# La creazione di un assistente virtuale per lo studio

Questo lavoro di tesi ha l’obiettivo di realizzare un assistente virtuale (*Learny*) che supporti gli studenti nello svolgimento di un compito, nella preparazione di un esame e, più in generale, nell’approfondimento di uno specifico argomento.

L’idea è nata dall’osservazione del lavoro pomeridiano di mia moglie, insegnante di sostegno nella scuola primaria, che la impegna sia nella realizzazione di materiali didattici personalizzati sia nella ricerca delle più opportune strategie didattiche mirate a stimolare l’attenzione e agevolare l’apprendimento, anche attraverso l’uso di giochi, quiz e varie attività che consentano d’imparare, divertendosi.

In tale contesto, l’intelligenza artificiale (AI – “Artificial Intelligence”) può rappresentare un formidabile strumento, tramite cui realizzare applicazioni interattive, innovative e flessibili, di facile utilizzo quando rivolte ai bambini ma anche in grado di fornire risposte accurate a svariati problemi di business, in molteplici contesti di maggiore complessità.

Naturalmente, più elevata è le complessità degli argomenti affrontati e maggiore sarà la difficoltà nell’ottenere risposte precise e verificabili rispetto alle fonti di provenienza. Sorge quindi l’esigenza di adottare tecniche efficaci per controllare l’AI stessa.

# La metodologia RAG

*Learny* è un assistente che deve saper acquisire la conoscenza di un argomento o materia d’interesse per lo studente ed interagire con lui utilizzando la sua stessa lingua. Per realizzarlo è quindi necessario far ricorso ad un sistema di intelligenza artificiale generativa (GenAI), specializzato nella generazione di testi e in grado di rispondere utilizzando sia la propria conoscenza (interna) sia la conoscenza (esterna) fornitagli dall’utente.

Nella categoria dei sistemi GenAI si possono in generale annoverare tutti i sistemi AI in grado di generare varie tipologie di contenuti, siano essi testi, immagini, video o audio. In questo lavoro verrà utilizzata una particolare declinazione dei sistemi GenAI, appartenente alla categoria dei "Large Language Model" (LLM), cioè un tipo di intelligenza artificiale progettata per comprendere e generare testo in linguaggio naturale. I modelli LLM utilizzano tecniche di apprendimento automatico, per svolgere svariati compiti, come generare parole e frasi in modo coerente e contestuale, rispondere a domande, scrivere testi e tradurre. Essi

vengono implementati attraverso reti neurali (NN – “Neural Networks”) specificamente addestrate su enormi quantità di dati testuali (da cui l’attributo “large”) contenente miliardi (o migliaia di miliardi) di parole recuperate da Internet e da altre fonti. A differenza dei precedenti sistemi di generazione *basati su regole*, la principale caratteristica distintiva di un LLM è che può produrre un testo nuovo e originale che si legga in modo naturale.

Poiché possono apprendere un’infinità di schemi, gli LLM acquisiscono le competenze linguistiche presenti nei dati di training, dalle sfumature grammaticali alla conoscenza degli argomenti, riuscendo a generare testo scritto in modo sintatticamente corretto e con contenuti aderenti al contesto.

I LLM guidano attualmente l’evoluzione della tecnologia NLG (“Natural Language Generation”): li troviamo nei chatbot, nei motori di ricerca, negli assistenti per la generazione di codice nei diversi linguaggi di programmazione, e in generale in tutte le applicazioni che coinvolgono la generazione di testo scritto. Essi sono inoltre il motore che consente il funzionamento degli *agent*, automi in grado di percepire e interpretare stimoli provenienti dall’ambiente, non solo quello digitale, per prendere decisioni e agire di conseguenza, scegliendo automaticamente quali strumenti (*tools*) utilizzare, fra quelli resi loro disponibili all’interno dell’applicazione.

Non mancano, tuttavia, gli aspetti problematici.

Fra questi, l’inadeguatezza nel fornire dati aggiornati in tempo reale (essendo gli LLM addestrati su grandi dataset statici raccolti in precedenza) rappresenta certamente una forte limitazione, in quanto le risposte prodotte potrebbero essere datate e quindi potenzialmente irrilevanti o persino errate e comunque fuorvianti.

Altri problemi sono legati all’incapacità di distinguere i fatti veri dalle falsità nonché le difficoltà che sorgono quando ci si muove in ambiti di conoscenza relativi a settori molto specifici o particolarmente complessi: senza un opportuno controllo sulle risposte, i LLM incorrono frequentemente in “allucinazioni”, generando risposte tanto inattendibili quanto peraltro molto convincenti, specialmente quando si trovano a gestire domande che esulano dai dati di training o che richiedono informazioni aggiornate.

Una metodologia che cerca di superare queste limiti è la RAG (“Retrieval - Augmented Generation”), fondata sull’idea di partire dal recupero (retrieval) di informazioni rilevanti rispetto ai temi da trattare, estraendole da una o più fonti di dati “esterne”[[1]](#footnote-2), contenenti la cosiddetta “conoscenza proprietaria”, e sfruttarle per generare una risposta significativa, aggiornata e aderente al contesto: integrando questa conoscenza, il modello risulta di fatto “potenziato” ed in grado di fornire risposte più accurate e pertinenti, migliorando la sua capacità di generare contenuti.

La metodologia RAG ha iniziato ad affermarsi nella sua forma più semplice (*Naive* *RAG*) subito dopo l'adozione diffusa di ChatGPT, ma la ricerca è in continua evoluzione ed ha portato a forme più complesse e raffinate di RAG, nate dall’esigenza di superare le limitazioni del modello iniziale. Il Naive RAG può essere descritto come un processo basato sulle fasi di *indicizzazione*, *recupero* e *generazione*.

Il processo inizia con l’acquisizione delle informazioni, che vengono estratte da file di diverso formato (Word, Pdf, HTML, Markdown, etc.) contenenti i dati “grezzi”, i quali vengono “puliti”, convertiti in documenti di testo semplice e successivamente segmentati in blocchi più piccoli (*chunk*) per renderli più agevolmente gestibili da parte del LLM.

I blocchi vengono quindi codificati secondo la rappresentazione vettoriale prevista dal modello di *embedding* del LLM, indicizzati e memorizzati in un file o in un database.

Se la trasformazione in vettori numerici è un passaggio cruciale per consentire ricerche di similarità nella fase successiva di recupero, la loro indicizzazione è fondamentale per renderle efficienti. Al ricevimento di una richiesta (*query*) da parte dell'utente, il sistema RAG utilizzerà lo stesso modello di codifica per estrarne una rappresentazione vettoriale, che verrà confrontata con i vettori memorizzati nel database, attribuendo a ciascuno di essi un punteggio (calcolato applicando un criterio di somiglianza algebrica con il vettore query) e selezionandone infine un certo numero. Questi chunk codificati costituiranno un “contesto” di conoscenza proprietaria, da aggiungere alla query iniziale dell’utente per preparare le istruzioni (*prompt*) da inviare al LLM, al fine di ottenere una risposta. In pratica si tratta di porre un certo quesito al LLM, chiedendogli di formulare la risposta, tenendo anche conto del contesto fornitogli.

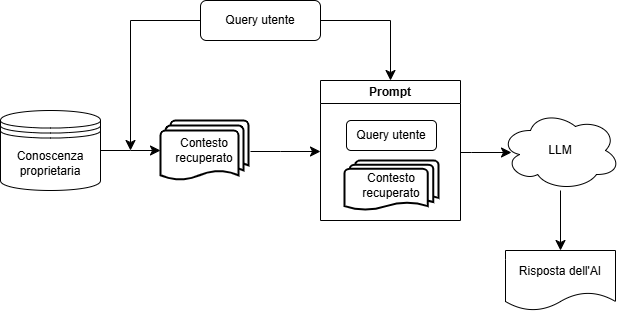


Figura 1 - Naive RAG

L'approccio alla risposta può variare a seconda dei criteri stabiliti per lo svolgimento di una particolare attività: al LLM può essere richiesto di attingere alla propria conoscenza oppure di circoscrivere le risposte alle informazioni contenute all'interno dei documenti forniti. Se si realizza un chatbot, è invece possibile integrare nel prompt la cronologia delle conversazioni precedenti, che fungerà da contesto, permettendo al modello di generare dialoghi fluidi, naturali e non ripetitivi o banali.

Il Naive RAG presenta tuttavia diversi inconvenienti. Il retrieval è talvolta inefficace in termini di precisione e recall[[2]](#footnote-3), con conseguente selezione di chunk irrilevanti e perdita di quelli rilevanti. In fase di generazione possono ancora presentarsi *allucinazioni* (un tipico problema dei LLM che la RAG si propone di risolvere)*,* cioè risposte errate, irrilevanti o inattendibili, tanto più frequenti quanto meno i contenuti utilizzati per la risposta siano supportati dal contesto recuperato dalla conoscenza proprietaria e fornito al LLM all’interno del prompt. Viceversa, il recupero d’informazioni simili da sorgenti differenti, può condurre a risposte ripetitive, che ne inficiano anche la qualità dal punto di vista stilistico. Inoltre, di fronte a materie complesse, un unico retrieval basato sulla query originale potrebbe non essere sufficiente per acquisire adeguate informazioni di contesto. Infine, basarsi eccessivamente sulla conoscenza proprietaria, potrebbe generare risposte che la riproducano pedissequamente, senza integrarla con ulteriori approfondimenti o sintetizzarne opportunamente i contenuti.

Ecco perché, come già evidenziato, la ricerca ha raffinato nel tempo la metodologia RAG, sviluppando ulteriori e più sofisticati approcci, come l’Advanced RAG e il Modular RAG, con l’obiettivo di colmare le carenze del Naive RAG.

# L’applicazione

L’applicazione *Learny* è stata realizzata utilizzando **LlamaIndex**[[3]](#footnote-4), un framework open source progettato per sviluppare, fra l’altro, sistemi che richiedono l’integrazione fra l’uso di contenuti informativi basati su una conoscenza di tipo proprietario e la capacità di generazione di testi tipica di un LLM, rappresentando di fatto un eccellente strumento per realizzare un’applicazione RAG.

Il software è stato scritto utilizzando il linguaggio Python, standard de facto per la realizzazione di applicazioni di intelligenza artificiale, grazie ad una vasta comunità di sviluppatori e ricercatori che contribuiscono costantemente con strumenti e librerie specifiche che agevolano enormemente lo sviluppo.

Learny si presenta all’utente tramite una semplice interfaccia grafica basata su Streamlit[[4]](#footnote-5), chiedendogli di comunicargli l’argomento che vuole preparare e invitandolo a caricare i relativi materiali di studio (file in vari formati, fra cui testo semplice, .docx, .pdf, etc.), al fine di poter successivamente interagire con lui tramite una chat e un generatore di quiz.

Il chatbot risponde sia a domande generiche sia specifiche, cioè riconducibili al materiale fornito. Le domande del quiz vengono generate basandosi sia sul materiale fornito sia, più in generale, sull’argomento di studio. Per realizzare le funzionalità necessarie, LlamaIndex interagisce con alcuni LLM di OpenAI (Ada v2 e GPT 3.5).

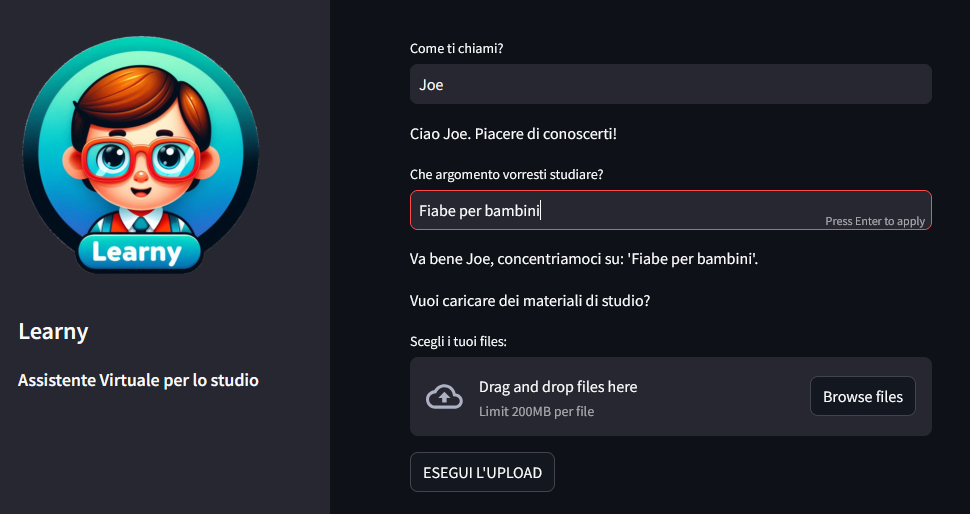


Figura 2 - Interfaccia grafica dell'applicazione

## Acquisizione dei materiali di studio e creazione dei nodi

Il modulo *document\_uploader.py* contiene la funzione *ingest\_documents*, responsabile della gestione del processo di acquisizione (*ingestion*) dei materiali di studio forniti dall’utente, sotto forma di file di differenti formati, e della loro trasformazione al fine di renderli utilizzabili nella successiva fase di retrieval.

Viene innanzitutto creato un lettore (reader) di directory di tipo *SimpleDirectoryReader*, che si adatta automaticamente al caricamento di differenti tipi di file e ha fra i suoi parametri la cartella in cui cercare i file da acquisire:

**reader = SimpleDirectoryReader(STORAGE\_PATH, filename\_as\_id=True)**

Invocando il metodo *load\_data()*, il reader estrae le informazioni contenute nei file presenti nella cartella e genera una lista di oggetti di tipo *Document*:

**documents = reader.load\_data()**

Il numero di elementi della lista, cioè di *documents* generati, dipende dal reader che viene selezionato a run-time, in base al tipo di file che si sta caricando. Nei test sono stati utilizzati file Pdf: eseguendo il debug dell’applicazione si può notare che il metodo *load\_data()* della classe *SimpleDirectoryReader* invoca il metodo statico *load\_file()* della stessa classe, che a sua volta crea a run-time un’istanza di *PDFReader*[[5]](#footnote-6), sulla quale verrà invocato il metodo *load\_data()* che effettivamente esegue il caricamento.

Nell’istanziare l’oggetto *PDFReader*, *SimpleDirectoryReader*  utilizza il costruttore di default: di conseguenza, invocando su questo oggetto *load\_data(),* verranno generati tanti documents quante sono le pagine del file Pdf caricato. Se si utilizzasse direttamente la classe PDFReader, sarebbe possibile passare al costruttore un argomento *return\_full\_document = True* e ottenere un solo *document* per file.

In LlamaIndex un oggetto di tipo *Document* rappresenta una sorta di container con diversi attributi, fra cui *text* (che contiene il contenuto testuale del documento), *metadata* (che contiene informazioni addizionali, come il nome del file) e *id* (che rappresenta un identificativo univoco per ciascun documento).

Durante i primi test, in esito alle risposte fornite dal sistema ci si è resi subito conto che era necessario procedere a ripulire i dati prima di sottoporli al LLM. Trattandosi di file di libero accesso, erano presenti “inserzioni pubblicitarie”, ripetute varie volte e su tutte pagine, che condizionavano fortemente la generazione del quiz, rendendolo di fatto inutilizzabile. La **qualità dei dati** di partenza è fondamentale in tutte le metodologie che li elaborano per derivarne conoscenza utile, e probabilmente lo è ancora di più nella RAG, che può essere considerata la sintesi di alcune di esse: il Machine learning, alla base delle tecniche di retrieval; il Data Mining, per l’estrazione di informazioni da grandi volumi di dati; il NLP, per la capacità di elaborare testo naturale e fornire risposte coerenti, etc.

Dopo aver dunque applicato ai documenti un filtro per eliminare le parti di testo note e indesiderate, viene definita una **pipeline di acquisizione**, cioè una sequenza di trasformazioni da effettuare sui singoli documenti, alla fine della quale si otterrà una lista di *nodi*:

1. utilizzando un *TokenTextSplitter* il testo viene suddiviso, rispettando i limiti della frase, in blocchi più piccoli (chunk) basati su *token* (parole, segni di punteggiatura, spazi). I blocchi vengono poi utilizzati per creare nodi, ciascuno dei quali rappresenta un'unità di informazione più complessa, che può contenere uno o più blocchi di testo. I nodi possono rappresentare concetti, entità o altre strutture semantiche all'interno di un testo, come una frase, un paragrafo o un'entità specifica come un nome proprio. I nodi sono utilizzati per organizzare e strutturare le informazioni in modo più significativo, possono includere metadati aggiuntivi, come riassunti o embedding e sono adatti per ulteriori elaborazioni con un LLM:

**TokenTextSplitter(**

**# dimensione massima di ciascun blocco   
# (chunk) in termini di numero di token.**

**chunk\_size=1024,  # default**

**# numero di token che si sovrappongono tra blocchi  
# consecutivi. Aiuta a mantenere la continuità**

**# del contesto tra i blocchi, migliorando la  
# coerenza delle risposte generate dal modello.**

**chunk\_overlap=20  # default**

**)**

Esistono altri metodi, oltre a quello basato sui token, per effettuare il *chunking*. In LlamaIndex troviamo vari ***text splitters****,* fra cui il *SentenceSplitter*, che suddivide il testo in frasi ed è utile per mantenere il contesto naturale, e il *CodeSplitter*, che lo suddivide in base al linguaggio di programmazione ed è ideale per la gestione della documentazione tecnica o del codice sorgente.

Troviamo anche strumenti più sofisticati, che suddividono un testo in nodi e sono stati progettati per elaborare vari formati standard di file e per essere utilizzati per contenuti più specifici, i ***node parser,*** fra cui:

* il *SentenceWindowNodeParser,*basato sul *SentenceSplitter*, divide il testo in singole frasi, includendo anche una finestra di frasi adiacenti nei metadati di ciascun nodo e creando quindi più contesto attorno ad ogni frase. Durante il processo di query, tale contesto verrà inserito nel prompt per il LLM per ottener risposte migliori.
* L’*HTMLNodeParser* analizza i file HTML e li converte in nodi, estraendo il testo dai tag HTML selezionati.
* Il MarkdownNodeParserparser elabora la parte testuale del markdown e genera nodi che ne riflettono la struttura e il contenuto.

È stato osservato che fornire troppi chunk come contesto al LLM può portare al problema “**Lost in the middle**”, cioè il modello accusa difficoltà nel mantenere il contesto e la rilevanza delle informazioni centrali, polarizzandosi verso le informazioni che appaiono all’inizio e alla fine del contesto. Lo strumento che LlamaIndex fornisce in proposito è un *node post processor* che interviene a valle del context retrieval, il *LongContextReorder*, particolarmente utile per le query in ambito accademico, dov’è comune l’utilizzo di documenti lunghi: se ad esempio si eseguisse una query su eventi storici dettagliati, un sistema RAG potrebbe recuperare nodi lunghi che presumibilmente comprendono maggiore contesto e LongContextReorder potrebbe riorganizzarli, assicurando che i quelli più rilevanti siano posizionati all'inizio o alla fine, sfruttando a vantaggio dell’utente la caratteristica del modello sopra descritta.

1. tramite un estrattore di metadati, per ciascun nodo viene generato un breve riassunto, basato sul contenuto testuale del nodo stesso. Ciò è utile per fornire un’idea del contenuto, senza dover leggere l'intero testo. I riassunti possono anche essere utilizzati durante la fase di retrieval per determinare rapidamente quali nodi sono rilevanti per una query specifica:

**SummaryExtractor(**

**summaries=['self'], # riassunto basato sul nodo stesso**

**language='it'**

**)**

1. dopo che i documenti sono stati suddivisi in blocchi di testo (chunk) e questi ultimi sono stati trasformati in nodi, un oggetto *OpenAIEmbedding* genera rappresentazioni vettoriali (embedding) per ciascun nodo. Gli embedding catturano il significato semantico del testo contenuto nei nodi e li rappresentano come vettori numerici: in un sistema RAG, vengono utilizzati per trovare nodi rilevanti per una query specifica.
2. infine, viene utilizzata una cache per evitare di ripetere elaborazioni costose (tokenizzazione, generazione di embedding e riassunti) sui dati che non cambiano. Se gli *hash* dei documenti nella cache corrispondono a quelli dei documenti attuali, la pipeline può utilizzare i risultati memorizzati nella cache invece di riapplicare la pipeline ai documenti. Se ci sono nuovi documenti o documenti modificati, la pipeline eseguirà l'elaborazione solo su quei documenti e aggiornerà la cache di conseguenza.

## Creazione di un indice vettoriale sui nodi

Disporre di un indice vettoriale è fondamentale per migliorare l'efficienza e l'efficacia delle operazioni di ricerca e recupero delle informazioni:

* *ricerca semantica*: gli indici vettoriali permettono di eseguire ricerche basate sulla somiglianza semantica. Ciò consente di individuare documenti o nodi che sono concettualmente simili alla query, anche se non contengono esattamente le stesse parole.
* *efficienza*: gli indici vettoriali sono ottimizzati per eseguire ricerche rapide su grandi quantità di dati. Utilizzando strutture dati specializzate, come gli alberi di ricerca o le tabelle hash, è possibile recuperare informazioni rilevanti in modo molto più veloce rispetto a una semplice scansione lineare dei nodi.
* *scalabilità*: quando si lavora con grandi dataset, un indice vettoriale permette di gestire e organizzare i dati in modo più efficiente. Questo è particolarmente utile in applicazioni di machine learning e intelligenza artificiale, dove la quantità di dati può essere enorme.
* *precisione*: gli indici vettoriali migliorano la precisione delle ricerche, poiché tengono conto delle relazioni semantiche tra i termini. Questo permette di ottenere risultati più pertinenti e accurati rispetto a una ricerca basata solo su parole chiave.

In sintesi, mentre i nodi contengono le informazioni di base, l'indice vettoriale organizza e ottimizza queste informazioni per rendere le operazioni di ricerca e recupero più efficienti e precise.

Il modulo *index\_builder.py* crea e gestisce un indice vettoriale (*VectorStoreIndex*) per i nodi elaborati nella precedente fase di acquisizione:

1. viene creato un oggetto StorageContext (contesto di archiviazione), responsabile della gestione della persistenza dei dati, ovvero di salvare e caricare i dati dell'indice da una directory specificata: **storage\_context = StorageContext.from\_defaults()**
2. viene creato un nuovo indice vettoriale, associandolo al contesto di archiviazione: ciò significa che tutte le operazioni di memorizzazione e recupero dei dati dell'indice verranno gestite da storage\_context:

**vector\_index = VectorStoreIndex(**

**nodes, storage\_context=storage\_context**

**)**

1. il contesto di archiviazione viene salvato nella directory di persistenza:

**storage\_context.persist(persist\_dir=INDEX\_STORAGE)**

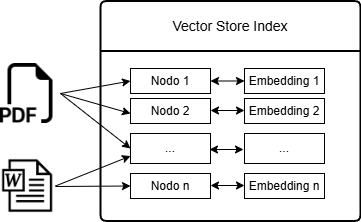


Figura 3 - Il processo di indexing

## Generazione del quiz

Con la creazione dell’indice vettoriale si conclude la fase preparatoria dei dati derivanti dalla conoscenza proprietaria, che potranno essere utilizzati per la produzione di un prompt da inviare al LLM, la cui risposta servirà a generare un quiz (modulo *quiz\_builder.py*):

1. utilizzando uno dei metodi esposti dall’indice viene innanzitutto definito il **motore di query** (*query engine*), un'interfaccia che permette di eseguire ricerche e recuperare (retrieval) informazioni dall'indice vettoriale in modo efficiente, basandosi sulla somiglianza semantica fra l’encoding di una query (query\_string), espressa in linguaggio naturale, e le rappresentazioni vettoriali dei nodi:

**query\_engine = vector\_index.as\_query\_engine()**

1. Il metodo *query* del query engine prende query\_string come input e la confronta con i vettori nel vector\_index per trovare i contenuti più rilevanti, utilizzando la similarità. Il query engine quindi produce un prompt esteso, combinando query\_string con il contesto estratto dal vector\_index. Questo prompt viene poi inoltrato al LLM per ottenere una risposta accurata e contestualizzata:

**response = query\_engine.query(query\_string)**

Anche in questo caso sono state sfruttate le potenzialità di LlamaIndex in ambito RAG, utilizzando l’indice vettoriale creato sulla base dei materiali caricati, successivamente trasformato in un query engine a cui si passa una query (query\_string) in linguaggio naturale, per chiedere al LLM di restituire le informazioni utili alla realizzazione del test. Questo aspetto, cioè la possibilità di interagire con il framework utilizzando linguaggio “umano”, dà la misura del livello raggiunto dalla tecnologia in ambito AI. Nel nostro caso, viene chiesto (in inglese) al LLM di creare un quiz in lingua italiana con un dato numero di domande sull’argomento indicato dall’utente. Per evitare che vertano tutte sul materiale caricato, viene indicato un limite. Poiché in tutte le pagine delle fiabe utilizzate per i test si fa riferimento al website che le mette a disposizione, viene precisato di non utilizzare tali informazioni. Infine si precisa che solo una risposta deve essere quella corretta e che la stessa deve essere riportata insieme alla motivazione (*rationale*), cioè in sostanza alla fonte dell’informazione stessa. Quest’ultimo elemento mostra che la RAG è uno strumento efficace rispetto all’esigenza di poter verificare la veridicità delle risposte fornite dal LLM:

**query\_string = (f"Create {QUIZ\_SIZE} different quiz questions relevant for testing a candidate's knowledge about {topic}. You must use Italian language. Each question will have 4 answer options. Each question will have different answers. No more than three questions should be specific to the provided text: in this case they should concern characters and stories.**

**Questions and answers must not refer to websites or URL.**

**For each question, provide also the correct answer and the answer rationale. Only one answer option should be correct.")**

Un altro aspetto interessante è legato alla possibilità di utilizzare il LLM anche per strutturare la risposta in modo utile per la realizzazione dell’interfaccia utente. Nel nostro caso viene prima definito un DataFrame Pandas vuoto, contenente nomi e tipi delle colonne per le domande del quiz. Successivamente viene utilizzato un parser di LlamaIndex (*DFRowsProgram*) che si appoggia ad un programma Pydantic[[6]](#footnote-7) (*OpenAIPydanticProgram*) per generare un nuovo dataframe nel formato Pandas precedentemente definito.

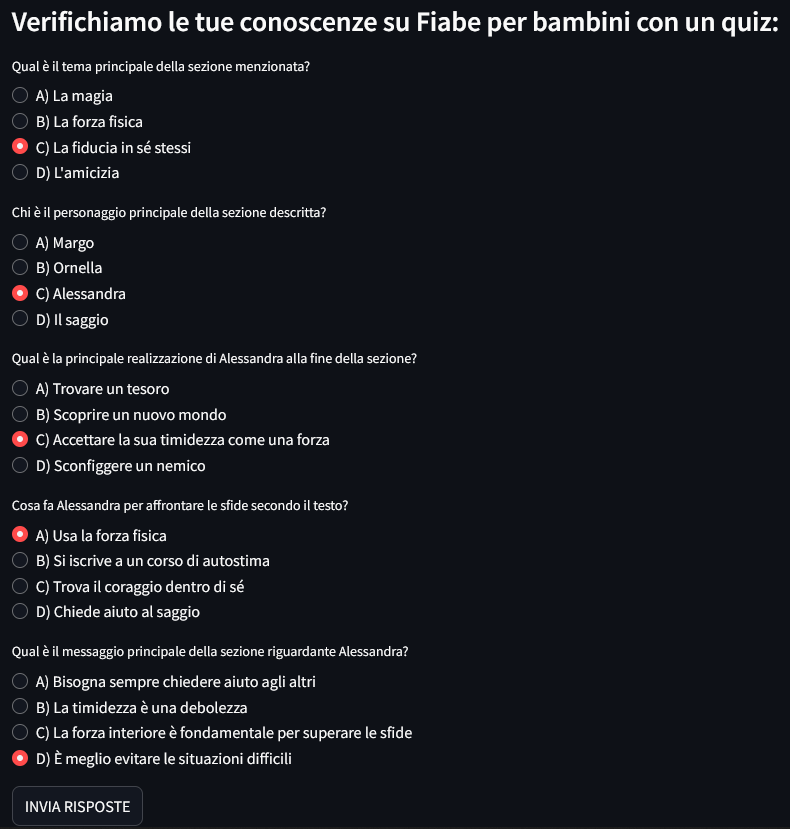


Figura 4- Generazione di un quiz

## Realizzazione del chatbot

Il modulo *conversation\_engine.py* gestisce attraverso un *agent* una chat interattiva che dialoga con l’utente, rispondendo alle sue domande. Il "motore di conversazione" è in grado di comprendere l'argomento ed e tiene traccia dell’interazione completa, salvandola su file per poterla riprendere fra sessioni differenti avviate dallo stesso utente.

La funzione *load\_chat\_store* è responsabile del recupero della conversazione del chatbot dalle sessioni precedenti: crea (o recupera) un'istanza di *SimpleChatStore*, una classe progettata per memorizzare su file una conversazione, e la restituisce al chiamante:

**def load\_chat\_store():**

**try:**

**chat\_store = SimpleChatStore.from\_persist\_path(CONVERSATION\_FILE)**

**except FileNotFoundError:**

**chat\_store = SimpleChatStore()**

**return chat\_store**

La funzione *initialize\_chatbot* inizializza un agent di classe *OpenAIAgent* e lo restituisce al chiamante:

1. utilizzando chat\_store, crea un buffer di memoria di tipo *ChatMemoryBuffer*:

**memory = ChatMemoryBuffer.from\_defaults(**

**token\_limit=3000,**

**chat\_store=chat\_store,**

**chat\_store\_key="0"**

**)**

1. carica il contesto di archiviazione:

**storage\_context = StorageContext.from\_defaults(**

**persist\_dir=INDEX\_STORAGE**

**)**

1. carica l'indice vettoriale dal contesto di archiviazione:

**vector\_index = load\_index\_from\_storage(**

**storage\_context, index\_id="vector"**

**)**

1. trasforma l’indice in un motore di query, che recupererà i 3 vettori più simili:

**study\_materials\_engine = vector\_index.as\_query\_engine(**

**similarity\_top\_k=3**

**)**

1. crea un Tool (*QueryEngineTool*), che incapsula il motore di query creato in precedenza e fornisce un accesso in sola lettura ai dati. Nei suoi metadati viene fornita una descrizione che ne specifica lo scopo, in modo che l'agent comprenda autonomamente quando utilizzarlo (nel caso in cui abbia a disposizione più tools fra cui scegliere): nel nostro caso, il tool ha lo scopo di fornire informazioni sui materiali di studio (study\_subject) e richiede una domanda in testo semplice come input:

**study\_materials\_tool = QueryEngineTool(**

**query\_engine=study\_materials\_engine,**

**metadata=ToolMetadata(**

**name="study\_materials",**

**description=(**

**f"Fornisce informazioni ufficiali su "**

**f"{study\_subject}. Richiede una domanda"**

**f"dettagliata in "**

**f"testo semplice come input."**

**)**

**)**

**)**

1. inizializza un **agente** di OpenAI (*OpenAIAgent*) con il tool precedentemente creato, la memoria per la chat e un prompt di sistema. Il prompt viene utilizzato per fornire al LLM informazioni di base per contestualizzare le sue risposte, garantire che siano pertinenti all’argomento di discussione attuale (study\_subject) e alle esigenze di approfondimento dell’utente (context).

**agent = OpenAIAgent.from\_tools(**

**tools=[study\_materials\_tool],**

**memory=memory,**

**system\_prompt=(**

**f"Ti chiami Learny, sei un assistente "**

**f"virtuale e parli in italiano. Il tuo "**

**f"scopo è di aiutare {user\_name} a studiare e "**

**f"comprendere meglio l'argomento: "**

**f"{study\_subject}. Stiamo discutendo in "**

**f"particolare il seguente contenuto: {context}"**

**)**

**)**



Figura 5 - Interfaccia del chatbot

## Test dell’applicazione

Per testare l’applicazione, sono stati utilizzati alcuni files Pdf (disponibili gratuitamente sul sito https://365favole.com/) contenenti favole per bambini ed utilizzati come conoscenza proprietaria. Interrogando Learny sul contenuto di una favola non ancora caricata, il chatbot risponde di non avere informazioni sufficienti:

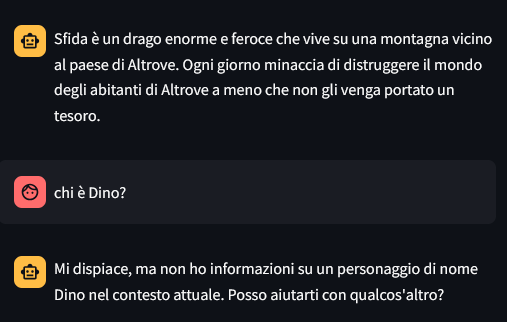


Figura 6 - Risposta in assenza di contesto

Dopo il caricamento, invece, risponde correttamente:

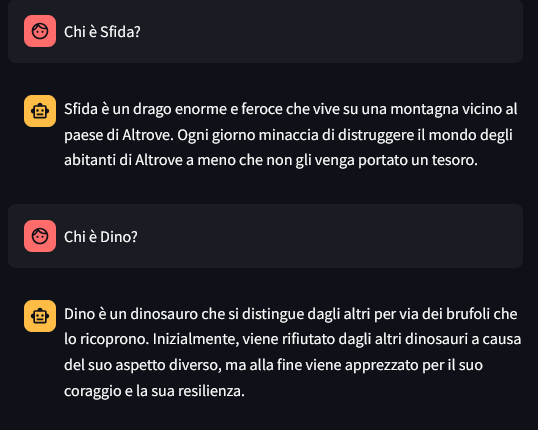


Figura 7 - Risposta supportata da contesto

Poiché la chat è stata dotata di “memoria”, è possibile intavolare una conversazione, utilizzando dei pronomi, anziché il nome proprio, per riferirsi ad un personaggio su cui si è già chiesto qualcosa nelle precedenti domande: Learny deduce dal contesto di chi si sta parlando e risponde correttamente.

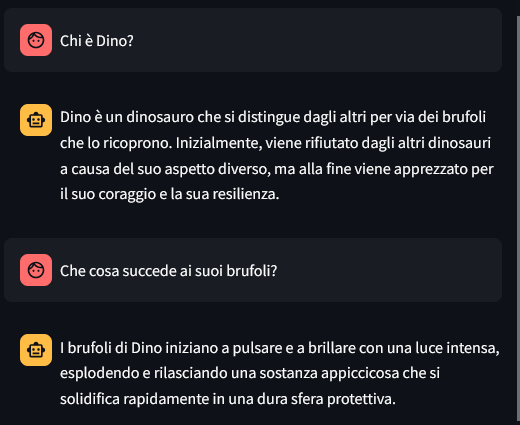


Figura 8 - Cronologia della conversazione

Inoltre, avendo indicato uno specifico argomento di studio, Learny è anche in grado di stabilire se un personaggio che gli viene indicato abbia o meno a che fare con la materia trattata: se ad esempio l’argomento è “Fiabe per bambini” e gli viene chiesto se “Albert Einstein” è un personaggio rilevante rispetto all’argomento di studio, risponde correttamente di no, aggiungendo anche qualche informazione sullo scienziato.

Talvolta è però necessario inserire nel sistema RAG uno o più **guardrail**, cioè controlli e pratiche implementati per assicurarsi che le risposte generate dal modello di linguaggio siano appropriate, pertinenti e sicure. Ad esempio potremmo introdurre delle **limitazioni contestuali**, cioè obbligare il LLM a rispondere basandosi esclusivamente sui documenti forniti in fase di retrieval. Per ottenere questo comportamento, possiamo modificare il query engine in questo modo:

study\_materials\_engine = vector\_index.as\_query\_engine(

similarity\_top\_k=3,

**strict\_retrieval=True**

)

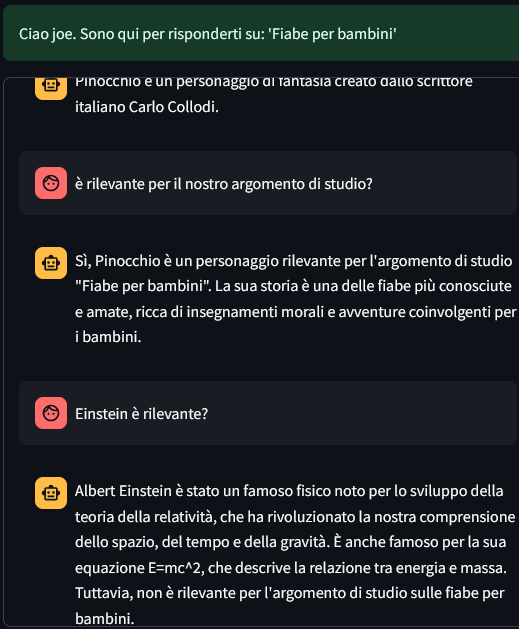


Figura 9 - Consapevolezza dell'argomento trattato

# Conclusioni

La realizzazione del software per implementare un assistente virtuale allo studio, basato su intelligenza artificiale, mi ha consentito di acquisire molteplici conoscenze nell’ambito dell’affascinante mondo della Retrieval-Augmented Generation (RAG), attraverso lo studio di quel formidabile framework open source che è LlamaIndex.

In particolare, ho potuto verificare sperimentalmente l'importanza della qualità dei dati di partenza e come essa influenzi direttamente l'efficacia delle risposte generate dall'assistente. Ho compreso come le tecniche di RAG possano potenziare un Large Language Model (LLM), migliorando la pertinenza e l'accuratezza delle risposte attraverso l'integrazione di conoscenze esterne al modello stesso. LlamaIndex fornisce molteplici strumenti di elevata complessità, che consentono, spesso con poche righe di codice, di controllare il comportamento del LLM e di modularne le risposte in base alle proprie esigenze, ad esempio filtrando quelle potenzialmente poco sicure, attraverso l’introduzione di limitazioni contestuali e altri guardrails.

Questo lavoro di tesi rappresenta solo una minima parte di ciò che potrebbe essere esplorato in questo campo: con più tempo a disposizione, ci sono diverse direzioni in cui mi piacerebbe proseguire la ricerca. Prima di tutto, mi piacerebbe approfondire ulteriormente le tecniche di chunking semantico per migliorare la coerenza e la rilevanza delle risposte. Inoltre, esplorerei l'integrazione di tecnologie di riconoscimento vocale per rendere l'interazione con l'assistente virtuale ancora più naturale e intuitiva.

Un'altra area di interesse sarebbe l'ottimizzazione del processo di retrieval in termini di precisione e recall, per garantire la corretta selezione del maggior numero possibile di informazioni effettivamente rilevanti da presentare al LLM. Infine, sarebbe interessante investigare l'applicazione della RAG in contesti educativi più specifici, come l'insegnamento delle materie STEM, per capire meglio come questa tecnologia possa essere utilizzata per supportare l'apprendimento negli ambiti di conoscenza maggiormente articolati e complessi, sia dal punto di vista della difficoltà intrinseca alla materia trattata sia da quello dei linguaggi formali in essi adottati.

1. che provengono cioè da fonti poste al di fuori del modello di linguaggio stesso: documenti, articoli, database o qualsiasi altra risorsa non facente parte del LLM. [↑](#footnote-ref-2)
2. Precisione e recall (sensibilità) sono concetti fondamentali in ambito di machine learning. La precisione misura il rapporto fra il numero di veri positivi e il totale dei positivi (veri + falsi) restituiti dal modello, indicando quindi quanto siano accurati i risultati che il modello ha identificato come rilevanti. La Sensibilità misura il rapporto fra il numero di veri positivi e il totale dei risultati rilevanti (veri positivi + falsi negativi), indicando quindi quanto il modello sia effettivamente in grado di identificare i risultati rilevanti. [↑](#footnote-ref-3)
3. https://www.llamaindex.ai/ [↑](#footnote-ref-4)
4. https://streamlit.io/ [↑](#footnote-ref-5)
5. llama\_index.readers.file.docs.base.PDFReader [↑](#footnote-ref-6)
6. I programmi Pydantic sono una forma di astrazione nel workflow del LLM che consente di generare output strutturato a partire da un input composto da stringhe di testo. [↑](#footnote-ref-7)