ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea in Ingegneria e Scienze Informatiche

Progettazione e sviluppo di un chatbot basato su tecniche RAG per la divulgazione di informazioni sulle specie marine

Tesi di Laurea in Ingegneria e Scienze Informatiche

Relatore
Prof.ssa Annalisa Franco

Candidato
Miriam Sonaglia

Sessione Marzo 2026
Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

Negli ultimi anni, grazie all'avvento dei Large Language Models (LLM), modelli linguistici di grandi dimensioni addestrati su enormi quantità di dati, si è assistito ad uno straordinario sviluppo dell'Intelligenza Artificiale e, grazie alle loro capacità sorprendenti, hanno aperto nuove possibilità in ambito di sistemi informativi intelligenti, rendendo possibile la creazione di chatbot sempre più sofisticati.

Tuttavia, nonostante le loro potenzialità, i Large Language Models presentano alcune importanti limitazioni quando impiegati in contesti che richiedono elevata affidabilità e precisione delle informazioni. La conoscenza di questi modelli è, infatti, limitata ai dati su cui sono stati addestrati, non possono accedere ad informazioni aggiornate o non presenti nelle fonti originali; per questo motivo possono generare risposte plausibili ma errate, fenomeno noto come "allucinazione". Ciò rappresenta un problema significativo in contesti informativi o divulgativi, nei quali l'affidabilità delle risposte costituisce un requisito fondamentale.

Per superare queste criticità, negli ultimi anni si è affermato un approccio denominato Retrieval-Augmented Generation (RAG), sistema che permette al modello di fornire una risposta soltanto dopo il recupero di informazioni da una base documentale verificata e autorevole e non generandola basandosi esclusivamente sulla conoscenza acquisita durante l'addestramento; in questo modo, le risposte del modello vengono vincolate a un insieme di fonti controllate, migliorando quindi l'affidabilità del sistema e le informazioni da esso divulgate.

La presente tesi si inserisce in questo contesto, esponendo lo sviluppo e la successiva valutazione comparativa di chatbot informativi basati su architettura RAG.

Il progetto nasce dalla collaborazione con il Dipartimento di Scienze Mediche Veterinarie di Cesenatico nell'ambito dell'applicazione *FinSpot*, piattaforma progettata per la raccolta di segnalazioni di avvistamenti di animali marini da parte dei cittadini.

Attraverso un'interfaccia conversazionale, gli utenti potranno, infatti, porre domande relative alle numerose specie marine e ottenere risposte accurate su aspetti morfologici, comportamentali, ecologici e sulle minacce che le riguardano, il tutto basandosi su una base documentale verificata.

La scelta di adottare un'architettura RAG è stata dettata dalla necessità di garantire che le risposte fornite dal chatbot fossero accurate e limitate esclusivamente al contenuto della base documentale fornita come riferimento.

Inoltre, per garantire all'utente un'esperienza offline e indipendente da servizi di cloud esterni, si è optato per uno sviluppo interamente locale del sistema.

1.2 Obiettivi della tesi

Il presente lavoro di tesi si propone due obiettivi principali: (i) lo sviluppo di un chatbot informativo basato su architettura RAG capace di fornire risposte accurate e affidabili sulle specie marine e (ii) la valutazione comparativa di diverse soluzioni tecnologiche per identificare i migliori trade-off tra qualità delle risposte ed efficienza computazionale.

Attraverso il raggiungimento di questi obiettivi, la tesi intende contribuire alla comprensione di come le scelte tecnologiche in un sistema RAG influenzino le prestazioni complessive di un sistema e quale configurazione risulti più adatta per applicazioni di divulgazione scientifica con vincoli di esecuzione locale.

Capitolo 2

Quadro tecnologico

2.1 Large Language Models

Con Large Language Models (LLM) si intendono modelli di apprendimento automatico addestrati su dataset di grandi dimensioni, in grado di comprendere e generare testo in linguaggio naturale e rappresentano una delle evoluzioni di maggior rilievo nel campo del Natural Language Processing (NLP). Il loro funzionamento si basa su architetture di reti neurali, in particolare sui Transformer, che funzionano prendendo in ingresso un testo e generando una risposta predicendo ripetutamente la parola successiva più probabile in base al contesto fornito.

2.1.1 Architettura Transformer

Prima dell'avvento dei Transformer, i modelli di NLP erano principalmente basati su architetture che processavano le sequenze di input token dopo token sequenzialmente, portando consequentemente a limitazioni quali difficoltà di ricordare informazioni, gestire dipendenze a lungo termine e impossibilità di parallelizzare l'addestramento, non potendo sfruttare appieno la potenza computazionale disponibile.

L'architettura Transformer, presentata in un articolo del 2017 da un gruppo di ricercatori di Google, è, invece, un modello di apprendimento che si basa sul meccanismo di *self-attention*, il quale pesa in modo dinamico l'importanza relativa di ogni elemento di una sequenza rispetto agli altri [5], utilizzando la capacità di alcuni neuroni di moltiplicare i propri output per pesi dinamici che variano in base al contesto; in questo modo è possibile catturare relazioni semantiche complesse tra parole e frasi e scalare a modelli con miliardi di parametri. La struttura di un Transformer è composta da più strati, ognuno dei

quali include i seguenti componenti principali: Meccanismo di multi-head self-attention, Feed-forward network e Normalizzazione

Self-Attention

Il meccanismo di self-attention trasforma, per mezzo di trasformazioni lineari, ogni input in tre vettori distinti: Query (Q), ovvero cosa sto cercando, Key (K), ossia l'etichetta e Value (V), cioè il contenuto effettivo della parola. L'attenzione viene quindi calcolata come:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

dove d_k è la dimensione dei vettori delle chiavi, utilizzata per normalizzare il prodotto scalare e stabilizzare i gradienti durante il training e *softmax* converte i punteggi di attenzione in probabilità (pesi) che sommano a 1.

Multi-Head attention

Il multi-head attention permette di eseguire il meccanismo di auto-attenzione più volte in parallelo, permettendo così al modello di concentrarsi simultaneamente su diverse parti dell'input.

Feed-forward network

La feed-forward network è una rete neurale completamente connessa che viene applicata a ogni posizione in modo indipendente dopo i meccanismi di attenzione.

Normalizzazione

La normalizzazione permette di stabilizzare e velocizzare l'addestramento delle reti neurali, assicurandosi che i gradienti non diventino troppo grandi o troppo piccoli, migliorando così la convergenza del modello.

2.1.2 Allucinazioni

Il problema delle allucinazioni consiste nella generazione di informazioni plausibili ma errate o completamente inventate da parte del modello. Esse possono manifestarsi in diverse forme a seconda del tipo di errore commesso dal modello: si parla di *allucinazioni di fatto* quando il modello genera risposte contenenti informazioni errate o di *allucinazioni*

di fedeltà quando il modello genera informazioni non presenti nel documento sorgente, ignorandolo o contraddicendo parti precedenti della stessa conversazione. Alle due tipologie appena descritte si aggiungono anche *confabulazioni*, nelle quali il modello riempie lacune nella sua conoscenza con informazioni inventate ma realistiche, poichè presentate con sicurezza, e *risposte non sensate*, dove il modello produce risposte che non hanno senso logico o connessione con ciò che viene chiesto dall'utente.

Le cause delle allucinazioni possono essere molteplici (overfitting o dati di addestramento rumorosi) e possono essere mitigate regolando parametri come la temperatura, impedendo al modello di essere troppo creativo e generare risposte più fattuali, oppure utilizzando tecniche come il Retrieval-Augmented Generation (RAG) che vincolano le risposte del modello a un insieme di documenti forniti e verificati.

2.2 Retrieval-Augmented Generation (RAG)

Letteralmente "Generazione Arricchita da Recupero", la tecnica RAG permette agli LLM di recuperare e incorporare nuove informazioni da fonti esterne; i modelli non rispondono, infatti, alle richieste dell'utente basandosi esclusivamente sulla conoscenza acquisita durante l'addestramento, ma accedendo a una base documentale esterna, usando quindi informazioni specifiche del dominio oppure aggiornate che non erano presenti nei dati di addestramento. [6]

2.2.1 Workflow

Un sistema RAG opera in quattro fasi principali:

1. *Pre-processing:* i documenti vengono suddivisi in blocchi (chunking) e trasformati in vettori numerici (embedding) che vengono poi indicizzati in un database vettoriale per consentire un recupero efficiente.

2. *Retrieval:* viene fatto l'embedding della query dell'utente e viene eseguita una ricerca semantica nel database vettoriale per recuperare i *chunk* di informazioni più pertinenti alla domanda, basandosi sulla similarità tra i vettori.

3. *Augmentation:* le informazioni recuperate vengono combinate con la query originale e fornite all'LLM come contesto addizionale per generare una risposta basata solamente su quelle informazioni.

4. *Generation:* l'LLM genera una risposta utilizzando sia la query originale sia i documenti recuperati.

2.2.2 Architettura e componenti

La tipica architettura di un sistema RAG è strutturata in una pipeline che trasforma i dati in una risposta contestualizzata. Alla radice di questa architettura si trova la base documentale, ossia l'insieme di documenti strutturati (database) o non (testi) che formano la conoscenza a cui il sistema fa riferimento per generare risposte. Questo insieme documentale viene elaborato dal *document loader*, che si occupa di preparare i documenti all'indicizzazione eseguendo parsing, chunking, estrazione dei metadati e normalizzazione del testo. Una volta avvenuta la segmentazione, i chunk vengono convertiti da un modello di embedding in vettori numerici che mappano la semantica del contenuto in uno spazio multidimensionale. Questi vettori vengono successivamente archiviati all'interno di un *Vector Store*, database specializzato che memorizza gli embedding dei documenti, consentendo un recupero basato sulla similarità vettoriale. Nel momento in cui l'utente pone una domanda, il *retriever* trasforma la query in un embedding e interroga il Vector Store per selezionare i frammenti più pertinenti alla domanda. Alla fine della pipeline troviamo l'LLM che, ricevendo come input sia la query originale, sia i documenti recuperati, genera una risposta rigorosamente vincolata alle informazioni fornite dal processo di recupero.

Strategie di Chunking

Il chunking dei documenti è una fase di fondamentale importanza in un sistema RAG, per questo esistono diverse strategie per poterlo eseguire al meglio. Tra le più comuni si può citare il *Fixed-size Chunking*, tecnica che suddivide il testo in blocchi di dimensione fissa. Nonostante sia computazionalmente efficiente, questa soluzione rischia di interrompere bruscamente una frase o un concetto, per questo si può scegliere di ricorrere al *Semantic Chunking*, che analizza le componenti del testo per individuare i punti dove si verifica una cambio di argomento o di concetto. Si può anche optare per una procedura che scompone il testo in modo ricorsivo, partendo da blocchi più grandi (paragrafi) e suddividendoli in blocchi sempre più piccoli (frasi, parole) fino a raggiungere una dimensione ottimale per l'embedding, tecnica nota come *Recursive Chunking*. È possibile anche sfruttare la formattazione naturale del documento, utilizzando la sua struttura (paragrafi, sezioni, capitoli) per creare blocchi di testo coerenti, in questo caso si parla di *Document-based Chunking*.

2.2.3 Vantaggi rispetto ai LLM standalone

Fornendo al modello come contesto dati aggiornati e verificati, RAG riduce la tendenza del modello a fornire informazioni errate poichè vincolato a basare le proprie risposte esclusivamente sui documenti forniti, piuttosto che sulla conoscenza acquisita in fase di training. Inoltre, a differenza degli LLM standalone, dove la conoscenza è fissa al momento del training, nei sistemi RAG è possibile aggiornarla modificando la base documentale, rendendo quindi il sistema molto più flessibile e adattabile.

2.3 Embeddings e rappresentazioni vettoriali

Permettendo di tradurre il significato semantico del testo in rappresentazioni numeriche dense e manipolabili, gli embeddings rappresentano un cardine del retrieval semantico all'interno dei sistemi RAG. Si basano sul concetto di spazio vettoriale dove parole, frasi o documenti con significati simili occupano regioni vicine dello spazio, permettendo così di quantificare la similarità semantica tramite operazioni algebriche sui vettori, come dimostrato da modelli come Word2Vec [7] e GloVe [8]. Tuttavia, questi approcci assegnano a ogni parola un singolo vettore statico, indipendentemente dal contesto nel quale essa compare, il che rappresenta una limitazione importante nei casi di polisemia. Per questo motivo sono stati introdotti modelli di *contextual embeddings*, come ELMo [10] e BERT [9], che generano rappresentazioni dinamiche in cui il vettore associato a una parola dipende dal contesto specifico in cui essa è inserita. BERT, in particolare, utilizzando un'architettura Transformer di tipo encoder-only, ovvero un modello che si concentra esclusivamente sulla comprensione del testo in ingresso senza generare output testuali, e venendo pre-addestrato con obiettivi come Masked Language Modeling (MLM) e Next Sentence Prediction (NSP), è in grado di apprendere sfumature semantiche molto complesse.

Nonostante ciò, nel retrieval è spesso necessario rappresentare non solo parole isolate, ma anche frasi o persino interi documenti. Inizialmente si è optato per il calcolo della media dei word embeddings ma questo approccio, non tenendo conto dell'ordine delle parole e delle interazioni tra esse, si è rivelato subottimale. Sono stati così introdotti modelli come Sentence-BERT (SBERT) [11], che modifica l'architettura di BERT per produrre sentence embeddings ottimizzati tramite strategie di addestramento basate su coppie di frasi semanticamente correlate e funzioni di perdita come la *ContrastiveLoss*, che minimizza la distanza tra frasi simili e massimizza quella tra quelle dissimili.

Ottenute le rappresentazioni vettoriali di query e documenti, il retrieval si occupa della dicerca dei vettori più simili tra loro. La similarità viene calcolata tramite metriche come la *similarità coseno* che misura l'angolo tra due vettori, indicando quanto due dati siano simili a prescindere dalla loro grandezza e assume un valore compreso tra -1 (vettori opposti) e 1 (vettori identici), con 0 che indica ortogonalità, ovvero nessuna similarità semantica:

$$\text{sim}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \, ||\vec{b}||} = \frac{\sum_i a_i b_i}{\sqrt{\sum_i a_i^2} \sqrt{\sum_i b_i^2}} \tag{2.2}$$

Nel caso di collezioni documentali di piccole o medie dimensioni, è possibile eseguire una ricerca esatta dei k nearest neighbors (k-NN) calcolando la similarità tra il vettore della query e tutti i vettori dei documenti indicizzati ma, con l'aumentare del numero di documenti, questa operazione diventa computazionalmente dispendiosa avendo una complessità lineare rispetto al numero di elementi indicizzati. Si sono quindi sviluppati i cosiddetti algoritmi di Approximate Nearest Neighbor (ANN) search, che, accettando una leggera perdita di accuratezza (recall), ottengono una velocità significativamente maggiore, migliorando consequentemente le prestazioni del sistema.

2.4 Database vettoriali

A differenza dei database tradizionali, sviluppati per query esatte, i database vettoriali sono progettati per gestire operazioni di ricerca per similarità su spazi vettoriali complessi.

Un database vettoriale è strutturato in un'architettura a più livelli: (i) il livello di storage, che si occupa dell'archiviazione fisica dei vettori, salvando generalmente sia gli embeddings che i metadati ad essi associati, (ii) il livello di indicizzazione, che trasforma i vettori in strutture dati indicizzate ottimali per la ANN e (iii) il query engine che interpreta la query dell'utente e calcola la similarità con i vettori indicizzati. Durante la fase di retrieval, il database vettoriale permette di associare a ciascun vettore un insieme di metadati e di utilizzarli per filtrare i risultati della ricerca. Questo meccanismo permette di restringere significativamente il dominio di ricerca poichè vengono combinati vincoli strutturati e similarità semantica. In un sistema RAG applicato a documenti testuali, ad esempio, ogni frammento di testo può essere corredato da informazioni quali la fonte, l'autore, la data di pubblicazione, la sezione del documento o la lingua; in fase di ricerca il sistema può ridurre il numero di documenti da prendere in considerazione filtrando quelli che non soddisfano i vincoli specificati.

Sono disponibili numerodi database vettoriali, sia open-source che commerciali, ognu-

no con caratteristiche necessarie a soddisfare esigenze specifiche come scalabilità, latenza, supporto per funzionalità avanzate (hybrid search o multi-tenancy) e integrazione con altri strumenti. Nel contesto del progetto descritto in questa tesi, è stato adottato ChromaDB per la sua semplicità di integrazione e la possibilità di utilizzo in ambiente locale.

Capitolo 3

Tecnologie e strumenti

Il presente capitolo descrive le tecnologie e gli strumenti adottati per la realizzazione del sistema. Esso è stato realizzato interamente in ambiente locale così da poter evitare dipendenze da servizi cloud esterni e facilitare l'utilizzo del sistema anche in assenza di connessione a internet o di hardware specializzato.

3.1 Framework e librerie

3.1.1 LlamaIndex

LlamaIndex [1] è un framework open-source progettato per la creazione di applicazioni che necessitano di integrare LLM con sorgenti di dati esterne. Fornisce strumenti che semplificano la costruzione di pipeline RAG grazie ad una forte astrazione delle componenti coinvolte come chunking, embedding, indicizzazione e retrieval. Oltre ad offrire molte facilitazioni, il framework supporta anche una vasta gamma di vector database, modelli di embedding e LLM.

3.1.2 Ollama

Ollama [2] è una piattaforma che esegue LLM e modelli di embedding direttamente su hardware locale. Funziona come un server locale e crea un ambiente isolato così da evitare qualsiasi conflitto con altri software installati rimanendo in ascolto su *localhost* e rispondendo a richieste per qualsiasi altro modello precedentemente scaricato.

Ai fini del progetto, Ollama ha offerto molti vantaggi tra cui: (i) indipendenza da servizi cloud esterni, (ii) possibilità di eseguire il sistema offline e (iii) flessibilità di

cambiare modelli di embedding ed LLM, argomento fondamentale che sarà approfondito in seguito nell'elaborato.

3.1.3 ChromaDB

ChromaDB [3] è un database vettoriale specializzato nell'archiviazione e nella conseguente ricerca per similarità di embedding. Esso utilizza un *in-memory database* nel quale i dati vengono mantenuti esclusivamente nella RAM, e quindi persi al riavvio, rendendo molto più efficienti le operazioni di lettura e scrittura. ChromaDB presenta però anche una memoria persistente che permette all'utente di salvare i dati su disco in una directory specificata. Quest'ultima modalità è stata adottata nel progetto attraverso `chroma_client = chromadb.PersistentClient(path=INDEX_DIR)`.

LlamaIndex integra ChromaDB tramite le classi `ChromaVectorStore` e `StorageContext` permettendo così di utilizzarlo come storage per il `VectorStoreIndex`.

3.1.4 Streamlit

Streamlit [4] è un framework Python per lo sviluppo di applicazioni web interattive utile per gli sviluppatori che necessitano di implementare rapidamente un'interfaccia utente senza dover utilizzare tecniche di front-end più complesse. Ad ogni interazione con l'utente, Streamlit esegue lo script da capo, aggiornando dinamicamente l'interfaccia. Il framework fornisce un ricco set di componenti predefinite e pronte all'uso tra cui `st.chat_message` e `st.chat_input`, che sono specifiche per le interfacce conversazionali.

3.2 Modelli di embedding

Influenzando la qualità del retrieval e di conseguenza le informazioni fornite all'LLM, la scelta del modello di embedding è un aspetto di fondamentale importanza in un sistema RAG.

Per il progetto sono stati presi in considerazione tre modelli: (i) *mxbai-embed-large*, (ii) *nomic-embed-text*, e (iii) *bge-m3*. Quest ultimo, è stato scartato nelle fasi iniziali di sviluppo causa degli elevati costi computazionali richiesti, mentre i primi due sono stati entrambi impiegati nelle diverse varianti del chatbot sviluppate.

3.2.1 mxbai-embed-large

Il modello *mxbai-embed-large* è un embedding model basato su architettura Transformer. Ha ottenuto risultati *state-of-the-art* nel bemchmark MTEB (Massive Text Embedding Benchmark) dimostrandosi molto competitivo nei compiti di retrival, classificazione e clustering.

È stato scelto come modello per una delle varianti grazie al buon trade-off che offre tra qualità di embedding e requisiti computazionali.

3.2.2 nomic-embed-text

nomic-embed-text è un modello di embedding rilasciato da Nomic AI, con dati di training, codice e pesi completamente open-source. Utilizza un regime a più fasi in cui un transformer pre-addestrato viene prima addestrato in modo contrastivo utilizzando un ampio corpus di dati debolmente accoppiati e successivamente perfezionato (finetuned) su dataset etichettati di dimensioni ridotte ma di qualità superiore. Il paradigma a due fasi migliora significativamente la qualità del modello poichè i dati debolmente accoppiati sono disponibili in quantità molto maggiori [12].

`Nomic-embed-text` è molto più leggero di `mxbai-embed-large`, inoltre può processare interi paragrafi senza perdita di informazione, il che risulta vantaggioso per documenti lunghi.

3.2.3 Confronto

La tabella 3.1 riassume le principali caratteristiche dei due modelli di embedding adottati nel progetto.

Tabella 3.1: Confronto tra i modelli di embedding

Caratteristica	**mxbai-embed-large-v1**	**Nomic Embed Text v1.5**
Sviluppatore	Mixedbread AI	Nomic AI
Context window	512 token	8192 token
Dimensioni embedding	1024 (default)	768 (default, regolabile)
Punti di forza	Precisione su chunk brevi	Lunghi contesti, versatilità
Licenza	Apache 2.0	Apache 2.0
Uso ideale	RAG classico con chunk piccoli, alta precisione	RAG con documenti lunghi o inesplorati

3.3 Large Language Models

Anche per la scelta dell'LLM è stato necessario valutare diversi modelli, considerando sia le prestazioni in termini di qualità delle risposte, sia i requisiti computazionali richiesti. I modelli in questione sono stati: (i) *Qwen 2.5*, (ii) *Phi-3 mini*, e (iii) *Llama 3.2*. Anche in questo caso Llama 3.2 è stato scartato nella fase iniziale a causa delle prestazioni inferiori rispetto agli altri due.

3.3.1 Qwen 2.5

Qwen 2.5 è un Large Language Model parte della serie Qwen sviluppata da Alibaba. Il rapporto tra dimensione del modello e qualità delle risposte lo rende molto competitivo.

Nel progetto è stato configurato con temperatura bassa al fine di rendere la risposta il più possibile deterministica; il modello seleziona infatti il token con probabilità più alta per rispettare rigorosamente il contesto documentale fornito.

3.3.2 Phi-3 mini

Phi-3 mini è uno Small Language Model (SML) sviluppato da Microsoft. La dimensione minore rispetto ad altri modelli gli permette di avere tempi di risposta più rapidi e requisiti computazionali contenuti. La serie Phi è stata sviluppata con l'obiettivo di dimostrare che modelli piccoli ma addestrati su dati di alta qualità possono raggiungere prestazioni comparabili a modelli molto più grandi [13, 14]. Le sue dimensioni ridotte comportano, però, alcune limitazioni quali: (i) una minore capacità di comprensione del contesto, (ii) una tendenza maggiore a discostarsi dal contesto fornito aggiungendo informazioni esterne, e (iii) una minor precisione lessicale in alcuni domini specialistici.

3.3.3 Confronto

La tabella 3.2 sintetizza le principali caratteristiche dei due LLM adottati nel progetto.

Tabella 3.2: Confronto tra i Large Language Models

Caratteristica	**Qwen2.5**	**Phi-3 Mini**
Sviluppatore	Alibaba	Microsoft
Context window	32K–128K	4K o 128K
Dimensione (Quantizzato)	~4–5 GB (4-bit)	~2.4–3 GB
Punti di forza	Coding, Matematica	Efficienza estrema
Licenza	Apache 2.0	MIT
Uso ideale	Coding agent, RAG	Dispositivi Edge

Capitolo 4

Progettazione del sistema

Il capitolo seguente tratterà la fase di progettazione del chatbot RAG sviluppato basato sull'utilizzo del modello di embedding *nomic-embed-text* e dell'LLM *Phi-3 mini*.

4.1 Requisiti e vincoli

Il sistema sviluppato deve essere in grado di rispondere alle domande poste dall'utente in linguaggio naturale fornendo informazioni affidabili riguardo alle specie marine presenti nel documento di riferimento. Al tempo stesso, deve mantenere coerenza relativa al contesto della conversazione in corso, riconoscendo quando l'utente fa riferimento ad una specie già menzionata negli scambi precedenti senza doverla citare esplicitamente, e, nell'eventualità in cui la risposta alla domanda non sia presente nella base documentale, deve essere in grado di comunicarlo direttamente per evitare di incorrere nel rischio di allucinazioni. Come ultimo requisito funzionale, il sistema deve fornire risposte in un tempo accettabile per un'interazione il più fluida possibile.

Il vincolo che più ha influenzato la scelta delle tecnologie del sistema è stato quello di garantire l'esecuzione locale orientando quindi la scelta in modelli abbastanza leggeri da poter essere eseguiti su hardware senza GPU dedicata.

4.2 Architettura del chatbot

L'architettura del sistema è quella di un sistema RAG, descritto nel Capitolo 2 e il flusso di lavoro può essere suddiviso in due macro-fasi: (i) una, eseguita una sola volta all'avvio del sistema, di preparazione e indicizzazione dei documenti e (ii) una, eseguita ad ogni interazione dell'utente, di interrogazione e conseguente generazione della risposta.

Nella prima fase, il documento fornito come base documentale viene caricato dal fylesistem locale e suddiviso in blocchi logici, ciascuno corrispondente a una specie marina. Ogni blocco estratto viene quindi corredato da metadati, che identificano il nome comune e il nome scientifico della specie, e poi suddiviso in chunk di dimensione fissa con un overlap del 20% tra chunk consecutivi per non perdere relazioni semantiche importanti ai fini del retrieval.

Tramite `nomic-embed-text` si procede poi alla trasformazione di ogni chunk in un vettore di embedding, che viene memorizzato in ChromaDB insieme ai metadati e al testo originale dando così origine all'indice vettoriale che viene salvato su disco così da evitare di doverlo ricostruire ad ogni esecuzione del sistema.

La seconda fase, invece, si ha quando l'utente formula una domanda attraverso l'interfaccia Streamlit. Il sistema analizza quindi la query alla ricerca di menzioni esplicite di specie marine che, se identificate, definiscono il contesto corrente della conversazione, in caso contrario il sistema mantiene il contesto della specie menzionata nell'interazione precedente. La query viene quindi trasformata in un embedding e viene eseguita una ricerca per similarità vettoriale nell'indice creato nella fase precedente. Se viene individuata la specie di riferimento, la ricerca viene filtrata per recuperare solamente i chunk relativi ad essa avvalendosi dei metadati. I top-5 chunk semanticamente più simili alla query vengono quindi reperiti e assemblati in un prompt insieme alla query originale e a istruzioni che guidano il metodo di risposta dell' LLM.

Il prompt completo viene così inviato al modello linguistico `Phi-3 mini` che, basandosi sui chunk recuperati, genera la risposta finale da restituire all'utente. Se questi non contengono informazioni utili alla risposta, il sistema restituisce un messaggio predefinito che invita l'utente a riformulare la domanda.

Le due figure sotto riportate illustrano l'architettura descritta: la figura 4.1 mostra lo schema generale di un sistema RAG, mentre la figura 4.2 riporta l'architettura specifica del sistema implementato con tutte le relative componenti adottate.

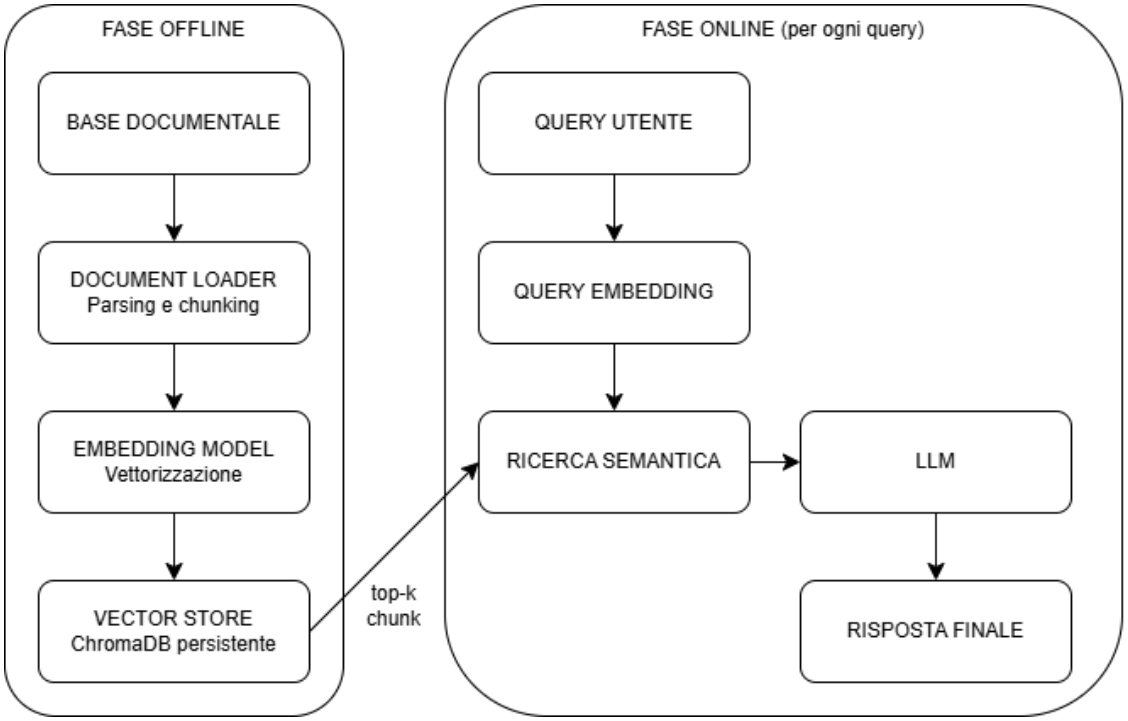

Figura 4.1: Architettura generale di un sistema RAG. La fase offline (sinistra) indicizza i documenti nel vector store; la fase online (destra) trasforma la query in embedding, recupera i chunk più rilevanti e li fornisce all'LLM per generare la risposta.

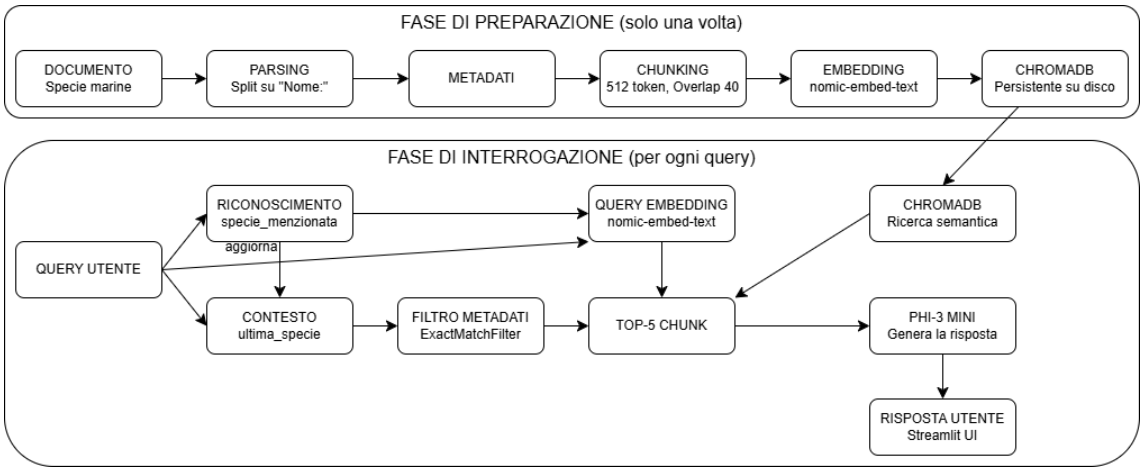

Figura 4.2: Architettura specifica del sistema implementato. La pipeline di preparazione (in alto) trasforma il documento in chunk indicizzati tramite `nomic-embed-text` e memorizzati in ChromaDB. La fase di interrogazione (in basso) mostra il flusso completo dalla query dell'utente alla risposta finale, con il meccanismo di riconoscimento della specie, l'aggiornamento del contesto e il filtraggio del retrieval tramite `ExactMatchFilter` metadati.

4.3 Scelte progettuali

La progettazione del sistema si è sviluppata seguendo una serie di scelte motivate da compromessi tra qualità delle risposte e sostenibilità computazionale. Le scelte relative al chunking sono state di fondamentale importanza poichè con chunk di dimensione troppo ridotta si rischia di frammentare eccessivamente le informazioni e quindi di rendere difficile al modello comprendere il contesto completo, mentre con chunk troppo grandi il rischio è quello di introdurre rumore informativo, diminuendo la precisione di riconoscimento semantico durante il retrieval. La dimensione ottimale deve tenere conto della context window del modello di embedding e della natura del contenuto. È stato quindi scelto di mantenere una parziale sovrapposizione tra chunk consecutivi (del 20%) per evitare che le informazioni agli estremi vengano perse.

Per il parsing è stata utilizzata una strategia che sfrutta la struttura del documento fornito. La descrizione di ogni specie segue infatti uno schema uniforme con marcatori testuali espliciti, per questo si è optato per un approccio basato su *pattern matching*. Ciò ha permesso di estrarre in modo molto più affidabile i nomi comuni delle specie che, essendo poi estratti come metadati, semplificano notevolmente i meccanismi di *filtering* contestuale evitando quindi di dover ricorrere a tecniche più complesse.

Ottenere un buon compromesso tra completezza e precisione ha guidato la scelta del numero di chunk da recuperare durante la fase di retrieval. Recuperare pochi chunk potrebbe, infatti, portare a una perdita di informazioni ma, al contrario, recuperarne troppi potrebbe introdurre rumore che andrebbe a confondere il modello o saturare la context window, rendendo molto difficile per l'LLM identificare le informazioni pertinenti alla domanda. La scelta del valore ottimale dipende quindi anche dalla capacità dell'LLM di filtrare informazioni irrilevanti.

Essendo il contesto del chatbot di natura informativa, per la configurazione dell'LLM si è optato per un'impostazione che privilegia l'affidabilità delle risposte rispetto alla creatività. Si è scelto quindi di impostare la temperatura a zero, eliminando di conseguenza ogni componente stocastica nella generazione, sacrificando varietà stilistica ma garantendo che le risposte ad una stessa domanda siano sempre le stesse.

La formulazione del system prompt è stata calibrata per sottolineare l'importanza di aderire strettamente alle informazioni presenti nella base documentale, imponendo al sistema di dichiarare l'assenza di informazioni piuttosto che inventare risposte plausibili ma errate e di incorrere quindi in allcuinazioni.

Il meccanismo di gestione del contesto conversazionale è stato implementato per offrire agli utenti una conversazione più fluida, permettendo a questi ultimi di non esplicitare il

soggetto della conversazione ad ogni domanda. Il sistema mantiene quindi traccia della specie attualmente in focus e la utilizza per filtrare il retrieval tramite un meccanismo di fallback che prende come riferimento la specie del chunk più rilevante quando questa non è esplicitata nella query. Questa scelta introduce una leggera complessità implementativa ma migliora la user experience rendendo la conversazione con il chatbot più naturale.

Si possono identificare tre scenari principali di gestione del contesto che influenzano il comportamento del sistema: (i) specie esplicitata nella query, nel quale la funzione `specie_menzionata` riconosce un riferimento esplicito ad una specie, quella diventa il nuovo contesto e il retrieval viene filtrato esclusivamente sui chunk ad essa relativi, (ii) nessuna specie nella query, tipico delle domande follow-up, dove l'utente pone una domanda senza menzionare alcuna specie, il sistema mantiene il contesto della specie precedente e applica nuovamente il filtro per metadati, e (iii) nessuna specie e nessun contesto, quando non è presente alcun contesto precedente e la query non contiene riferimenti espliciti, il retrieval avviene sull'intera base documentale senza filtri e il contesto viene aggiornato con la specie del chunk più rilevante restituito.

Un caso particolare si presenta quando vengono poste domande comparative tra due o più specie; la funzione `specie_menzionata` riconosce la prima specie con l'alias più lungo che compare nella query e imposta quello come contesto, applicando il filtro di retrieval sui chunk ad essa relativi. Le risposte risultano comunque accettabili grazie ad una proprietà strutturale della base documentale in cui ogni scheda include nella sezione *Riconoscimento* riferimenti espliciti alle specie morfologicamente simili con i loro principali tratti differenziali. I chunk della specie filtrata contengono quindi già alcuni elementi utili al confronto con le altre specie citate.

Questa soluzione presenta un limite di completezza: le informazioni sulle specie non filtrate vengono ricavate dai cenni presenti nella scheda della specie di contesto e non dai loro chunk dedicati. Le risposte comparative sono quindi corrette nei tratti principali, ma meno esaustive rispetto a quanto il sistema potrebbe produrre recuperando chunk da tutte le specie coinvolte nel confronto.

La scelta di utilizzare ChromaDB in modalità persistente invece di ricostruire l'indice ad ogni esecuzione, permette di minimizzare i tempi di avvio dopo la prima indicizzazione visto che i documenti di riferimento sono fissi e non rendono necessario rigenerare gli embedding ad ogni avvio.

Capitolo 5

Implementazione

Il presente capitolo si propone di descrivere nel dettaglio le fasi legate all'implementazione del sistema, dalla raccolta dei dati alla realizzazione dell'interfaccia finale. Il codice è stato sviluppato in Pyhton, utilizzando le librerie e i framework descritti nel Capitolo 3.

5.1 Raccolta e organizzazione dei dati

La prima fase del progetto ha riguardato la raccolta delle informazioni relative alle specie marine presenti nei mari italiani; esse sono state raccolte tramite una ricerca su fonti scientifiche e divulgative affidabili consultando siti web di organizzazioni scientifiche riconosciute e materiale divulgativo di enti di ricerca.

Per ciascuna specie, sono state raccolte informazioni riguardanti caratteristiche morfologiche, habitat, comportamento, alimentazione, riproduzione e principali minacce; che sono state organizzate, dapprima in documenti distinti per ogni specie, successivamente uniti in un unico documento Word mantenendo, per ciascuna specie, un blocco testuale indipendente identificato dal marcatore testuale "Nome:" antecedente al nome comune della specie, seguito da sezioni introdotte da etichette come "Nome scientifico:", "Descrizione:", "Dimensioni:", ecc.. Questa scelta è stata fondamentale per facilitare le successive operazioni di parsing e di estrazione dei metadati.

Il documento finale risulta quindi una raccolta delle principali informazioni riguardanti 14 specie marine comuni nei nostri mari.

A titolo esemplificativo, il listato 5.1 sotto riporta una porzione di documento relativa al Tursiope, mostrando la struttura del file che utilizza i marcatori testuali utili al parsing.

Listing 5.1: Estratto del documento di testo – Tursiope

```
Nome: Tursiope

Nome scientifico: Tursiops Truncatus

Descrizione: Colorazione grigia con varie sfumature sul dorso e bianca
sul ventre. Il melone pronunciato e mascella e mandibola allungate
formano un rostro di 8cm corto e tozzo da cui il nome "Truncatus".
La pinna dorsale e' ricurva e alta circa 23cm, le pinne pettorali
misurano circa 30-50cm e quella caudale circa 60cm.

Dimensioni: da 2.5 a 3.8 metri di lunghezza con un peso fino a 650kg

Riconoscimento: Puo' essere confuso con il delfino comune o con la
stenella striata. Il tursiope si distingue per l'assenza di striature
bianche sui fianchi.

Distribuzione e habitat: La specie abita principalmente zone di
piattaforma continentale, lungo le coste. In Italia sono frequenti
lungo le coste siciliane, nell'Adriatico e in alcune porzioni del
Santuario dei Cetacei.

Comportamento: Sono animali sociali, vivono in branchi composti
generalmente da 2-6 individui, solitamente femmine con i loro piccoli.

[... Suoni, Apparato sensoriale, Alimentazione, Riproduzione ...]

Minacce: Classificata come "a rischio minimo" dalla lista rossa IUCN;
tuttavia sono noti problemi di conservazione per la sottopopolazione
vulnerabile del Mar Mediterraneo.
```

Ogni specie segue esattamente lo stesso schema: il marcatore `"Nome:"` delimita l'inizio di ogni specie e le etichette delle sezioni consentono l'estrazione dei metadati.

5.2 Preparazione e parsing dei documenti

Il documento testuale è stato suddiviso in blocchi logici e trasformato in una struttura dati tramite il parsing.

Il caricamento del documento è stato gestito tramite `SimpleDirectoryReader` di LlamaIndex, configurato per leggere i file presenti nella directory `data/`; in questo modo

si è cercato di mantenere una certa flessibilità per poter espandere in futuro la base documentale, aggiungendo nuovi file.

Listing 5.2: Caricamento del documento

```
DATA_DIR = "data"

documents = SimpleDirectoryReader(DATA_DIR).load_data()
if not documents:
    raise ValueError("Nessun documento trovato in 'data/'")

full_text = documents[0].text
```

Il testo completo del documento viene memorizzato in una variabile stringa che costituisce l'input per la fase successiva di parsing, che suddividerà il testo in blocchi logici.

Come accennato precedentemente, il parsing sfrutta i marcatori testuali per riconoscere i confini tra le parti di testo riguardanti le diverse specie; il delimitatore è utilizzato è la stringa "Nome:", che introduce il nome comune di ogni specie.

Listing 5.3: Suddivisione in blocchi per specie

```
species_documents = []
blocks = full_text.split("Nome:")

for block in blocks:
    block = block.strip()
    if not block:
        continue

    lines = block.splitlines()
    species_name = lines[0].strip()
```

Per ciascun blocco testuale, la prima riga contiene il nome comune della specie, mentre le righe successive contengono le relative sezioni informative. Per estrarre il contenuto di ogni sezione, è stata definita una funzione `extract` che, data un'etichetta, cerca il pattern che inizia con quella, seguita da due punti e prende tutta la sezione di testo fino all'etichetta successiva.

Listing 5.4: Estrazione dei metadati tramite regex

```
def extract(label: str) -> str:
    pattern = rf"{label}\s*:\s*([\s\S]*?)(?=\n[A-Z\u00c0-\u00dc
       ][^\n]*?:|\Z)"
    m = re.search(pattern, block, re.IGNORECASE)
    return m.group(1).strip() if m else ""

nome_scientifico = extract("Nome scientifico")
```

Ogni blocco viene a questo punto trasformato in un `Document` di LlamaIndex, che racchiude sia il testo che un dizionario dei relativi metadati che includono: (i) nome comune della specie nella forma originale, (ii) nome comune normalizzato in minuscolo con spazi multipli rimossi, (iii) nome scientifico nella forma originale, e (iv) nome scientifico normalizzato. La doppia rappresentazione (originale e normalizzata) si rivelerà utile per il riconoscimento delle menzioni nelle query dell'utente, grazie alla possibilità di effettuare confronti case-insensitive.

5.3 Chunking e generazione degli embedding

Per migliorare il recupero semantico, i documenti sono stati ulteriormente suddivisi in segmenti di dimensione controllata (chunk). Il chunking è stato sviluppato utilizzando `SentenceSplitter` di LlamaIndex, configurato con una dimensione di 512 token e un overlap di 40 token tra chunk consecutivi. I confini delle frasi vengono identificati tramite punteggiatura o pattern linguistici, unendo frasi consecutive fino a raggiungere la dimensione target senza però spezzare le frasi a metà.

Listing 5.5: Configurazione del chunking

```
from llama_index.core.node_parser import SentenceSplitter

parser = SentenceSplitter(chunk_size=512, chunk_overlap=40)
nodes = parser.get_nodes_from_documents(species_documents)
```

L'overlap viene gestito includendo le ultime frasi del chunk precedente all'inizio di quello successivo; questo meccanismo converte ogni `Document` in uno o più `Node`, unità atomiche indicizzate, che ereditano i metadati del documento padre così da poter tracciare la provenienza di ogni chunk.

Gli embedding vengono generati tramite il modello `nomic-embed-text` eseguito localmente grazie ad Ollama. Per ogni chunk il testo viene inviato al modello che

restituisce un vettore che ne rappresenta il significato semantico. Questo avviene per ogni chunk durante la fase di creazione dell'indice.

Gli embedding così generati permettono quindi di recuperare informazioni attinenti alla query dell'utente anche quando esso utilizza termini che non coincidono esattamente con quelli presenti nella base documentale.

5.4 Database vettoriale

La memorizzazione degli embedding è stata affidata a ChromaDB, configurato in modalità persistente, così da migliorare l'efficienza del sistema ed evitare di dover rigenerare gli embedding ad ogni esecuzione. Viene configurato quindi un client che salva i dati su disco specificando il percorso della directory.

ChromaDB viene integrato con LlamaIndex tramite `ChromaVectorStore` e viene creato uno `StorageContext` che unisce il vector store all'infrastruttura di storage generale di LlamaIndex così da poter salvare e caricare indici indipendentemente dal backend utilizzato.

Listing 5.6: Configurazione di ChromaDB

```
import chromadb
from llama_index.vector_stores.chroma import ChromaVectorStore
from llama_index.core import StorageContext

INDEX_DIR = "index_nomic"
COLLECTION_NAME = "specie_marine_nomic"

chroma_client = chromadb.PersistentClient(path=INDEX_DIR)
collection = chroma_client.get_or_create_collection(name=
    COLLECTION_NAME)

vector_store = ChromaVectorStore(chroma_collection=collection)
storage_context = StorageContext.from_defaults(vector_store=
    vector_store)
```

Si verifica se la collection contenga già dati o meno; se è vuota, viene creato l'indice vettoriale (`VectorStoreIndex`) a partire dai nodi, dallo storage context e dal modello di embedding, se, invece, contiene già dati, l'indice viene semplicemente caricato.

La fase di creazione dell'indice è computazionalmente costosa e dipende dal numero di chunk da indicizzare e dalla velocità di generazione degli embedding, mentre la fase

di caricamento è molto più rapida non richiedendo la rigenerazione degli embedding ma solo il caricamento dei vettori già presenti in ChromaDB.

Listing 5.7: Creazione o caricamento dell'indice

```
from llama_index.core import VectorStoreIndex

if collection.count() == 0:
    print("Creazione indice...")
    index = VectorStoreIndex(
        nodes=nodes,
        storage_context=storage_context,
        embed_model=embed_model
    )
else:
    print("Caricamento indice esistente...")
    index = VectorStoreIndex.from_vector_store(
        vector_store=vector_store,
        embed_model=embed_model
    )
```

Anche la query dell'utente viene trasformata in embedding tramite lo stesso modello utilizzato in fase di indicizzazione e il vettore risultante viene confrontato con quelli presenti nel database per recuperare quelli più semanticamente simili. I chunk vengono quindi ordinati per similarità e forniti come contesto al modello linguistico per la generazione della risposta.

5.5 LLM

La generazione delle risposte è a carico del modello `Phi-3 mini`, eseguito in locale tramite Ollama.

Listing 5.8: Configurazione del LLM

```
from llama_index.llms.ollama import Ollama

LLM_MODEL_NAME = "phi3:mini"

llm = Ollama(
    model=LLM_MODEL_NAME,
```

```
    temperature=0.0,
    request_timeout=300,
    additional_kwargs={
        "num_ctx": 4096,
        "num_predict": 512,
    },
    system_prompt=(
        "Rispondi esclusivamente usando le informazioni
           contenute nei documenti. "
        "Puoi riformulare in modo chiaro e naturale. "
        "Se l'informazione non e' presente, rispondi: "
        "'L'informazione non e' presente nei documenti forniti.'
           "
    ),
)
```

Il modello è stato configurato con temperarura pari a zero per ridurre la variabilità e aumentare il determinismo. Il `request_timeout` di 300 secondi permette al sistema di gestire query complesse che necessitano di tempi di elaborazione prolungati senza dover interrompere la generazione prematuramente.

Per specificare i parametri di contesto del modello viene utilizzato `additional_kwargs`, mentre `num_ctx` limita la context window a 4096 token; il modello, nella sua configurazione di default, infatti, potrebbe tentare di allocare cache per context window molto più ampie, saturando la RAM disponibile nel dispositivo utilizzato. `num_predict` invece limita a 512 token la lunghezza massima della risposta generata così da evitare risposte eccessivamente lunghe e prolisse.

Per imporre al modello di utilizzare esclusivamente le informazioni fornite nei documenti recuperati, è stato formulato un system prompt con tre direttive principali: (i) vincolo di aderenza ai documenti forniti, (ii) libertà di riformulazione stilistica per migliorare la leggibilità, e (iii) gestione esplicita dell'assenza di informazioni attraverso una risposta predefinita.

La creazione del query engine avviene tramite il metodo `as_query_engine` dell'indice, che configura i processi di retrieval e generazione.

Listing 5.9: Creazione del query engine

```
from llama_index.core.vector_stores import MetadataFilters,
    ExactMatchFilter

def make_query_engine(specie: str | None):
    if not specie:
        return index.as_query_engine(
            llm=llm,
            similarity_top_k=5,
            response_mode="simple_summarize"
        )

    filters = MetadataFilters(
        filters=[ExactMatchFilter(key="specie", value=specie)]
    )
    return index.as_query_engine(
        llm=llm,
        similarity_top_k=5,
        response_mode="simple_summarize",
        filters=filters
    )
```

`similarity_top_k`, impostato a 5, è il parametro che specifica quanti chunk recuperare per ogni query, mentre `response_mode` è configurato come `simple_summarize`, una modalità in cui tutti i chunk recuperati vengono concatenati insieme alla query in un unico prompt da cui il modello genera una risposta sintetica. Altre modalità, come quelle iterative, richiederebbero molte chiamate all'LLM portando quindi a tempi di risposta più lunghi.

Quando il contesto di specie è attivo, viene creato un filtro sui metadati con `MetadataFilters` ed `ExactMatchFilter`, che indica che soltanto i chunk il cui metadato `specie` corrisponde esattamente al valore del contesto corrente devono essere recuperati. Questa procedura avviene prima del retrieval semantico, a livello di vector store, così da ridurre lo spazio di ricerca e rendere il processo di recupero molto più efficiente.

Il ciclo di interrogazione del chatbot è implementato nella funzione `loop_boot`, che coordina tutte le componenti del sistema.

Listing 5.10: Loop di interrogazione principale

```
def specie_menzionata(user_query: str) -> str | None:
    q = normalize(user_query)
    for alias in sorted(alias_to_specie.keys(), key=len, reverse
       =True):
        if alias and alias in q:
            return alias_to_specie[alias]
    return None

ultima_specie = None

def loop_bot(user_query: str) -> str:
    global ultima_specie
    # Riconosco menzioni di specie nella query
    sp_in_query = specie_menzionata(user_query)
    if sp_in_query:
        ultima_specie = sp_in_query

    # Creo query engine con o senza filtering
    qe = make_query_engine(ultima_specie)
    response = qe.query(user_query)
    source_nodes = response.source_nodes or []

    if not source_nodes:
        return (
            "Non trovo informazioni su questo nei materiali "
            "a mia disposizione. Prova a riformulare la domanda.
               "
        )
    # Aggiorno contesto se non era esplicito
    if not sp_in_query:
        nuova_specie = source_nodes[0].node.metadata.get("specie
           ")
        if nuova_specie:
            ultima_specie = nuova_specie

    return str(response).strip()
```

Ad ogni richiesta dell'utente viene prima invocata `specie_menzionata` per controllare se la query contenga riferimenti espliciti a specie marine, in caso affermativo, la variabile globale `ultima_specie` viene aggiornata diventando il nuovo contesto della conversazione, altrimenti viene mantenuto quello precedente. A questo punto viene creato un query engine che include sia il testo generato sia la lista di `source_nodes`, chunk effettivamente utilizzati per la generazione della risposta. Se la lista è vuota, viene restituito un messaggio predefinito che informa l'utente dell'assenza di informazioni rilevanti alla domanda all'interno della base documentale.

5.6 Sviluppo dell'interfaccia utente

Lo script principale `app.py` è quello che definisce l'intera applicazione tramite una sequenza di chiamate alle funzioni messe a disposizione dal framework Streamlit.

Il titolo della pagina e l'icona visualizzata nel browser vengono impostati tramite `st.set_page_config`, mentre con `st.title` viene definita l'intestazione principale dell'applicazione.

È stata implementata una sidebar che contiene al suo interno due funzioni principali: (i) un selettore che permette di scegliere quale variante del chatbot utilizzare tra le tre implementate, e (ii) un pulsante per resettare la conversazione corrente. Il selettore utilizza `st.sidebar.selectbox`, che presenta all'utente un menu dropdown con le opzioni disponibili e restituisce la stringa che corrisponde alla selezione corrente. Il caricamento del chatbot selezionato avviene attraverso `importlib.import_module` che importa il modulo Python corrispondente alla scelta dell'utente.

Lo stato conversazionale viene gestito tramite `st.session_state`, dizionario persistente che mantiene in memoria i dati tra le riesecuzioni dello script. Vengono mantenute due variabili di stato: (i) `chat`, una lista di dizionari che rappresentano i messaggi della conversazione, e (ii) `last_bot`, che memorizza quale variante di chatbot era attiva nell'esecuzione precedente. Se l'utente cambia in itinere la selezione del chatbot, la cronologia viene resettata automaticamente per evitare confusione tra conversazioni avute tra modelli diversi.

Le richieste dell'utente vengono raccolte con `st.chat_input` che presenta il campo di testo in basso allo schermo, e quando l'utente invia un messaggio questo viene visualizzato come messaggio utente e aggiunto alla cronologia. Per la generazione della risposta viene creato un nuovo messaggio assistente che attiva uno spinner (`st.spinner`) il quale mostra un'animazione di caricamento e il testo "Il bot sta pensando..." durante l'elaborazione.

La risposta generata viene visualizzata tramite `st.markdown`, seguita da due caption indicanti la variante di chatbot che ha generato la risposta e il tempo impiegato.

Listing 5.11: Gestione dell'interazione utente

```
user_input = st.chat_input("Scrivi una domanda...")

if user_input:
    # Messaggio utente
    st.session_state.chat.append({"role": "user", "content":
       user_input})
    with st.chat_message("user"):
        st.markdown(user_input)

    # Risposta del bot
    with st.chat_message("assistant"):
        with st.spinner("Il bot sta pensando..."):
            start_time = time.perf_counter()
            response = loop_bot(user_input)
            end_time = time.perf_counter()
            response_time = end_time - start_time

        st.markdown(response)
        st.caption(f"Risposta generata da: {BOT}")
        st.caption(f"Tempo di risposta: {response_time:.2f}
           secondi")

    st.session_state.chat.append({"role": "assistant", "content"
       : response})
```

Ad ogni nuova interazione, lo script viene eseguito da capo ma, grazie alla persistenza del `session_state`, la cronologia viene preservata e la conversazione rimane visibile così che l'utente abbia sempre la possibilità di controllare gli scambi precedenti.

Capitolo 6

Architetture alternative

Per evidenziare e valutare l'importanza delle scelte architetturali sulle prestazioni del sistema, si è scelto di sviluppare due varianti alternative del chatbot, utilizzando diverse combinazioni di modelli di embedding e Large Language Models rispetto all'originale. Nel Capitolo 7 verrà condotta una valutazione comparativa tra le tre varianti, mentre nel presente capitolo si descriveranno le differenze implementative di queste rispetto al chatbot originale.

6.1 Motivazioni

Un'architettura RAG, grazie alla sua natura modulare, rende molto semplice testare diverse configurazioni di modelli di embedding e LLM senza la necessità di dover implementare l'intero sistema da zero. Grazie a questa caratteristica è stato possibile testare diverse combinazioni, al fine di identificare il trade-off migliore tra qualità delle risposte ed efficienza computazionale.

Per quanto riguarda la scelta del modello di embedding, nel chatbot originale si è optato per `nomic-embed-text` in primis per la sua efficienza computazionale, tuttavia, `mxbai-embed-large`, avendo dimensioni maggiori, offre una precisione semantica superiore; per questo motivo, si è scelto di sviluppare una variante che lo utilizza per verificare se questa precisione superiore fosse compatibile con i requisiti del sistema e potesse apportare un miglioramento alle prestazioni del chatbot.

Per i Large Language Models, `Qwen 2.5` è stato scelto come alternativa a `Phi-3 mini` per la capacità di aderire strettamente alla base documentale fornita e per la precisione nella generazione di risposte in lingua italiana, a differenza di Phi-3 mini che, pur essendo più veloce, risulta meno preciso e aderente al contesto.

Il confronto tra questi due LLM permette quindi di valutare il miglior compromesso tra velocità, efficienza e affidabità delle risposte fornite.

Le tre combinazioni finali scelte per i chatbot, compreso l'originale, sono state quindi: (i) nomic-embed-large + Phi-3 mini (originale), (ii) mxbai-embed-large + Qwen 2.5 e (iii) mxbai-embed-large + Phi-3 mini.

6.2 Variante con mxbai-embed-large + Qwen 2.5

Verranno descritte ora le differenze implementative della prima variante del chatbot che coinvolge l'utilizzo dei modelli `mxbai-embed-lagre` e `Qwen 2.5`. Questa configurazione, a livello teorico, rappresenta la combinazione migliore in termini di qualità delle risposte, a scapito dell'efficienza computazionale.

È stato creato quindi un nuovo file molto simile all'originale ma con alcuni parametri modificati per adattarsi alle caratteristiche dei nuovi modelli utilizzati.

Le directory di indice sono state separate per evitare conflitti tra le varianti ed evitare di rigenerare gli embedding ad ogni esecuzione e per ogni variante; alla prima esecuzione di questa soluzione, dovendo generare gli embedding da zero, si evidenzierà un tempo di avvio più lungo rispetto al chatbot originale a causa delle maggiorni dimensioni del modello.

Il chunking utilizza parametri più conservativi:

Listing 6.1: Chunking per la variante mxbai + Qwen

```
parser = SentenceSplitter(chunk_size=256, chunk_overlap=20)
nodes = parser.get_nodes_from_documents(species_documents)
```

La dimensione dei chunk è stata ridotta a 256 token per adattarsi alla context window più limitata di *mxbai-embed-large* che è di 512, a differenza di *nomic-embed-text* che supporta fino a 8192 token.

Il modello produce, inoltre, vettori di dimensione 1024, contro i 768 di *nomic*, riuscendo potenzialmente a captare legami semantici più sottili tra i chunk.

A livello di LLM, `Qwen 2.5` differisce da `Phi-3 mini` in maggiore misura per l'assenza di limitazioni esplicite riguardanti la context window, il che permette di fornire al modello più contesto quando necessario senza rischiare di saturare la memoria.

Per quanto riguarda il retrieval, esso utilizza un numero maggiore di chunk rispetto al chatbot originale.

Listing 6.2: Query engine

```
def make_query_engine(specie: str | None):
    if not specie:
        return index.as_query_engine(
            llm=llm,
            similarity_top_k=10,
            response_mode="compact",
        )

    filters = MetadataFilters(
        filters=[ExactMatchFilter(key="specie", value=specie)]
    )
    return index.as_query_engine(
        llm=llm,
        similarity_top_k=10,
        response_mode="compact",
        filters=filters,
    )
```

È stata scelta una `similarity_top_k=10` per sfruttare la maggiore capacità di *Qwen 2.5* di gestire contesti maggiori senza confondersi, mentre la `response_mode="compact"` riduce i token ridondanti durante la formattazione dei chunk nel prompt, così da lasciare più spazio alla risposta, non saturando la context window.

La differenza più significativa di questa variante è l'aggiunta di un passaggio di post-processing sulla risposta generata, che utilizza nuovamente *Qwen 2.5* per riformulare la risposta prima di restituirla all'utente.

Listing 6.3: Post-processing della risposta con Qwen

```
response = query_engine.query(user_query)

prompt_rewrite = f"""
Sei un editor. Devi restituire SOLO la versione finale della
   risposta.

REGOLE OBBLIGATORIE:
- Output: SOLO la risposta al quesito.
- NON aggiungere informazioni nuove.
- NON cambiare specie.
```

```
- NON inventare numeri o dettagli.
- Se il testo dice che l'informazione manca, mantieni lo stesso
    significato.

Testo:
{response}
"""
text = llm.complete(prompt_rewrite).text.strip()
```

Questo approccio, nonostante non sia computazionalmente efficiente, permette di migliorare la qualità delle risposte, rendendole più concise e dirette.

6.3 Variante con mxbai-embed-large + Phi-3 mini

Questa variante rappresenta un compromesso tra quelle già desctitte, utilizzando lo stesso modello di embedding della precedente e mantenendo come LLM `Phi-3 mini`, come nell'originale.

La configurazione di base dell'embedding è identica alla variante precedente, come il chunking che mantiene le stesse dimensioni per la comatibilità con la context window limitata di `mxbai-embed-large`.

Anche per quanto riguarda `Phi-3 mini` si utilizzano le stesse configurazioni usate nel chatbot principale.

La differenza più rilevante di questa variante rispetto alle altre è che il retrieval viene configurato come un compromesso tra le due soluzioni precedenti.

Listing 6.4: Query engine con retrieval intermedio

```
def make_query_engine(specie: str | None):
    if not specie:
        return index.as_query_engine(
            llm=llm,
            similarity_top_k=5,
            response_mode="simple_summarize"
        )

    filters = MetadataFilters(
        filters=[ExactMatchFilter(key="specie", value=specie)]
    )
    return index.as_query_engine(
```

```
        llm=llm,
        similarity_top_k=5,
        response_mode="simple_summarize",
        filters=filters
    )
```

Recuperare 10 chunk come nella variante con *Qwen* potrebbe saturare la context window, per questo ne vengono recuperati solo 5 (`similarity_top_k=5`), mentre la `response_mode` è configurata come nel chatbot originale, ovvero `simple_summarize`, che fornisce al modello un contesto più ampio e meno ridondante rispetto alla modalità `compact` utilizzata con *Qwen*.

A differenza della variante precedente, questa configurazione non include il passaggio di post-processing, restituendo così com'è la risposta generata dall'LLM.

Capitolo 7

Valutazione sperimentale

Il presente capitolo si pone l'obiettivo di valutare le prestazioni delle tre varianti di chatbot sviluppate. L'analisi è stata condotta attraverso un insieme di domande di test pensate per coprire diversi aspetti funzionali del sistema, valutando la risposta in termini di accuratezza, completezza e tempi di risposta.

7.1 Metodologia di valutazione

Al fine di valutare in modo oggettivo le prestazioni dei chatbot, sono state definite cinque metriche di valutazione, ciascuna pesata in base alla sua importanza relativa per l'esperienza di utilizzo finale complessiva.

La prima metrica, il *tempo di risposta* (T), misura l'efficienza computazionale del sistema ed è stata quantificata su scala 0-3 con peso 2, convertendo il tempo misurato in secondi in un punteggio a fasce: 3 punti per risposte entro 10 secondi, 2 punti per risposte tra 10 e 25 secondi, 1 punto per risposte tra 26 e 50 secondi e 0 punti per risposte superiori a 50 secondi.

Il secondo parametro, *identificazione della specie corretta* (Sp), verifica se il modello ha correttamente identificato e mantenuto il contesto della specie oggetto della domanda e viene misurata su scala 0-2 con peso unitario, dove 2 indica che l'identificazione è avvenuta correttamente, 1 indica un'identificazione parziale o confusa e 0 indica un errore o una mancata identificazione.

L'*aderenza al documento* (Ad), è il terzo criterio di valutazione e misura quanto la risposta utilizzi esclusivamente informazioni presenti nella base documentale fornita senza aggiungere o inventare dettagli. È misurata su scala 0-2 con peso 2, dove un punteggio di 2 indica aderenza completa, 1 indica piccole imprecisioni o aggiunte e 0 indica allucinazioni o informazioni errate.

Come quarta metrica abbiamo la *completezza della risposta* (Cmp), che misura quanto la risposta copra tutte le richieste della domanda senza omettere informazioni. È valutata su scala 0-2 con peso 1, dove 2 indica completezza totale, 1 una copertura parziale e 0 una risposta incompleta.

L'ultimo parametro, *gestione del contesto* (Ctx), valuta la capacità del modello di mantenere coerenza conversazionale quando vengono poste domande follow-up e di gestire correttamente i cambi di specie tra domande successive, viene valutato su scala 0-2 con peso 1, dove 2 indica una gestione del contesto ottima, 1 una gestione parziale e 0 assente o errata.

Le metriche e i relativi punteggi sono riassunti nella tabella 7.1 sottostante:

Tabella 7.1: Criteri di assegnazione dei punteggi per ciascuna metrica

Metrica	**Punteggio**	**Criterio di assegnazione**
Tempo (T) peso 2	3	Risposta entro 10 secondi
	2	Risposta tra 10 e 25 secondi
	1	Risposta tra 26 e 50 secondi
	0	Risposta oltre 50 secondi
Specie (Sp) peso 1	2	Identificazione corretta
	1	Identificazione parziale
	0	Specie errata o assente
Aderenza (Ad) peso 2	2	Basata esclusivamente sui documenti
	1	Lievi imprecisioni o piccole aggiunte
	0	Allucinazioni o informazioni errate
Completezza (Cmp) peso 1	2	Risposta completa
	1	Copertura parziale
	0	Risposta incompleta
Contesto (Ctx) peso 1	2	Gestione corretta del contesto
	1	Gestione parziale o incoerente
	0	Gestione errata o assente

Il punteggio totale per ciascuna domanda è stato calcolato sommando i punteggi pesati delle cinque dimensioni con un massimo di 16 punti per domanda:
Totale $= 2T + Sp + 2Ad + Cmp + Ctx$.

7.2 Descrizione delle domande di test

Le domande di test sono 10 e coprono diversi scenari e livelli di difficoltà: (i) descrizioni base di singole specie, (ii) domande di follow-up che testano la gestione del contesto, (iii) confronti tra specie simili che richiedono implicitamente sintesi di più informazioni, (iv) domande su dettagli numerici specifici, (v) domande su informazioni particolari più difficili da recuperare, e (vi) una domanda test richiedente informazioni non presenti nei documenti per verificare il comportamento del sistema in assenza di dati rilevanti.

Ogni domanda è stata posta in sequenza ad ognuna delle tre varianti e, per ognuna, sono stati registrati i tempi di risposta tramite `time.perf_counter()` convertiti poi in punteggio secondo le fasce citate sopra.

Verranno ora descritte le domande di test corredate da una breve spiegazione del motivo per cui sono state scelte e quali aspetti del sistema intendono testare.

La domanda D1, *"Descrivimi il tursiope in modo semplice: aspetto e dimensioni"*, è una domanda descrittiva base che richiede al sistema di fornire informazioni morfologiche su una specie comune come il tursiope. Testa la capacità di retrieval semantico su concetti generali e di sintesi delle informazioni raccolte.

La domanda D2, *"E cosa mangia di solito?"*, è una domanda follow-up per testare la gestione del contesto: l'utente non ripete la specie ma si aspetta che il sistema leghi la domanda al tursiope menzionato in D1.

La domanda D3, *"Come riconosceresti una stenella striata e dove vive di solito?"*, cambia il soggetto della conversazione menzionando una nuova specie. Testa in questo modo la capacità del sistema di resettare e cambiare il contesto precedente, recuperando, al tempo stesso, informazioni sulla nuova specie menzionata.

La domanda D4, *"Differenze principali tra tursiope, delfino comune e stenella striata"*, richiede un confronto strutturato tra tre specie simili. È una domanda complessa che richiede al sistema di recuperare informazioni su tutte e tre le specie e sintetizzarle in modo comparativo, testando la capacità di gestire query multi-entità.

La domanda D5, *"Quanto è grande e quanto pesa un capodoglio?"*, richiede informazioni numeriche specifiche con distinzione tra maschi e femmine. Testa la precisione nell'estrazione di dettagli quantitativi e la capacità di fornire informazioni differenti per sottogruppi (maschi vs femmine) all'interno della stessa specie.

La domanda D6, *"E che tipo di gruppi sociali forma?"*, è un'altra domanda follow-up come la domanda D2.

La domanda D7, *"Perché lo zifio è considerato 'elusivo' e che tipo di immersioni fa?"*, richiede informazioni su caratteristiche comportamentali particolari testando la capacità

di retrieval su concetti più specifici.

La domanda D8, *"Qual è la caratteristica più riconoscibile della balenottera comune e come si nutre?"*, combina una richiesta di caratteristica distintiva con un'informazione comportamentale, testando la capacità di sintesi di informazioni eterogenee tra loro.

La domanda D9, *"Differenze tra Caretta caretta e Chelonia mydas: riconoscimento e alimentazione"*, è un confronto strutturato tra due specie di tartarughe marine, che testa nuovamente la capacità comparativa tra due specie simili e quindi facilmente confondibili.

Infine, la domanda D10, *"Qual è la velocità massima del pesce luna?"*, è un test di rifiuto progettato per verificare il comportamento del sistema quando l'informazione richiesta non è presente nella base documentale. Il documento contiene, infatti, informazioni sul pesce luna ma non sulla sua velocità massima, per questo ci si aspetta la dichiarazione esplicita da parte del sistema dell'assenza dell'informazione.

7.3 Risultati

I risultati della valutazione vengono ora presentati separatamente per ciascuna variante, con analisi delle prestazioni su ogni domanda.

7.3.1 Chatbot originale (mxbai + Qwen 2.5)

Il chatbot originale, basato su `nomic-embed-text` e `Phi-3 mini`, è quello che ha ottenuto la valutazione complessiva più alta tra le varianti. La tabella seguente 7.2 riassume i risultati dettagliati per ciascuna domanda.

Tabella 7.2: Risultati del chatbot nomic + Phi-3

Dom.	**Tempo (s)**	**T**	**Sp**	**Ad**	**Cmp**	**Ctx**	**Tot**	**Note**
D1	11.3	2	2	1	2	2	9	Misure imprecise
D2	5.99	3	2	2	2	2	11	
D3	6.38	3	2	1	2	2	10	Generalizza oceani
D4	7.58	3	2	1	2	2	10	Poco preciso
D5	6.42	3	2	2	2	2	11	
D6	6.07	3	2	2	2	2	11	
D7	7.07	3	2	2	2	2	11	
D8	7.42	3	2	2	2	2	11	
D9	7.58	3	1	1	1	2	8	Errore su dieta
D10	5.11	3	2	2	2	2	11	
Tot.	70.92	29	19	16	19	20	**103**	

T=Tempo (peso 2), Sp=Specie (peso 1), Ad=Aderenza (peso 2),
Cmp=Completezza (peso 1), Ctx=Gestione Contesto (peso 1)

Il chatbot ha ottenuto un punteggio non pesato di 103 su 110 ma, applicando i pesi alle diverse metriche, il punteggio pesato finale è di 148 su 160 che corrisponde al 92.5% delle prestazioni ideali. Il tempo medio di risposta è stato di 7.09 secondi, molto inferiore alla soglia dei 10 secondi, con 9 domande su 10 ottenenti il punteggio massimo nel parametro tempo, simbolo di un'efficienza computazionale valida e di una conseguente esperienza utente fluida.

Dal punto di vista di aderenza documentale, il chatbot ha mostrato generalmente una buona fedeltà alle informazioni presenti nel documento mostrando, tuttavia, alcune imprecisioni. Nella domanda D1, infatti, sono state riportate misure leggermente imprecise rispetto ai valori esatti del documento; inoltre, nella domanda D3 il sistema ha generalizzato la distribuzione geografica della stenella striata a *"tutti gli oceani"*, quando il documento fa riferimenti più specifici ad aree temperate e tropicali. Nella domanda D4 le descrizioni comparative sono risultate meno precise di quanto desiderato e nella domanda D9 il sistema ha commesso un errore descrivendo la dieta della Chelonia mydas come onnivora, mentre il documento specifica che gli adulti sono quasi esclusivamente erbivori.

La gestione del contesto conversazionale è stata eccellente. Le domande follow-up D2 e D6 hanno, infatti, mantenuto correttamente il riferimento alla specie precedente, e i cambi di specie espliciti sono stati gestiti senza problemi.

Infine, la domanda test D10 ha prodotto il comportamento atteso: il sistema ha correttamente dichiarato l'assenza dell'informazione richiesta nel documento.

7.3.2 Chatbot variante 1 (mxbai + Phi-3)

La variante che utilizza `mxbai-embed-large` e `Qwen 2.5`, ottimizzata per la qualità delle risposte, ha ottenuto risultati contrastanti, con un punteggio complessivo inferiore rispetto al chatbot originale. La tabella seguente 7.3 presenta i dettagli dei risultati ottenuti da questa variante.

Il punteggio totale di 92 su 100, pesato secondo i criteri definiti, diventa 126 su 160, ovvero il 78.75% delle prestazioni ottimali, risultato molto inferiore rispetto al chatbot originale.

Il fattore che più ha penalizzato questa variante è stato il tempo di risposta; con una media di 30.7 secondi per domanda, Qwen 2.5 risulta circa 5 volte più lento di Phi-3 mini, tempi che non permettono una user experience fluida. La domanda D4, in particolare, ha richiesto oltre 74 secondi per essere elaborata, ottenendo 0 punti nella metrica tempo.

Dal punto di vista qualitativo, invece, le risposte si sono dimostrate accurate, complete e aderenti al documento, ottenendo il punteggio massimo (20/20) sia nell'aderenza docu-

Tabella 7.3: Risultati del chatbot mxbai + Qwen 2.5

Dom.	Tempo (s)	T	Sp	Ad	Cmp	Ctx	Tot	Note
D1	25.01	2	2	2	2	2	10	Lento
D2	19.34	2	2	2	2	2	10	Lento
D3	33.54	1	2	2	2	2	9	Molto lento
D4	74.44	0	2	2	1	2	7	Troppo prolisso
D5	24.02	2	2	2	2	2	10	Lento
D6	22.14	2	2	2	2	2	10	Lento
D7	27.85	1	2	2	1	2	8	Prolisso
D8	27.49	1	2	2	2	2	9	Molto lento
D9	33.92	1	2	2	2	2	9	Molto lento
D10	19.46	2	2	2	2	2	10	Lento
Tot.	307.21	14	20	20	18	20	**92**	

mentale che nella gestione del contesto. Le risposte sono risultate spesso troppo prolisse, infatti, Qwen 2.5 tende a generare risposte molto dettagliate che, sebbene accurate, vanno in contrasto con l'obiettivo di fornire risposte rapide, penalizzando quindi il punteggio di completezza in alcune domande come D4 e D7.

7.3.3 Chatbot variante 2 (nomic + Phi-3)

I risultati peggiori sono stati ottenuti dalla variante intermedia con `mxbai-embed-large` e `Phi-3 mini`. La tabella 7.4 seguente evidenzia le problematiche emerse.

Tabella 7.4: Risultati del chatbot mxbai + Phi-3

Dom.	Tempo (s)	T	Sp	Ad	Cmp	Ctx	Tot	Note
D1	7.36	3	1	1	2	2	9	Confonde specie
D2	5.64	3	2	1	2	2	10	Aggiunge info
D3	6.43	3	2	2	2	2	11	
D4	7.46	3	2	1	1	2	9	Aggiunge info
D5	5.22	3	0	0	0	0	3	**Fallimento**
D6	—	0	0	0	0	0	0	**Fallimento**
D7	10.37	3	2	1	2	2	10	Aggiunge info
D8	6.39	3	2	1	2	2	10	Poco preciso
D9	6.13	3	1	1	1	2	8	Errore su dieta
D10	5.34	3	2	2	2	2	11	
Tot.	60.34	27	14	10	14	16	**81**	

Il punteggio non pesato di 81 su 110 diventa 118 su 160, ovvero il 73.75% delle prestazioni ottimali, dopo l'applicazione dei pesi, rivelandosi il più basso fra le tre varianti.

Nonostante i tempi medi di risposta eccellenti (6.7 secondi), il problema più rilevante è stato il fallimento ottenuto sulle domande D5 e D6, dove il sistema non è riuscito ad identificare e recuperare informazioni sul capodoglio, ottenendo di conseguenza un punteggio di 0 in tutti i parametri. Aumentare il numero di chunk recuperati (`similarity_top_k`) avrebbe potuto risolvere il problema ma, al contempo, avrebbe saturato la memoria disponibile.

Le altre domande hanno mostrato alcuni problemi di aderenza documentale (solo 20 punti pesati su 40, contro i 32 di nomic+Phi-3 e i 40 di mxbai+Qwen). Il sistema ha, infatti, spesso aggiunto informazioni non richieste come la cooperazione con i pescatori nella risposta su alimentazione del tursiope (D2), informazioni sulla riproduzione nel confronto tra specie (D4 e D7), e lo stesso errore sulla dieta della Chelonia mydas (D9).

Nella domanda D1, inoltre, è stata fornita una risposta errata, in cui il sistema ha confuso il tursiope con il delfino comune, indicando un problema di retrieval.

7.4 Analisi comparativa

La tabella 7.5 sintetizza i risultati complessivi delle tre varianti utilizzando i punteggi pesati.

Tabella 7.5: Confronto generale tra le varianti (punteggi pesati)

Variante	**Tempo (/60)**	**Specie (/20)**	**Aderenza (/40)**	**Compl. (/20)**	**Contesto (/20)**	**Totale (/160)**
nomic + Phi-3	58	19	32	19	20	**148** (92.5%)
mxbai + Qwen	28	20	40	18	20	**126** (78.75%)
mxbai + Phi-3	54	14	20	14	16	**118** (73.75%)

La tabella 7.6 fornisce, invece, maggiori dettagli sulle prestazioni.

Tabella 7.6: Metriche aggiuntive di confronto

Metrica	**nomic + Phi-3**	**mxbai + Qwen**	**mxbai + Phi-3**
Tempo medio (s)	7.09	30.72	6.70
Tempo totale (s)	70.92	307.21	60.34
Fallimenti critici	0	0	2
Domande perfette (11/11)	6/10	3/10	3/10

Grazie alla configurazione più equilibrata, il chatbot nomic + Phi-3 si è dimostrato il chatbot più performante. La combinazione di velocità eccellente (58/60 punti, con un tempo medio di 7.09 secondi) e la qualità delle risposte molto alta rappresenta il vantaggio principale di questa variante.

La variante mxbai + Qwen ottiene i punteggi massimi in aderenza (40/40) e gestione contesto (20/20), dimostrando la superiorità qualitativa di Qwen 2.5 rispetto a Phi-3 mini, ma il tempo di risposta estremamente lento (28/60 punti) penalizza pesantemente il punteggio complessivo, rendendo questa configurazione inadatta per applicazioni come quella in esame, dove la rapidità della risposta ricopre un ruolo fondamentale.

La variante mxbai + Phi-3, pur essendo la più veloce (54/60), soffre di problemi rilevanti in tutte le altre dimensioni: aderenza documentale insufficiente (20/40, metà dei punti massimi), identificazione delle specie problematica (14/20) e gestione del contesto non ottimale (16/20). I due fallimenti critici la rendono, di conseguenza, la soluzione meno affidabile.

7.5 Discussione dei risultati

L'analisi dei risultati fa emergere molte informazioni interessanti sui trade-off tra prestazioni complessive e scelte tecnologiche nei sistemi RAG.

Il primo punto da sottolineare è la rilevanza del tempo di risposta come metrica, infatti, nonostante Qwen 2.5 abbia prodotto risposte di qualità leggermente superiore rispetto a Phi-3 mini, la differenza non compensa i tempi di attesa 5 volte superiori. Per chatbot conversazionali come quello che ci si poneva di realizzare, la velocità di risposta è un fattore che influenza enormemente la fluidità dell'esperienza utente, e tempi superiori ai 30 secondi non possono essere considerati accettabili, indipendentemente dalla qualità delle risposte. Phi-3 mini, pur essendo significativamente più piccolo, riesce a mantenere un buon livello di qualità unito a tempi di risposta eccellenti, dimostrando che gli Small Language Models possono essere preferibili a modelli più grandi in scenari dove la rapidità è un vincolo fondamentale.

La seconda osservazione riguarda la precisione del retrieval semantico. La variante con mxbai-embed-large, nonostante abbia mostrato una precisione maggiore nei benchmark MTEB, non ha prodotto miglioramenti significativi nella qualità delle risposte nel dominio considerato. Questo dimostra che, oltre una certa soglia di qualità di embedding, altri fattori diventano limitanti, come la formulazione del prompt, la capacità di sintesi dell'LLM e la dimensione del chunking, che sono di grande impatto nella qualità finale delle risposte.

Si vuole inoltre porre l'attenzione sull'aderenza documentale, nella quale entrambe le varianti implementate con Phi-3 mini hanno mostrato la tendenza ad essere imprecise o ad aggiungere informazioni non presenti nel documento, mentre Qwen è risultato più fedele.

Per quanto riguarda la gestione del contesto conversazionale, tutte e tre le varianti hanno dimostrato una buona capacità di mantenere coerenza tra domande follow-up. Nonostante la variante mxbai + Phi-3 abbia fallito nel recuperare informazioni su una determinata specie, si può affermare che il problema sia attribuibile al retrieval piuttosto che alla gestione del contesto.

In sintesi, per il caso d'uso specifico, la configurazione nomic + Phi-3 si afferma come la scelta migliore grazie al bilanciamento che offre tra velocità, accuratezza e aderenza documentale. La variante mxbai + Qwen 2.5, nonostante in questo contesto non risulti adatta a causa dei tempi di risposta troppo elevati, potrebbe essere considerata in scenari dove la qualità della risposta è prioritaria e il tempo non viene considerato un vincolo critico. L'alternativa mxbai + Phi-3 mini, invece, non è raccomandata a causa dei fallimenti critici osservati che ne compromettono l'affidabilità complessiva.

Capitolo 8

Conclusioni

La presente tesi ha affrontato la progettazione, l'implementazione e la valutazione comparativa di chatbot informativi basati su architettura Retrieval-Augmented Generation descrivendone anche le tecnologie utilizzate e i risultati ottenuti.

Il lavoro si è sviluppato in diverse fasi, partendo dalla raccolta dei dati, per finire alla valutazione delle configurazioni implementate. La prima fase si è occupata della raccolta di informazioni scientifiche su specie marine presenti nei nostri mari, e della loro successiva organizzazione in un documento che costituisce la base di conoscenza su cui si fonda l'intero sistema.

Il sistema segue l'architettura RAG, al fine di garantire che le risposte fornite dal chatbot siano basate esclusivamente sui contenuti del documento redatto nella fase precedente così da evitare un fenomeno noto come allucinazione, frequente nei sistemi basati su LLM standalone.

Sono state sviluppate tre varianti del chatbot, ognuna con una diversa combinazione di modello di embedding e LLM per valutare la scelta tecnologica più adatta al caso d'uso richiesto: (i) la configurazione principale con nomic-embed-text e Phi-3 mini, che offre efficienza e velocità, (ii) una variante con mxbai-embed-large e Qwen 2.5, orientata alla massima qualità delle risposte e (iii) una variante intermedia con mxbai-embed-large e Phi-3 mini.

La valutazione comparativa delle tre soluzioni è stata condotta ponendo loro sequenzialmente un insieme di 10 domande di test progettate per coprire diversi scenari e livelli di difficoltà. Sono state poi definite cinque metriche di valutazione, con pesi differenti in base alla loro importanza relativa per la user experience complessiva: (i) tempo di risposta e (ii) aderenza al documento, con peso 2, (iii) identificazione della specie corretta, (iv) completezza della risposta e (v) gestione del contesto conversazionale, con peso 1.

I risultati ottenuti hanno identificato la configurazione nomic + Phi-3 mini come la

soluzione complessivamente migliore, ottenendo un punteggio pesato di 148 su 160, che costituisce il 92.5% delle prestazioni ideali, grazie a un ottimo bilanciamento tra velocità e qualità delle risposte.
La variante mxbai + Qwen 2.5, nonostante abbia prodotto risposte di qualità superiore, ha ottenuto un punteggio complessivo inferiore (126/160, 78.75%) a causa dei tempi di risposta troppo elevati per poter garantire all'utente un'esperienza fluida.
La variante mxbai + Phi-3 ha mostrato, invece, fallimenti critici in alcune domande che ne hanno compromesso l'affidabilità (118/160, 73.75%).

Alla luce dei risultati si può affermare che configurazioni teoricamente superiori, come quella con mxbai-embed-large e Qwen 2.5, non sempre offrono prestazioni complessive migliori quando si considerano vincoli come tempo di risposta e possibilità di esecuzione su ambienti con risorse limitate.

8.1 Limitazioni

Sebbene i risultati siano stati positivi,è importante riconoscere che il lavoro presenta alcune limitazioni che potrebbero costituire le fondamenta per eventuali sviluppi futuri. La prima limitazione riguarda le dimensioni della base documentale; l'espansione a un corpus più ampio di documenti potrebbe richiedere adattamenti nelle strategie di chunking e di retrieval per poter mantenere le prestazioni raggiunte.

La valutazione è stata condotta su un set limitato di 10 domande che, nonostante siano state pensate per coprire diversi scenari, non possono rappresentare l'intero insieme di possibili domande che un utente potrebbe porre; per questo sarebbe interessante testare il sistema con utenti reali e, di conseguenza, un numero maggiore di domande.

Il sistema è stato sviluppato e testato in lingua italiana e le sue prestazioni potrebbero variare se messo alla prova con domande in lingue diverse, considerando soprattutto che Phi-3 mini è specializzato principalmente per l'inglese.

8.1.1 Scalabilità del sistema

Il sistema implementato è stato progettato per una base documentale di dimensioni contenute; verrà analizzato ora come il sistema si comporterebbe in scenari di scala maggiore, identificando le potenziali problematiche e le revisioni architetturali necessarie.

Espansione del numero di specie

Con un numero molto maggiore di specie, si verificherebbe un aumento del numero di chunk nell'indice vettoriale. Il meccanismo di riconoscimento della specie nella query , basato su confronto a stringa, su tutti i nomi ordinati per lunghezza, rappresenta un problema molto più rilevante poichè la lista potrebbe contenere anche migliaia di voci, aumentando il rischio di falsi positivi.

Una soluzione potrebbe essere quella di adottare un modello di *Named Entity Recognition* (NER) specializzato su specie marine, oppure un classificatore di intenzione per identificare la specie di riferimento senza dipendere dal confronto lessicale diretto.

Documenti più corposi e multi-fonte

Se ogni specie fosse documentata con fonti più articolate, come studi scientifici, la strategia di chunking a dimensione fissa risulterebbe meno adeguata. Con documenti molto lunghi ed eterogenei, sarebbe infatti preferibile ricorrere al *semantic chunking*, che individua automaticamente i punti di cambio semantico nel testo, o al *document-based cunkuing*, che sfrutta la struttura del documento come confini dei chunk.

Il parser attuale dipende fortemenre dalla struttura del documento di input. Per poter supportare formati diversi si dovrebbero utilizzare parser dedicati per ogni fonte o un approccio basato su LLM per l'estrazione automatica dei metadati indipendentemente dalla struttura.

Context window e memoria

Un corpus documentale più ampio aumenta il rischio di saturare la context window del modello linguistico. Con Phi-3 mini, configurato a 4096 token e `top_k=5` chunk da 512 token ciascuno, si utilizzano già circa 2500 token di contesto, lasciando poco spazio per query elaborate. Aumentare `chunk_size` o `similarity_top_k` richiederebbe di adottare un LLM con context window più ampia.

Inoltre, il sistema attuale carica in memoria l'intero testo del documento all'avvio per eseguire il parsing. Per questo, con una base documentale molto estesa, sarebbe efficiente eseguire il parsing e l'indicizzazione una volta soltanto e in una fase di setup separata, delegando poi a ChromaDB l'intero recupero dei dati, senza necessità di mantenere i documenti originali nella RAM durante l'esecuzione.

Sviluppi futuri

Le limitazioni trattate sopra aprono, di conseguenza, diverse strade per possibili futuri sviluppi. Potrebbero essere implementate strategie di retrieval più avanzate e un chunking semantico con dimensione variabile che rispetti la struttura logica delle informazioni, che potrebbe sostituire il chunking a dimensione fissa attuale.

Si potrebbero inoltre integrare al chatbot modelli di vision-language che permetterebbero agli utenti di caricare foto di avvistamenti e ricevere identificazioni in tempo reale, o estendere il dominio di conoscenza del sistema a ulteriori specie marine o altri ambiti correlati.

In conclusione, il presente lavoro ha mostrato lo sviluppo di un sistema RAG eseguibile localmente, valutandone e identificandone la configurazione ottimale tra le tre sviluppate. Eventuali sviluppi futuri potrebbero estendere le capacità del sistema, che contribuirebbe all'espansione di strumenti di divulgazione scientifica affidabili e sempre più accessibili, consolidando un rapporto saldo tra intelligenza artificiale e scienza.

Bibliografia

[1] LlamaIndex Documentation, `https://developers.llamaindex.ai`

[2] Ollama Documentation, `https://docs.ollama.com/`

[3] ChromaDB Documentation, `https://docs.trychroma.com`

[4] Streamlit Documentation, `https://docs.streamlit.io`

[5] A. Vaswani et al., *Attention Is All You Need*, NeurIPS, 2017.

[6] *Why Google's AI Overviews gets things wrong*, technologyreview.com, 31 May 2024.

[7] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *Efficient estimation of word representations in vector space.*.

[8] Pennington, J., Socher, R., & Manning, C. D. (2014). GloVe: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 1532-1543.

[9] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). *BERT: Pre-training of deep bidirectional transformers for language understanding.*.

[10] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). *Deep contextualized word representations.*.

[11] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 3982-3992.

[12] Nussbaum, Z., Morris, J., Duderstadt, B., & Mulyar, A. (2024). *Nomic embed: Training a reproducible long context text embedder.*.

[13] Abdin, M., Jacobs, S. A., Awan, A. A., Aneja, J., Awadallah, A., Awasthi, A., ... & Zhou, X. (2024). *Phi-3 technical report*.

[14] Gunasekar, S., Zhang, Y., Aneja, J., Chaudhary, A., Del Giorno, A., Gopi, S., ... & Mitra, A. (2023). *Textbooks are all you need.*.