a diversi progetti nel corso della loro carriera. Questo li porta a passare da progetti a cui hanno contribuito a sviluppare sin dalle prime fasi del quale hanno una conoscenza completa ad applicativi già in produzione che sono stati sviluppati da altri consulenti, per i quali i clienti richiedono l'aggiunta di specifiche funzionalità. Tuttavia, per implementare queste modifiche in modo efficace e con il minimo impatto, è fondamentale una conoscenza approfondita e pregressa del sistema. Per affrontare le problematiche esposte finora, è fondamentale disporre di una documentazione software completa, aggiornata e facilmente accessibile per la consultazione. Tuttavia, anche quando è disponibile, spesso risulta obsoleta, difficile da reperire e complessa da consultare, compromettendo l'efficacia dei processi di sviluppo, ricerca e risoluzione dei bug.

1.2 Impatti, problematiche e possibili miglioramenti

Dato il contesto descritto nel paragrafo precedente, è facile intuire che l'inserimento di una nuova risorsa all'interno del flusso lavorativo è un percorso tutt'altro che semplice. Questo dovuto in parte alla mancata esperienza lavorativa dei giovani consulenti ed in parte alla mancanza di mentoring nel contesto specifico dell'ambito di lavoro. Inoltre lo stato della documentazione associata ad un applicativo, su cui comunque una risorsa potrebbe formarsi autonomamente, sovente è poco aggiornato e difficile da reperire. Nonostante ciò la documentazione viene comunque utilizzata ed è sempre la prima risorsa che si cerca in vista di nuovi sviluppi su un componente gia attivo o guasti su un particolare flusso di cui non si ha una conoscenza dettagliata. Gli impatti di una documentazione non facilmente accessibile e non ben aggiornata sono molteplici. In primo luogo potrebbe portare ad una inefficiente gestione del tempo nel caso in cui non venga aggiornata coerentemente con le modifiche apportate al codice, rendendo il lavoro più ostico al consulente che non ha mai visionato quel punto del codice costringendolo a fare reverse-engeneering per la risoluzione di un eventuale problematica. Inoltre potrebbe essere un ostacolo agli architetti che progettano una nuova feature del software basandosi su quella documentazione che ora non è più valida.
Assodato che mantenere i dati documentali il più coerenti possibile con l'applicativo che descrivono è fondamentale, è altrettanto importante concentrarsi sulla loro

accessibilità. Infatti, la principale problematica non è la mancanza di documentazione in sé, ma la difficoltà nel reperirla. Inoltre, una volta trovata, spesso risulta difficile consultarla a causa della specificità degli argomenti trattati, soprattutto per le nuove risorse che, come detto in precedenza, non hanno familiarità con progetti di grandi dimensioni e con le integrazioni che un software può avere con servizi di terze parti, operanti nel settore dei pagamenti.
Sulla base delle motivazioni esposte fin'ora ho deciso di orientare il mio lavoro di tesi verso il miglioramento all'accesso delle informazioni documentali. Infatti migliorare la reperibilità delle informazioni, non solo influisce direttamente su una maggiore efficienza nella gestione dei progetti, ma incide indirettamente sulla qualità del codice che viene sviluppato per l'implementazione di nuove feature su moduli software gia esistenti, e sull'efficacia della formazione autonoma da parte delle nuove risorse. Inoltre, aumentando la velocità di reperimento delle informazioni relative alla documentazione, si migliora anche l'efficienza e la qualità con cui un consulente può intervenire su un componente in caso di guasto per risolvere un problema, contribuendo così a elevare complessivamente la qualità del supporto tecnico che l'azienda è in grado di offrire.

1.3 Soluzione proposta

Una possibile soluzione a questa specifica esigenza potrebbe essere l'implementazione di un chatbot, che consenta di consultare facilmente le diverse parti dei progetti in carico all'azienda. Infatti grazie all'implementazione di questa soluzione sarebbe possibile ottimizzare in modo significativo l'accesso alle informazioni da parte dei consulenti, sia alle prime armi che più esperti, riguardo le parti fondamentali del progetto. Per di più renderebbe più semplice l'onboarding dei dipendenti sui vari prodotti software riducendo da un lato, il carico lavorativo di chi è incaricato al mentoring delle nuove risorse, e aumentando dall'altro la sicurezza e l'atuonomia operazionale che quest'ultime potrebbero avere supportate dal chatbot.
Entrando nella specifico un' assistente virtuale potrebbe, aiutare a reperire velocemente informazioni riguardanti un flusso specifico di operazioni che avvengono sul prodotto, quando, a causa di guasti, si è costretti ad intervenire in modo repentino, diminuendo di conseguenza il MTTR (Mean Time to Repair) e aumentando quindi l'efficienza nella gestione degli incidenti. In aggiunta può aiutare i consulenti a localizzare e isolare meglio le fonti di errore, che molte volte, nel caso di particolari tipi di anomalie che impattano flussi dati che coinvolgono third-party services, sono complesse da individuare e possono ritardare la risoluzione del problema anche di molti giorni. Infine un assistente digitale "esperto" nella documentazione potrebbe essere di grande aiuto ai progettisti software per pianificare in modo più appropriato e veloce gli sviluppi e l'architettura di nuove specifiche richieste da un cliente su un

progetto già esistente, evitando da parte di quest'ultimi di indagare all'interno del codice per reperire ad esempio la risposta di una determinata Api o il Diagramma ER di un database. Cosi facendo il lavoro dei progettisti e sviluppatori verrebbe semplificato, facendo concentrare i primi esclusivamente sulla progettazione dell'infrastruttura e aiutando indirettamente i secondi nell'avere direttive nei tempi previsti e quanto piu chiare possibili per progredire in modo efficace negli sviluppi. In definitiva, un assistente virtuale progettato per reperire in modo più efficiente le informazioni tecniche potrebbe migliorare in generale il benessere di tutti i dipendenti, la loro preparazione complessiva rispetto ai progetti su cui sono allocati e aiuterebbe a rientrare in modo agevole nelle stime proposte per il completamento di una commessa.
Un software che risponde a queste specifiche può essere realizzato grazie all' Intelligenza Artificiale, che consente di interpretare la documentazione disponibile, alimentando il sistema con le informazioni estratte, rendendo così più accessibili le richieste tecniche da parte dell'utente. Infatti, grazie ai significativi sviluppi degli algoritmi di Machine Learning negli ultimi anni, in particolare nel campo dell'analisi di contenuti testuali, è oggi possibile sviluppare un software basato su AI capace di "comprendere" e archiviare testi, per poi recuperarli in modo rapido ed efficiente quando necessario.

1.3.1 Obiettivi

L'obiettivo del lavoro di tesi è quindi lo sviluppo di un chatbot che può essere consultato dalle risorse per agevolare la produzione software, la progettazione dell'architettura di nuovi componenti e migliorare l'efficienza dell'intervento in caso di incidenti. Essendo un'applicazione destinata a un cospicuo numero di utenti, l'intento è di progettare un'architettura a microservizi, in cui ciascuno dei componenti coinvolti è responsabile di un compito specifico, mirato a soddisfare le richieste in modo rapido ed efficiente. Inoltre, i moduli che comporranno l'architettura dovranno essere sviluppati con una certa flessibilità per facilitare l'integrazione con possibili sviluppi futuri pianificati su questa base. Un ulteriore proposito sarà quello di sfruttare un'interfaccia Front-End il più semplificata possibile, accessibile con facilità da tutti gli utenti e che formatti correttamente le informazioni relative alla documentazione software, come codice, tabelle e immagini, per garantire una rappresentazione adeguata dei dati.
Un risultato atteso da questo progetto è fornire un sistema di gestione dei file relativi alla documentazione, suddividendoli in partizioni logiche (*progetti*). Questo permetterà all'utente, durante l'interrogazione del chatbot, di selezionare i documenti da cui intende recuperare le informazioni. Inoltre, durante l'inserimento di un nuovo documento nel sistema, sarà possibile scegliere il progetto a cui il documento fa riferimento, contribuendo ad arricchire la "base dati del chatbot". L'obiettivo

finale dell'attività sarà permettere alle risorse dei team di sviluppo dell'azienda di utilizzare l'applicativo creato, al fine di ottimizzare tutti i processi di produzione del software, ricerca e risoluzione dei bug.

Capitolo 2

Chatbot

2.1 Storia dei chatbots

Nonostante questo tipo di tecnologia sia diventata di uso comune solo negli ultimi anni, con l'avanzare del progresso tecnologico nell' ambito dell'Intelligenza Artificiale, i primi tentativi di sviluppare un chatbot che simulasse quanto piu possibile il linguaggio umano risalgono agli anni 60. Infatti tra il 1964 e il 1967 viene sviluppato il primo "natural language processing computer program", chiamato ELIZA. Quest'ultimo non aveva una vera e propria comprensione dell'input dell'utente, ma, grazie ad algoritmi di pattern matching e metodologie di sostituzione delle parole, dava l'illusione all'utilizzatore di capire cio che quest'ultimo le stava comunicando.

Figura 2.1: Prompt Chatbot Elisa

Dopo Eliza un passo significativo verso lo sviluppo di un software in grado di generare frasi in linguaggio umano è Alice. Questo chatbot che è stato pubblicato nel 1995, funziona grazie allo schema XML noto come linguaggio di markup dell'intelligenza artificiale (AIML), creato appositamente per definire i pattern di

conversazione e le risposte del chatbot. AIML consentiva di scrivere regole che associano determinati input (domande o frasi) a risposte predefinite.
Nel 2010 il grande passo in avanti. verso la realizzazione di un chatbot in grado di comprendere il linguaggio umano è stato fatto da Siri. Infatti l'assistente virtuale sviluppato da Apple non si limita ad essere solo un chatbot , riesce a riconoscere anche la voce, riuscendo a interagire con diverse App e dispostivi. Nel 2021, OpenAI ha lanciato GPT-3, uno dei chatbot più utilizzati dal grande pubblico, con 100 milioni di utenti attivi settimanali a livello globale. Questo modello di linguaggio impiega reti neurali avanzate per generare risposte simili a quelle umane e può essere considerato, insieme alle sue versioni più moderne (GPT-3.5 e GPT-4), lo stato dell'arte nel campo dei chatbot conversazionali. Attualmente queste tecnologie stanno vivendo una fase di rapida evoluzione grazie ai progressi nell'Intelligenza Artificiale e nell'Elaborazione del Linguaggio Naturale rendendo i chatbot più sofisticati e capaci di comprendere e generare linguaggio umano con maggiore precisione. Per capire la velocità del progresso di questa tecnologia e l'impatto che sta avendo sulla società basti pensare che nel 2024 il mercato globale dei chatbot è stato valutato 7.01 miliardi di dollari con una crescita prevista fino ai 20,81 miliardi entro il 2029. Questa previsione ottimistica è dovuta all'adozione crescente dei chatbot in vari settori, tra cui assistenza clienti , e-commerce e sanità.

2.2 Transformer

2.2.1 Principi di funzionamento

Un Transformer è un particolare modello di intelligenza artificiale che è utilizzato all'interno della maggior parte dei chatbots attualmente in commercio (ChatGpt, Falcon, ClaudeAi etc..). Questa tecnologia, infatti, è ampiamente utilizzata nell'ambito dell'elaborazione del linguaggio naturale ed è stata originariamente proposta nel celebre lavoro di Vaswani et al. [1]. Un Transformer può possedere un'architettura estremamente complessa, arrivando a includere anche centinaia di miliardi di parametri. Nell'ambito del Natural Language Processing (NLP), il task principale di questo modello è di generare in output la parola (o token) più probabile in base ai token ricevuti in ingresso.

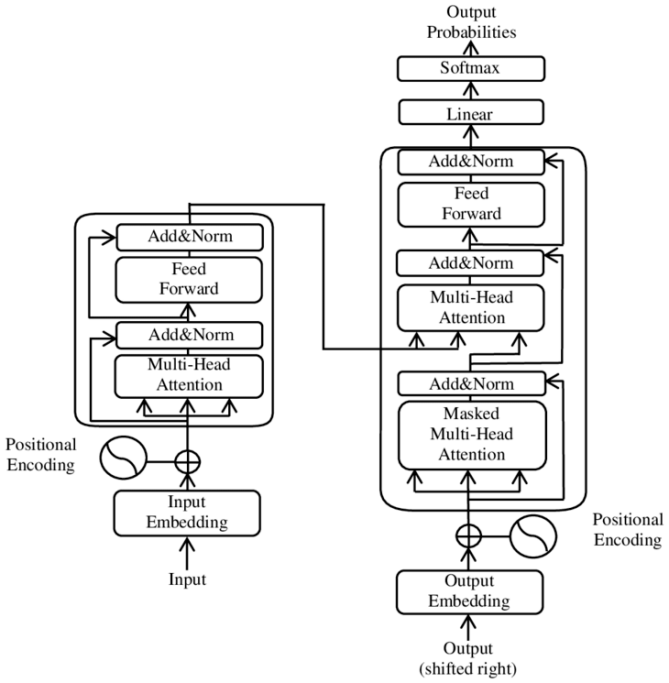

Figura 2.2: Schema architetturale Transformer

La *Figure 2* mostra uno schema ad alto livello di tutte le componenti che compongono il modello. Quest'ultimo è composto da 2 blocchi fondamentali l'Encoder e il Decoder. L'Encoder è il blocco che si occupa di ricevere l'input testuale, dividerlo in token, ovvero piccole porzioni di testo che possono essere una parola o anche una parte di una parola, e associare ogni token generato ad un *contextual-embedding*. Quest'ultimo è un vettore numerico di lunghezza d che racchiude intrinsecamente il significato semantico del token, anche in base alla posizione che questo occupa all'interno della frase in input. Il Decoder riceve in ingresso sia i contextual embeddings della frase di input, che tutti i token generati da lui stesso fino a quel momento. Successivamente, produce un vettore di probabilità di dimensione n, indicizzato a corpus di token (dizionario). In questo modo, riesce a determinare il token successivo più probabile, generandolo come output. Di seguito verranno analizzate le parti fondamentali che compongono il Transformer separatamente.

2.2.2 Generazione embeddings

Il primo layer che entra in gioco durante l'inferenza e il training di un Transformer è il layer responsabile della generazione degli embeddings. Il suo compito è creare una rappresentazione vettoriale per ciascun token che compongono la frase in input.

La rappresentazione vettoriale finale non deve contenere solo il dato semantico del token che rappresenta, ma anche l'informazione contestuale del token all'interno della frase, in modo da catturare il significato complessivo nel contesto specifico in cui appare.
Ricevuta una frase in input, questo layer inizia il processo di tokenizzazione, che consiste nel suddividere la frase in tokens. Successivamente, ogni token generato viene convertito in un intero che rappresenta il suo identificativo numerico. Questo ID viene recuperato da un dizionario, che costituisce l'insieme di tutti i token che il modello è in grado di comprendere. Il dizionario, o vocabolario, del Transformer è una mappa statica che associa a ciascun token un identificativo numerico univoco. La chiave di questa mappa è l'ID numerico, mentre il valore corrispondente è il token stesso. Questo vocabolario definisce l'insieme di tutte le parole, simboli e unità linguistiche che il modello può riconoscere e elaborare.

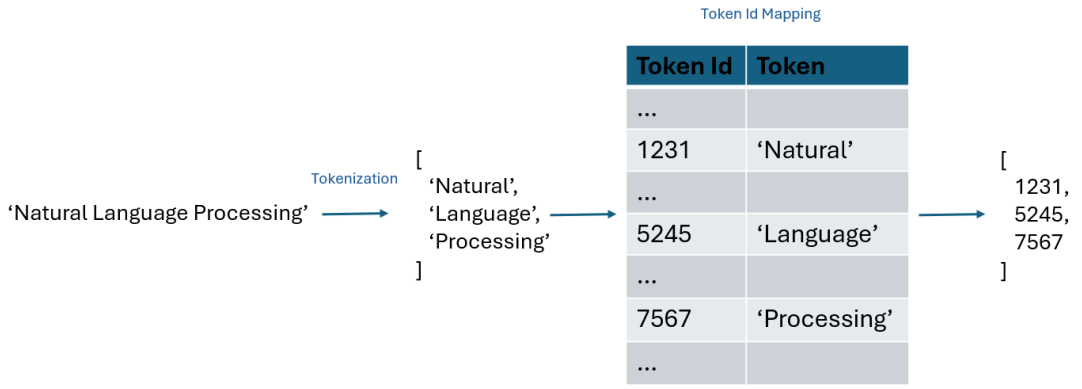

Figura 2.3: Processo di estrazione del tokenId dal corpus del modello

Una volta ottenuto il vettore composto dagli ID dei token, gli embeddings corrispondenti a ciascuno di essi vengono prelevati dalla *Static Word Embeddings Matrix* (SWEM). Questa matrice, di dimensione (n, m), è una componente fondamentale del Transformer e contiene un numero di voci pari al numero di token presenti nel dizionario. La SWEM fa parte del grafo di apprendimento del modello e quindi i suoi valori vengono aggiornati durante la fase di training, rientrando a tutti gli effetti all'interno dei parametri dell'architettura. In uscita dalla fase di selezione degli embeddings statici, si ottiene una sotto matrice (p, m) dove:

- p, è il numero di token della frase in input.
- m, è la dimensionalità degli embeddings che il modello utilizza.

Ogni voce della sotto-matrice cosi ottenuta rappresenta il significato semantico statico del token, ovvero il suo significato decontestualizzato. Il primo passo che il

Transformer prevede per ottenere una rappresentazione vettoriale del token che ne racchiuda il significato contestuale è quello di sommare la matrice degli embeddings statici con un'ulteriore matrice ottenuta tramite la seguente formula statica:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{2.1}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{2.2}$$

dove:

- d_{model}, è la dimensione degli embeddings all'interno del modello.
- PE, è la matrice dei *positional embeddigs* di dimensione (p, n).

La matrice cosi ottenuta, sommata alla sotto-matrice di embeddings statici ricavata al passo precedente, produce una nuova matrice della stessa dimensione, all'interno del quale ogni riga rappresenta un embeddings che contiene sia l'informazione semantica del token associato, che l'informazione della sua posizione all'interno della frase in input.

2.2.3 Attention layer

Le rappresentazioni vettoriali in uscita dallo stage precedente racchiudono esclusivamente il significato statico del token e l'informazione della loro posizione all'interno della frase. Per ottenere un risultato più accurato occorre capire la correlazione semantica che una determinata parola ha con il resto delle altre parole all'interno della frase. Questo è il lavoro dell'*Attention Layer*. Quest'ultimo è composto da piu *head di attenzione*, dove un singolo head di attenzione non è nient'altro che un modulo composto da tre matrici W_Q, W_K,W_V che sono parametri allenabili del modello, che cercano di dare un punteggio alla correlazione tra le parole prendendo in considerazione una particolare relazione. Si può dire che ogni head lavora su una particolare relazione tra le parole della frase e cerca di dare uno score al particolare rapporto che intercorre tra una parola e l'altra. Il primo passo che viene fatto in questo stage per ottenere questo particolare tipo di relazione è calcolare tramite una trasformazioni lineari i vettori Q, K e V nel modo che segue.
Siano:

- $X \in \mathbb{R}^{n \times d}$ vettore degli embeddings in input all'attention layer.

- $W_Q^{(h)}, W_K^{(h)}, W_V^{(h)} \in \mathbb{R}^{n \times d_k}$ le matrici di proiezione apprese per il calcolo dei vettori di query,key e value della $h-esima$ testa di attenzione, con $d_k = d/n_{head}$ dimensione dello spazio latente delle teste di attenzione.

I vettori Q,K e V si ottengono tramite trasformazion lineari degli embeddings di input:

$$Q^{(h)} = XW_Q^{(h)} \tag{2.3}$$
$$K^{(h)} = XW_K^{(h)} \tag{2.4}$$
$$V^{(h)} = XW_V^{(h)} \tag{2.5}$$

dove:

- $Q^{(h)} \in \mathbb{R}^{n \times d_k}$ contiene le query per ogni token, ovvero le rappresentezioni che cerano informazioni rilevanti tra la altre parole per la determinata relazione che l'head *h-esima* rappresenta. In altre parole è una rappresentazione matematica di cio che il token *n-esimo* sta cercando per una determinata relazione.
- $K^{(h)} \in \mathbb{R}^{n \times d_k}$ contiene le chiavi per ogni token, che forniscono il criterio per il confronto con le query. Ogni entry di questo vettore è la rappresentazione matematica delle informazioni che il token *n-esimo* può offrire per il particolare tipo di relazione che l'head *i-esima* cerca di rappresentare.
- $V^{(h)} \in \mathbb{R}^{n \times d_k}$ contiene i valori, ovvero le informazioni oggettive che verranno aggregate in base al peso assegnato dall'attenzione.

L'operazione successiva che è prevista nella pipeline di esecuzione è il calcolo della matrice di attenzione usando il prodotto scalare tra Q e K^T, normalizzato con la softmax per ottenere i pesi di attenzione nel modo che segue:

$$A^{(h)}(Q, K, V) = softmax\left(\frac{Q^{(h)}K^{(h)T}}{\sqrt{d_k}}\right) \tag{2.6}$$

Il risultato di questa operazione è una matrice $A_{(i)}^{(h)} \in \mathbb{R}^{n \times n}$, dove l'elemento $A_{(i,j)}^{(h)}$ da un'indicazione del punteggio di correlazione che il token *i-esimo* ha con il *j-esimo*, riguardante la relazione espressa dall' *h-esima* testa di attenzione. Una volta terminata questa procedura il passo successivo è quello di calcolare l'output della singola testa di attenzione, moltiplicando la matrice ottenuta al passo precedente per la matrice $V^{(h)}$:

$$Z^{(h)} = A^{(h)}V^{(h)} \tag{2.7}$$

Dove $Z^{(h)} \in \mathbb{R}^{n \times d_k}$ è una matrice contentente una sottoporzione di dimensione d_k di tutti gli n tokens rielaborati, contenenti informazioni contestuali provenienti da tutti gli altri tokens della sequenza.

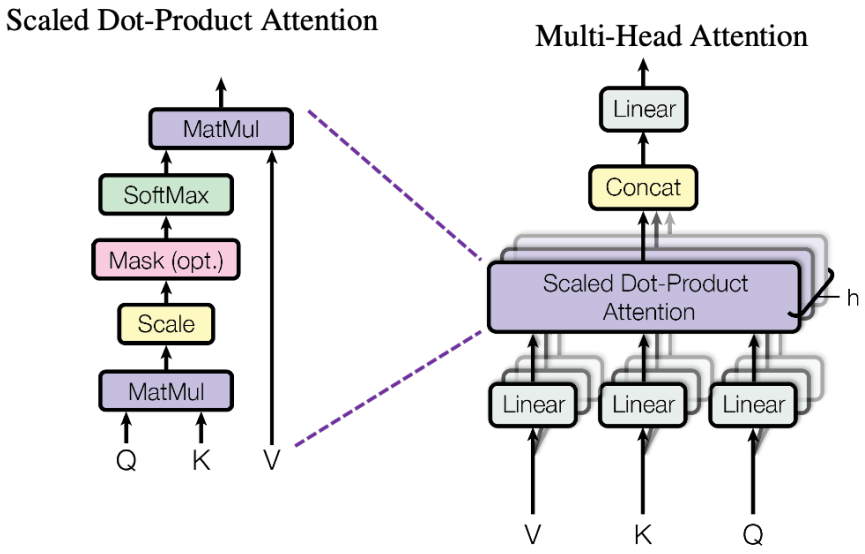

Figura 2.4: Schema architetturale Multi-Head Attention Layer

L'operazione di generazione dei contextual-embeddings si conclude applicando una trasformazione lineare alla concatenzione dell'outputs $Z^{(h)}$ di tutte le teste di attenzione con una matrice di pesi W_O per mescolare le informazioni informazioni tra le teste.

$$E_{contextual} = Concat(Z^{(0)}, Z^{(1)}, ..., Z^{(n_{head})})W_O \tag{2.8}$$

dove:

- $E_{contextual} \in \mathbb{R}^{n \times d}$, è l'ouput finale dello stage contenente le rappresentazioni vettoriali contestualizzate di lunghezza d di tutti gli n token.
- $W_O \in \mathbb{R}^{n \times n}$, è una matrice di trasformazione lineare applicata per combinare le informazioni tra gli output delle teste di attenzione.
- $Z^{(0)}, Z^{(1)}, ..., Z^{(n_{head})} \in \mathbb{R}^{n \times d_k}$, sono le matrici contenenti le sotto-porzioni dei token per ogni testa di attenzione.

Questa elaborazione, che permette la generazione di rappresentazioni vettoriali relative ai token via via piu complessi, viene ripetuta sia nell'encoder che in parte decoder. La spiegazione del funzionamento di questi due componenti verrà esposta nei paragrafi che seguono.

2.2.4 Encoder

Il compito dell'encoder è fornire una rappresentazione vettoriale dei tokens in input, rielaborata in modo che quest'ultimi contengano le informazioni di contesto all'interno della frase. Per ottenere questo risultato la frase in input viene inizialmente pre-elaborata come descritto nel paragrafo 2.2.2.

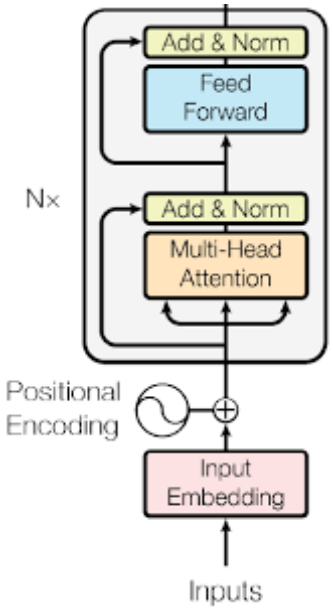

Figura 2.5: Schema architetturale Encoder

Successivamente gli embeddings ottenuti vengono mandati avanti in input al primo *encoder-block*. All'interno di questo blocco gli elementi vengono processati dal *Multi-Head-Attention layer* come descritto nel paragrafo 2.2.3 e in seguito il risultato fornito in output viene processato da un *residual-layer* che ha il compito di sommare a quest'ultimi gli embeddings in input all'attention-layer e normalizzare il risultato della somma per preservare le informazioni originali dei dati in ingresso e mantenere una distribuzione coerente dei valori. Il risultato di quest'ultima computazione viene fornito a seguire ad un *Feed-Forward-Layer* con lo scopo di introdurre una non linearità nei dati in modo tale da codificare dipendenze complesse tra essi. Infine l'elaborazione all'interno di un encoder-block viene conclusa tramite il passaggio da un'ulteriore residual-block con l'obiettivo di portare avanti nella catena di elaborazione le informazioni precedentemente calcolate. L'output finale in uscita da un ecoder-block sarà una matrice $E_{eb} \in \mathbb{R}^{n \times d}$ contenente la rappresentazione vettoriale di contesto $E_{eb}(i)$ dell'*i-esimo* elemento all'interno dell'input.
L'architettura dell'Encoder può essere composta da N encoder-block successivi

utili a raffinare progressivamente la rappresentazione numerica dei dati in ingresso, infatti N è un parametro dell'architettura che ne definisce la complessità.

2.2.5 Decoder

Il compito del Decoder è, una volta ricevuti gli embeddings dei tokens processati dall' Encoder, processare quest'ultimi insieme alle informazioni dei tokens che ha precedentemente generato per creare un nuovo elemento in output.

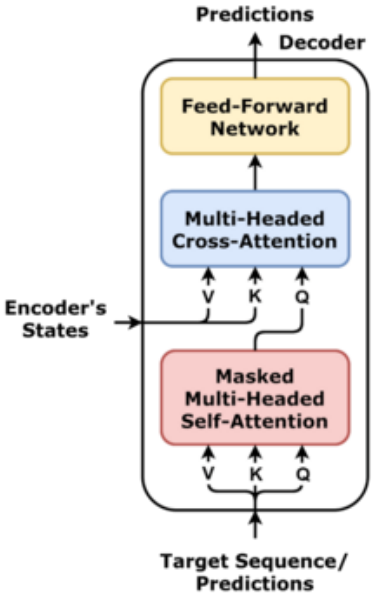

Figura 2.6: Schema architetturale Decoder

L'unità funzionale di questo componente è chiamata *decode-block* e riceve in input la sequenza di token che ha predetto fino a quel momento e gli embeddings dell'input generati dall'encoder.
In una prima fase il decoder prende la successione di elementi predetti fino a quel momento e ne calcola i *positional-embeddings* come visto nel paragrafo 2.2.2. Di seguito quest'ultimi vengono forniti al decode-block che prevede un primo step del calcolo dell'attenzione per processare gli embeddings in modo tale da avere le informazioni di tutta la sequenza. In questo caso però a differenza di quanto visto nel paragrafo 2.2.3 alla matrice $A^{(h)}$ viene applicata una maschera $M \in \mathbb{R}^{n \times n}$ dove M è una matrice triangolare inferiore con i valori non nulli settati ad 1.

$$A^{(h)}(Q, K, V) = softmax\left(\frac{Q^{(h)}K^{(h)T}}{\sqrt{d_k}}\right)M \tag{2.9}$$

Questo viene fatto affinchè i punteggi di correlazione all'interno della matrice $A^{(i)}(n, m)$ con $m > n$ dell'elemento $n-esimo$ con gli altri elementi che lo succedono nella sequenza siano 0, poichè la predizione fatta dell'elemento n-esimo deve essere riferita solo ai token che lo precedono nella sequenza.

Successivamente le rappresentazioni vettoriali in uscita, dopo essere state processate attraverso un residual-layer (come descritto anche in 2.2.4), vengono date in input al *Multi-Head-Cross-Attention-Layer*. Il meccanismo interno con cui vengono processate le informazioni all'interno di questo layer è il medesimo del *Self-Attention-Layer* utilizzato all'interno dell'encoder, con la differenza che i vettori $K^{(h)}$ e $V^{(h)}$ sono calcolati a partire dagli embeddings forniti in input dall'encoder mentre il vettore $Q^{(h)}$ viene creato a partire dagli embeddings generati dal decoder. Una volta ottenuti in output gli embeddings rielaborati dal Multi-Head-Cross-Attention-Layer, l'elaborazione prosegue passando i dati in input in successione ad un feed-forward-layer e un residual-layer. L'output ottenuto da un decoder-block è una matrice $E_{dec} \in \mathbb{R}^{m \times d}$, dove d è la dimensione degli embeddings che il modello supporta e m è il numero di token generati dal decoder fino a quel momento.
L'architettura del decoder-block descritta può essere ripetuta N volte all'interno del modello influenzandone l'accuratezza della predizione e la generazione delle rappresentazioni numeriche riferite ai token in input al decoder.

2.2.6 Predizione finale

Una volta ottenuti gli embeddings rielaborati in uscita dell'ultimo Decoder, quest'ultimi vengono fatti processare da un layer lineare per ottenere una dimensione in output in $\mathbb{R}^{n \times v}$, dove v è la dimensione del dizionario V. Dopodichè i dati vengono processati un'ultima volta applicando la softmax ad ogni riga della matrice ottenuta dal calcolo precedente.

$$R = softmax(E_{dec} W_{proj}) \tag{2.10}$$

dove $R \in \mathbb{R}^{n \times v}$ e l'elemento $R_{(i,j)}$ rappresenta la probabilità che il token nella posizione *j-esima* del vocabolario, possa occupare l'i-esima posizione all'interno della sequenza. Dopodichè per ogni riga della matrice R viene selezionato l'indice con massima probabilità.

$$r = argmax_j(R_{(i,j)}), \quad \forall i \tag{2.11}$$

dove l'elemento *i-esimo* di $r \in \mathbb{R}^n$, rappresenta l'indice del token all'interno del vocabolario V, che ha la massima probabilità di occupare la posizione i nella

sequenza di output. L'algoritmo, infine, prevede di prendere l'ultima entry di questo vettore per ricavare il token successivo del quale vi è necessità. L'inferenza prosegue ritornando in input i tokens trovati in tutti i passi precedenti, fino alla recezione del token speciale $' < END >'$ che indica la fine della sequenza generata.

2.3 Retrieval-Augmented Generation

I modelli generativi di linguaggio naturale, per quanto precisi, comodi e discrosivi, presentano alcuni limiti. Il principale svantaggio di un modello di linguaggio naturale è che, per aggiornarlo con informazioni recenti, è indispensabile riaddestrarlo utilizzando nuovi dati. Sebbene esistano tecniche, come il fine-tuning, che semplificano il processo di aggiornamento di un modello pre-addestrato con nuovi dati, eseguire un training completo per ogni nuovo documento che l'utente desidera inserire non è comunque una soluzione praticabile nel contesto del Documentation Chatbot. Per superare questi limiti, è stato introdotto negli ultimi anni un paradigma noto come *Retrieval-Augmented Generation* spiegato nel lavoro di Yunfan Gao et. al [2]. Questo approccio prevede una fase iniziale di elaborazione dei dati, in cui, ad esempio, un documento testuale viene suddiviso in piccoli frammenti e per ciascun frammento viene calcolata una rappresentazione vettoriale chiamata embedding. Una volta calcolati, gli embeddings vengono poi archiviati in coppia al frammento di testo al quale si riferiscono in un particolare tipo di database, denominato Database Vettoriale. Quest'ultimo permette, dato un embedding in ingresso, di recuperare quelli più "vicini" all'interno del database, confrontando quello in input con quelli già salvati usando specifiche metriche di similarità.

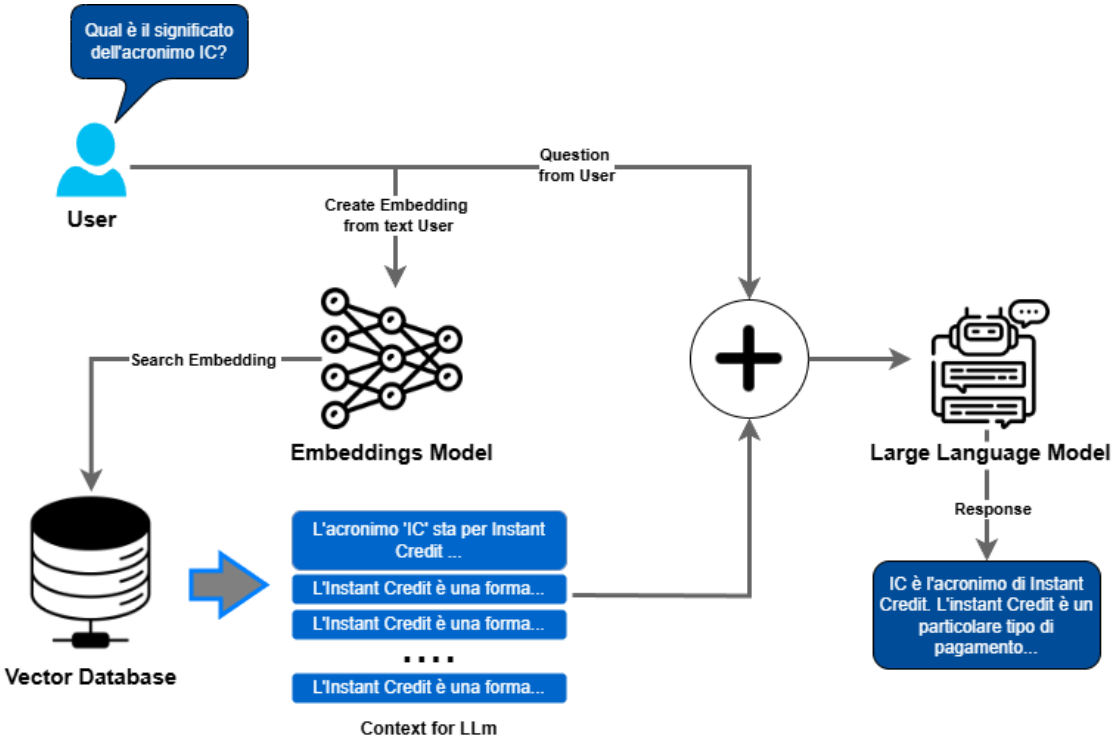

Figura 2.7: Flusso di esecuzione paradigma RAG

Nella fase in cui l'utente interroga il sistema, per generare una risposta viene inizialmente calcolato l'embedding del testo in input. Successivamente, il sistema esegue una ricerca nel Database Vettoriale per individuare i frammenti testuali più pertinenti alla richiesta. Questi ultimi, insieme alla domanda dell'utente, vengono forniti in input al LLM, che utilizza i frammenti recuperati come contesto per elaborare una risposta. Una volta generata, la risposta viene restituita all'utente. Il paradigma RAG è il meccanismo adottato dal Documentation Chatbot per fornire risposte in modo efficace. Questo è reso possibile da un'architettura software progettata per essere resiliente e ottimizzata, capace di gestire un elevato numero di richieste simultanee da parte di più utenti. I dettagli di questa architettura verranno approfonditi nei capitoli successivi.

Capitolo 3

Architettura software

3.1 Introduzione

L' evoluzione delle architetture software ha attraversato diverse fasi, adattandosi alle crescenti esigenze di scalabilità, manutenibilità e rapidità nello sviluppo. Inizialmente le applicazioni sono state sviluppate seguendo il paradigma di *architettura monolitica*, in cui tutti i componenti come l'interfaccia utente, la logica del server e l'interazione con i database sono integrati in un'unica base di codice unificata. Questo approccio facilita l'avvio dello sviluppo grazie alla sua semplicità, ma con il crescere della complessità dell'applicazione, questa diventa sempre più difficile da mantenere e scalare. Infatti piccole modifiche possono ripercuotersi su ampie porzioni del sistema, rendendo il processo di aggiornamento e bug-fixing lento e tedioso.

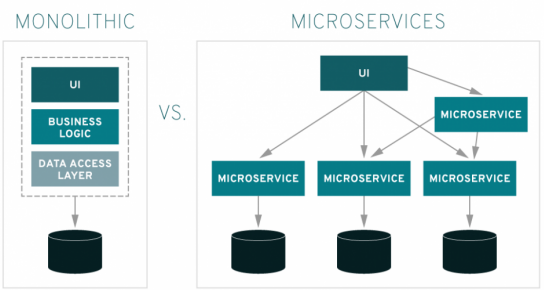

Figura 3.1: Confronto ad alto livello tra architettura monolitica e architettura a microservizi

Per ovviare alle limitazioni che intrinsecamente il paradigma dell'architettura monolitica introduce, si è fatto strada un nuovo modello di architettura chiamata *architettura a microservizi*. Quest'ultimo prevede che l'applicazione venga divisa in

una serie di *servizi indipendenti*, ognuno focalizzato su una specifica funzionalità logica. Questa filosofia di costruzione software infatti prevede che i servizi cosi sviluppati comunichino e collaborino tra loro, attraverso chiamate API o messaggi asincroni, per realizzare una specifica funzionalità. Questa modularità consente a ciascun servizio di essere sviluppato, distribuito e scalato in modo autonomo, migliorando la flessibilità e la resilienza del sistema complessivo.
Un'ulteriore evoluzione è rappresentata dai *monoliti modulari*, che cercano di combinare i vantaggi dei microservizi, come la modularità e il disaccoppiamento della logica complessiva di business, con una minore complessità tecnologica, cercando di diminuire quanto più possibile le comunicazioni via web di cui l'architettura a microservizi necessità introducendo un ulteriore elemento fault-prone. Infatti seguendo questo approccio, l'applicazione rimane un'unica unità deployabile, ma è strutturata in moduli ben definiti che possono essere sviluppati e gestiti in modo indipendente, facilitando la manutenzione e lo sviluppo del software.
In questo lavoro di tesi, si è scelto di seguire il paradigma di architettura a microservizi per lo sviluppo dell'applicativo, cercando di suddividere quanto più possibile i compiti di ogni microservizio in maniera chiara, mantenendo le logiche interne più semplici possibili. Tale scelta è stata fatta per garantire che il software finale sia altamente performante, scalabile e facilmente aggiornabile con nuove funzionalità. Per rendere più chiara la motivazione di questa scelta, nei paragrafi successivi del capitolo verranno esaminati in dettaglio i due principali paradigmi architetturali più diffusi oggi, approfondendone vantaggi e svantaggi [3].

3.2 Architettura monolitica

3.2.1 Caratteristiche principali e ciclo di vita

L'architettura monolitica rappresenta un approccio tradizionale allo sviluppo software, in cui tutte i componenti, dall'interfaccia utente al lato server e al database, sono integrate in un codice sorgente unificato. Questa struttura implica che i vari moduli dell'applicazione siano strettamente interconnessi e interdipendenti, rendendo l'intero sistema una singola unità indivisibile.
Date queste premesse è facilmente intuibile che questo approccio comporta diverse problematiche in fase di sviluppo, a causa del fatto che ogni componente del team di sviluppo si ritrova a lavorare sulla stessa base di codice rendendo più macchinosa l'integrazione e l'unificazione delle varie features che una singola risorsa sta sviluppando. Un altro punto fondamentale è che in fase di sviluppo si tende ad adeguare le tecnologie e i linguaggi che si utilizzano per l'intero progetto, facendo scendere a compromessi gli sviluppatori che progettano una particolare feature che potrebbe essere resa al meglio con una tecnologia o linguaggio specifico, diverse da quelle imposte all'interno del progetto. Questo è dovuto al fatto che il codice è

accoppiato e deve funzionare all'interno di uno stesso ambiente, detto che, in molti casi, esistono soluzioni per integrare diversi linguaggi e tecnologie all'interno di uno stesso monolite ma vengono spesso scartate perche troppo macchinose e poco manutenibili.
La fase successiva a quella di sviluppo è quella di testing del software, che viene eseguita in questa particolare architettura contemporaneamente su tutta l'applicazione. Questo impatta direttamente su una maggiore difficoltà nella scrittura dei test di unità dei vari componenti, poichè all'interno di un monolite ad esempio il comportamento di una singola funzione potrebbe dipendere direttamente da altre parti del sistema, rendendola difficile da testare e mockare in modo isolato.
Poichè tutte le funzionalità e componenti sono contenute in un unico repository, l'accesso e la gestione del codice sono agevolati, semplificando il delploy dell'applicativo in un ambiente essendoci un'unica pipeline di distribuzione, anche se una piccola modifica in una parte del sistema richiede la ricompilazione e la ridistribuzione dell'intera applicazione.

3.2.2 Scalabilità

Una volta che l'applicazione monolite è in produzione, si possono avere grandi flussi di utenti che sfruttano un particolare servizio, magari più utenti di quanto si era preventivato in fase di progettazione a causa ad esempio di un aumento del volume business. Se l'architettura hardware su cui l'applicativo è in esecuzione era stata pensata per supportare un determinato carico di richieste, un aumento significativo di quest'ultime può portare a gravi problematiche che possono compromettere il business dell'applicazione. Gli incidenti più comuni che sono legati ad un sovraccarico di richieste sono:

- **saturazione della CPU**: a fronte di tante richieste, la CPU viene sovraccaricata e non riesce ad elabrarle tutte in un tempo accettabile aumentando cosi il tempo di risposta e impattando direttamente sulla qulità del prodotto. Se il server ha un limite massimo di thread per le richieste, le nuove che arrivano vengono messe in coda o scartate.

- **esaurimento della RAM**: se le singole richieste conusmano molta memoria, ad esempio salvando le sessioni utente in RAM, si potrebbero avere continui *page swap* che il server fa per terminare processi e liberare memoria, ma questo renderebbe il sistema complessivo molto lento.

- **overload del Database**: un aumento di richieste, comporta quasi in modo proporzionale un aumento delle query che l'applicativo fa verso il database service. Un incremento significativo del numero di query potrebbero bloccare il servizio database e il tempo di esecuzione delle query.

Per questo motivo se si prevede un aumento di volume del business quando l'applicativo è gia in produzione, è molto importante agire prima per migliorarne le prestazioni, in modo tale da gestire il sovraccarico previsto.
Per potenziare le prestazioni del nostro applicativo, quando quest'ultimo è stato progettato con un'architettura monolitica vi sono due opzioni principali. La prima è scalare il sistema verticalmente, ovvero migliorare le prestazioni dell'hardware sul quale il software è in esecuzione, aumentandone la RAM o aggiungendo più CPU dove possibile o nel caso sostiuire quella attuale con una più performante.

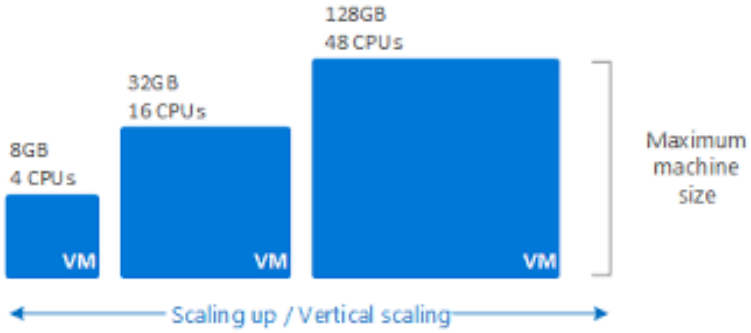

Figura 3.2: Esempio dell'andamento aumento risorse Hardware su nodo computazionale.

Il limite della scalabilità verticale è che ogni sistema ha un livello massimo di risorse che fisicamente può gestire. In altre parole non può essere aggiunta RAM all'infito all'hardware che abbiamo a disposizione, o non possono essere aggiunte piu CPU, poiche la scheda madre supporta una quantità massima di socket dedidicati. Inoltre il costo di scalare verticalmente cresce in modo non lineare, infatti oltre ad un certo limite cambiare CPU o sostituire la RAM con una più potente potrebbe diventare molto costoso, ottenendo un aumento di performance poco significative in relazione al budget speso. Un 'ultima criticità della scalabilità verticale è che se tutti i costi e gli effort vengono indirizzati verso un unico componente hardware su cui il software va in esecuzione, questo diventerebbe automaticamente un *single point of failure* dell'intero applicativo che in caso di guasto comprometterebbe tutto il business. Quindi per tutte le motivazioni spiegate, nonostante l'architettura monolitica sembra prestarsi bene per essere scalata in maniera verticale, molte volte questa pratica risulta poco vantaggiosa. La seconda opzione disponibile utile ad aumentare le prestazioni del nostro sistema, è scalarlo in modo orizzontale. Questo comporta la creazione di una o più istanze dell'applicativo che vanno in esecuzione su macchine differenti, con l'aggiunta di un altro componente software, il *load-balancer*, che funge da punto di accesso unificato per le chiamate in ingresso, e gestisce il carico tra le varie istanze dell'applicativo.

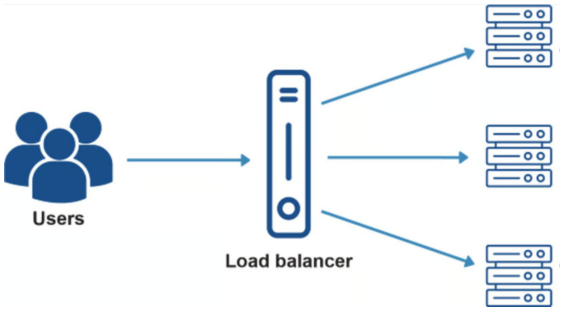

Figura 3.3: Schema funzionalità Load Balancer

In questo modo è possibile risolvere molte delle problematiche legate alla scalabilità verticale. Infatti l'applicativo puo essere replicato su piu istanze hardware che non devono essere necessariamente performanti, garantendo comunque un aumento delle performance del sistema complessivo in quanto vi sono più entità replicate che lavorano in parallelo.
Nonostante ciò, per un software progettato in modo monolitico neanche questa soluzione è completamente indolore, infatti dato che vi è una base di codice unificata il sistema va replicato per intero anche se solo una piccola parte di questo è in sovraccarico, provacando sostanzialmente uno spreco di risorse. Inoltre è importante notare, che in un'applicazione di questo genere, tutte le funzionalità spesso condividono lo stesso database quindi è importante gestire i problemi di concorrenza quando più istanze cercano di accedere ad una stessa risorsa condivisa sul database, il che comporta l'aggiunta di un ulteriore overhead di complessità non prevista inizialmente.

3.2.3 Manutenzione e gestione errori

Come detto anche nei paragrafi precedenti all'interno di un'applicazione con architettura monolitica le varie parti di codice che ne implementano le funzionalità sono strettamente accoppiate. Questo può rendere molto più ostico fare manutenzione del codice poichè aumenta il rischio di incorrere in errori non previsti a causa delle modifiche effettuate. Infatti sovente, la correzione di un bug relativo a un flusso può introdurre un altro bug più grave in altri flussi in cui l'applicativo è coinvolto compromettendone le funzionalità, quindi ogni manutenzione ha un effort non indifferente anche per quanto riguarda il test della modifica apportata. Di conseguenza è facilmente intuibile che in generale, anche in fase di sviluppo, si deve prestare particolare attenzione alle situazioni critiche, come la gestione degli errori in un particolare flusso, poichè un bug all'interno dell'applicazione puo rendere l'intero sistema non funzionante.

3.3 Architettura a micro-servizi

3.3.1 Caratteristiche principali e ciclo di vita

L'architettura a microservizi rappresenta un'evoluzione del modello monolitico, suddividendo l'applicazione in una serie di servizi indipendenti e autonomi. Ogni microservizio è responsabile di una specifica funzionalità e comunica con gli altri attraverso API ben definite o eventi asincroni e ad ognuno è associato un database specifico per garantirne l'indipendenza.
Il modo in cui comunicano tra loro i microservizi impatta direttamente alcune caratteristiche dell' applicativo che implementano. Una modalità di comunicazione che può essere implementata tra quest'utltimi è quella *sincrona*, dove i vari componenti comunicano tra di loro attraverso API REST. Con questo approccio, però, aumenta la dipendenza tra servizi aprendo a scenari di fallimenti in cascata tra questi, infatti se un servizio fallisce anche il servizio che dipende da quest'ultimo fallirà. Di conseguenza, vi possono essere anche colli di bottiglia, dato che, in presenza di un servizio particolarmente carico di richieste, si avrà il rallentamento anche di tutti i servizi che attendono una risposta da quest'ultimo. La comunicazione tra microservizi può essere realizzata in alternativa, attraverso un protocollo di tipo **asincrono**, implementando un paradigma di programmazione e architettura software *Event-Driven* basato sulla gestione di eventi.

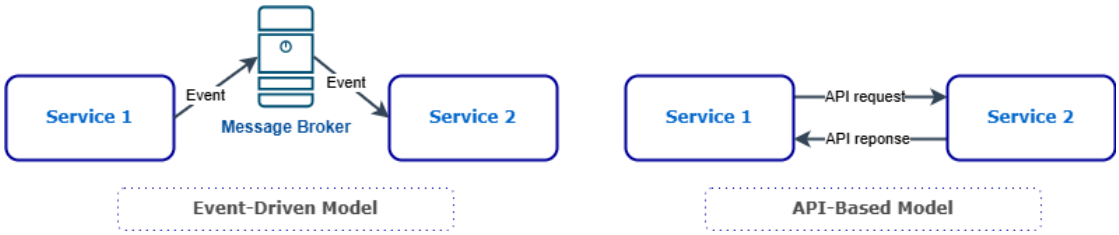

Figura 3.4: Reppresentazione comunicazione sincrona e asincrona.

Con questo tipo di strategia un microservizio non aspetta una risposta immediata, ma invia un messaggio a un *Message Broker*, che è uno specifico componente che si occupa di gestire i vari tipi di messaggi instradandoli verso un 'ulteriore servizio a cui interessa quel particolare tipo di evento e che è in grado di elaborarlo nel modo corretto. Si può notare che implementando questo tipo di tecnologia i macroservizi diventano a tutti gli effetti indipendenti l'uno dall'altro, aumentando di conseguenza il parallelismo dell'elaborazioni delle richieste ed evitando la latenza della comunicazione sincrona.
Questa struttura consente un'elevata modularità e scalabilità, permettendo che ogni servizio possa essere svilppato, testato e distribuito separatamente. In questo contesto, grazie alla modularità che il modello introduce, ogni risorsa del team

può lavorare separatamente su una particolare feature dell'applicazione senza interferire con parti scritte da altre risorse evitando i conflitti sul codice. Inoltre ogni microservizio può essere scritto in un linguaggio differente e adottare tecnologie più adatte al suo scopo.
Per quanto riguada la fase di testing, i test di unità risultano più semplici da scrivere, da che per loro natura i microservizi sono indipendenti e possono essere isolati facilmente, anche se la necessità di test di integrazione aumenta, poichè si deve essere certi che i servizi comunichino tra di loro correttamente, quest'utlimo aspetto di conseguenza rende il debugging piu complesso, perchè un problema potrebbe derivare dall'interazione tra più servizi.
Uno dei grandi vantaggi che i microservizi offrono è che ogni microservizio può essere rilasciato separatamente in ambiente e questo riuce il rischio di impatti sull'intero sistema, anche se la pipeline di CI/CD diventa più articolata , richiedendo strumenti di orchestrazione come Kubernetes. In fase di deploy inoltre, bisogna prestare attenzione alle versioni e alle compatibilità tra microservizi affinchè si abbia la sicurezza che le modifiche che si stanno per rilasciare su un microservizio siano compatibili con le versioni degli altri servizi con cui interagiamo.

3.3.2 Scalabilità

Un'applicazione basata su architettura a microservizi si presta naturalmente per essere scalata in modo orizzontale, ma in questo caso al contrario di quanto accade per l'architettura monolitica, è possibile replicare in più istanze solo i microservizi che hanno un carico eccessivo di richieste. In questo modo le istanze possono anche essere eseguite con hardware di basso costo riducendo i costi dell'infrastruttura e ottenenendo un livello di performance del nostro applicativo adeguato a qualsiasi esigenza di buisiness in termini di carico supportato, dato che la scalabilità orizzontale è sempre possibile e non vi sono limiti teorici al numero di istanza parallele che si possono creare dello stesso servizio. Un altro vantaggio che si ottiene grazie alla facilità con cui un microservizio può essere scalato è quello di non avere più un *single point of failure*, ma, avendo più repliche dello stesso servizio, in presenza di un problema hardware a un nodo, le funzionalità dell'applicativo non sono compromesse grazie alle repliche del servizio che sono in esecuzione su altri nodi.
Una delle principali problematiche legata all'architettura in esame è la gestione delle varie istanze dei microservizi, il bilanciamento del carico tra esse e la gestione della comunicazione tra servizi di tipo diverso. Infatti un componente deve poter fare una richiesta ad un'altro senza dover specificare un'istanza precisa, ad esempio specificando un particolare indirizzo. Inoltre le chiamate fatte ad uno stesso servizio devono essere adeguatamente bilanciate tra le istanze di quest'ultimo, altrimenti si perderebbero i benefici della scalabilità orizzontale. Per far ciò si utilizzano *sistemi di orchestrazione* che gestiscono in modo automatico il deploy, lo scaling,

il networking e il bilanciamento di carico delle applicazioni sviluppate. Una delle principali piattaforme utilizzate è Kubernetes, e ne verrà approfondita l'utilità e il funzionamento nei capitoli successivi.

3.3.3 Manutenzione e gestione errori

Le modifiche all'interno un software basato su microservizi sono frequenti ed è possibile gestirle più agevolmente rispetto al caso monolitico. Questo è dovuto al fatto che un componente all'interno di questa architettura ha per definizione una logica semplice ed essenziale perchè deve implementare una specifica funzionalità, quindi è facile per gli sviluppatori apportare modifiche in previsione di una nuova funzionalità che l'applicativo nel complesso dovrà implementare. Inoltre qual'ora venisse richiesta una nuova feature che il software dovrà implementare, la parte più importante della logica può essere implementata su un altro componente senza impattare significativamente le dinamiche preesistenti.
Per quanto concerne la gestione degli errori l'infrastruttura è completamente resiliente in quanto vi sono più istanze per la stesso componente, quindi nell'eventualità che un nodo si guasti il servizio è ancora disponibile.

3.3.4 Kubernetes e l'Orchestrazione dei Microservizi

All'interno di un'architettura a microservizi è fondamentale che vi sia la presenza di un elemento che coordini le varie istanze e gestisca la comunicazione tra microservizi. Ad oggi esistono varie piattaforme che permettono una gestione ottimizzata dei componenti, ma la più comune ed utilizzata è Kubernetes.

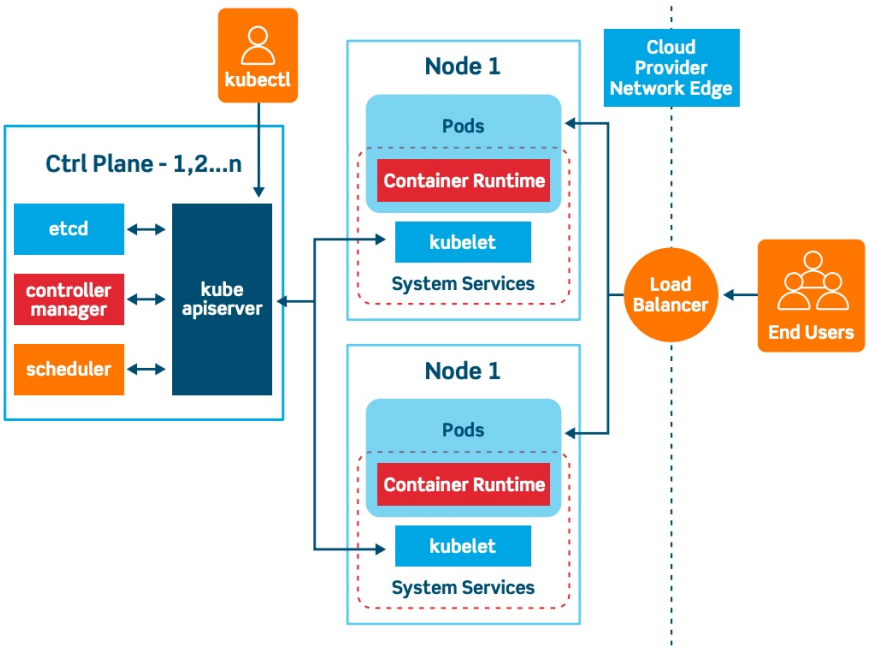

Figura 3.5: Schema architetturale Kubernetes

Kubernetes è una piattaforma che aiuta a gestire applicazioni containerizzate, come quelle eseguite in Docker, distribuendole su più macchine, dette nodi, in modo automatico. Questo orchestratore esegue un'architettura *master-worker*, in cui vi sono due livelli principali, il *Control Plane* che gestisce e coordina i nodi e i *woker-node* che si occupano di gestire i containers all'interno del nodo. Il Control Plane è composta dalle seguenti parti che hanno un compito ben specifico:

- **API Server**, punto di ingresso per i comandi utente per gestire i cluster e creare nuove istanze di un servizio .

- **Scheduler**, componente che decide su quali nodi (fisici o virtuali) avviare i container Docker.

- **Controller Manager**, controlla lo stato del cluster e reagisce a eventi (ad esempio cerca di instanziare i container su altri nodi, quando quello su cui erano in esecuzione termina a causa di guasti).

Inoltre, come detto brevemente in precedenza, all'interno di ogni nodo vi è un componente che gestisce i container. Nello specifico questa unità, chiamata worker-node è composta dalle seguenti parti:

- **Kubelet**, che comunica con il Control plane eseguendo i suoi comandi con il compito specifico di avviare i container all'interno del nodo quando e come richiesto.
- **Container runtime**, esegue i container all'interno dei nodi.
- **Kube Proxy**, gestisce il networking tra container.

Un pod può essere instanziato tramite il comando *kublectl* indicando come parametro un file YAML di tipo *deployment* dove vengono descitti all'interno i parametri e le risorse con cui il pod deve essere instanziato. Quest'operazione fa partire una richiesta verso l'API Server che salva i dati relativi al file in *etcd* il database distribuito che mantiene lo stato dell'intero cluster. Lo *Scheduler* monitora periodicamente etcd e rileva il nuovo Pod creato ma non ancora assegnato a nessun nodo, quindi ne sceglie uno adatto in base alle risorse disponibili e ad altre politiche di instanziamento configurabili su cui fare il deploy del pod richiesto, aggiornado l'oggetto Pod del database distribuito assegnandoli nodo selezionato. Il kubelet di quel nodo specifico, che interroga periodicamente l'API Server per capire se ci sono modifiche nel suo stato, rileva il cambiamento e di conseguenza recupera l'immagine del container dal repository specificato. Recuperata quest'ultima kubelet incarica il *container runtime* di eseguirla, notificando successivamente all'Api che il Pod è in stato *Running*.
Su ogni Pod quindi Kubernetes istanzia un servizio diverso. Quest'ultimi hanno la necessità di comunicare tra loro attraverso API, e per far ciò è possibile creare i Service, che sono un'astrazione di rete che implementano regole di routing per distribuire il traffico tra i Pod. Quando un Pod A vuole inviare una richiesta a un'applicazione B, il Pod A manda la richiesta all'IP indicato nel Service che identifica un cluster di Pod in esecuzione per gli applicativi di tipo B; questa richiesta arriva direttamente al kube-proxy del nodo su cui il Pod A è in esecuzione, il quale verifica le regole di routing per il Service corrispondente, identificando i Pod associati a quel Service attraverso un selettore che utilizza etichette, le quali sono coppie chiave-valore associate ai Pod e consentono di raggrupparli in base a caratteristiche comuni. Ottenuti questi Pod, il kube-proxy applica un algoritmo di bilanciamento del carico per selezionare uno dei Pod disponibili per gestire la richiesta e la inoltra all'indirizzo IP di quello scelto; il Pod B, che esegue l'applicazione di tipo B, riceve la richiesta e la elabora in base alla logica dell'applicazione, generando una risposta che viene restituita al kube-proxy, il quale la invia indietro al Pod A, completando così il ciclo della richiesta.
Kubernetes fornisce meccanismi di autoscaling per adattare dinamicamente le risorse di un'applicazione in base alla domanda. Le due principali strategie di autoscaling sono il Horizontal Pod Autoscaler (HPA) e il Vertical Pod Autoscaler (VPA). L'HPA regola automaticamente il numero di Pod in esecuzione in base a metriche

come utilizzo della CPU, della memoria o indicatori personalizzati. Il suo obiettivo è mantenere le prestazioni dell'applicazione garantendo che le risorse allocate siano sufficienti al carico di lavoro. L'HPA è applicato come una risorsa separata in Kubernetes e può essere configurato per definire il numero minimo e massimo di repliche per un Deployment attraverso un file yml di tipo *HorizontalPodAutoscaler* da applicare attraverso il solito comando *kublectl*. L'HPA utilizza il Metrics Server per raccogliere i dati sulle risorse consumate e, se il carico supera una soglia definita, aumenta il numero di Pod per bilanciarlo. Quando la domanda diminuisce, l'HPA riduce automaticamente il numero di istanze per ottimizzare l'uso delle risorse. A differenza dell'HPA, il VPA non modifica il numero di Pod, ma regola le risorse CPU e RAM assegnate a ciascun Pod. Questo è utile per applicazioni con carichi di lavoro variabili ma con un numero fisso di istanze. Inoltre le politiche di HPA e VPA possono essere usate insieme per garantire un bilanciamento ottimale delle risorse: l'HPA gestisce la scalabilità orizzontale, mentre il VPA adatta le risorse dei singoli Pod. Tuttavia, per evitare conflitti, Kubernetes consente di usare HPA con metriche basate su CPU solo se il VPA non modifica la CPU, oppure di combinare HPA su metriche personalizzate con un VPA che gestisce CPU e memoria. L'applicazione di HPA e VPA avviene attraverso file di configurazione YAML separati, che vengono forniti a Kubernetes tramite kubectl apply. Questi file definiscono le regole di autoscaling, le soglie di attivazione e i limiti minimi e massimi delle risorse o delle repliche.

3.3.5 Comunicazione Event-Driven tra Microservizi con Kafka

Per *Event-Driven Communication* si intende un particolare tipo di comunicazione in cui più microservizi comunicano in modo asincrono tramite eventi pubblicati e consumati attraverso un *message-broker*. Per la gestione degli eventi e il loro corretto instradamento vengono usualmente utilizzate piattaforme di supporto progettate per coordinare la produzione e il consumo dei messaggi da parte dei microservizi. Una tra le più utilizzate è **Kafka**. Quest'ultima è basata su un'architettura *publisher-subscribe* e memorizza i dati in modo persistente e scalabile ed è composta dalle seguenti componenti:

- **Producer**, che è un qualsiasi servizio che pubblica eventi su Kafka.
- **Consumer**, che è un qualsiasi servizio che legge eventi su Kafka.
- **Topic**, canali logici in cui i producer scrivono i messaggi e i Consumer li leggono.

- **Brokers**, nodi di Kafka che memorizzano e gestiscono i messaggi, instradandoli verso i consumer che hanno fatto la subscribe su un determinato topic in cui l'evento è stato pubblicato.
- **Partitions**, suddivisioni di un topic che permettono il parallelismo e la scalabilità nel processamento degli eventi.
- **Consumer Group**, insieme di consumer che collaborano per processare i messaggi di un topic in parallelo. Ogni consumer all'interno del consumer group è assegnato a una o più partizioni di un determinato topic, in modo che ogni partizione venga letta da un solo consumer per garantire l'elaborazione parallela e senza duplicazione dei messaggi.

Quando ci sono più istanze dello stesso microservizio che devono consumare dalla stessa Topic, Kafka bilancia automaticamente il carico tra di esse. In particolare, quando uno o più Producers pubblicano eventi su un Topic, questi eventi vengono distribuiti tra le partizioni in cui quest'ultimo è stato diviso. I Consumer appartenenti a un Consumer Group leggono gli eventi dalla partizione che è stata loro assegnata da Kafka ed elaborano i messaggi in modo parallelo, garantendo l'elaborazione scalabile e bilanciata.

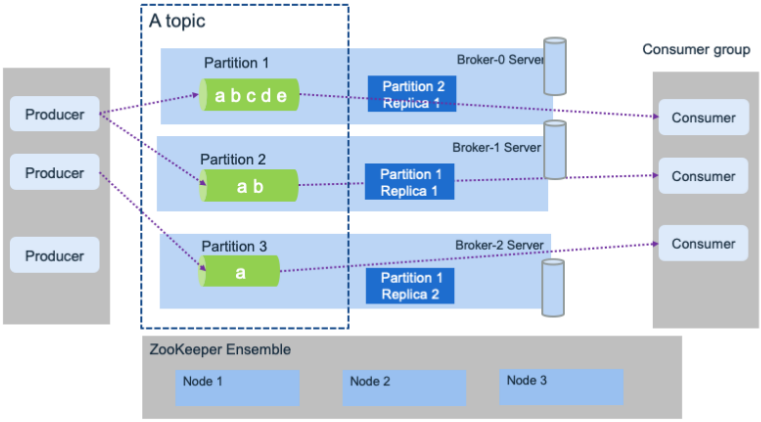

Figura 3.6: Rappresentazione scambio eventi tra Producers e Consumers

All'interno di questo sistema infatti, è la piattaforma che assegna preventivamente a ogni Consumer all'interno di un Consumer Group la partizione della Topic di interesse, con la regola che una partizione puo essere letta solo da un Consumer, ma quest'ultimo puo essere assegnato a più partizioni della stessa Topic. Inoltre da una coda possono consumare più Consumer Group in modo indipendente, ovvero un

evento E che arriva su una specifica partizione P su cui consumano due Consumer Group, CG1 e CG2, verrà letto sia da un Consumer all'interno di CG1 che da un Consumer all'interno di CG2. Come accennato precedentemente un broker è un server Kafka che memorizza i messaggi e gestisce la comunicazione con i producer e consumer. Quest'ultimo è parte di un cluster che può essere composto da uno o più nodi, dove ogni broker all'interno del cluster gestisce un certo numero di partizioni. Infatti le partizioni in cui viene suddiviso un Topic, possono essere distribuite su diversi broker nel cluster, questo consente di bilanciare il carico e migliorare la scalabilità. Inoltre per migliorare la tolleranza ai guasti ogni partizione puo essere replicata su più broker, per far questo per ogni partizione replicata Kafka designa un broker *leader* e uno o più *follower*, dove:

- **Leader**: E' il broker che gestisce tutte le operazioni di lettura e scrittura per quella partizione.
- **Follower**: Sono broker che mantengono copie della partizione leader per garantire la tolleranza ai guasti.

La gestione dei broker all'interno del cluster è svolta da *ZooKeeper* che è un ulteriore componente software che tiene traccia di quali brrker fanno parte di un determinato cluster, decidendo quali sono i leader e quali i follower per una determinata partizione. Inoltre questo applicativo monitora continuamente i broker, affinchè se uno di questi dovesse fallire, possa notificare a Kafka il guasto in modo che quest'ultimo possa assegnare un nuovo leader per le partizioni colpite che sono fallite [4].

Capitolo 4

Descrizione dell'Applicazione

4.1 Panoramica Generale

Come specificato nell'introduzione, l'applicativo sviluppato per questo lavoro di tesi è un chatbot automatico che consente di recuperare informazioni richieste tramite chat riguardo la documentazione software di progetti seguiti da una società di consulenza. Il software permette quindi sia la creazione di sotto-canali separati dedicati ad ogni progetto, sia l'upload all'interno del sistema dei file di documentazione software associandoli ad ogni canale. I documenti inseriti possono essere selezionati dall'utente in una fase preliminare alla conversazione in chat, per far scegliere all'utente su quali specifici testi, all'interno del canale selezionato, chiedere informazioni. Inoltre è data la possibilità all'utente di eliminare documenti inseriti all'interno di un progetto o eliminare l'intero canale.
L'interazione del chatbot con l'utente basata sulle informazioni dei file inseriti, è possibile grazie al paradigma RAG. Infatti una volta inserito il file, quest'ultimo sarà processato da un componente software dedicato che si occuperà di dividere il file in sottoporzioni in un determinato modo. Per garantire un'elaborazione efficiente delle informazioni in input, si è scelto di utilizzare il formato Markdown per i file in ingresso, poiché la sua struttura semplice ma versatile ne facilita l'analisi e la suddivisione in modo personalizzabile. Una volta che il testo in ingresso è diviso, l'applicativo si occuperà di generare gli *embeddings* relativi alle sotto-parti estratte dal file, prodotti grazie al supporto di un servizio di Encoding esterno, e salvare la coppia *embeddings - porzione di testo* all'interno di una entry nel *database vettoriale*. Durante la conversazione con il chatbot, dopo che l'utente ha inviato la propria domanda, l'applicativo genera l'embedding della richiesta e avvia una ricerca nel database vettoriale. In particolare, individua i 10 frammenti

di testo semanticamente più affini confrontando l'embedding dell'input con quelli archiviati, utilizzando la distanza L2 come metrica di similarità. Questi frammenti costituiscono il contesto su cui si baserà la risposta. Il contesto selezionato, insieme alla domanda dell'utente, viene quindi fornito in ingresso a un LLM, che elabora un output basandosi sulle informazioni ricevute.
Per garantire che le funzionalità descritte in precedenza fossero semplici e intuibili a livello di interfaccia utente, si è optato di implementarle all'interno di un chatbot Telegram. Inoltre, questa tipologia di interfaccia facilita il trasferimento di file tra il dispositivo dell'utente e l'applicativo, garantendo al contempo una rappresentazione ottimizzata di immagini e codice. Quest'ultimo viene formattato correttamente in base al linguaggio di programmazione specifico durante la generazione della risposta.

4.2 Struttura dell'Applicazione

Come descritto nel *Capitolo 3*, l'architettura di un'applicazione è un aspetto fondamentale dello sviluppo software. Decidere come l'intero sistema deve essere costruito è una scelta che va presa saggiamente poichè una volta che il software è in produzione è molto difficile, se non impossibile, ritornare sui passi percorsi cambiando architettura. Per lo sviluppo del *Documentation Chatbot* si è deciso di optare per un'architettura Event-Driven a microservizi. La motivazione dietro questa scelta sta nel fatto che grazie a questa configurazione, l'applicativo è in grado di reggere alti carichi di lavoro che possono presentarsi con intensità variabile nell'arco del tempo. Inoltre, separando le funzionalità tra i diversi microservizi è possibile potenziare in maniera mirata quelli che ne hanno una reale esigenza. Questo modello, per di più, consente di elaborare le richieste con elevate prestazioni, poiché i servizi possono operare in parallelo senza dover attendere la risposta di altri componenti, a differenza della comunicazione *API-based*.
Una delle motivazioni che hanno guidato questa scelta è il fatto che il progetto presenta ancora ampi margini di miglioramento e potrebbe richiedere l'integrazione di nuove funzionalità in futuro. Adottando questa soluzione architetturale, l'estensione del *Documentation Chatbot* risulterebbe più agevole, poiché, nella maggior parte dei casi, sarebbe sufficiente aggiungere un nuovo servizio dedicato alla funzionalità desiderata. Per questi motivi, insieme ad altri fattori rilevanti, si è ritenuto opportuno adottare l'approccio descritto.
L'applicativo è strutturato in due componenti principali: il *Frontend* e il *Backend*. L'interfaccia User, sviluppata in *Spring Boot*, utilizza l'API di Telegram per gestire l'interazione con l'utente, ricevendo le richieste e inviando le risposte del chatbot. Oltre a curare la comunicazione con la piattaforma di messaggistica, si occupa anche dell'elaborazione preliminare delle domande, inoltrandole ai servizi del *Backend*

per il recupero delle informazioni necessarie.
Il *Back-End* è composto da 4 servizi che hanno un compito ben definito:

- **API-interface**. Questo servizio ha il compito di esporre le Api verso il *telegram bot* e iniziare il flusso richiesto, instradando la richiesta del *Frontend* verso la giusta coda iniziando la vera elaborazione.

- **document-manager**. Questo componente ha il compito di processare il file markdown, dividerlo in frammenti semantici e metterli sulla coda verso l'**AI-adapter**.

- **AI-adapter**. Unità responsabile di generare gli *embeddings* relativi ai frammenti semantici e di generare le risposte dal contesto e la domanda in ingresso chiamando il LLM tramite API.

- **vector-database-adapter**. Questo componente si occupa di memorizzare gli embeddings generati da **AI-adapter**, insieme ai frammenti testuali ad essi associati, all'interno del database vettoriale. Durante l'interazione con l'utente, il sistema ricerca i chunks testuali nel database, confrontando gli embeddings salvati con quello derivante dal testo della richiesta dell'utente.

I servizi che compongono il *Back-End* comunicano tra di loro per l'elaborazione di determinati flussi tramite invio e ricezione di determinati eventi, utilizzando delle code **Kafka**. La comunicazione tra la parte *Front-End* e i componenti interni del *Back-End* avviene tramite **Rest-Api**, adottando la strategia architetturale del *Back-End For Front-End*. Questa strategia prevede la creazione di un'interfaccia che consente di adattare il flusso sincrono tipico del *Front-End* al flusso asincrono di un'architettura a microservizi. Nei paragrafi successivi verranno introdotte le tecnologie utilizzate all'interno del progetto e come queste sono state integrate all'interno dei servizi approfondendone per ciascuno il funzionamento.

4.3 Tecnologie Utilizzate

I microservizi che compongono l'applicativo utilizzano diverse tecnologie di supporto per svolgere i loro compiti. In questo paragrafo verranno analizzati questi strumenti, con una descrizione generale del loro funzionamento e dell'utilizzo all'interno del sistema. In particolare, vengono impiegati database relazionali, database in-memory per la sincronizzazione tra microservizi, database vettoriali per la memorizzazione e la rapida ricerca di frammenti testuali e dei relativi embeddings, oltre a sistemi che permettono la comunicazione tramite eventi asincroni. Infine, verranno descritte le principali caratteristiche del framework di sviluppo utilizzato, che consente l'integrazione dei vari servizi con queste tecnologie.

4.3.1 Framework di Sviluppo: Spring Boot

Spring Boot è il framework di sviluppo utilizzato nel progetto. Si tratta di un framework open source basato su Java, facente parte dell'ecosistema Spring, progettato per semplificare e accelerare lo sviluppo di applicazioni Java, con un focus particolare sulle applicazioni web e le architetture a microservizi. L'obiettivo principale di Spring Boot è ridurre la necessità di configurazioni manuali e codice boilerplate, fornendo funzionalità come configurazione automatica, server embedded e una gestione semplificata delle dipendenze.
In un'architettura a microservizi, Spring Boot si integra perfettamente con strumenti come *Spring Cloud*, *Kafka*, *Redis* e *Docker*, favorendo la comunicazione tra i vari servizi e la gestione distribuita dei dati. Il framework utilizza un meccanismo di auto-configurazione che determina dinamicamente le impostazioni necessarie in base alle dipendenze presenti nel progetto. Ad esempio, se nel file pom.xml viene inclusa la dipendenza spring-boot-starter-web (utile per sviluppare applicazioni web), Spring Boot configura automaticamente un Rest Template, un client per effettuare chiamate REST, senza necessità di configurazione manuale.
Inoltre, Spring Boot offre pacchetti chiamati Starter, che raggruppano tutte le dipendenze necessarie per specifici scenari applicativi e configurano automaticamente i Bean associati. Il framework integra anche server web come Tomcat, eliminando la necessità di configurare un application server esterno. Questo è particolarmente utile in ambienti containerizzati e serverless, dove i microservizi vengono eseguiti in ambienti isolati.
Una delle caratteristiche più vantaggiose di Spring Boot è la possibilità di utilizzare profili di configurazione. Ciò permette agli sviluppatori di definire variabili di configurazione specifiche per diversi ambienti (sviluppo, test, produzione, ecc.), semplificando la gestione delle configurazioni in base al contesto di esecuzione. Ad esempio, è possibile configurare il sistema per interagire con servizi diversi a seconda che si tratti dell'ambiente di sviluppo o di produzione.
Nel contesto di microservizi, Spring Boot rende più semplice l'implementazione e la gestione di applicazioni scalabili e distribuite, integrandosi facilmente con le tecnologie moderne necessarie per ottimizzare le performance e la comunicazione tra i vari componenti del sistema.

4.3.2 Database in-memory: Redis

Redis è un database in-memory che offre prestazioni ad alte performance per la gestione di dati strutturati. Si distingue per la sua capacita di memorizzare e recuperare dati in "tempo reale", rendendolo ideale per applicazioni che richiedono bassa latenza e alta velocità di accesso. Questo database supporta diverse strutture dati, tra cui stringhe, liste, set, hashset e sorted set, permettendo una gestione flessibile delle informazioni.

Uno degli utilizzi principali di Redis è come cache distibuita, riducendo il carico sui database tradizionali e migliorando le prestazioni delle applicazioni. Inoltre, grazie alle sue funzionalità di pub/sub e gestione dei lock distribuiti, Redis è ampiamente utilizzato per code di messaggi e, gestione delle sessioni utente e coordinazione di sistemi distribuiti. Nel contesto del lavoro di tesi questo tipo di Database è stato utilizzato principalmente per la coordinazione tra microservizi e cache a bassa latenza per il passaggio di dati tra servizi.

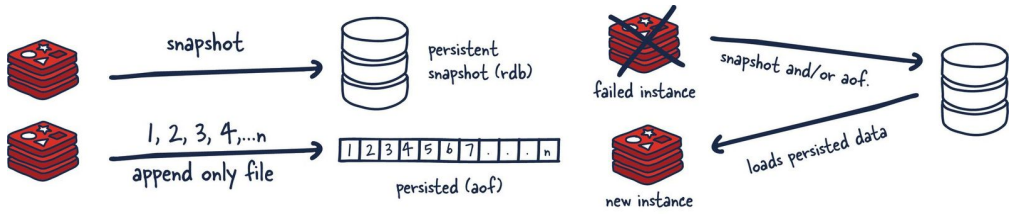

Figura 4.1: Rappresentazione meccanismi di persitenza in Redis

Redis per garantire affidabilità e persistenza dei dati, fornisce diversi meccanismi di replica, peristenza su disco e clustering, essenziali per prevenire la perdita di dati in caso di guasti.
Per salvaguardare la memorizzazione dei dati nel tempo, questo database, utilizza due principali strategie:

- Approccio basato su **Snapshotting** dei dati, dove Redis salva i dati su disco in un file binario RDB (Redis Database File) a intervalli regolari. L'uso di questo metodo riduce il consumo di memoria e l'impatto sulle prestazioni, poichè il salvataggio è gestito in background e permette il ripristino veloce del sistema in caso di riavvio di quest'ultimo. Questa soluzione è ideale per i sistemi in cui non è fondamentale salvare ogni operazione, ma si ha bisogno comunque di garantire un backup periodico.

- Strategia **Append-Only File (AOF)**, in cui Redis registra ogni opreazione di scrittura eseguita in un file log, garantendo un livello di persistenza maggiore rispetto all'RDB. In questo caso quindi ogni comando di modifica viene registrato sequenzialmente in un file log. Usando questa strategia di persistenza è consentita una configurazione per la sincornizzazione dati su disco in tre modalità differenti. Infatti l'amministratore può decidere tra:

  - una configurazione di tipo **always** in cui ogni comando viene registrato direttamente su disco garantendo una gestione sicura ma lenta della persistenza.

- un setting in modalità **everysec** in cui Redis scrive su disco tutti i comandi eseguiti ad intervalli di un secondo, questo tipo di controllo aumenta le prestazioni ma rende il ripristino meno sicuro.
- un'impostazione in cui delega la scrittura dei log al sistema operativo, rendendo il sistema complessivo molto più veloce ma meno sicuro.

Questo meccanismo garantisce una persistenza quasi totale, anche in caso di crash improvvisi, permettendo di ricostruire lo stato del database leggendo il log all'avvio, ma di contro può causare un rallentamento delle performance in caso di elevato traffico di scritture e un aumento di consumo d i spazio su disco.

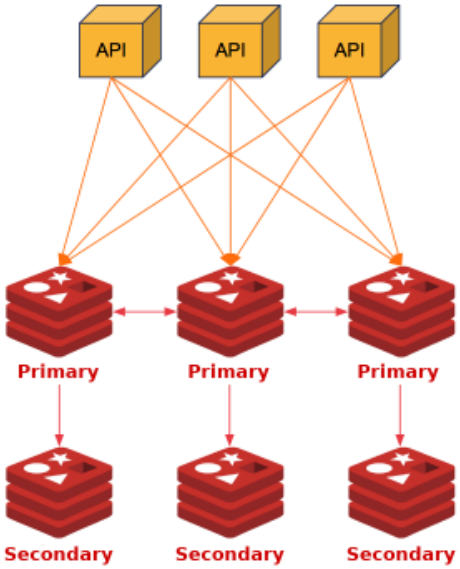

Figura 4.2: Struttura architetturale nodi Redis

Per garantire alta disponibilità e fault tolerance, Redis supporta un meccanismo di replica master-slave. Questo meccanismo funziona grazie ad un'istanza *Master* di Redis che gestisce tutte le operazioni di scrittura e una o più istance *Slave* che replicano automaticamente i dati dal master. In questo modo in caso di guasto, uno degli *slave* può subentrare come nuovo master. Il sistema descritto riduce il carico sulla macchina principale migliornado la scalabilità distribuendo le letture sui vari slave, inoltre garantisce anche la disponibilità dei dati in caso di guasti al nodo principale. Le repliche *slave* che Redis mette a disposizione vengono però aggiornate in modo asincrono, quindi in caso di fallimento improvviso del sistema si possono perdere alcune scritture, in più per avere il failover automatico delle repliche

Redis richiede l'uso di un altro servizio chiamato *Redis Cluster*. Quest'ultimo è un ulteriore componente software che l'eco-sistema Redis sfrutta, che permette di distribuire i dati su più nodi per garantire lo *shreding automatico* dividendo i dati tra più nodi evitando i colli di bottiglia, e il *fileover automatico* riassegnando i nodi automaticamente quando uno di questi fallisce [5].

4.3.3 Database Vettoriale: Milvus

Milvus è un database vettoriale open-source progettato per gestire e ricercare dati non strutturati convertiti in vettori numerici, noti come embeddings. Questa tecnologia consente di rappresentare informazioni complesse come testi, immagini e audio in formati numerici che ne catturano le caratteristiche essenziali, facilitando analisi e ricerche efficienti.

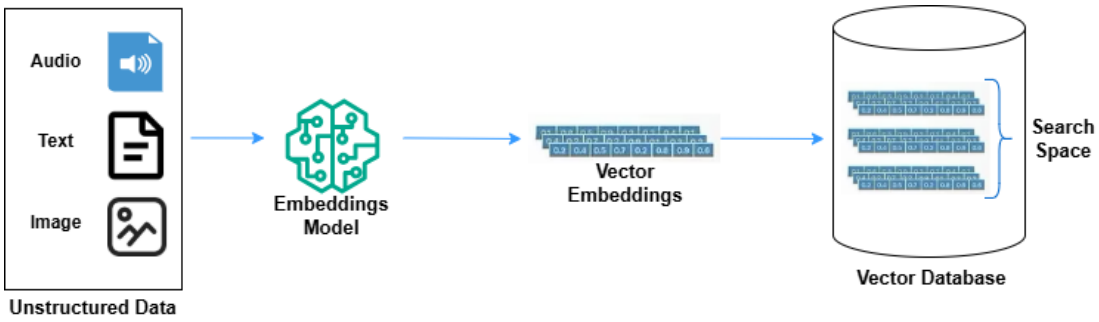

Figura 4.3: Sequenza di operazione per ricerca dati non -strutturati in Milvus

Le ricerche all'interno di un database vettoriale vengono eseguite confrontando la similarità tra l'embedding di input (target) e i vettori presenti nel database. Per fare questo, vengono utilizzate specifiche metriche di similarità che permettono di misurare quanto i vettori siano "vicini" tra loro in uno spazio multidimensionale. Milvus supporta nativamente diverse metriche per calcolare questa similarità, che permettono di ottimizzare e personalizzare la ricerca in base al tipo di dati e alle necessità specifiche dell'applicazione. Tra queste quelle più rilevanti sono:

- **Distanza Euclidea (L2)**, che misara la distanza lineare tra due punti nello spazio vettoriale.

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{4.1}$$

- **Prodotto Interno (IP)**, valuta la similarità basandosi sul prodotto scalare tra vettori.

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i \tag{4.2}$$

- **Similarità coseno (COSINE)**, dove per valutare la similarità viene calcolato l'agolo tra i due vettori confrontati, indipendentemente dalla loro magnitudine.

$$\cos\theta = \frac{\langle \mathbf{x}, \mathbf{y} \rangle}{\|\mathbf{x}\|\|\mathbf{y}\|} \tag{4.3}$$

  dove:

  - $\langle \mathbf{x}, \mathbf{y} \rangle$, è il prodotto interno dei due vettori.
  - $\|\mathbf{x}\|$ e $\|\mathbf{y}\|$ sono le norme euclidee dei vettori, calcolate come $\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}$.

Il motore di ricerca offre diverse modalità di ricerca per individuare i vettori simili all'interno del database. L'opzione di ricerca usata più comunemente è la *Top-K Search* che restituisce i K vettori più simili al vettore di query, ordinati in base alla distanza calcolata con la metriche di similarità scelta (L2, coseno, prodotto interno). Questa metodo di ricerca all'interno del database è quella che viene usata dal *Documentation Chatobot* per restituire i *top-10* frammenti più vicini alla rappresentazione vettoriale relativa alla domanda dell'utente. Di seguito verranno descritti ulteriori metodi di ricerca che Milvus implementa permette di usare:

- **Range Search**, metodo di ricerca che restituisce i vettori la cui distanza dal vettore di query rientra in un intervallo specificato. Risulta una scelta efficace quando si vogliono recuperare tutti i riultati simili entro una certa soglia di similarità senza specificare un numero fisso K.

- **Hybrid Search**, questa soluzione combina la ricerca vettoriale con filtri sui dati strutturati. Infatti permette di integrare condizioni booleane sui metadati associati ai vettori.

- **Multi-Vector Search**, questo metodo permette di effettuare una ricerca con più vettori contemporaneamente, ritornando un risultato combinato integrando meccanismi di aggregazione o di fusione.

- **Inverted File Index**, un tipo di ricerca che utilizza un indice approssimato per ridurre il numero di confronti tra vettori durante l'esplorazione. È molto utilizzato in caso di dataset di grandi dimensioni e per esigenze di alta velocità..

- **Brute-force Search**, metodo che confronta il vettore di query con tutti i vettori nel database. Questo approccio restituisce i risultati con la massima precisione, ma con un costo computazionale elevato.

Per velocizzare la ricerca all'interno del Database, Milvus supporta vari tipi di indici. La scelta dell'indice determina sia il metodo di ricerca all'interno della base dati, sia la struttura dati di supporto che il motore di ricerca crea implicitamente per effettuare quella ricerca. Nello sviluppo dell'applicazione si è deciso di usare un indice di tipo FLAT, che permette di effettuare una ricerca esaustiva confrontando il vettore di query con tutti i vettori all'interno del database. A primo impatto questa scelta potrebbe sembrare azzardata e inefficiente proprio perchè confrontare il vettore di query con tutte le entry disponibili potrebbe rallentare la ricerca, ma tenendo conto che quest'ultima viene effettuata per ogni richiesta, all'interno di una piccola sottoporzione del database (*partition*), si è deciso di optare per una soluzione che garantisse la massima precisione a discapito della velocità. Oltre all' indicizzazione di tipo FLAT, Milvus propone altre tipologie di indici, orientati ad una maggiore efficienza in fase di ricerca. Tra questi quelli più usati sono:

- **IVF FLAT**, tipo di indicizzazione che combina l'approccio dell'indice invertito (IVF) con la ricerca esaustiva all'interno di ciscun cluster. I vettori vengono suddivisi in cluster utilizzando l'algoritmo k-means, riducendo il numero di confronti necessari durante la ricerca. In pratica quando viene assegnato questo indice alla colonna associata alla rappresentazione vettoriale di un qualsiasi contenuto, Milvus esegue l'algoritmo del K-Means sulle entry all'interno del database, dividendo i vettori del database in N cluster e assegna ogni vettore al cluster più vicino in base alla distanza di quest'ultimo dai (*centroidi*). Dopodichè vene costruita una tabella che associa ogni vettore al cluster a cui appartiene.

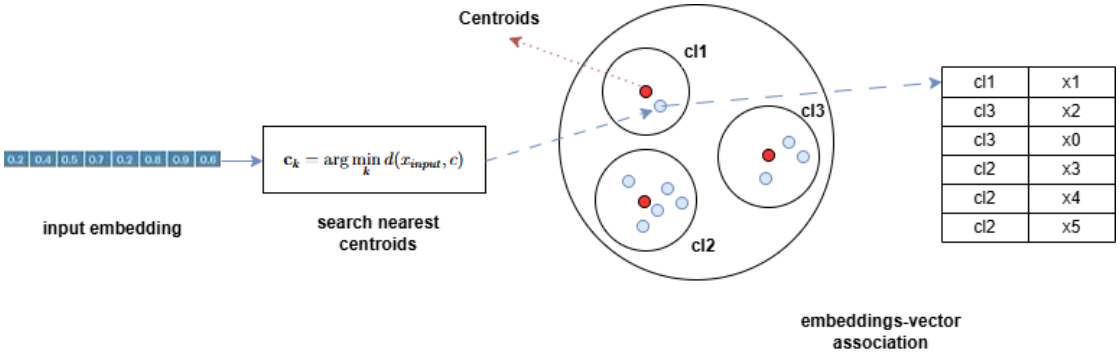

Figura 4.4: Processo costruzione indice IVF FLAT

Durante la fase di interrogazione, il vettore in ingresso viene inizialmente

confrontato con la rappresentazione vettoriale di ciascun cluster. Successivamente, in base al cluster più simile, viene eseguita una ricerca approfondita confrontando il vettore con tutte le rappresentazioni all'interno di quel cluster. In questo modo, il numero di confronti da effettuare si riduce notevolmente.

- **HNSV**, questo tipo di indicizzazione si basa sulla costruzione di un grafo navigabile che consente di trovare i vettori più vicini senza dover confrontare ogni elemento del database.L'idea del principio è di organizzare i vettori in un grafo dove ogni nodo rappresenta un vettore e i collegamentri tra i nodi rappresentano relazioni di vicinanza. Il grafo risultante è organizzato a livelli gerarchici dove i **livelli alti**, contengono nodi con connessioni che rappresentano una distanza grande tra i nodi a cui sono connessi (*connessioni lunghe*), mentre i livelli più bassi contengono nodi con connessioni che rappresentano un distanza più piccola tra essi (*connessioni strette*). In fase di ricerca, si

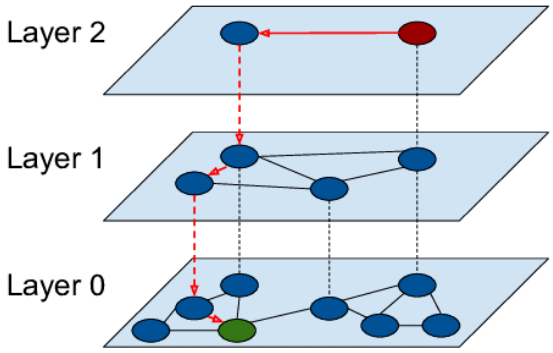

Figura 4.5: Ricerca tramite indicizzazione IVF FLAT

parte a confrontare il query vector con i vettori che si trovano in alto nella gerarchia, passando poi man mano ai nodi che si trovano nella parte più bassa seguendo le connessioni. Una volta arrivati ad un certo livello di profondità si fa la ricerca su quella zona del grafo per trovare i Top-K risultati più simili.

Questi due approcci di ricerca sono stati presi in considerazione durante lo sviluppo dell'applicativo. Tuttavia, si è optato per una ricerca esaustiva, poiché l'utente, selezionando i documenti su cui vuole interrogare il chatbot, restringe già implicitamente il sottoinsieme dei vettori. Di conseguenza, si è scelto di privilegiare la precisione della ricerca.

Dal punto di vista architetturale, la struttura di Milvus è basata su microservizi permettendo la scalabilità e alte performance. Esistono numerosi componenti, ognuno con un ruolo fondamentale. Di seguito verranno descritti i principali e le relative funzionalità. Il primo componente rilevante è il *Query Node*, responsabile

dell'elaborazione delle query sui vettori e del calcolo delle distanze, utilizzando gli indici per ottimizzare la ricerca. Quest'ultimo riceve richeste di ricerca dal *Query Coordinator* che provvede a smistare le richieste ai vari *Query Node*. Per la logica di memorizzazione dei dati Milvus utilizza 2 ulteriori servizi, il *Data Node* e il *Data Coordinator*. Il primo riceve nuovi dati, li compatta per migliorare le prestazioni e memorizza temporaneamente il risultato in segmenti che successivamente salva in un sistema di storage esterno, il secondo, ha il compito di sincronizzare la scrittura dei dati sullo storage persistente e coordinare il ciclo di vita dei segmenti. Infine, tra i componenti più importanti è necessario citare l' *Index Node*, che si occupa della creazione degli indici durante l'inserimento dei dati e l' *Index Coordinator* che invece coordina il bilanciamento di carico tra i vari nodi, decidendo quando e dove costruire i nuovi indici.

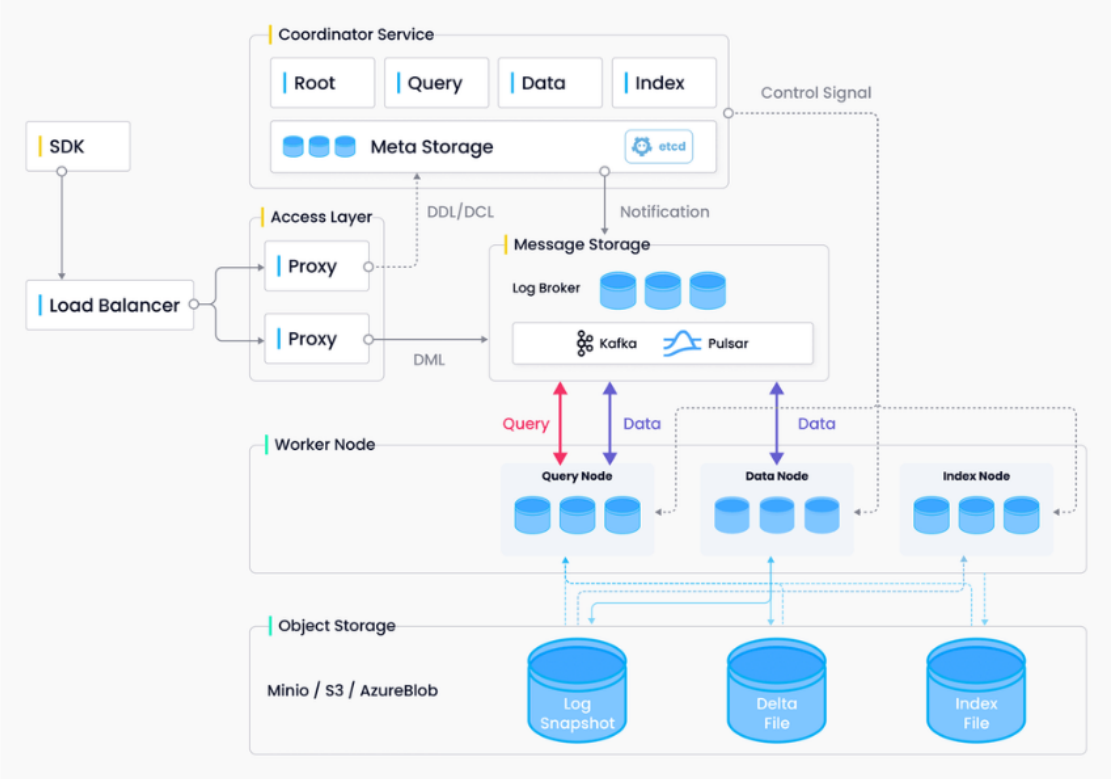

Figura 4.6: Schema architetturale dell' interazione servizi Milvus

È importante sottolineare che Milvus si basa su diversi servizi esterni per il suo funzionamento, tra cui **etcd**, utilizzato per la memorizzazione della configurazione del cluster e la sincronizzazione tra i nodi, e **MinIO**, un sistema di storage distribuito impiegato per salvare i segmenti in modo persistente [6].

4.4 Descrizione Microservizi Documentation Chatbot

L'applicativo prodotto per questo lavoro di testi, come detto precedentemente, è composto da 4 microservizi fondamentali che verranno descritti singolarmente nei paragrafi successivi.

4.4.1 Api-Interface

All'interno dell'applicativo quando il *Front-End* deve contattare il *Back-end* lo fa attraverso API-Rest. Quest'ultime vengono esposte dall'Api-Interface, che ha il compito di ricevere le richieste e instradare in modo asincrono gli eventi corrispondenti verso le giuste code per dare il via ai flussi dati necessari per soddisfare le richieste.

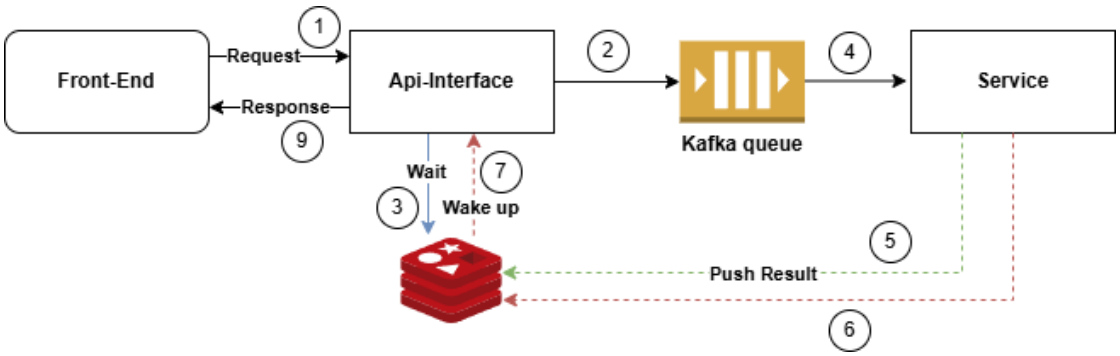

Figura 4.7: Diagramma di Flusso generico interazione Api Interface.

L'Api-Interface riceve una richiesta dal Front-end tramite REST-API, estrae i dati necessari e genera un evento. Questo evento viene quindi inoltrato al servizio responsabile dell'avvio del flusso di esecuzione specificato nel body della richiesta. Dopo aver avviato l'evento, l'Api-Interface entra in uno stato di attesa, sospendendosi su un latch registrato su Redis, associato all'identificativo del processo (processId) del flusso di esecuzione. Nel frattempo, il microservizio incaricato dell'elaborazione esegue il flusso e, una volta completato, memorizza il risultato in una mappa su Redis, utilizzando il processId come chiave e il risultato dell'elaborazione come valore. A questo punto, il microservizio rilascia il latch su cui l'Api-Interface era in attesa. Al suo risveglio quest'ultimo recupera i dati necessari dal database e invia la risposta al Front-end.

4.4.2 Document-Manager

Il *Document-Manager* è un microservizio che ha il compito di elaborare il documento che viene fornito dall'utente per poter essere inserito all'interno del Database Vettoriale. Per far questo il componente riceve l'identificativo del file all'interno di Google Drive per poterne fare il download. Ottenuto il file, il Document Manager si occupa di dividerlo in porzioni in modo personalizzato. La divisione in frammenti viene fatta sfruttando la struttura intrinseca dei documenti con cui spesso viene scritta la Documentazione Software. Infatti all'interno di un tipico documento ci sono dei Paragrafi che a sua volta sono divisi in sotto-paragrafi che a loro volta sono suddivisi in altri sotto-paragrafi. In altre parole, l'organizzazione del documento può essere assimilata a una struttura dati ad albero gerarchico, in cui ogni nodo rappresenta un paragrafo: i nodi figli corrispondono ai paragrafi contenuti all'interno di un paragrafo principale, mentre i nodi padre rappresentano i paragrafi che li includono. Si esegue una *Depth-First Search* (DFS) all'interno di questo albero per ottenere tutti i percorsi che collegano il nodo radice ai nodi foglia.

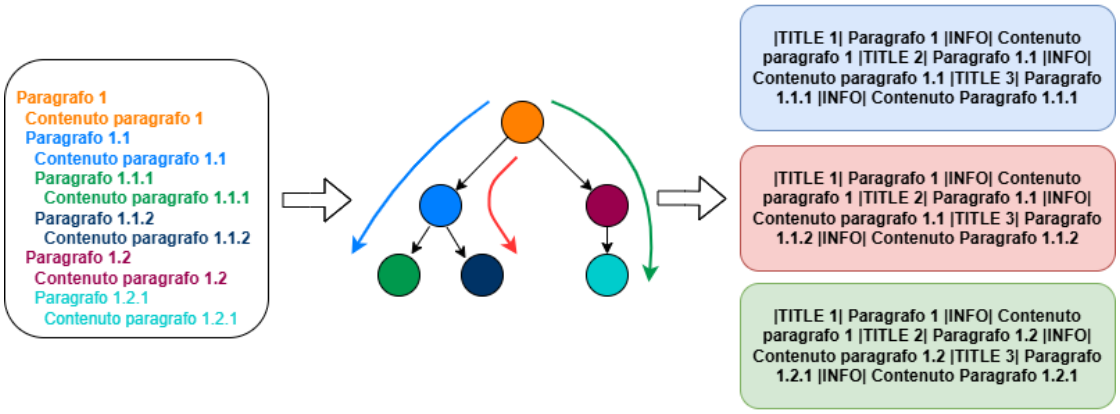

Figura 4.8: Processo di divisione documento in Input in frammenti testuali

Lavorando sui path ottenuti, e quindi seguendo ciascuno di essi dal nodo radice al nodo foglia, concatenando le informazioni che ogni nodo incamera in modo adeguato, si ottengono i frammenti di testo, come illustrato nella *Figure 4.8*. All'interno di un paragrafo è possibile trovare informazioni testuali, frammenti di codice, immagini e tabelle. Ogni Tipo di informazione è inserita all'interno del frammento testuale

associandola ad un Tag specifico, affinche il **LLm** abbia un'indicazione su come sfruttare i contenuti che gli arrivano in input. Di seguito verranno esposti i tag utilizzati e il relativo significato:

- **|TITLE LEVEL_NUMBER|**, tag che indica la presenza del titolo del paragrafo, come possiamo notare all'interno del tag è presente il *Level Number*, che indica la profondità.
- **|IMAGE|**, tag che indica la presenza di un'immagine ed è seguito dal *fileID* relativo a google Drive di quest'ultima. Viene chiesto infatti al modello di inserire nella risposta il *fileID* dell'immagine che appare in un frammento fornitogli, solo se lo ritiene necessario.
- **|TEXT|**, suggerisce la presenza di un contenuto testuale.
- **|TABLE|**, indica la presenza di una tabella, quest'ultima viene inserita all'interno del frammento in formato testuale Markdown.
- **|LIST|**, denota una lista all'interno del frammento anche questa inserita in formato testuale Markdown.
- **|CODE|**, evidenzia la presenza di codice, che può appartenere a qualsiasi linguaggio di programmazione. Poiché il file di input è in formato Markdown, che supporta nativamente una formattazione specifica per il codice, il contenuto associato al tag sarà anch'esso strutturato in Markdown.

Come si può intuire, i frammenti generati potrebbero avere parti in comune se fanno riferimento a sotto-paragrafi che hanno un paragrafo in comune.
Il vantaggio di avere una parte comune tra i diversi frammenti è che, in caso di richieste generiche su un determinato argomento, che farebbero riferimento a un paragrafo di livello superiore, il database vettoriale avrà maggiori probabilità di recuperare tutti frammenti contenenti quel paragrafo più generico. Questo permette di fornire al LLM un contesto più ricco e coerente, migliorando la qualità delle risposte generate.

4.4.3 VectorDatabase-Adpater

Quando l'utente pone una domanda al chatbot o aggiunge un nuovo documento come fonte di informazioni, è necessario un meccanismo efficace per conservare e recuperare i dati richiesti o inseriti, in modo da garantire risposte pertinenti e contestualizzate. Questo è il compito del *VectorDatabase-Adapter*, che è la parte software che si interfaccia con il motore di ricerca vettoriale *Milvus*, per rendere più fruibile il salvataggio e il recupero delle informazioni gestite.

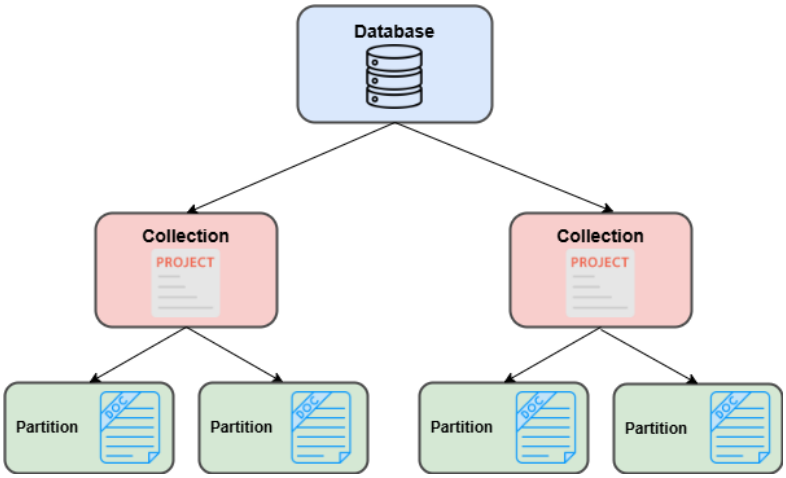

Figura 4.9: Organizzazione delle informazioni all'interno del Vector Database

Il VectorDatabase-Adapter ha il compito di organizzare in modo efficiente i frammenti dei documenti relativi alla documentazione software forniti dall'utente. Milvus consente di strutturare il database in unità chiamate *Collections*, le quali possono essere ulteriormente suddivise in *Partitions*. Questi sottoinsiemi logici permettono di ottimizzare la gestione dei dati e migliorare le prestazioni delle ricerche, riducendo il numero di confronti necessari durante le interrogazioni. Sfruttando queste strutture fornite da Milvus, il VectorDatabase-Adapter associa una Collection al progetto a cui fa riferimento la documentazione, mentre ogni Partition viene associata a un singolo file relativo alla documentazione di quel progetto. Questa organizzazione consente all'utente di interrogare il chatbot su documenti specifici all'interno di uno stesso progetto, aumentando l'efficacia e le prestazioni della ricerca.
Le informazioni presenti nei documenti caricati dall'utente vengono trasmesse al VectorDatabase-Adapter dall'AI-Adapter sotto forma di frammenti testuali, ciascuno accompagnato dai relativo embedding. Una volta ricevute le informazioni da salvare, il componente inizialmente si occupa di generare una nuova Partition per quel documento, all'interno della Collection specifica del progetto a cui si riferisce, dopodichè imposta uno schema con degli attributi specifici, come **embeddings** e **sentence** e infine costruisce un indice di tipo L2 su embeddings per rendere possibile la ricerca tramite distanza **L2** sul campo embeddings. Finita la parte di caricamento della Partitions vuota su Db inserisce in modo iterativo i dati forniti da AI-Adapter all'interno del DB.
In fase di richiesta da parte dell'utente il componente riceve dall'Ai-Adapter un embedding, che è la rappresentazione vettoriale della domanda dell'utente, il nome di una collections e di una o più partitions. Una volta ricevute queste informazioni, il suo compito è cercare all'interno del Database i Top-10 frammenti testuali i cui

embeddings sono più simili, in base alla metrica di similarità utilizzata, all'embedding relativo alla domanda dell'utente. Prelevate le informazioni dal database, queste vengono organizzate all'interno di un evento che viene mandato poi verso l'Ai adapter che formulerà una risposta testuale.

4.4.4 Ai-Adapter

L'applicativo, così come progettato, necessita di interfacciarsi con un provider di servizi per l'uso di LLM e modelli di generazione di embeddings, al fine di produrre le risposte per l'utente e generare gli embeddings relativi sia alle informazioni contenute nei documenti, che alle domande poste dagli utenti. Questo compito viene eseguito dall'**Ai-Adapter**, che ascolta gli eventi in ingresso dalla coda e, in base al tipo di flusso, contatta il provider per effettuare una richiesta al LLM o per generare gli embeddings relativi ai dati in ingresso.

Per implementare le funzionalità descritte è stata utilizzata la libreria *Spring-Ai*, una libreria di supporto per i task di Intelligenza Artificiale facilmente integrabile nell'ecosistema Spring, che permette un'interazione facilitata con i modelli di Ai generativa offerti da vari provider.
All'interno di questo applicativo, è stato scelto OpenAI come fornitore dei servizi di Generative AI, poiché offre una vasta gamma di soluzioni, dalle più economiche alle più costose, permettendo al cliente di selezionare quella più adatta alle sue necessità. Il Documentation Chatbot integra il modello di embeddings *Ada-002*, sviluppato da OpenAI, che è in grado di gestire in ingresso un vettore di massimo 8192 token e restituisce un embeddings con una dimensione di 1536. Questo consente di rappresentare frammenti di testo molto ampi con un singolo vettore numerico. Questo consente di rappresentare frammenti di testo molto ampi con un singolo vettore numerico.
Il modello linguistico impiegato è *GPT-3.5-turbo*, il quale supporta un input fino a 15.000 token e genera risposte testuali che possono arrivare fino a 32.000 token di lunghezza. Questo modello è stato scelto per il suo buon rapporto tra costi contenuti e la capacità di gestire un ampio numero di token in ingresso. Inoltre, l'architettura del sistema è stata progettata per essere flessibile, consentendo l'eventuale integrazione di modelli più avanzati in futuro, per migliorare le performance o adattarsi a nuove esigenze.
Per garantire che il LLM generi contenuti nel modo desiderato, ovvero rispondendo in maniera professionale a domande sulla Documentazione Software, è stato configurato un *System Prompt*. Questo prompt rappresenta un'istruzione predefinita fornita al modello di intelligenza artificiale per orientarne il comportamento e le modalità di risposta. In particolare, il *System Prompt* utilizzato è il seguente:

You are an AI assistant specialized in software documentation. Your primary goal is to provide detailed, accurate, and well-structured responses to the user's queries.
You must return the response exclusively in the following JSON format: {response_example}
Do not include any other text or formatting outside of the specified format above. Within the information provided to you for the responses, data regarding images and tables will be provided in the following format:
- '/IMAGE/->/id_image/' indicates an image with the identifier 'id_image'. You should provide the image ID as a string in the 'image_id' field of the JSON object. Do not include any Markdown formatting in the 'image_id' field. The 'image_id' should simply be the ID of the image (for example, 'image_12345'). If no image is associated with the chunk, leave the 'image_id' field empty as ".
- '/table/->/formatted_table/' represents a table with the specified content in 'formatted_table'. If this is relevant, incorporate the table in your response.
-' /CODE/->/code like json, python etc.. in markdown format/' represent the code inside the documentation.
- Any other text should be treated as explanatory information.
When generating a response, ensure:
1. You provide the 'image_id' in the JSON object without any Markdown formatting. The 'image_id' should just be a string representing the image identifier (for example, 'image12345') or " if no image is associated with the chunk.
2. You integrate the specified elements (images, tables) seamlessly into the response as if they were part of a professional document.
3. Your response is concise, well-structured, and written in professional English.
You also need to split your response into smaller chunks, where each chunk: 1. Contains at most 4076 characters.
2. Is formatted as a JSON object with three fields: - 'chunk_answer': A string containing the text of the chunk (maximum 4076 characters).
- 'image_id': A string that represents an image identifier. If there is no image associated with the chunk, use an empty string ". Each 'image_id' should be unique across all chunks.
- 'code' : A string containing the code in Markdown Format, in the same format in which they appear in the information
3. The chunks should be divided in a way that ensures the text is logically coherent and does not break in the middle of sentences or concepts.
4. Ensure that you do not exceed the token limit when dividing the text

and include all necessary elements in each chunk.
Your response should look like this: {response_list_example} Now,
process the following information: {info}

Come si può dedurre dal prompt, il LLm è configurato per ricevere in input una lista di informazioni, che vengono inserite al posto del placeholder *info*, e una lista di Dto (Data Transfer Object) in formato JSON, rappresentato dal placeholder *response_list_example*. Questi esempi servono a definire il formato di risposta richiesto. L'obiettivo è che il modello generi risposte nel medesimo formato, facilitando così la deserializzazione del risultato in un oggetto Java. L'uso di questi placeholder consente al chatbot di rispondere in modo strutturato e coerente con il formato previsto, garantendo che i dati possano essere successivamente elaborati correttamente come oggetti Java.
Grazie al prompt fornito il LLm è in grado di rispondere alla richieste dell'utente in formato JSON scrivendo in campi separati i frammenti di codice, il contenuto testuale ed eventuali *image_id*. In questo modo il modello viene impostato per rispondere cercando le informazioni necessarie esclusivamente dai frammenti forniti in input. Una volta ottenuta la risposta del modello questa viene deserializzata in un oggetto Java e mandata in avanti all'interno del flusso.

4.4.5 Telegram Bot

Questo componente costituisce il Front-End dell'applicativo e implementa la logica attraverso il quale il Bot Telegram risponde all'utente. Per la sua realizzazione è stato necessario utilizzare la libreria ***telegrambots-spring-boot-starter***, che può essere integrata facilmente nell'ecosistema Spring Boot grazie alle classi astratte che propone, che forniscono implementazioni utili per interagire direttamente con l'interfaccia del Bot su Telegram. La specifica interfaccia utilizzata è la *TelegramLongPollingBot*. Grazie all'implementazione di quest'interfaccia l'applicativo usa un meccanismo chiamato *long-pollling*, che consiste nell'inviare una richiesta HTTP al server di Telegram chiedendo aggiornamenti (updates). Il server di Telegram mantiene aperta la connessione finchè arriva un aggiornamento oppure finchè non passa un certo tempo massimo di attesa. Quando il server Telegram risponde con gli aggiornamenti il componente li elabora e poi manda una nuova richiesta per ricevere i prossimi aggiornamenti.
Il Telegram Bot riceve gli aggiornamenti dal server di Telegram, generati dalle interazioni degli utenti. L'applicazione memorizza i dati di ciascun utente in un database relazionale, includendo informazioni dettagliate tra cui lo user_id, il chat_id e lo stato dell'utente. Il sistema salva inoltre nel database le informazioni relative ai documenti caricati dagli utenti e i dettagli sui progetti, archiviandoli rispettivamente nelle tabelle **documents** e **projects**. Queste ultime sono collegate

da una relazione uno a molti, in cui un progetto può contenere più documenti, mentre ogni documento è associato a un unico progetto, rendendo la relazione unidirezionale dal progetto ai documenti.
Il Telegram-Bot gestisce gli utenti attraverso una macchina a stati. Quando un utente invia una richiesta, il sistema recupera dal database lo stato in cui si trovava e inizializza la macchina a stati partendo da quel punto. In base all'update ricevuto dall'utente, la macchina a stati avanza verso un nuovo stato, attivando un'azione che esegue una serie di operazioni legate alla logica dell'evento in arrivo, tra cui il salvataggio del nuovo stato dell'utente nel database.
La *figura 4.10*, mostra la macchina a stati che il Telegram-Bots implementa per gestire le operazioni degli utenti.

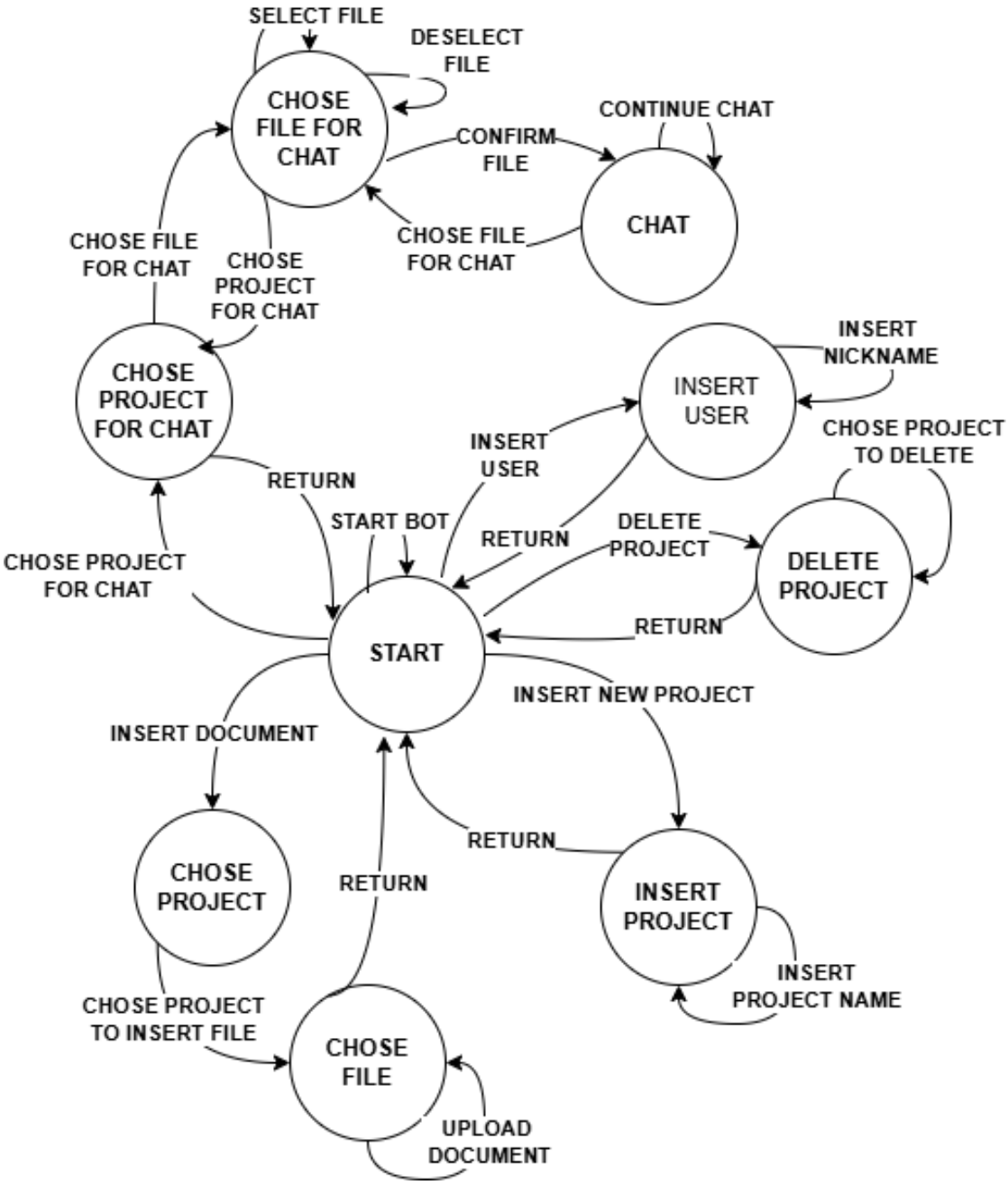

Figura 4.10: Macchina a stati per la gestione dell'Utente

Gli eventi che l'utente manda attraverso l'interfaccia telegram sono triggerati da una qualsiasi azione che esso compie. Ma la logica dell'applicativo fa si che il bot risponda solo a quelli consentiti. Ad esempio se lo stato in cui si trova l'utente prevede che quest'ultimo effettui una scelta tramite la pressione di un

button esposto, ma scrive del testo, questo viene ignorato facendo si che il bot rimanga in attesa di un evento previsto.
A seconda dello stato in cui si trova, l'utente può interagire con il bot premendo un pulsante mostrato dall'applicazione oppure inviando un messaggio di testo, come nel caso dello stato CHAT. Il componente risponde all'azione dell'utente inviando un messaggio nella chat corrispondente, che può contenere una tastiera con diversi pulsanti o un semplice testo. Questo messaggio guiderà l'utente nelle successive azioni possibili.

Capitolo 5

Cominicazione tra Microservizi

5.1 Panoramica flussi di comunicazione

In questo capitolo verranno analizzati i principali flussi di comunicazione tra microservizi che vengono eseguiti per soddisfare le richieste dell'utente. Tutti i processi di esecuzione partiranno dal *Telegram Bot* che comunicherà con l'*Api-Interface* per usufruire dei servizi di Backend. Durante un qualsiasi flusso di esecuzione l'Api-Interface si mette in attesa che l'ultimo servizio che lavora su quello specifico flusso carichi il risultato finale su Redis e segnali la fine del lavoro, come descritto nel paragrafo 4.4.1. Di seguito verranno descritte le principali operazioni che l'applicativo permette e il loro funzionamento.

5.1.1 Creazione istanza Progetto

Questo flusso di esecuzione permette all'utente di inizializzare un nuovo progetto, il quale funge da contenitore per tutti i documenti caricati e correlati a uno specifico progetto software.
A questo scopo, si registrano all'interno del database relazionale i metadati realtivi al progetto, all'interno dello schema projects, e si richiede al *VectorDatabase-Adapter* di istanziare una nuova collection all'interno di Milvus.

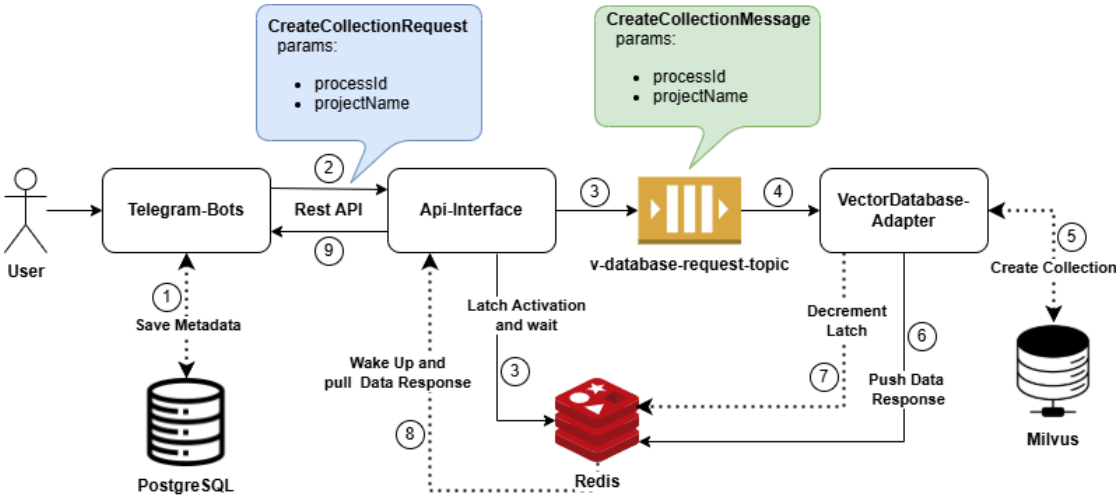

Figura 5.1: Diagramma di flusso: Creazione nuovo Progetto

La *Figure 5.1* rappresenta il flusso di creazione di un progetto all'interno del sistema. Inizialmente l'utente, esegue il comando di inserimento fornendo il nome del progetto e l'update arriva al Telegram-Bot. Quest'ultimo, in prima battuta salva i metadata relativi al progetto all'interno dello schema *projects*, poi esegue una richiesta HTTP verso **Api-Interface**, inserendo un *processId* e il *projectName* all'interno del body.
Quando l'Api-Interface riceve la richiesta, genera un *evento* all'interno del quale inserisce le informazioni del request-body in input e lo inoltra sulla coda, dopodichè crea un Redis latch e lo incrementa, caricandolo successivamente su una RedisMap all'interno di una entry che ha come key il *processId* legato al flusso e si mette in attesa.
Quando l'evento raggiunge il VectorDatabase-Adapter, viene elaborato per estrarre il projectName, che viene utilizzato per creare una collection con lo stesso nome all'interno di Milvus. Una volta completata la creazione della collection, il componente salva il risultato dell'operazione in una RedisMap condivisa, dove la key è il processId e il value è il *dto-response* associato. Successivamente, il componente decrementa il latch precedentemente inserito dall'Api-Interface, recuperandolo dalla mappa dedicata ai latch tramite il processId.
A questo punto l'Api-Interface è operativo è può recuperare da Redis il risultato dell'operazione e tornarlo al Telegram-Bot.

5.1.2 Inserimento Documento

Il processo di Inserimento Documento consente all'utente di caricare un file in formato Markdown nel sistema, affinché il chatbot possa utilizzare i contenuti

di tali documenti durante la generazione delle risposte. Anche in questo caso il Telegram-Bot ha il compito di registrare i metadati relativi al documento e di metterli in relazione con il progetto a cui appartengono. A tale scopo verrà chiesti all'utente, in fase di inserimento, all'interno di quale progetto caricare il documento tra quelli disponibili.

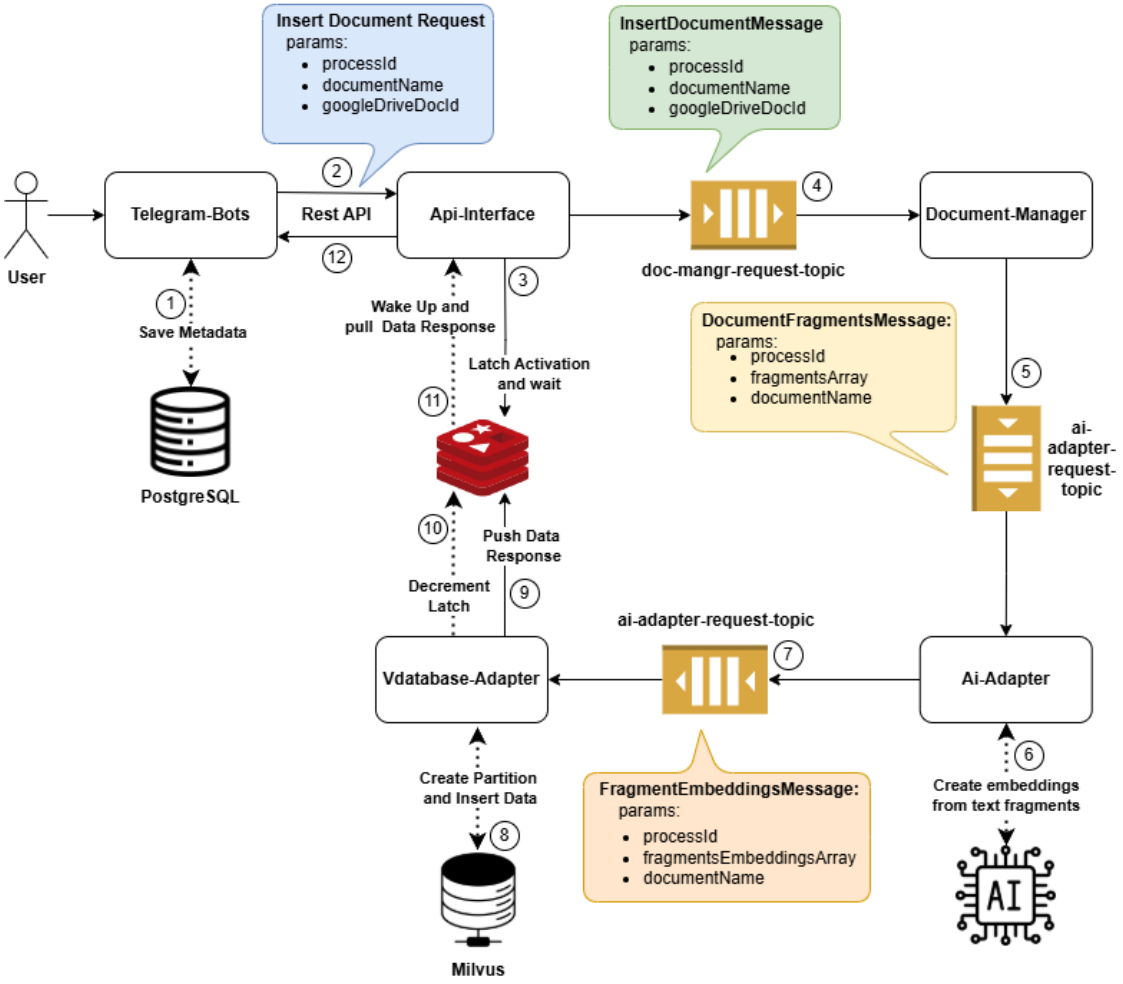

Figura 5.2: Diagramma di flusso: Inserimento Documento

Come mostrato nella *Figure 5.2*, il **Telegram-Bot**, dopo aver salvato i metadati nel database relazionale, invia una richiesta HTTP di tipo POST all'Api-Interface. Nel request body vengono inclusi i dati essenziali per la corretta elaborazione della richiesta, come il *processId*, il nome del documento e il *googleDriveDocId*. Quest'ultimo rappresenta l'identificativo con cui i componenti possono scaricare il file dal Google Drive, dove il Telegram-Bot lo ha precedentemente caricato..

L'Api-Interface, una volta ricevuta la richiesta, inoltra un evento sulla coda monitorata dal *Document-Manager*, ed entra nello stato di attesa. Questo evento contiene le stesse informazioni presenti nel request body, garantendo così il corretto trasferimento dei dati necessari per l'elaborazione.
Una volta ricevuto il messaggio, il Document-Manager scarica il documento da Google Drive utilizzando il *googleDriveDocId*, estrae i frammenti come descritto nel paragrafo 4.4.2 e li organizza all'interno di un vettore. Successivamente, il componente invia un messaggio sulla coda monitorata dall'*AI-Adapter*, includendo il vettore con tutti i frammenti, il processId e il documentName.
L'*AI-Adapter* elabora il messaggio in ingresso generando un embedding per ciascun frammento testuale, utilizzando un servizio esterno specializzato nella creazione di embeddings per contenuti testuali. Successivamente, associa ogni frammento alla propria rappresentazione vettoriale e inserisce le coppie risultanti all'interno di un vettore. Dopo questa fase di elaborazione, il microservizio costruisce un nuovo messaggio contenente il vettore con i frammenti testuali e i rispettivi embeddings, il processId e il nome del documento. Infine, inviando infine il messaggio sulla coda osservata dal *VectorDatabase-Adapter*.
Quest'ultimo, alla ricezione del messaggio, crea una *partition* su MIlvus, assegnandole il nome indicato dal campo **documentName**, all'interno della *collection* indicata da **projectName**. All'interno di questa partizione, il *VectorDatabase-Adapter* inserisce i frammenti testuali insieme ai relativi embeddings ricevuti, oltre ad altre informazioni di utilità. Al termine del processo, il componente notifica la fine dell'elaborazione all'Api-Interface, che a sua volta informa il Telegram-Bot riguardo l'esito dell'operazione. All'utente infine verrà segnalato il risultato del processo, proponendo inoltre a quest'ultimo un button per farlo ritornare al menù.

5.1.3 Chat con Utente

Il flusso descritto in questo paragrafo consente all'utente di porre delle domande al chatbot riguardo uno o più documenti selezionati, tutti legati allo stesso progetto. In una fase preliminare, all'utente verrà chiesto di selezionare il progetto e i documenti al suo interno su cui desidera porre le domande. Una volta completata questa fase, viene avviata la sessione di chat, durante la quale l'utente potrà scrivere il testo della domanda.

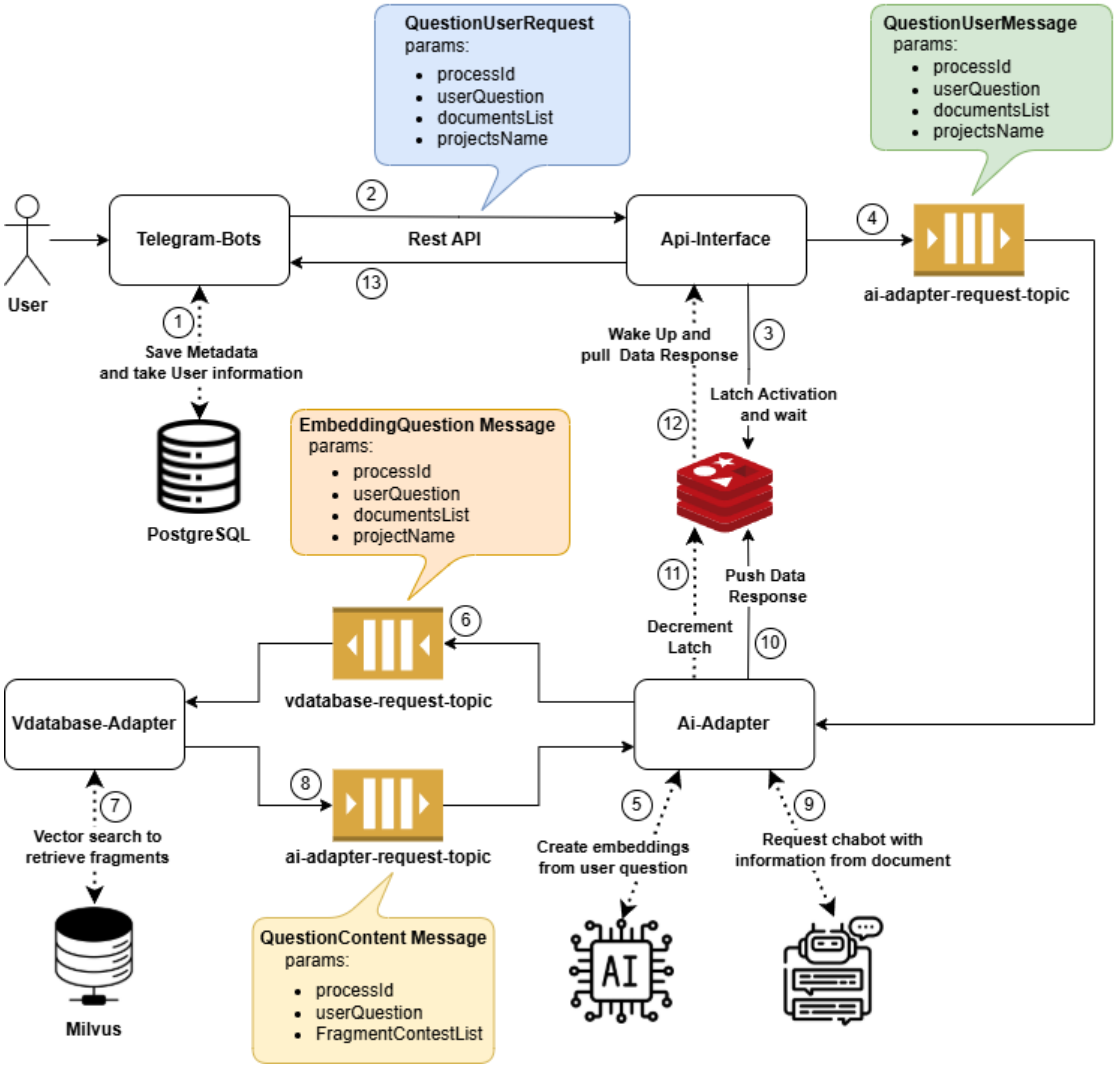

Figura 5.3: Diagramma di flusso: Chat con Utente

Quando l'utente invia la domanda, il Telegram-Bot invia una richiesta HTTP all'Api-Interface, includendo nel request body il processId, la domanda dell'utente, i documenti selezionati su cui è posta la domanda e il progetto a cui appartengono quei documenti. L'Api-Interface, come anche negli altri casi, crea un messaggio a partire dai dati che riceve in input e lo inserisce all'interno di una coda su cui l'Ai-Adapter è in ascolto.
Una volta ricevuto l'evento, l'Api-Interface estrae la domanda dell'utente e chiama il servizio di generazione degli embeddings per ottenere la sua rappresentazione vettoriale. Completata questa operazione, il microservizio prepara un messaggio da inviare al VectorDatabase-Adapter, includendo gli embeddings della domanda alle

informazioni precedentemente ottenute in input.
Alla ricezione del messaggio il *VDatabase-Adapter*, esegue una ricerca all'interno del database vettoriale cercando i *Top-10 frammenti testuali* semanticamente più *"vicini"* alla domanda dell'utente, utilizzando una metrica di similarità di tipo L2. I frammenti cosi ottenuti compogono il contesto dal quale il LLm attingerà per generare la risposta. Il lavoro del microservizio si conclude organizzando innanzitutto tutti i frammenti estratti all'interno di un vettore in ordine di rilevanza. Successivamente, crea un messaggio contenente la domanda dell'utente, il vettore di contesto e il processId, e infine inserisce il messaggio nella coda su cui l'*AI-Adapter* è in ascolto.
Ottenuto l'ultimo messaggio descritto, l'*AI-Adapter* estrae il contesto e la domanda dell'utente, e invia una richiesta al LLM utilizzando questi dati. Una volta ricevuta la risposta, l'AI-Adapter la serializza all'interno di un oggetto Java, come descritto nel paragrafo 4.4.4, e inserisce il risultato in una RedisMap dedicata. Successivamente, notifica la fine del processo all'Api-Adapter, che, una volta terminato lo stato di attesa, restituisce la risposta generata al Telegram-Bot.

5.1.4 Rimozione Progetto o Documento

Quando l'utente vuole eliminare un progetto o un documento inserito, puo farlo attraverso il menù esposto su Telegram indicando quale progetto oppure, quale documento all'interno di un progetto eliminare.

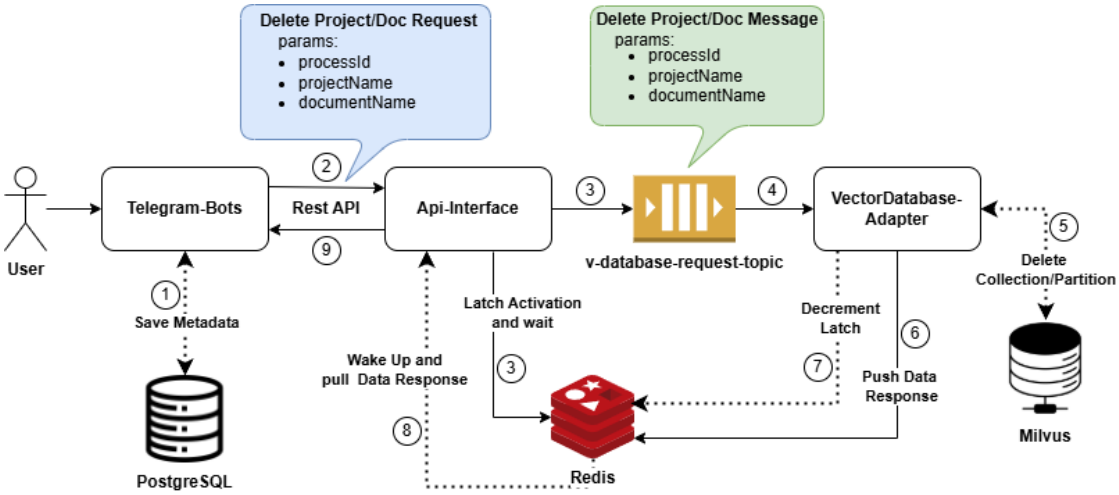

Figura 5.4: Diagramma di flusso: Rimozione Documento

In questo caso il Telegram-Bot fa una richiesta all'Api-Interface inserendo all'interno del request Body il nome del progetto e il documento (nel caso in cui si volesse cancellare un documento).

L'Api-Interface, una volta ricevuta la richiesta, inoltra un messaggio sulla coda in cui il VectorDatabase-Adapter è in ascolto. Alla ricezione del messaggio, il VectorDatabase-Adapter estrae il nome del documento e del progetto, nel caso in cui l'utente abbia richiesto l'eliminazione di un documento. Utilizzando queste informazioni, il microservizio invia una richiesta a Milvus per l'eliminazione del documento o del progetto. Successivamente, il VectorDatabase-Adapter notifica all'Api-Interface il completamento dell'operazione. Come nei flussi precedenti, l'Api-Interface inoltra la risposta al Telegram-Bot, che a sua volta invia una risposta appropriata all'utente.
Il flusso descritto riguarda l'eliminazione di un documento all'interno del sistema, ma le operazioni per eliminare un progetto sono molto simili. Pertanto, non verrà trattato il caso specifico per l'eliminazione di un progetto.

Capitolo 6

Test e Validazione

6.1 Test Software

6.1.1 Test di Unità

Per verificare le singole funzionalità dei microservizi sono stati implementati gli *unit-test*. Quest'ultimi sono una tecnica di testing del software in cui singole parti di un programma vengono testate in isolamento per verificarne il corretto funzionamento. I test di unità non servono solo per garantire che le funzionalità implementate reagiscano come previsto, ma servono anche per prevenire le regressioni in caso di modifiche successive, documentare trasversalmente il comportamento del codice mostrando come ci si aspetta che una funzione si comporti in diversi scenari e può migliorare generalmente la progettazione del software poichè spingono a creare codice più modulare e meno dipendente da altri componenti. All'interno del progetto i test di unità sono stati implementati utilizzando la nota libreria **JUnit**, fondamentale per la stesura dei test e l'implementazione della loro logica.

6.1.2 Test di Integrazione

Per verificare che i microservizi comunicassero correttamente tra loro, sono stati effettuati dei *test di integrazione*. Questo tipo di tecnica verifica il funzionamento correto dell'interazione tra più unità di codice o moduli, concentrandosi nel testare il comportamento del sistema quando questi moduli collaborano tra loro. Questo tipo di test sono necessari per verificare problemi di comunicazione tra componenti, infatti, anche se i singoli microservizi funzionano correttamente, potrebbero esserci problemi nella loro comunicazione. Inoltre verificano anche l'integrazione con i Database o servizi esterni con cui un microservizio potrebbe interagire, funzionalità che non possono essere testate tramite test di unità.
Vi sono varie tipologie di test di Integrazione, che si differenziano tra loro per

l'approccio che lo sviluppatore ha durante la loro esecuzione. L'approccio utilizzato per testare il progetto in questo caso è stato il *test Bing Bang*. Questo approccio prevede che tutti i microservizi e i servizi esterni a cui questi sono associati vengano testati insieme. Infatti quello che è stato concretamente fatto è stato testare tutti i flussi descritti nel capitolo procedente tenendo attivi tutti i componenti.

6.2 Validazione Algoritmo di Retrieve

La logica più importante presente all'interno del progetto è quella dell'algoritmo di retrieve dei frammenti di testo presenti all'interno del database vettoriale più semanticamente vicini alla domanda dell'utente. Per constatare la qualità dell'algoritmo esistono in letteratura diverse metriche che si possono adattare per valutare questo algoritmo spcecifico. Nei successivi paragrafi verranno spiegate le metriche esistenti utilizzate, e le modifiche che sono state effettuate su quest'ultime affinche fossero più aderenti alla valutazione dell'algoritmo nel caso specifico del Documentation Chatbot.

6.2.1 Dataset

Il Dataset utilizzato per la validazione dell'algoritmo di retrieve è composto da 100 campioni. Ogni campione è formato da:

- Domanda di Test.
- Embedding della domanda di Test.
- Lista degli IDs dei frammenti testuali che occorrono per rispondere alla domanda.

Il Dataset cosi composto fa riferimento ad un set di documenti di Test che rappresentano la documentazione del DocumentationChatbot stesso, caricati all'interno dell'appicativo.
La collezione di campioni, per quanto modesta, contiene domande di diversa natura che ricoprono gran parte della documentazione di Test che si vuole interrogare ed è composta, sia da domande specifiche a cui per rispondere è necessario un frammento specifico, che da domande di natura generica a cui per rispondere è necessario ottenere diversi frammenti. Il Dataset cosi composto è stato utilizzato per valutare l'algoritmo attraverso 2 diverse metriche che verranno descritte in seguito.

6.2.2 Recall@K

Questa metrica è molto utile nel contesto della valutazione delle performance del DocumentationChatbot poichè misura la frazione di documenti rilevanti che sono stati effettivamente recuperati tra i primi K risultati. Per ogni sample all'interno del Dataset si viene effettuata la misura nel modo che segue:

$$Recall@K_{(i)} = \frac{\#frammenti\ rilevanti\ tra\ i\ primi\ K\ risultati}{\#totale\ di\ documenti\ rilevanti\ disponibili} \tag{6.1}$$

Dove il numeratore indica i primi K frammenti di testo recuperati dal motore di ricerca vettoriale e il denominatore corrisponde a $len(y_{true})_{(i)}$, ovvero il numero di frammenti testuali che occorrono per rispondere correttamente alla domanda riportata nel sample $i - esimo$. Quindi per avere una valutazione complessiva della $Recall@K$ su tutto il Dataset proposto, vengono sommati tutti i valori di $Recall@K_{(i)}$ calcolati sui singoli sample e poi viene divisa la somma totale per il numero di campioni presenti [7]. Il calcolo complessivo della metrica viene eseguito quindi come segue:

$$Recall@K = \frac{\sum_{i=0}^{NSample} Recall@K_{(i)}}{NSample} \tag{6.2}$$

In questo modo viene ottenuta una valutazione complessiva secondo la metrica $Recall@K$ dell'algoritmo di retrieve implementato su tutto il Dataset.

6.2.3 Positional Coverage Score

Nel tentativo di dare una valutazione anche sulla qualità della logica di retrive, cercando quindi di dare un punteggio sulla base dell'ordine di importanza con cui l'algoritmo propone i frammenti ricercati, è stata implementata una metrica custom che prende il nome di *Positional Coverage Score*.

Questa metrica opera su ogni sample dividendo i Top-10 chunks recuperati dal Database in n Set differenti. La divisione prevede che nel primo Set vengano inseriti i primi $len(y_{true})$ frammenti presenti nella Top-10 restituita dal motore di ricerca mentre gli $n - 1$ Set successivi vengono costruiti dividendo la restante parte della Top-10 in uguale misura tra i diversi Set. Ogni Set così costruito avrà un punteggio associato proporzionale alla posizione che la divisione occupa all'interno della Top-10. Se il frammento j-esimo all'interno di pred $y_{pred_{(j)}}$ è contenuto nel Set $i - esimo$, gli verrà assegnato il punteggio corrispondente al Set.

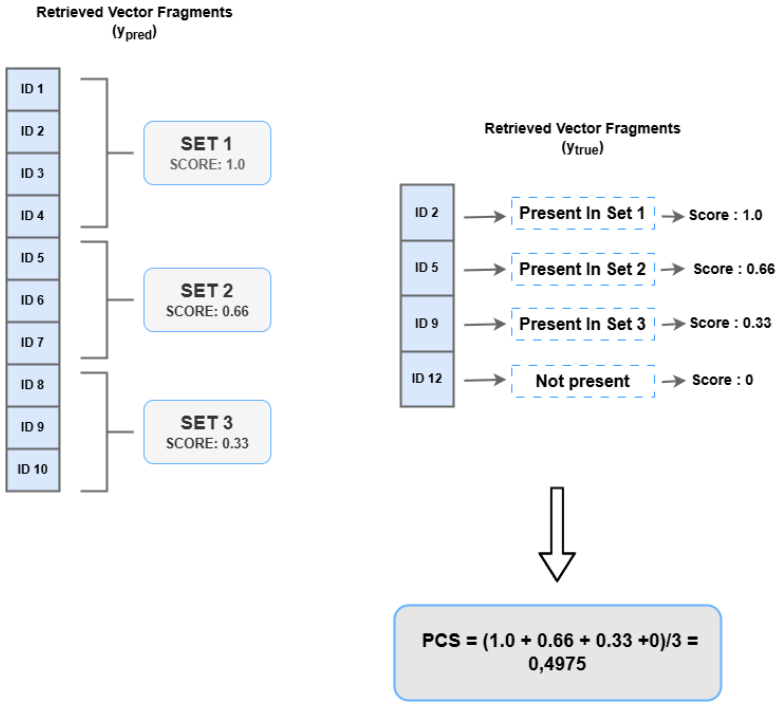

Figura 6.1: Rappresentazione calcolo PCS per sample *i-esimo*

Una volta ottenuti i punteggi per ogni frammento all'interno di y_{pred} questi vengono sommati e normalizzati per la lunghezza di y_{pred}, come riportato nella formula generale *(6,6)*. La valutazione dell'algoritmo attraverso il *Positional Coverage Score* sul Dataset complessivo viene calcolata come segue:

$$PCS_j = \frac{\sum_{i=0}^{NSample} |FragmentSet_{(i)} \cap y_{true_{(j)}}| * score_i}{len(y_{true_{(j)}})} \tag{6.3}$$

dove:

- $FragmentSet_{(i)}$ è il set i-esimo in cui vi è una parte della previsione.
- $y_{true_{(j)}}$ è il vettore *j-esimo*, prelevato dal Dataset, contenente i frammenti testuali necessari per rispondere correttamente alla domanda.
- $score_i$ è il punteggio assegnato al frammento predetto se è contenuto all'interno del $FragmentSet_{(i)}$.
- $len(y_{true_{(j)}})$ è il numero di frammenti testuali presenti all'interno di $y_{true_{(j)}}$.

6.2.4 Risultati e valutazioni

L'algoritmo di Retrieve è stato valutato utilizzando entrambe le metriche citate nel paragrafo precedente. La *Recall@K* è stata calcolata impostando il parametro K = 4. In questo modo, il valore della metrica descrive la qualità dell'algoritmo nel ritornare i frammenti di testo più rilevanti all'interno delle prime 4 posizioni della Top-10 ritornata. E' stato deciso di settare K=4 per la valutazione di questa metrica, poichè all'interno di un sample nel Dataset, la lunghezza massima possibile è $len(y_{true}) = 4$. Questa metrica ha riportato un risultato di 0.95 sul Datset costruito dimostrando l'efficacia del recupero delle parti di testo più rilevanti.
Per una valutazione più approfondita l'algoritmo di retrieve è stato utilizzata la *Positional Coverage Score*. La metrica è stata impostata con $NSet = 3$ e con il seguente vettore di punteggio [1,0.3,0]. Questa impostazione è stata scelta per avere una valutazione più puntuale dell'ordine con cui i frammenti più rilevanti vengono restituiti all'interno della Top-10, premiando l'algoritmo se questi cadono all'interno delle prime $len(y_{true})$ posizioni, e dando punteggi minori se invece rientrano nei Set che coincidono con posizioni più basse della classifica. L'algoritmo di Retrieve valutato con *Positional Coverage Score*, con il setup descritto precedentemente ottiene un punteggio di 0.85 sul Dataset creato. Questo punteggio dimostra la qualità dell'algoritmo nel comprendere le informazioni più rilevanti che sono necessarie per rispondere alla domanda di un utente.

Capitolo 7

Conclusioni

7.1 Risultati raggiunti e stato dell'applicativo

Allo stato attuale degli sviluppi l'applicativo progettato è in grado di rispondere alle domande degli utenti sulla documentazione caricata in modo efficiente, proponendo a quest'utimo risposte in vario formato come immagini, codice e testo.

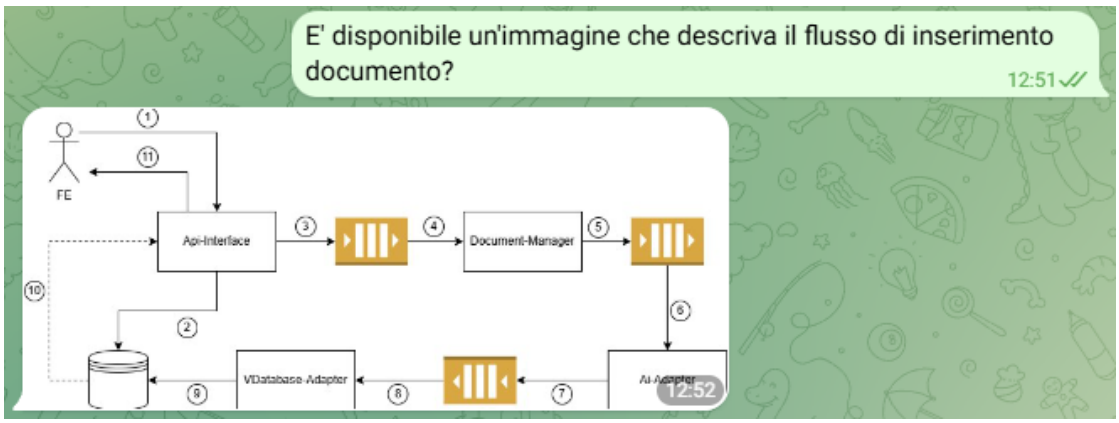

Figura 7.1: Esempio Risposta Chatbot in cui l'utente richiede immagine relativa a Flusso di Inserimento Documento.

Inoltre grazie all'algoritmo di retrieve dei frammenti testuali, e a come questi sono stati costruiti prima di essere inseriti nel Database, al LLM può essere fornito in fase di risposta un contesto più ampio e coerente per poter rispondere correttamente a domande di carattere generico, come mostra la *Figure 7.2* o a domande che contengono intrinsecamente più richieste come nell'esempio illustrato nella *Figure 7.3*.

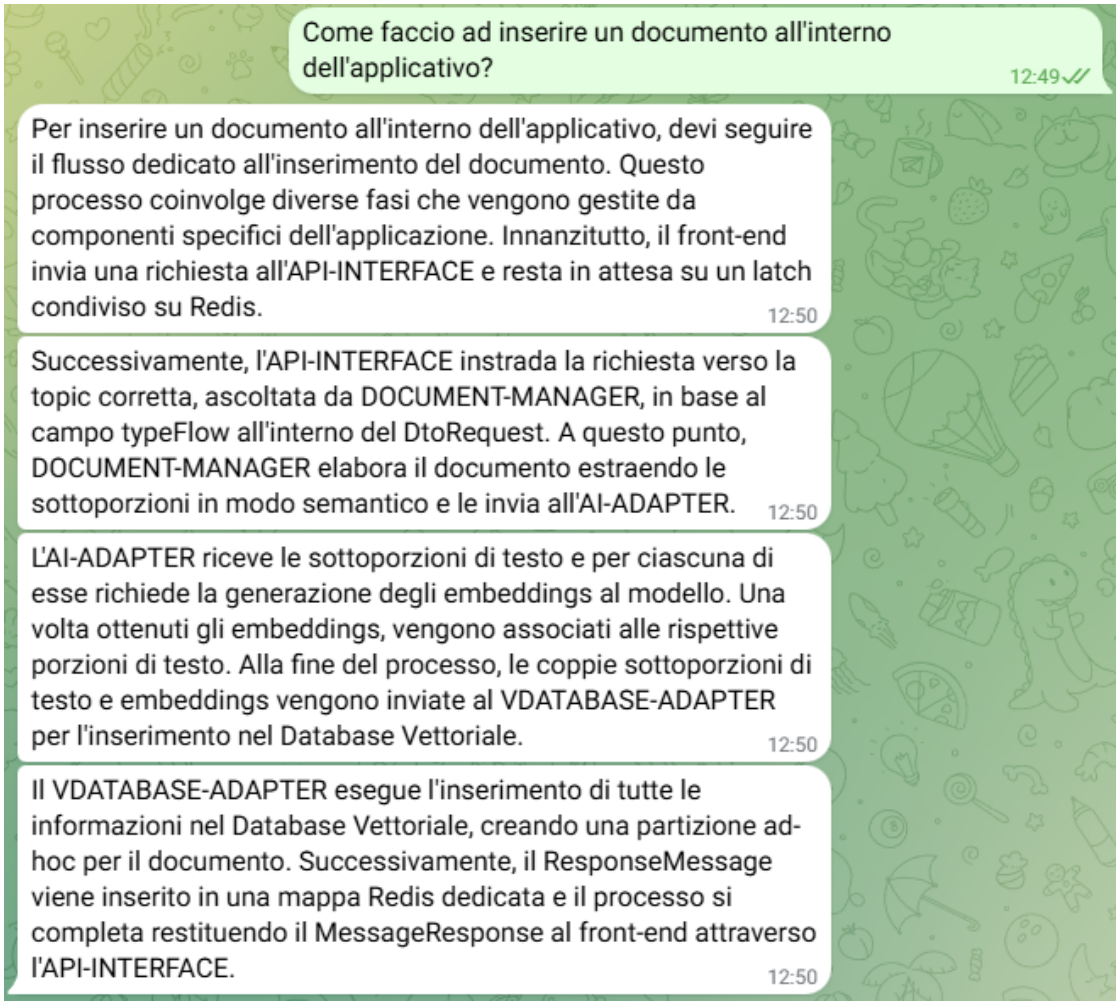

Figura 7.2: Esempi di risposta del Chatbot in cui l'utente fa una domanda di carattere generico, richiedendo la descrizione di un flusso dati, e l'applicativo risponde con tutti i particolari.

Inoltre come è possibile notare dalla medesima figure, l'applicativo formatta correttamente il codice in messaggi separati, supportando diversi linguaggi di programmazione.

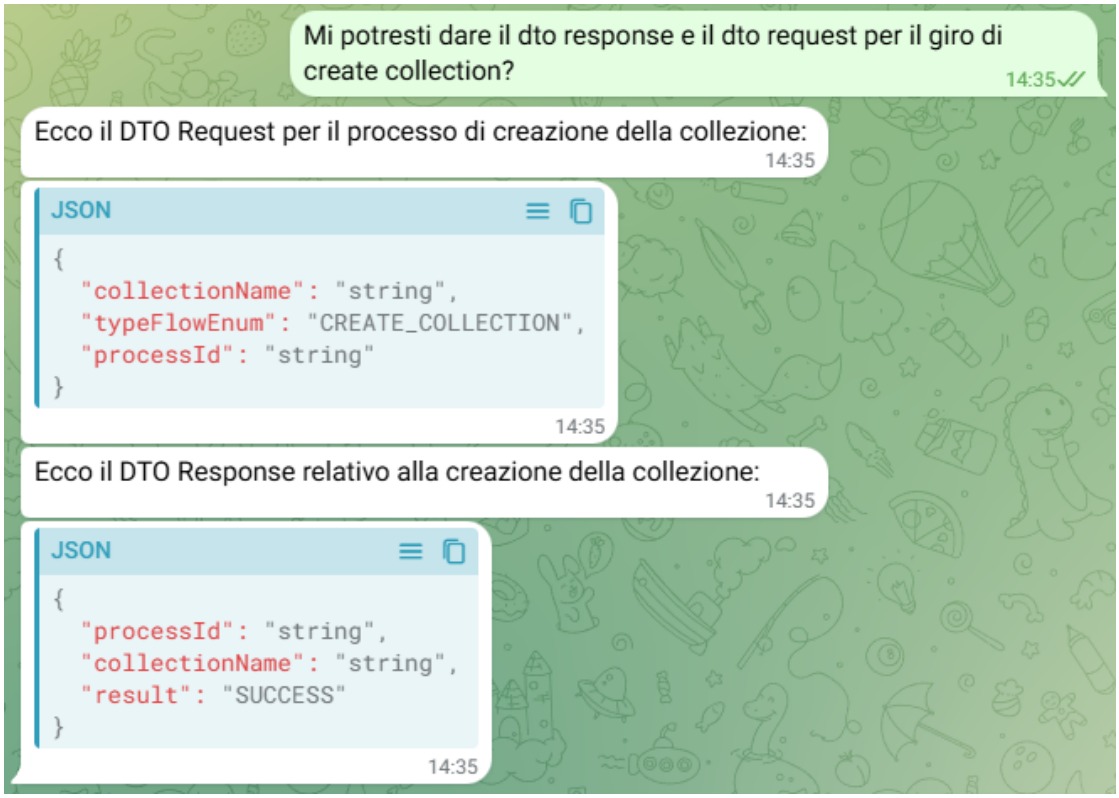

Figura 7.3: Esempio di risposta Documentation Chatbot, in cui l'applicativo risponde fornendo del testo formattato in JSON.

L'applicativo permette inoltre di gestire la documentazione software tramite file Markdown dividendoli in partizioni logiche. Questo avviene tramite un menu a bottoni proposto all'utente nella sezione di gestione dei documenti dei vari progetti. Inoltre in questa sezione l'utente ha anche la possibilità di eliminare o creare documenti e progetti.

7.2 Sviluppi Futuri

Il Documentation Chatbot allo stato attuale degli sviluppi presenta alcuni limiti. In primo luogo si potrebbe migliorare la gestione delle utenze, permettendo di avere varie utenze con diversi privilegi di lettura o scrittura, in modo tale da fornire i diritti per la gestione dei documenti solo ad alcune risorse specifiche del personale. Inoltre, sarebbe possibile introdurre un algoritmo ad-hoc per la gestione dei documenti in memoria, infatti ogni volta che Milvus effettua una ricerca all'interno del Database, le partizioni coinvolte attraverso cui la ricerca viene effettuata devono

essere caricate in RAM. Al momento, le partizioni interrogate vengono automaticamente caricate in memoria al momento dell'interrogazione da parte di un utente, poichè è stato realizzato un algoritmo di *paging* dei documenti assegnando magari una quantità limitata di memoria per la gestione delle partizione interrogate.
L'utilizzo di un'interfaccia utente come Telegram, nonostante sia comunque intuitiva presenta comunque delle limitazioni in termini di facilità e velocità d'uso. L'implementazione di un Front-End dedicato all'applicativo potrebbe permettere l'inserimento di nuove feature, come ad esempio lo storico delle conversazioni con l'utente. Questo potrebbe essere facilmente realizzato mantenendo il Back-End attuale, poichè, durante gli sviluppi del progetto, si è cercato di separare quanto più possibile le "responsabilità" che ha ogni microservizio.
In futuro, potrebbe essere implementata, in aggiunta, una memoria del chatbot per conservare le conversazioni, permettendo così di rispondere in modo coerente rispetto agli scambi di messaggi precedenti. Questo potrebbe essere reso possibile integrando nel sistema un database non relazionale, in cui vengono salvati i "riassunti" delle conversazioni. Questi riassunti verrebbero poi utilizzati come contesto all'interno del prompt inviato al LLM, migliorando la coerenza e la continuità nelle risposte del chatbot.

Bibliografia

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser e Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cit. a p. 7).

[2] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997 (cit. a p. 16).

[3] OpenLegacy. *Monolithic Application: What It Is How It Works*. 2023. URL: https://www.openlegacy.com/blog/monolithic-application (cit. a p. 19).

[4] Medium-Cobch7. *Kafka Architecture*. 2023. URL: https://medium.com/@cobch7/kafka-architecture-43333849e0f4 (cit. a p. 30).

[5] Medium-Nerd For Tech. *Understanding Redis in System Design*. 2024. URL: https://medium.com/nerd-for-tech/understanding-redis-in-system-design-7a3aa8abc26a (cit. a p. 37).

[6] Milvus. *Panoramica dell'Architettura di Milvus*. 2024. URL: https://milvus.io/docs/it/architecture_overview.md (cit. a p. 41).

[7] Evidently AI. *Precision Recall at K: Ranking Metrics Explained*. 2024. URL: https://www.evidentlyai.com/ranking-metrics/precision-recall-at-k (cit. a p. 61).