Indice

1 Introduzione

Questo lavoro di tesi adotta una prospettiva principalmente **sperimentale**, anzi: l'obiettivo non è realizzare un sistema production-ready né ottimizzare un'architettura rispetto a latenza, costi o vincoli di governance dei dati. Lo scopo è invece capire quali scelte incidano di più sulla qualità del retrieval e, di conseguenza, sulle risposte prodotte da un Language Model in una pipeline RAG. Il punto di partenza è che le prestazioni di un sistema RAG dipendono in gran parte dal retrieval e dal modo in cui rendi l'informazione interrogabile; per questo, si valutano e confrontano approcci RAG diversi: da soluzioni semplici a pipeline più strutturate, discutendo di come queste si collochino rispetto alle criticità fondamentali del RAG moderno. L'analisi si concentra sulle famiglie: RAG gerarchico, Naive ed Adattivo. Questi approcci sono particolarmente interessati quando la base documentale è eterogenea, come nei Data Lake: non solo testo, ma anche tabelle relazionali, metadati... che difficilmente si riducono senza eccessiva perdita d'informazione ad un insieme di chunk indipendenti; le strutture graph based permettono di modellare il retrieval, rispetto a query non SQL like, anche per dati strutturati come questi. Dal punto di vista metodologico, la tesi propone un confronto sperimentale osservando come cambino le prestazioni al variare di:

- **strategia di indicizzazione** (indice piatto vs strutture gerarchiche / a grafo),
- **strategia di retrieval** (single-shot top-k vs comunità/percorsi),
- **modalità di integrazione delle fonti** (testo non strutturato e dati strutturati tabellari),

L'implementazione usa più tecnologie: Azure AI Foundry come baseline per un Naive RAG, LangChain ed ElasticSearch per costruire varianti più articolate (ad esempio CoT e GraphRAG) e Neo4j per gestire strutture graph-based. Anche queste scelte seguono l'impostazione del lavoro: non cercare "la" soluzione, ma misurare il tradeoff che c'è tra prestazioni e retrieval.
Molte criticità attribuite ai modelli generativi (allucinazioni, incoerenze, risposte generiche) dipendono spesso dal problema che c'è a monte: recuperare informazione pertinente alla domanda; i capitoli successivi formalizzano il paradigma RAG, discutono i limiti degli approcci naïve e presentano il confronto sperimentale tra i metodi considerati, con particolare enfasi sulle soluzioni gerarchiche e graph-based.

2 Prerequisiti

Questo capitolo ha l'obiettivo di fornire le nozioni essenziali per interpretare correttamente le scelte progettuali e le componenti tecniche del lavoro. Alcuni concetti verranno introdotti da zero durante la discussione, altri richiamati per

coerenza, ma per la maggior parte dedicare ulteriore tempo, ogni volta, alle definizioni necessarie appesantirebbe la trattazione e distoglierebbe eccessivamente l'attenzione dal tema principale; per questo motivo, spezziamo i prerequisiti in una sezione dedicata.

2.1 Inverted index

L'*inverted index* (indice invertito) è la struttura dati principe dei motori di ricerca tradizionali: associa a ogni termine l'elenco dei documenti in cui compare e si basa interamente sull'idea che una query possa essere risposta da un dato set di documenti che puoi definire rilevanti poiché posseggono una sovrapposizione lessicale esatta rispetto alla richiesta stessa; difatti, così la valutazione di una query evita la scansione completa del corpus e si riduce alla combinazione delle *posting lists*, ovvero le liste di documenti, associate ad ogni termine della query. Ciascuna lista contiene tipicamente gli identificativi dei documenti (docID) e, spesso, informazioni utili al ranking come la frequenza del termine nel documento. L'indice invertito rende, quindi, efficiente la ricerca anche su collezioni molto grandi perché sposta il costo computazionale dalla lettura dei documenti all'operazione di *merge* tra liste ordinate; nella stragrande maggioranze dei motori di ricerca, come Google ad esempio, si sfrutta un'intersezione delle posting lists, ovvero default AND trai termini della query, appunto richiamando il concetto precedente di sovrapposizione lessicale.

2.2 Positional index e Phrase Queries

Un *positional index* estende l'indice invertito memorizzando, per ogni termine e per ogni documento, anche le posizioni (offset) in cui esso appare. Questa informazione aggiuntiva abilita anche query di tipo diverso: le *phrase queries*; queste sono concettualmente semplici, poiché a te non interessa la sovrapposizione lessicale esatta indipendentemente dalla posizione, nel caso precedente l'AND trai termini della query impone solo che questi debbano comparire nei documenti di riferimento ma, potenzialmente, anche in ordine diverso, ma vuoi mantenere anche l'ordine posizionale. Infatti, richiedono che i termini compaiano adiacenti e nell'ordine corretto, basando la verifica sul confronto tra le informazioni posizionali: dato un documento candidato, si controlla tipicamente se, per ogni coppia di termini, la differenza tra le posizioni relative è la stessa nel documento e nella query. Le phrase queries estendono il concetto stesso alla base della ricerca cosiddetta full - text: sovrapposizione lessicale esatta è rilevanza, difatti coincidono con il caso in cui la ricerca token - based performi, in realtà, meglio in generale; su questo concetto e sui problemi che ne derivano ci torneremo successivamente.

2.3 Tokenizzazione ed Analyzers

Discutendo di Inverted e Positional Index abbiamo presupposto che quei 'termini' a cui associ le posting lists tu li abbia già: in realtà questo comporta spesso

delle vere e proprie pipeline che applichi al testo grezzo in ingresso, formalmente queste vengono definite *analyzer*. Tra le trasformazioni più comuni rientrano il *case-folding*, la rimozione di *stopword* e procedure di stemming o lemmatizzazione per ricondurre varianti morfologiche a una forma più stabile, con l'obiettivo principale di ridurre la variabilità del testo e aumentare la *recall*, accettando però situazioni in cui parole semanticamente diverse vengono ricondotte alla stessa radice sintattica. Gli analyzer più comuni differiscono soprattutto per la strategia di tokenizzazione: un analyzer *standard* divide il testo basandosi sui confini "parola" secondo lo standard Unicode, mentre un analyzer basato su *whitespace* si limita a spezzare sugli spazi, risultando semplice ma spesso troppo grezzo in presenza di punteggiatura soprattutto; esistono altre varianti come il simple, stop analyzer... ma le due precedenti sono le più famose.

Sebbene la tokenizzazione sembri un processo semplice, in realtà è particolarmente complesso in relazione alle esigenze: quando si parla di tokenizer orientati al retrieval, tipicamente si sfruttano set di regole semplici, come quelle Unicode o Whitespace discusse precedentemente, diverso è nel caso di modelli di NLP come Transformer, che discuteremo successivamente. In questi casi, sono diffusi metodi di tokenizzazione subword come *Byte Pair Encoding* (BPE) e *WordPiece*. Senza scendere eccessivamente nel dettaglio, il che non è lo scopo di quest'introduzione, i suddetti metodi spezzano le parole in sotto - unità per gestire vocaboli rari, addirittura 'inventati' e la variabilità morfologica nel caso generico, migliorando la generalizzazione del modello alla variabilità stessa del testo che può trovarsi a dover processare. Tuttavia, questi schemi sono spesso poco adatti alla ricerca full-text classica: introducono un disallineamento tra l'intuizione dell'utente (che formula query in parole) e l'unità effettivamente indicizzata (subword), riducono l'interpretabilità di *highlight* e snippet e, soprattutto, aumentano il numero di token per documento, con un impatto diretto sulla dimensione dell'indice e sulla lunghezza delle posting lists. È interessante notare come le conseguenze siano quasi complementari nei Transformer e nei sistemi di retrieval tradizionali. Nei modelli Transformer, la segmentazione in subword (ad esempio BPE) è vantaggiosa perché riduce drasticamente la dimensione del vocabolario: invece di memorizzare un token distinto per ogni parola rara o morfologicamente complessa (es. `fortissimo`), il modello può comporla a partire da unità più frequenti (es. una radice `fort` e un suffisso `##issimo`, esempio esclusivamente intuitivo), ottimizzando la copertura lessicale a parità di grandezza del vocabolario. Nel caso di un *inverted index*, invece, l'obiettivo è diverso: si desidera che l'unità indicizzata coincida il più possibile con l'unità di ricerca dell'utente e che i termini siano *stabili*. Segmentare sistematicamente in subword tende ad avere l'effetto opposto: aumenta il numero di termini indicizzati, rende le posting lists più dense (perché subword frequenti compaiono in molte parole diverse) e può peggiorare la precisione del matching, dato che frammenti condivisi da parole non correlate generano corrispondenze rumorose dall'intersezione. Mentre, quindi, nei Transformer la scomposizione subword è una strategia ottima per comprimere il vocabolario e generalizzare meglio, nel retrieval full-text classico rischia di trasformare il problema in un matching su

unità troppo atomiche, incentivando un qualcosa che in letterature si indica come *Vocabulary Mismatch Problem*: la bontà dei risultati restituiti da un motore di ricerca dipende da come l'utente formula la query, o meglio, da quanto la distribuzione lessicale della stessa sia allineata rispetto a quella dei documenti, per questo si cerca di mantenere il più possibilmente la tokenizzazione allineata con quello che viene naturalmente espresso dall'utente e la segmentazione in sotto - unità crea più problemi che vantaggi; ciò non vuol dire che tu non possa agire sulla query, ad esempio tramite riscritture, vuole semplicemente dire che tokenizzare come fanno i Transformer non è una grande idea.

2.4 Metriche nella Full-Text Search

I documenti ottenuti dalle posting lists vengono, poi, ordinati sulla base di degli score derivanti dall'importanza che i token della query assumono all'interno del documento; come il retrieval si basa su sovrapposizione lessicale, anche gli score riprendono questo concetto per costruire il cosiddetto *ranking*. Due delle metriche più utilizzate in questo contesto sono *TF-IDF* e *BM25*, entrambe fondate sull'idea che l'informatività di un termine dipende mutuamente dalla sua importanza locale e globale: se è un termine molto diffuso tra tutti i documenti, allora non è una discriminante, altrimenti se è molto diffuso all'interno del singolo documento e poco/mediamente negli altri è una discriminante.

TF-IDF TF-IDF (*Term Frequency – Inverse Document Frequency*) assegna un peso ai termini della query in funzione della: frequenza del termine nel documento (*TF*), catturando quanto quel termine è rappresentativo dello stesso, e la frequenza del termine nell'intera collezione (globale) (*IDF*), che riduce l'importanza di parole eccessivamente comuni in corpus soprattutto tematici, si pensi ad un corpora di documenti medici ad esempio. Concettualmente, il punteggio di un documento rispetto a una query cresce quando i termini della query compaiono spesso nel documento, ma soprattutto quando tali termini sono rari nel corpus, in maniera totalmente allineata con quanto accennato prima. TF-IDF è semplice, efficace e costituisce una base naturale per il ranking lessicale, pur soffrendo del fatto che la componente TF può crescere in modo eccessivo per documenti molto lunghi o ripetitivi: più un documento è lungo e maggiore sarà, mediamente, la TF dei termini; questo crea un bias negli score attribuiti, che tendono a preferire sistematicamente documenti più lunghi e non più rilevanti.

BM25 Aggiunge a TF-IDF una normalizzazione non lineare alla Term Frequency: meno dura di quella diretta rispetto alla lunghezza del documento, ma con gli stessi vantaggi. L'idea è sempre la stessa: un documento che ha un buon compromesso tra frequenza locale e globale dei termini della query è un documento rilevante. La normalizzazione per lunghezza vuole semplicemente evitare che i documenti più lunghi vengano sempre restituiti e ridurre il bias che discutevamo precedentemente; questa, come suddetto, viene applicata al termine TF della TF - IDF, essendo che l'IDF non dipende dalla lunghezza del documento.

Con il Chunking, discusso nei capitoli successivi, si vede un'indicizzazione per 'porzioni locali' del documento: generalmente questo ti permette anche di avere una normalizzazione strutturale per TF-IDF senza ricorrere a BM25, tuttavia quest'ultima rimane lo standard de facto essendo che, dipendentemente dalla strategia, potresti non avere tutti chunk di dimensione identica.

2.5 Problemi Tipici del Full-Text Retrieval

Abbiamo già discusso il *Vocabulary Mismatch Problem* e tutte le criticità della Full-Text search si riconducono a questo; possiamo studiarlo anche in modo diverso, in funzione prevalentemente della distribuzione lessicale. Infatti, vale l'osservazione complementare: una maggiore sovrapposizione lessicale non implica necessariamente, in generale, una maggiore sovrapposizione semantica; la presenza delle stesse parole in query e documento non garantisce che stiano facendo riferimento allo stesso significato. Questo fenomeno è strettamente legato alla *polisemia* del dominio, per cui un termine può assumere accezioni diverse a seconda del contesto, aumentando l'ambiguità e degradando la precisione del retrieval basato su sovrapposizione lessicale; al contrario, in domini più *monosemici*, si osserva spesso che i termini sono più stabili e meno ambigui, di conseguenza il matching lessicale risulta più affidabile semanticamente.

Già da qui si capisce un concetto su cui torneremo più volte nel RAG: il retrieval deve essere progettato, tarato e ottimizzato in funzione del dominio di appartenenza, a meno di accontentarsi di prestazioni subottimali. Quando è possibile stimare con buona accuratezza la distribuzione lessicale del dominio, si possono spesso ottenere prestazioni elevate anche con approcci relativamente semplici, evitando di introdurre complessità non necessaria oltre la ricerca full-text tradizionale. Questa dipendenza dal dominio diventa ancora più evidente quando si adotta la *hybrid search*: combinare segnali lessicali e densi richiede di pesare opportunamente i rispettivi score e tali pesi non sono universali ma emergono soprattutto dalle caratteristiche del corpus e delle query attese; sostanzialmente quindi, la scelta tra full-text, dense retrieval o una combinazione dei due non è fortemente guidata dal dominio e dalla distribuzione delle query.

2.6 Dense retrieval

Il *dense retrieval* è un paradigma di ricerca in cui documenti e query vengono rappresentati vettorialmente in uno spazio continuo ed il retrieval avviene tramite meccanismi di vicinanza trai vettori definenti query e documenti; si sfruttano misure di similarità, principalmente di tipo coseno: metrica che quantifica l'angolo compreso tra due vettori, più è piccolo e più i vettori che tracci dall'origine del piano sono coincidenti e, quindi, i punti nello spazio rappresentanti query o documenti coincidono nella stessa posizione e sono simili. Operativamente, il sistema calcola tale vettore (embedding) per la query, interroga un indice vettoriale e restituisce i k documenti con massima similarità. Il vantaggio principale

rispetto al full-text classico è la capacità di catturare corrispondenze *semantiche*: un documento può risultare rilevante anche senza condividere esattamente le stesse parole della query, riducendo strutturalmente il *Vocabulary Mismatch Problem*, difatti la rappresentazione densa si concentra proprio sul catturare il significato del testo e della query, evitando di modellarlo in maniera 'latente' rispetto alla sovrapposizione lessicale.

2.7 Embeddings

Quando si parla di *embedding* si fa riferimento ai suddetti vettori; più formalmente: un embedding è una funzione che mappa un oggetto discreto (parola, frase, documento) in un vettore reale $\mathbf{e} \in R^d$, con l'obiettivo proiettare in uno spazio dove la geometria dello stesso rifletta relazioni utili: punti vicini dovrebbero corrispondere a contenuti semanticamente simili, viceversa lontani. In quest'ottica, la semantica emerge proprio rispetto alla disposizione dei punti nello spazio: concetti simili tendono a formare regioni dense o *cluster*, mentre concetti distanti si collocano in aree che devono essere il più possibilmente separate dello spazio; la qualità di questo dipende non solo dal modello, ma soprattutto dal criterio di addestramento. Nei sistemi moderni di retrieval si usano spesso obiettivi di addestramento *contrastivi*: l'idea è "modellare" lo spazio vettoriale in modo che una query risulti vicina ai documenti veramente rilevanti e lontana da quelli irrilevanti. Nel concreto, dato un esempio positivo (q, d^+) e un insieme di negativi $\{d_1^-, \ldots, d_m^-\}$, si ottengono gli embedding $\mathbf{q}$, $\mathbf{d}^+$ e $\mathbf{d}_i^-$ e si ottimizza una loss che aumenta $s(\mathbf{q}, \mathbf{d}^+)$ e riduce $s(\mathbf{q}, \mathbf{d}_i^-)$, dove $s(\cdot, \cdot)$ è la misura di similarità adottata. Intuitivamente, la loss forza la query a "scegliere" il documento corretto tra molte alternative, o meglio, gli stiamo direttamente indicando cosa per noi è semanticamente correlato da cosa non lo sia.

Nelle discussioni successive richiameremo esplicitamente la relazione tra **anisotropia** e loss *contrastive*, quindi non è utile anticipare ora tutti i dettagli; è però importante evidenziare un punto chiave: il solo pre-training raramente è sufficiente per ottenere embedding adatti al retrieval semantico. Modelli come BERT, ad esempio, apprendono rappresentazioni vettoriali perché vengono addestrati a modellare le relazioni grammaticali, semantiche, posizionali... nel linguaggio tramite cosiddetti auxiliary task come il *masked language modeling*. Tuttavia, il fatto che un modello sappia rappresentare bene il linguaggio non implica automaticamente che gli embeddings prodotti siano anche geometricamente ben strutturati secondo le necessità precedenti: detta informalmente, il pre-training insegna al modello a rappresentare il significato contestuale del linguaggio, ovvero che la stessa parola può assumere una semantica diversa a seconda del contesto (ad esempio *apple* come frutto oppure come azienda), insegnando indirettamente anche il significato delle parole, tuttavia questo non implica automaticamente che lo spazio degli embeddings venga ben separato come ci aspettiamo e come vuole un retrieval denso; difatti, la nozione di "differenza" che aiuta a separare lo spazio è un qualcosa di relativo alla fase di post-training e strettamente collegato alla loss che utilizzi. Questo passaggio è

cruciale per rendere efficace l'interrogazione tramite nearest neighbors e chiarisce perché un buon language model non coincida necessariamente con un buon modello di retrieval; d'altronde, le contrastive losses corrispondono spesso e volentieri a fine tunings successivi, dove come obiettivo ulteriore hai anche quello di modellare sfumature di significato complesse nel tuo dominio e che un pre-training generalista non ti consente.

2.8 Transformer

Non l'abbiamo detto direttamente, però leggendo un po' tra le righe: gli embeddings, ad oggi, vengono tutti ricavati ma modelli di Deep Learning Transformer based; questo non necessariamente implica che si sia sempre fatto così, anzi: approcci come Word2Vec o GloVe non si basano su Transformer, ma più sull'idea *dimmi con chi vai e ti dirò chi sei*, modellando in diversi modi ul significato contestuale. Il grosso 'boost' a livello di bontà degli embeddings prodotti storicamente è avvenuto proprio con le architetture Transformer: l'attention permette di risolvere il problema dei suddetti approcci tradizionali di 'staticità' dell'embedding prodotto; GloVe, come Word2Vec, ad inferenza forniscono matrici statiche tali per cui puoi ottenere un embedding come prodotto vettoriale tra una rappresentazione OneHotEncoded della singola parola e la matrice di embedding stessa, sostanzialmente un lookup, dove l'embedding di frase coincideva spesso con una media aritmetica di quelli dei token singoli; il problema è che, qui, hai un embedding singolo per ogni parola ed indipendentemente da quello che sta nell'intorno della stessa: non c'è significato contestuale, l'attention, invece, parte da un embedding statico e lo modifica in funziona degli embedding di tutti gli altri token vicini, producendone una rappresentazione enhanced.

2.8.1 BERT

Discutere di BERT è fondamentale per comprendere meglio i Bi e Cross-Encoder che discuteremo nei capitoli successivi, essendo che quest'ultimi, sebbene più sofisticati, si basano comunque sull'architettura Encoder only introdotta proprio da BERT. *Bidirectional Encoder Representations from Transformers* è un modello basato sull'omonima architettura Transformer che utilizza esclusivamente lo *stack* di **encoder**, elabora una sequenza testuale tramite **self-attention** e produce, per ogni token, una rappresentazione vettoriale contestuale. La caratteristica distintiva di BERT è la natura **bidirezionale**: ogni token può integrare informazione proveniente sia dal contesto a sinistra sia da quello a destra, poiché durante il training non viene applicata una *causal mask* come avviene nei modelli generativi come GPT; BERT, appunto, non è auto - regressivo ed il grosso vantaggio nell'NLP dell'utilizzo di architetture di questo tipo sta, proprio, nell'interazione tra ogni token. Nel caso di modelli generativi questo non avviene: c'è il vincolo di causalità, il token $T + 1$ viene generato sulla base del token T prodotto a passo precedente e, di conseguenza, sfruttare un meccanismo di attention che permetta di far interagire i token precedenti con quelli futuri

perde di senso semplicemente perché non li conosci; ci sono, poi, considerazioni che si potrebbero fare a livello di 'peaking' nel training, ma intuitivamente il concetto rimane solido anche così. La bidirezionalità non dipende, quindi, dalla self-attention "in sé", ma dal fatto che non venga limitata a vedere solo il passato e questo è coerente con l'obiettivo di *comprensione* più che generazione.

Il pre-training originale di BERT è formulato come *multitask learning* e combina principalmente due compiti: **Masked Language Modeling (MLM)** e **Next Sentence Prediction (NSP)**. Nel task MLM si seleziona tipicamente circa il 15% dei token e si applica la regola 80/10/10: nell'80% dei casi il token viene sostituito con `[MASK]`, nel 10% con un token casuale e nel restante 10% rimane invariato. La sequenza così perturbata viene data in input all'encoder e il modello deve ricostruire il token originale nelle posizioni selezionate; la loss (cross-entropy) viene calcolata *solo* su tali posizioni. Complessivamente, l'addestramento tramite MLM induce BERT a produrre embedding abbastanza ricchi: per predire correttamente i token mascherati, il modello deve sfruttare dipendenze sintattiche e semantiche presenti in tutta la frase, cosa che la self-attention permette in modo naturale; questo rende BERT particolarmente efficace in compiti di language understanding, come classificazione del testo e Named Entity Recognition (NER).

Il task NSP, introdotto nella versione originale, mira a modellare relazioni tra frasi: dato un input del tipo `[CLS] frase_A [SEP] frase_B [SEP]`, il modello predice se la seconda frase segue effettivamente la prima nel testo sorgente. La decisione viene presa a partire dall'embedding del token speciale `[CLS]`, che non ha significato lessicale ma, grazie alla self-attention, può raccogliere informazione dall'intera sequenza; il 'come' faccia questo è abbastanza semplice: è un token neutro (inventato) che non compare nei testi di addestramento, viene posto all'inizio della sequenza in modo, proprio, che possa assorbire la totalità del significato di tutti i token successivi, quindi l'interezza della frase, grazie all'attention stessa. Per questo motivo, `[CLS]` tende a diventare una rappresentazione "globale" utile in compiti a livello di frase. In successive implementazioni, come RoBERTa, il task NSP è stato rimosso: l'idea di fondo è che ottenere un embedding `[CLS]` *general purpose* già ben calibrato per molti compiti sia un obiettivo troppo ambizioso per il solo pre-training, e che l'utilità pratica emerga soprattutto dopo un *fine-tuning* supervisionato sul task specifico. BERT è concepito come modello *general-purpose* per la comprensione del linguaggio, tuttavia i grossi vantaggi vengono quando si specializza l'utilizzo ad un determinato task: si noti che per specializzare non si intende per forza qualcosa di relativo ad un dato dominio, come i fine tunings per sfumature di singificato, ma proprio un compito 'operativo'. Ci sono diversi esempi:

- task di *sentence classification* (ad esempio sentiment analysis), dove si aggiunge una semplice testa di classificazione che prende in ingresso l'embedding finale del token speciale `[CLS]`. Poiché la funzione di perdita supervisiona direttamente questa rappresentazione, il training spinge `[CLS]`

a condensare l'informazione dell'intera sequenza in un vettore unico, utile per decisioni "globali" a livello di frase

- Nei task *token-level* come la *Named Entity Recognition*, invece, l'obiettivo non è assegnare un'etichetta al testo nel suo complesso, ma produrre una predizione per ciascuna posizione. Si applica quindi una testa di classificazione a ogni embedding contestuale in uscita dall'encoder, ottenendo una sequenza di etichette (ad esempio `B-PER`, `I-ORG`, `O`, ...). In questo scenario, la qualità del modello dipende soprattutto dalla capacità di rappresentare accuratamente il contesto *locale* di ogni token, più che dall'esistenza di un singolo vettore riassuntivo.

- Nel retrieval invece, BERT viene adattato con un fine-tuning che comprende le loss contrastive suddette; architetture come SBERT introducono questa tendenza esplicita ad ottimizzare le rappresentazioni del testo affinché risultino confrontabili con una metrica di similarità (coseno o prodotto scalare). L'embedding testuale, ottenuto tipicamente da `[CLS]` oppure tramite pooling sulle posizioni

Per concludere, a mio avviso, è utile dare una panoramica generale di ciò che avviene "all'interno" di BERT durante la costruzione degli embeddings. In ingresso, ciascun token viene trasformato in un embedding iniziale tramite una matrice W, anch'essa appresa durante il training: per ciascuno, viene portata in conto l'informazione posizionale tramite una strategia di positional encoding, dipendente dalla versione di BERT che si considera; evitiamo di discutere ulteriormente l'encoding posizionale poiché è la cosa che interessa di meno in realtà. Il meccanismo di **Self-Attention** opera proprio su tali embeddings iniziali: operazione chiave con cui viene costruito per token una rappresentazione contestuale integrando informazione dagli altri token della sequenza tramite, sostanzialmente, combinazioni lineari opportunamente pesate. Per ciascun embedding si calcolano tre proiezioni lineari: **Query** ($\mathbf{Q}$), **Key** ($\mathbf{K}$) e **Value** ($\mathbf{V}$). L'attenzione si ottiene confrontando ogni query con tutte le key tramite prodotto scalare, applicando poi una softmax per ottenere pesi normalizzati:

$$\mathrm{Attn}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}.$$

Il risultato è una combinazione pesata dei *value*: ogni token "decide" quanta informazione assorbire dagli altri, assegnando pesi maggiori alle posizioni più rilevanti per il contesto corrente. Intuitivamente, la **Query** ($\mathbf{Q}$) è "la domanda" che ogni token pone al resto della sequenza: per il token i-esimo, q_i chiede "quali altri token sono rilevanti per me?". La **Key** ($\mathbf{K}$) è invece la "chiave" con cui ogni token si descrive: il vettore k_j del token j indica "in che modo io potrei essere rilevante per gli altri". Infine, la **Value** ($\mathbf{V}$) è il contenuto informativo che viene realmente aggregato: il vettore v_j contiene l'informazione che, pesata dall'attenzione, contribuisce a costruire l'output; la rappresentazione enhanced contestualmente dell'embedding di ogni token viene, infatti, da una combinazione lineare dei pesi di attention, dati dalla softmax, rispetto ai values V. La

Multi-Head Attention ripete lo stesso calcolo su più teste in parallelo con proiezioni differenti: teste diverse possono specializzarsi nel catturare segnali anche diversi dalla semplice semantica, ad esempio dipendenze sintattiche o anche posizionali più complesse. Le uscite delle teste vengono concatenate e riproiettate, l'intero blocco è accompagnato da *residual connections* e *layer normalization*, utili a stabilizzare l'ottimizzazione e a preservare il flusso informativo tra strati. Accanto all'attenzione, ogni layer include una **Feed-Forward Neural Network** (FFNN) *position-wise*, applicata indipendentemente a ciascun token: è tipicamente una MLP a due strati con non linearità che espande la dimensionalità nel layer hidden e, poi, la riporta a quella originale:

$$\mathrm{FFN}(\mathbf{x}) = \mathbf{W}_2\,\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2.$$

Se l'attenzione realizza il *mixing* dell'informazione tra token, la FFNN svolge la trasformazione non lineare che *modella* e riorganizza l'informazione per ciascun token, aumentando la capacità espressiva del modello e rendendo le rappresentazioni via via più complete ed informative. Anche in questo caso, residui e normalizzazione facilitano la composizione di molti strati encoder senza degradare incorrere nei classici problemi del training nelle architetture Deep.

2.9 Hybrid Search

Alla luce di quanto discusso, è facile capire l'idea di *hybrid search*: combinre segnali lessicali (full-text) e segnali semantici (dense retrieval) per sfruttare la complementarità trai problemi di uno ed i vantaggi dell'altro. L'idea, appunto, è che i modelli lessicali (ad esempio BM25) eccellono quando la query contiene termini discriminanti e quando l'utente richiede corrispondenze esatte, mentre il dense retrieval è più efficace nel gestire sinonimia, parafrasi e query formulate in modo diverso dal testo del documento. Operativamente infatti, un sistema ibrido può recuperare candidati da entrambi i canali e poi fonderli, calcolando un punteggio combinato; in entrambi i casi, la fusione mira ad ottenere un ranking che porti strutturalmente conto sia dello score full-text che di quello denso, restituendo i documenti migliori per compromesso.

Un approccio comune è recuperare due liste top-k (lessicale e vettoriale) e unificarle con:

- **Reciprocal Rank Fusion (RRF)**: invece di combinare direttamente gli score, si combinano i *ranghi*. Se un documento d compare in posizione $r_1(d)$ nella lista lessicale e in posizione $r_2(d)$ nella lista densa, il punteggio finale può essere definito come

  $$\mathrm{RRF}(d) = \sum_{i \in \{1,2\}} \frac{1}{k_0 + r_i(d)},$$

  dove k_0 è una costante che smorza l'effetto delle primissime posizioni. In questo modo vengono premiati i documenti che risultano "buoni" in

entrambi i ranking full-text e denso, ottenendo una fusione abbastanza robusta soprattutto al caso in cui gli score singoli delle liste non sono confrontabili; RRF è potente, proprio perché non devi preoccuparti a priori di normalizzare gli score in una scala comune e questo si capisce bene anche guardando la combinazione convessa, dove avresti un termine molto più preponderante dell'altro e che, indipendentemente dal peso, rischierebbe di dominare nella definizione del ranking

- **Combinazione convessa (weighted sum)**: si normalizzano gli score dei due canali e si calcola un punteggio aggregato tramite una somma pesata:
$$s(d) = \alpha\, s_{\text{lex}}(d) + (1-\alpha)\, s_{\text{dense}}(d), \qquad \alpha \in [0,1].$$
Il parametro α controlla il compromesso tra corrispondenza lessicale e affinità semantica. Questa strategia è molto semplice ed efficace, tuttavia richiede attenzione alla normalizzazione e scelta di α

In generale, l'hybrid search è particolarmente utile nei casi in cui una delle due componenti fallisce in modo sistematico: ad esempio, una query con termini molto specifici (codici, nomi propri, acronimi) tende a favorire la ricerca lessicale, mentre query "concettuali" o parafrasate beneficiano del canale denso; la fusione mira quindi a ridurre la probabilità di *zero-hit* e a rendere mediamente migliore la qualità del retrieval.

2.10 Approximate Nearest Neighbor (ANN) e HNSW

La ricerca dei *k-Nearest Neighbors* (KNN) è l'operazione fondamentale alla base del *dense retrieval*: dato un embedding di query $\mathbf{q}$ e un insieme di chunk/document embeddings $\{\mathbf{d}_1, \ldots, \mathbf{d}_N\}$, l'obiettivo è trovare i k vettori "più vicini" secondo una data metrica, tipicamente similarità coseno; l'assunzione chiave del retrieval denso, come suddetto, è geometrica: se lo spazio degli embedding è ben strutturato, la prossimità tra punti nello spazio approssima la rilevanza rispetto all'*information need* della query, quindi i documenti più vicini a $\mathbf{q}$ sono buoni candidati da usare come contesto in un processo di retrieval. Tuttavia, il KNN *esatto* diventa rapidamente proibitivo su corpora molto estesi: una ricerca brute-force richiede di confrontare $\mathbf{q}$ con tutti i N vettori, con costo lineare in N, oltre al costo di calcolo della similarità rispetto alla dimensione d degli embeddings; sebbene spesso trascurato può diventare rilevante.

In scenari reali di retrieval (milioni o miliardi di vettori, vincoli di latenza real-time), questa scansione completa è assolutamente ingestibile: serve quindi un indice che consenta di limitare drasticamente il numero di confronti, mantenendo comunque alta la *recall*. Gli algoritmi di *Approximate Nearest Neighbor* (ANN) nascono proprio per rispondere a quest'esigenza: accettano un'approssimazione dei vicini, abbandonando l'idea di risolvere il problema all'ottimo, ma restituire un parco di candidati *quasi* ottimali in cambio di una riduzione importante delle risorse di calcolo necessarie alla procedura. Molti approcci si basano sulla

costruzione e popolazione di una struttura dati che organizzi i vettori in modo da esplorare, a query-time, solo una piccola porzione dello spazio, evitando la scansione completa ed ottimizzando il trade-off esplicito tra accuratezza e prestazioni classico delle approssimazioni.

Tra i metodi di ANN più diffusi, *HNSW* (*Hierarchical Navigable Small World*) è uno standard de facto per la dense search: costruisce un grafo di prossimità in cui ogni nodo è un vettore e gli archi collegano vettori "vicini"; la struttura è *gerarchica*: ai livelli superiori il grafo è più sparso e serve per una navigazione ad alto livello verso regioni promettenti per la query, mentre ai livelli inferiori è più denso e consente di rendere la ricerca più precisa ragionando localmente; si parla spesso di alternanza tra *coarse* e *fine search*. Per essere proprio precisi: "più sparso" significa che i livelli alti contengono molti meno nodi, difatti non tutti i vettori compaiono a tutti i livelli, e l'assegnazione di un nodo ad un livello è probabilistica, con un nodo che può comparire anche in più livelli. La gerarchia funge da riassunto ad alto livello dello spazio nel suo complesso, utile per fare, intuitivamente, "salti lunghi" verso la regione corretta, mentre i livelli bassi, più popolati, consentono di rispondere alla query più precisamente. A query-time, si parte da un *entry point* nei livelli alti e si procede con una visita greedy: si esplora un insieme limitato di candidati, si aggiorna iterativamente il miglior vicino e si scende di livello man mano che ci si avvicina alla regione più promettente, fino al livello base dove si selezionano i top-k finali. Questo schema ottiene tipicamente alta recall con latenze molto basse, rendendo HNSW particolarmente adatto alle condizioni applicative nei casi reali di dense search. Non stupisce, infatti, come HNSW sia estremamente usato anche nelle SDK fornite dai maggiori cloud providers: ad esempio, Azure AI Search espone HNSW come algoritmo ANN di riferimento per la vector search, proprio perché offre il migliore trade-off.

2.11 PageRank

PageRank è un algoritmo classico per stimare l'importanza (autorevolezza) di un nodo in un grafo, originariamente introdotto nel contesto del web, dove i nodi rappresentano pagine e gli archi hyperlink [Page et al., 1999]. L'intuizione di base è che una pagina sia tanto più autorevole quanto più viene "citata" da altre pagine e che una citazione proveniente da una pagina già autorevole valga più di una proveniente da una pagina marginale; in questo senso, PageRank è principalmente una metrica di *ranking* dei risultati e non di pertinenza semantica rispetto ad una specifica query. Formalmente, PageRank si interpreta tramite il modello del *Random Surfer*: un navigatore che esegue una random walk sul grafo seguendo casualmente i link uscenti, ma che con una certa probabilità λ effettua un salto (*teleport*) verso una pagina scelta a caso; il valore stazionario associato a tale processo viene poi assunto come misura di centralità del nodo [Langville and Meyer, 2004].

L'aspetto interessante è che, nelle strutture a grafo, centralità/autorevolezza di-

pendono sempre dalle proprietà che codifichi tramite gli archi: nel caso del web sono collegamenti e PageRank rappresenta quanto bene una pagina è collegata; se invece gli archi modellano relazioni strutturali o semantiche, il significato di "importanza" cambia, ma il concetto rimane invariato: PageRank quantifica la rilevanza di ciascun nodo rispetto alla specifica struttura del grafo e alle proprietà veicolate dagli archi.

2.12 Community Detection e Clustering Semantico

Clustering e *Community Detection* sono concetti molto simili, ma non equivalenti. Il clustering, nel senso tradizionale, parte da una rappresentazione di punti in uno spazio e cerca di raggrupparli in insiemi: con algoritmi come K-means Clustering, DBSCAN... la nozione di "vicinanza" dipende interamente dalla rappresentazione e dalla metrica adottata, o meglio, da quello che abbiamo chiamato 'geometria' dello spazio discutendo degli embeddings; quando si parla di **clustering semantico**, infatti, non ci sono sostaziali differenze dal clustering tradizionale, se non lo spazio nel quale disponi punti: se tale è uno spazio di embeddings, clusterizzare significa, quindi, identificare gruppi di chunk/documenti che risultano *semanticamente allineati*, ovvero che occupano regioni vicine dello spazio vettoriale. Da qui è facile capire come la qualità dei cluster dipenda criticamente dal modello di embedding e dal modo in cui questo struttura lo spazio.

La *Community Detection*, invece, assume che i dati siano già rappresentati come grafo e mira a individuare insiemi di nodi **densamente connessi** al loro interno e **debolmente connessi** verso l'esterno; la differenza con il clustering non sta, infatti, nel concetto di 'gruppo', quanto più sulla struttura dati alla base, ovvero la scelta di modellare possibili interazioni trai nodi o meno. Le community vengono spesso definite come "moduli" del grafo e sono oggetto di un'ampissimo studio in letteratura e soprattutto nel retrieval, ai sensi del fatto che il Web è rappresentabile come un grafo dove i nodi sono pagine e gli archi collegamenti tra di esse, oltre che l'estensivo utilizzo in domini come social network, recommender systems... [Fortunato, 2010].

2.12.1 Louvain e Leiden

Un algoritmo molto noto è *Louvain*, che mira a massimizzare la **modularità** (denso internamente, poco esternamente) tramite una procedura greedy in due fasi ripetute iterativamente [Blondel et al., 2008]. Nella prima fase (*local moving*), si parte da una partizione banale in cui ogni nodo è una community e, uno alla volta, ciascun nodo viene spostato nella community di un suo vicino se tale spostamento produce un incremento della modularità; il processo continua finché non si ottengono ulteriori miglioramenti locali. Nella seconda fase (*aggregation*), ogni community trovata viene contratta in un *super-nodo* e i pesi degli archi tra super-nodi corrispondono alla somma dei pesi degli archi tra le rispettive community originali (gli archi interni diventano self-loop). Il risultato è un grafo più piccolo su cui si ripete lo stesso schema, ottenendo una decom-

posizione gerarchica che tende a produrre comunità via via più macroscopiche ad ogni livello.

Nelle pipeline di RAG che presenteremo successivamente, facciamo riferimento soprattutto a **Leiden**: nasce come miglioramento di Louvain, mantenendo sempre l'idea dell'ottimizzazione di una funzione obiettivo (ancora modularità tipicamente), introducendo, però, un passo di *refinement* che corregge una criticità ben nota di Louvain, ossia la possibilità di ottenere comunità non ben connesse; se guardi interamente alle communities trovate da Louvain, potresti vedere che queste si 'clusterizzano': comunità formate da gruppi di nodi debolmente connessi tra loro o, addirittura, non connessi, il che va contro l'assunto fondamentale di massimizzazione della densità interna delle communities, tendenti ad un sottografo fortemente connesso nel caso ottimo infatti. Leiden, quindi, alterna le stesse fasi di *local moving* dei nodi per migliorare l'obiettivo ed *aggregation* in super-nodi, aggiungendo un *refinement* che garantisce comunità meglio connesse e, ripetendo il processo in modo gerarchico, fornisce migliori garanzie sulla connettività interna delle comunità rispetto a Louvain, nonché risulti, spesso, anche più efficiente computazionalmente [Traag et al., 2019]. Per essere più precisi, la fase di refinement interviene all'interno di ogni community prodotta dal local moving: invece di accettare la community come blocco unico, Leiden guarda internamente e 'spezza' componenti o sottoinsiemi che non risultano sufficientemente coesi; i nodi così ottenuti possono essere riallocati in sotto-community "figlie" derivanti dalle precedenti, in modo da garantire che ciascun gruppo risultante sia ben connesso. Solo dopo questa pulizia strutturale si procede alla contrazione in super-nodi: così l'aggregazione avviene su comunità che rispettano almeno un certo livello di connettività interna, evitando che sotto-gruppi mal connessi vengano trascinati ai livelli successivi della gerarchia.

2.13 Communities nel Retrieval

Dal punto di vista del retrieval, communities e cluster possono essere impiegati in modo quasi sovrapposto: entrambi definiscono *regioni* semanticamente coerenti su cui instradare una query prima di fare un drill-down verso le componenti che le costituiscono (chunk testuali, ad esempio), ricordando che l'information need risiede, proprio, in quest'ultime.

L'uso più comune, infatti, è quello di recupero *coarse-to-fine*: si restringe il retrieval ai soli chunk contenuti nelle top communities per rilevanza rispetto alla query, migliorando precisione e riducendo rumore; su questo meccanismo torneremo nei capitoli successivi sulle Architetture RAG. Le communities fungono, quindi, da "contenitore tematico" per ridurre lo spazio di ricerca e rendere più facile collegare informazione distribuita nel singolo documento o tra documenti diversi; anche quando la community detection è tecnicamente diversa dal clustering, nel retrieval la si intende spesso come un clustering *strutturato*: invece di raggruppare punti solo per vicinanza nello spazio vettoriale, si raggruppano

nodi in base a come sono connessi, ottenendo comunque insiemi coerenti rispetto alle proprietà alla base degli archi che modelli (semantiche, posizionali...).

2.14 Language Models Generativi

Con *Language Model* (LM) ci si riferisce a modelli basati sull'architettura Trasformer, addestrati a stimare distribuzioni di probabilità su un vocabolario di 'token' facenti riferimento a testo o, come abbiamo visto, sub-words da concatenare; un LM può essere generativo o meno, BERT è un modello non generativo ad esempio. Come richiamato nella sezione sui Transformer, l'architettura di riferimento è quella introdotta in [Vaswani et al., 2017], basata su blocchi di self-attention, feed-forward position-wise, residual connections e layer normalization: sebbene i moderni modelli generativi GPT-like siano **decoder-only**, l'architettura nel complesso non è così tanto differente dagli **encoder-only** presentati con BERT, se non per l'utilizzo atteso ad inferenza del modello stesso; l'obiettivo, adesso, è di generazione di token, difatti se in BERT la backbone è fondamentale e la testa viene spesso cambiata all'occorrenza, qui anche in fine tunings successivi il modulo che utilizzi per generare token rimane identicamente lo stesso. Un LM, quindi, è riassumibile proprio guardando la distribuzione di probabilità che apprende sul vocabolario: dato un contesto $x_{<t}$, il modello produce una distribuzione $p(x_t \mid x_{<t})$ sui token possibili e, nel caso auto-regressivo, la probabilità dell'intera sequenza fattorizza come $\prod_t p(x_t \mid x_{<t})$. Il modello *non* "memorizza" risposte, ma apprende una funzione che associa una distribuzione sul prossimo token da produrre e data, ovviamente, una certa sequenza in input; ciò che ti aspetti il modello impari, quindi relazioni sintattiche, grammaticali, posizionali... è lo stesso che in LM non generativi.

La generazione, come suddetto, avviene nella *testa* (*head*) del modello: così si intende il modulo di output che trasforma la rappresentazione nascosta finale prodotta dalla backbone Transformer in una predizione nel dominio del task; nel caso dei LM generativi, la *LM head* è tipicamente una proiezione lineare W_{out} (spesso con bias) che mappa lo stato nascosto $h_t \in R^d$ in logits $z_t \in R^{|V|}$ sul vocabolario, seguita da una softmax:

$$z_t = W_{\text{out}} h_t + b, \qquad p(x_t \mid x_{<t}) = \text{softmax}(z_t).$$

Questa testa, quindi, implementa esplicitamente l'operazione di mappatura verso il vocabolario e definisce la distribuzione che il modello utilizza ad inferenza. Esistono diversi modi di sfruttare la distribuzione appresa: scegliere sempre il token con maggiore probabilità (la softmax somma ad 1), piuttosto che introdurre parametri, come la *temperatura* T, che evitino questo bias strutturale di scegliere sempre il token migliore, aumentando l'eterogeneità delle risposte.

L'attention funziona similmente a BERT, ma stavolta applica una **causal mask** vincolando ogni posizione t a "vedere" solo i token precedenti $< t$; in questo modo, la generazione è **auto-regressiva**: avviene un token alla volta, campionando. Durante il training, questo obiettivo viene implementato in modo

diretto tramite *next-token prediction*: dato un testo $x_{1:T}$, si costruiscono coppie input/target *shiftando* la sequenza di una posizione, cioè il modello riceve in input $x_{1:T-1}$ e deve predire $x_{2:T}$ (equivalentemente, per ogni posizione t predice x_t dato $x_{<t}$), ottimizzando una cross-entropy sui token target. La causal mask è quindi essenziale per evitare *peeking*: se la self-attention fosse bidirezionale, la rappresentazione al passo $T-1$ potrebbe già "vedere" il token T durante il forward pass, rendendo il compito di predizione artificialmente facile e inducendo il modello a imparare corrispondenze inesistenti; ad inference-time, d'altronde, non hai informazione sui token futuri, quindi creeresti un mismatch ta training ed utilizzo e la causalità serve proprio a questo.

LLM vs SLM In letteratura non esiste una soglia universalmente accettata che separi *Small Language Models* (SLM) e *Large Language Models* (LLM): la distinzione è in larga parte relativa allo stato dell'arte. Detto questo, una convenzione quantitativa ragionevole è trattare come **SLM** i modelli nell'ordine di **pochi miliardi di parametri** (ad esempio ∼1–8B), che rimangono deployable con requisiti più contenuti e spesso adatti a scenari con minore disponibilità di risorse [Subramanian et al., 2025]; molti lavori e survey usano **LLM** per riferirsi a modelli cosiddetti *foundational* con **oltre ∼10B parametri** e spesso anche nell'ordine di decine o centinaia di miliardi [Subramanian et al., 2025]. In una prospettiva più "industriale", si trovano anche definizioni operative che collocano gli SLM tipicamente tra **100M e 10B** parametri, mentre gli LLM arrivano a **centinaia di miliardi o trilioni** [Chen, 2025], il che, personalmente, preferisco.

2.15 Chain-of-Thought

Il *Chain-of-Thought* (CoT) è una famiglia di tecniche di prompting che induce il modello a simulare una sequenza di passaggi intermedi prima di produrre la risposta finale; in realtà, l'approccio a cui si ispira è molto simile al principio di *divide et impera*: scomporre un problema complesso in sottoproblemi più semplici e risolverli in sequenza; empiricamente il ragionamento step-by-step rende il Language Model molto meno error prone, efficace soprattutto per modelli grandi e task complessi: d'altronde, il CoT stesso coincide con quello che comuenemente si chiama *reasoning* del modello, ovvero token intermedi che utilizza per scomporre il problema ed approcciarlo, come suddetto, step wise [Wei et al., 2022].

Operativamente, il CoT può essere realizzato in diversi modi:

- **Few-Shot**: esempi nel prompt in cui, oltre alla risposta, viene mostrata una possibile catena di passaggi da seguire [Wei et al., 2022]
- **Zero-Shot**: istruzioni minimaliste del tipo "let's think step by step", inducendo il reasoning anche senza esempi [Kojima et al., 2022]

- **Self-Consistency**: invece di usare una singola catena, si sfruttano diverse, chiamiamole, 'run' del modello, ovvero più ragionamenti con temperature > 0, aggregando la risposta più frequente o più coerente; questo fa riferimento a concetti simili al majority vote e con forte integrazione nell'ensembling di LLM, mitigando la variabilità della singola generazione [Wang et al., 2022]

Se si guarda alle tecniche modere di training dei LLM, principalmente i cosiddetti post-trainings, si vede come i modelli vengano direttamente allenati su problemi (matematica, programmazione...) dove è direttamente disponibile uno o più ragionamenti con cui costruire la soluzione: nei modelli moderni il reasoning non è solamente indotto dal prompting, bensì si cerca di instillare direttamente capacità di ragionamento nella distribuzione appresa ed in maniera tale che la CoT stessa sia più efficace.

In questo lavoro, il CoT è stato utilizzato soprattutto come meccanismo di **adattività del retrieval**: prima di generare la risposta, il modello sviluppa una catena di ragionamento scomponendo la query in sotto - domande più semplici per interpretare meglio la richiesta e stimare quali evidenze siano necessarie, quali siano già coperte dal contesto recuperato e, soprattutto, che cosa *manca* ancora per soddisfare l'*information need*; così, il CoT non serve solo a "ragionare meglio" ma a guidare la ricerca stessa, rendendo la pipeline, appunto, adattiva alla variabilità della complessità delle query, con buon aumento dei costi di gestione.

2.16 LLM-as-a-Judge

Valutare un motore di ricerca non è semplice: nelle sezioni precedenti abbiamo discusso dei problemi che ci sono nel retrieval full-text, come il Vocabulary Mismatch, monosemia o polisemia del dominio... tutte caratteristiche ragionevolmente mitigate dalla ricerca densa; trapela un aspetto che, qui, diventa fondamentale: le performance di un motore di ricerca dipendono prevalentemente da come gli utenti formulano le query, quindi dovresti fissare una distribuzione di riferimento per ottenere metriche significative al contesto applicativo; questo non è sempre possibile e né facile fare, nonché anche costruire una ground truth è arbitrariamente complesso: se ti riduci a metriche come Precision, Recall, F1-Score... presupponi che, per ogni query che valuti fissata la distribuzione, tu possa avere un ground-truth sulla base di tutti i documenti che rispondono, o meno, all'informatio need. In corpora molto grandi, anche se non valutassi direttamente chunking, dovresti, per ogni query, avere etichette rispetto a tutti gli elementi del corpora: i costi di labeling esplodono, tuttavia necessari per ottenere, soprattutto sulla Recall, valutazioni significative.

Per superare criticità del genere, indipendentemente dal contesto di retrieval, si è diffuso il paradigma *LLM-as-a-Judge*: utilizzare un Language Model come *valutatore* che, dato un input ed una o più risposte candidate, produce

una label seguendo criteri di valutazione espliciti indicati tramite prompting. Sostanzialmente, le capacità del modello non vengono sfruttate solo per generare una risposta alla query, ma anche per *valutarla*: l'LLM può confrontare l'output con una reference, oppure stimarne la qualità assegnando etichette discrete o punteggi numerici facenti riferimento a metriche indicate e spiegate nel prompt; in generale decidere quale tra più risposte candidate sia la migliore. Questo approccio è diventato popolare soprattutto per la semplicità: consente valutazioni rapide e relativamente economiche rispetto ad annotazione umana, pur mantenendo significatività rispetto alle valutazioni prodotte. [Zheng et al., 2023, Liu et al., 2023].

I pattern tipici di LLM-as-a-Judge includono:

- **Pairwise preference**: dato (prompt, risposta A, risposta B), il giudice sceglie la migliore e/o fornisce una motivazione; spesso più stabile di uno score assoluto, d'altronde puoi sfruttare la descrizione per audit successivi e prompt engineering
- **Rubric-based scoring**: si fornisce una griglia di criteri (es. correttezza, completezza, grounding, chiarezza) e si chiede un punteggio per ciascuno [Liu et al., 2023].
- **Reference-based grading**: si confronta la risposta con un gold answer; utile quando hai una reference affidabile e come metrica per comprendere le performance del modello nel judging, non sempre hai dei gold answer però
- **Multi-judge / majority vote**: si usano più giudizi (variazioni di prompt o modelli) e si aggrega per ridurre la varianza nelle risposte; stesso concetto di ensembling nel ML, maggiori costi ovviamente

Quando si tratta di valutare una pipeline di RAG, come spiegheremo successivamente, il Judging con LLM è molto interessante nel valutare soprattutto: il **grounding**, cioè se la risposta è effettivamente supportata dalle evidenze recuperate e non le contraddice, piuttosto che la **factuality**, cioè l'assenza di allucinazioni o affermazioni non giustificate dal contesto e la **completezza**, intesa come copertura dei punti richiesti dalla domanda senza introdurre divagazioni o contenuti superflui; oltre che potresti pensare di utilizzarlo anche per valutare la bontà dei chunk prodotti, quindi come valutazione per la strategia di chunking. Insomma: il vantaggio principe dei LLM, coincidente con le ottime prestazioni generaliste (zero - shotting), si vede anche e soprattutto usandoli come valutatori.

2.16.1 Bias

Molti vantaggi sicuramente, ma altrettante criticità spesso "velate" e difficili da interpretare; per spiegare meglio quello che voglio dire: spesso modelli cinesi (ad es. Qwen, Kimi) non sono buoni *judge* in contesti general-purpose, non tanto per

limiti prestazionali, quanto per questioni di *alignment* del giudizio relativo al pre-training di questi modelli. La funzione di giudice richiede prevalentemente coerenza nel comprendere e nell'applicare criteri: per dirla proprio informalmente, un LLM è tanto più un bravo Judge quanto più è imparziale, quindi, se il training ha insegnato a rispondere seguendo determinate policies, automaticamente riduci l'imparzialità e la bontà del giudizio; ho fatto precedentemente l'esempio di modelli cinesi proprio perché sono abbastanza emblematici di questo: basta provare a chiedergli un giudizio sulla qualità argomentativa di due testi che parlano di argomenti politicamente sensibili, come Taiwan ad esempio, per rendersi presto conto di quello che stiamo dicendo.

I problemi più noti ed interessanti del LLM Judging, ricollegandoci all'intuizione precedente, a mio avviso si riassumono in:

- **Bias di stile**: il LM può preferire risposte più lunghe, sicure nel tono o meglio formattate anche quando sono meno rilevanti rispetto all'information need
- **Prompt Sensitivity**: qui non ha senso fare fine tuning, o meglio, l'utilizzo stesso del LM come Judge impone voler sfruttare le capacità generaliste e fine tunings ti appesantiscono situazioni nelle quali vorresti, ad esempio, cambiare facilmente diverse metriche valutative; tuttavia, piccoli cambiamenti nel prompt possono alterare di molto i giudizi ed è necessario fornire valutazioni stabili; per questo, spesso si parla di 'settare la *temperatura* a 0': questo è un parametro che indica, intuitivamente, la 'creatività' del modello, ovvero quanto vuoi diverse le risposte che genera a parità di prompt. Per judging, ovviamente, auspicabilmente vorresti che queste siano il più possibilmente deterministiche, low variance insomma
- **Calibrazione degli score**: punteggi numerici (es. 1–10) sono spesso poco calibrati e non comparabili tra prompt/modelli; questi sono abbastanza pericolosi, poiché inducono un'illusione di precisione: il LM risponde, ad esempio, 5 non perché valuti realisticamente rispetto ad una scala di cui ha cognizione, ma perché è la distribuzione di probabilità sui token da generare, acquisita durante il pre-training, che gli indica tale score; per questo, a mio avviso, quando si sfrutta un LM come Judge è sempre meglio far riferimento ad etichette categoriche: la semantica dei termini è appresa ed è, per il modello, più facile modellare un'associazione termine-input che valutazione-input, in quanto i numeri stessi non hanno una semantica ben definita o collegabile al prompt stesso

Un aspetto particolarmente interessante è la forte analogia con i problemi classici del *crowdsourcing*: quando si vuole stimare l'affidabilità di un worker si introducono tipicamente gold standard, majority vote, misure di agreement e score di qualità. Il judging con LLM, difatto, ripropone problematiche molto simili e concettualmente potresti estendere le suddette mitigazioni, usatissime nella pratica.

2.17 SDK e Cloud Providers

Qui tocchiamo una parte più 'tecnica' relativa agli strumenti utilizzati per costruire le pipeline di RAG discusse successivamente; manteniamo sempre un tono concettuale, ai sensi del fatto che discutere a livello tecnico di 'codice' implementativo, di fondo, non aggiunge nulla al lettore ed appesantisce inutilmente la discussione. Per *Cloud Provider* si intendono aziende che offrono prevalentemente risorse di calcolo, storage e servizi software tramite infrastrutture distribuite (data centers), tipicamente con l'obiettivo primario di garantire **high availability** e continuità di servizio: ridondanza, tolleranza ai guasti, scalabilità elastica e, più in generale, qualità di servizio difficilmente ottenibili on-premise. Nonostante, ad oggi, si parli molto più di AI che di Cloud, quest'ultimo riveste comunque un'importanza fondamentale: le necessità computazionali introdotte da molti degli strumenti presentati per il retrieval, come modelli di embedding, LLM, pipeline di Indexing... sono tali da rendere il cloud un prerequisito in molti contesti applicativi per tutti i vantaggi suddetti.

Tali Cloud Providers forniscono, spesso, cosiddette *Software Development Kits* (SDK): sono collezioni di librerie, API e strumenti che facilitano l'accesso ai servizi del provider, permettendo di interagire con essi ai sensi di creazione, eliminazione, monitoraggio... delle risorse. Nel nostro caso applicativo di RAG, questa astrazione si traduce nella possibilità di costruire rapidamente una pipeline end-to-end usando componenti predefiniti per storage, ingestion, indicizzazione, retrieval e integrazione con endpoint LLM/embedding, riducendo drasticamente il *time-to-first-result*; la principale criticità, che discuteremo meglio successivamente, riguarda proprio la standardizzazione stessa e la sua tendenza a vincolare molte scelte critiche per la qualità del retrieval: pre-processing, chunking, ranking... mitigando, di fondo, molto la semplicità progettuale che viene dalle SDK stesse.

2.18 LangChain

LangChain è una libreria progettata per costruire applicazioni LLM-centric, offrendo, anch'essa, un livello di astrazione importante nella gestione di unità di lavoro LLM-based all'interno di prodotti più grandi; le considerazioni che ho tratto lavorando con LangChain mirano, proprio, sul quello che, a mio avviso, è il valore principale: il controllo; permette di definire in modo esplicito loader, trasformazioni di pre-processing, strategie di chunking avanzate, pipeline di embedding e costruzione di indici vettoriali, oltre a comporre catene più articolate con step intermedi (query rewriting, routing, multi-step retrieval...), rendendo LangChain adatto a riprodurre ed estendere pipeline RAG "gestite" dai provider, mantenendo però la possibilità di intervenire su punti progettualmente critici. Sostanzialmente quindi, da una parte le SDK cloud accelerano l'integrazione infrastrutturale, dall'altra LangChain accelera l'esplorazione sperimentale e principalmente quando, appunto, non devi sottostare a tutti quei vincoli che derivano dall'immissione sul mercato di un prodotto.

2.19 ElasticSearch

ElasticSearch (ES) è un query engine distribuito costruito su *Apache Lucene*, progettato per indicizzare e interrogare grandi volumi di dati privilegiando scalabilità e bassa latenza; è molto adottato in contesti industriali, offrendo un compromesso molto bilanciato tra prestazioni e semplicità nell'utilizzo, per questo ho deciso di adottarlo anche nel mio lavoro. Dal punto di vista del retrieval, ES eredita l'impostazione dei motori di ricerca classici basati su *inverted index*: il testo è sottoposto a pipeline di indicizzazione definite tramite analyzers (tokenizzazione, normalizzazione, eventuale stemming...), con una forte integrazione con il formato JSON per i documenti. Su questa base, ElasticSearch sfrutta metriche di ranking consolidatissime nella pratica e già discusse, come BM25, consentendo di gestire diverse tipologie di query: full-text, booleane, exact match... oltre che vettoriali secondo i meccanismi di ANN precedentemente trattati [Elastic, 2026].

2.20 Groq

Groq è un provider di inferenza focalizzato su **bassa latenza** e alto **throughput** nell'esecuzione di LLM a costi competitivi, esponendo endpoint compatibili con API standard ed un parco modelli abbastanza ampio per coprire casi d'uso anche diversi da quello classico di QA: modelli general-purpose, varianti più leggere per real-time, Speech-to-Text.. Il principale vantaggio che ho riscontrato nell'utilizzo è la semplicità di gestione dell'API tramite libreria Python di Groq, risultando generalmente meno verbosa rispetto a soluzioni anche più blasonate come OpenAI, piuttosto che Google. Nel lavoro, Groq è stata utilizzata prevalentemente per il *judging* e per la generazione della risposta finale a partire dal contesto recuperato dal retrieval.

3 Retrieval Augmented Generation

I Large Language Models (LLM) hanno reso possibile interagire dinamicamente con una grande quantità di informazione codificata, grazie al pre - training, nei parametri stessi del modello. Quando si parla di **allucinazioni**, si fa riferimento ad un fenomeno in cui il Language Model restituisce una risposta 'inventando' informazioni che sono sì, spesso e volentieri, semanticamente correlate alla richiesta ma non sono supportate da evidenze reali. L'idea del RAG di fondo è semplice: prima di rispondere, il sistema valuta la richiesta dell'utente su un corpus documentale, poi condiziona la generazione sulla base di cosa la ricerca restituisca, ottenendo risposte potenzialmente più accurate e, soprattutto, verificabili; c'è una forte interazione, quindi, tra retrieval e Language Models, anzi: in un certo senso, quest'ultimi servono solo come mezzo di organizzazione dell'informazione pertitente, difatti con system prompt opportuni puoi istruirli ad utilizzare la conoscenza pregressa non per rispondere direttamente ma per interpretare l'informazione e costruire una risposta basata unicamente sul 'ground' fornito dalla ricerca.

L'importanza del RAG è principalmente riconducibile, oltre al concetto precedente di mitigazione delle allucinazioni, ad una necessità strettamente pratica: allenare continuamente LM è costoso, anzi, inutile ogniqualvolta l'informazione sia caratterizzata da una forte componente di **dinamismo**; il fatto che un LM possa essere allenato su testo ed imparare da esso implica che questi abbiano un'estensione abbastanza naturale al *Question Answering*, infatti i modelli di maggiore utilizzo ad oggi vengono definiti *Chat Completion Models* proprio per questo, tuttavia non necessariamente l'allenamento è la strada migliore per farlo. In contesti, come quelli aziendali ad esempio, l'informazione tipicamente evolve molto velocemente, di conseguenza il RAG si concentra sul progettare una pipeline che comprenda dapprima una ricerca efficiente sulla base documentale, in quanto se riesci sempre a fornire un contesto corretto ottieni buone risposte alle query dell'utente. Generalmente, si pensa che questo avvenga indipendentemente non solo dall'allenamento del modello ma anche dal modello stesso, permettendo performance migliori anche con Small Language Models: torneremo su questo concetto con i risultati, in quanto, seppur naturalissima, non necessariamente è un'intuizione corretta.

3.1 Semantic Collapse, Hubness e Chunk Dependency

In una implementazione *naïve*, il RAG procede tipicamente così:

1. i documenti vengono segmentati in *chunk* di lunghezza fissa,
2. ogni chunk viene trasformato in un vettore di embedding e inserito in un indice vettoriale
3. a query time si calcola l'embedding della domanda e si recuperano i top-k vicini più simili rispetto a metriche come, prevalentemente, la *cosine similarity*, seguendo sempre un approccio Nearest Neighbor, o meglio, versioni approssimate e più efficienti computazionalmente come ANN ma non cambia il discorso
4. i chunk recuperati vengono passati all'LLM per la risposta

Questo schema funziona bene in molte applicazioni, ma tende a degradare quando la base di conoscenza cresce e diventa eterogenea, oltre a particolarità derivanti dalla distribuzione lessicale del caso; è proprio in questi casi che emerge il cosiddetto *semantic collapse*, ovvero la perdita di separazione utile tra contenuti davvero rilevanti e contenuti solo semanticamente simili. Il retrieval denso (dense retrieval) si basa sull'ipotesi che la similarità geometrica nello spazio degli embedding sia un buon approssimante della rilevanza rispetto la query (information need). Tuttavia, quando l'indice aumenta di dimensione, l'accuratezza del recupero può peggiorare sensibilmente: Reimers e Gurevych mostrano teoricamente ed empiricamente che, con indici grandi, le rappresentazioni dense possono incorrere in un aumento di *false positive*, fino a un punto di svolta in cui approcci full - text (es. classic BM25) possono diventare competitivi o

addirittura superiori [Reimers and Gurevych, 2021]. Questo comportamento è perfettamente coerente con aspetti più generali della *curse of dimensionality*, dove la nozione di "vicinanza" in spazi ad alta dimensionalità perde di importanza rispetto a quello che normalmente si osserva in un piano euclideo ad esempio [Chen et al., 2025a]. Di conseguenza quindi, quando sfrutti un indice vettoriale hai di base un limite alla dimensionalità degli embedding, e quindi alla loro rappresentatività, per il problema della *curse of dimensionality* e, all'aumentare della grandezza del corpora, ti imbatti di base in un *semantic collapse*, il che aiuta a capire meglio, poi, quando discuteremo di approcci Hierarchical.

Un secondo problema strettamente correlato è la *Hubness*: in spazi ad alta dimensionalità, alcuni punti, cosiddetti hub, tendono a comparire con frequenza anomala come vicini di molti altri punti, distorcendo la ricerca dei nearest neighbors [Feldbauer and Flexer, 2019]; in un contesto RAG come il nostro, gli hub sono sostanzialmente chunk molto "generici" e che si trovano in posizioni definiamole 'centrali' dello spazio degli embedding, entrando spesso nel top-k a prescindere dalla specificità della domanda, riducendo la qualità dell'informazione restituita. Quest'ultimo problema di Hubness è fortemente correlato con la dipendenza del dense retrieval dal *chunking*: la segmentazione del testo determina l'unità informativa che indicizzerai, quindi il tuo corpus, e conseguentemente il vantaggio prestazionale che la ricerca semantica aggiunge a metodi tradizionali; sembra banale che il retrieval denso basato su embedding dipenda da che cosa usi per costruirli, ma proprio perché è un concetto basilare l'approccio al chunking è fondamentale. I chunk non sono praticamente mai auto esplicativi, difatti se troppo piccoli frammentano il contesto e spezzano il discorso, catene logiche... mentre chunk troppo grandi aumentano rumore, mischiano concetti diversi e riducono molto più facilmente l'allineamento domanda - testo rilevante. D'altronde, la stessa informazione può essere distribuita su parti lontane di un documento: il chunking ti fa intrinsecamente perdere relazioni del genere. Sostanzialmente quindi, parte della qualità di un RAG è "decisa" prima ancora del retrieval, al momento della definizione dei chunk e dei metadati.

Torneremo in maniera più approfondita su molti dei concetti precedenti nelle sezioni successive.

3.2 Proposta Iniziale

La proposta iniziale prevedeva una combinazione di tre tecniche:

1. Recursive Chunking come alternativa al Fixed per ottenere una prima segmentazione "strutturale" del documento

2. Clustering semantico dei chunk tramite distanza tra embedding, con l'obiettivo di raggruppare contenuti semanticamente simili

3. LLM Stitching per riscrivere i chunk del cluster semantico in manierà più diretta, oltre che risolvere errori derivanti, ad esempio, da modelli OCR preliminari

Il lavoro si è esteso rispetto alla suddetta formulazione, mantenedo gran parte della struttura ma integrando le intuizioni rispetto alle proposte del relatore; in particolare: l'analisi e l'utilizzo di strutture a grafo (GraphRAG) si integra molto bene con il clustering semantico, difatti per identificare sottoinsiemi di chunk del genere è possibile utilizzare algoritmi di **Community Detection** come Louvain/Leiden. Il primo passo della tesi rimane comunque l'analisi di strategie di chunking alternative al fixed chunking, privilegiando segmentazioni *structure-aware* come il Recursive Chunking, estendendo anche a soluzioni *semantic-aware*. Un punto emerso durante lo studio riguarda, poi, l'impatto delle trasformazioni testuali sul retrieval ibrido: non adottiamo più LLM stitching, in quanto una riscrittura, sia questa aggressiva o meno, può indebolire la componente *full - text* della ricerca, determinante per una certa classe di query, riducendo proprio la sovrapposizione lessicale tra query e testo, rendendo complessivamente la qualità delle risposte ottenute molto minore per query difatto semplici; sintesi ed astrazioni vengono, tuttavia, considerate unicamente come supporto per ridurre lo 'scope' della ricerca e, comunque, parte integrante dell'approccio Hierarchical di cui diamo un'intuizione nel paragrafo che segue.

La struttura che ho costruito coincide con un *semantic chunk graph*, coerentemente con la proposta del relatore: i nodi rappresentano chunk e gli archi codificano sia relazioni strutturali (es. adiacenza) sia relazioni semantiche (es. similarità tra embedding); su questa base, valutiamo la componente di *Clustering Community Based*: l'idea è instradare la query primariamente su una dimensione aggregata data da un riassunto ad alto livello della tematica di community, per poi valutare un drilldown successivo sui chunk che la compongono. Questa impostazione è coerente con la direzione di approcci gerarchici nella letteratura RAG (ad es. RAPTOR), che mostrano i benefici del retrieval a diversi livelli di astrazione [Sarthi et al., 2024]; analogamente, lavori come GraphRAG evidenziano come una rappresentazione a grafo e l'aggregazione a livello di community possano supportare domande più "globali" e multi-hop [Edge et al., 2024].

4 Chunking

Molti aspetti del chunking e delle criticità che introduce sono già stati discussi; resta però naturale chiedersi perché sia davvero necessario. Il primo motivo è strutturale: i language model hanno una finestra di contesto limitata, che nella pratica impedisce quasi sempre di fornire in input un documento intero (o, a maggior ragione, un'intera base documentale); di conseguenza, la scomposizione del testo in unità più atomiche non è una scelta "opzionale", ma una necessità. C'è poi una motivazione legata agli embedding: se l'obiettivo è rappresentare in vettori la semantica del testo, allora un contenuto troppo esteso e ricco di argomenti tende a mescolare segnali diversi, in quanto l'informazione specifica

che vorremmo recuperare viene "diluita" in una rappresentazione più generica, al contrario chunk eccessivamente brevi rischiano di perdere la semantica globale e il contesto necessario per interpretarli correttamente. Il chunking, infatti, nasce proprio come compromesso tra questi due estremi: preservare abbastanza contesto da mantenere coerenza rispetto al documento di riferimento, ma non così tanto da perdere l'informazione specifica. Grosso del perché, a livello pratico, la ricerca sia spesso una *Hybrid Search*, combinando sia ricerca densa che full - text, deriva proprio dal garantire maggiore specificità del retrieval quando la sovrapposizione lessicale lo permette. In questo senso, il RAG si fonda sull'assunzione che grandi collezioni documentali non possano essere incluse integralmente nel contesto del modello; idealmente, basterebbe fornire ogni volta l'intero corpus come contesto al language model e ottenere risposte perfettamente accurate. Nella pratica, questo approccio è proibitivo per vincoli soprattutto relativi al costo (i LM si pagano per token in input e generati), rendendo il chunking l'unica alternativa praticabile rispetto a tutti i vincoli.

4.1 Sentence Transformers

I Sentence Transformers sono modelli pensati per produrre embedding di porzioni testuali, ottimizzati per compiti di similarità e retrieval: l'architettura tipica riusa un backbone Encoder di tipo Transformer (es. BERT/RoBERTa), anche se ad oggi si vedono anche riutilizzi di porzioni delle architetture Decoder di Language Models generativi, ed aggiunge un meccanismo di pooling (mean/max/CLS o varianti) per comprimere le rappresentazioni token-level in un singolo vettore a dimensione fissa. Gli embeddings risultanti sono proiezioni in uno spazio, tipicamente alto - dimensionale, che mirano a catturare la semantica "a livello di frase", in modo che testi con significato simile abbiano vettori vicini secondo una metrica (spesso similarità coseno) e testi semanticamente lontani siano ben separati. L'addestramento avviene con losses contrastive o ranking-oriented: in configurazioni Siamese/bi-encoder si usano, ad esempio, cosine/MSE per regressione di similarità, triplet loss, contrastive loss e soprattutto Multiple Negatives Ranking Loss (o InfoNCE), che spinge le coppie positive ad avvicinarsi e separa le negative nello stesso batch. Tramite le suddette loss risolvi un problema fondamentale alla base del pre - training di architetture Transformer come BERT: lo spazio degli embedding prodotto è molto schiacciato, tanto che le differenze di cosine similarity che determinano la differenza o similarità semantica tra due testi sono talmente piccole che, informalmente, 'tutto sembra uguale a tutto'; ci si riferisce spesso a questo come **anisotropia** dello spazio.

Il modello che si usa è molto importante in relazione alle necessità: se la pipeline di RAG vive in contesti con vincoli real - time, allora non puoi aspettarti di utilizzare Sentence Transformer molto grandi, o meglio, quelli che occupano le prime posizioni delle leaderboard in MTEB (benchmark), in quanto sono molto molto lenti nella costruzione degli embedding, il che va totalmente contro le condizioni di alto traffico. Nel nostro caso, abbiamo utilizzato `BAAI/bge-large-en-v1.5`:

modello con circa $600M$ di parametri, relativamente lento ma molto utilizzato tra quelli open - source per progetti di ricerca e complessivamente garantisce un buon tradeoff rispetto ad i vincoli discussi precedentemente. Come suggerisce il nome, il modello è adatto solamente per testi in lingua inglese: abbiamo fatto quest'assunzione sulla scelta dei dataset di testing, sebbene non necessariamente il mondo reale ti mette davanti situazioni sempre di questo tipo, infatti l'utilizzo di Decoder di modelli generativi come SentenceTransformer viene proprio dal fatto che il pre - training di questo li rende molto più robusti alla variabilità della lingua.

4.2 Strategie

L'importanza del chunking è già stata trattata a livello concettuale e, ai fini di questa sezione, l'infarinatura è sufficiente: importante evidenziare, però, che non esista una strategia universalmente ottimale, poiché la bontà della scelta dipende fortemente dal dominio e dalla natura del testo. In documenti tecnici, come articoli scientifici ad esempio, la struttura esplicita di titoli, paragrafi, elenchi... è spesso affidabile per segmentare preservando segnale utile, definendo un'unità semantica fondamentale; al contrario, in un romanzo la coesione semantica segue dinamiche narrative (scene, dialoghi, personaggi, flashback) che rendono i confini precedenti un modo di chunkare troppo "meccanico", rischiando di spezzare la continuità sia strutturale che logica. D'altronde, testi provenienti da OCR o con una formattazione particolare sono spesso soggetti a vari errori di conversione da modelli commerciali e richiedono una fase di pre - processing prima della segmentazione, in quanto ciò che da questi viene dichiarato come capitolo, sezione... potrebbe essere un semplice testo in **bold** come questo, quindi non è sempre un'alternativa viabile.

Questo è un punto di discussione importante per inquadrare il lavoro fatto: l'obiettivo è quello di ricavare risultati sperimentali relativamente a procedure che non vengono costruite specificamente per un singolo dominio, anche se per rendere una pipeline di RAG competitiva le scelte che vai a fare devono tassativamente essere verticali al particolare contesto. Difatti, sebbene non esista un **gold - standard** ho provato a costruirne uno, quindi degli approcci che indipendentemente dal caso ti garantiscono delle performance ragionevolmente decenti ed i discorsi che andiamo a fare sono intendibili come un punto di partenza sul cui basarsi per scelte progettuali successive e, appunto, più specifiche al dominio di riferimento.

4.2.1 Fixed e Recursive Chunking

Il **Fixed Chunking** è, probabilmente, la strategia più famosa: segmenta il testo in blocchi di lunghezza costante (tipicamente per numero di caratteri o token), spesso con una overlap per non perdere contesto tra chunk adiacenti; è semplice, veloce e facile da parametrizzare, ma introduce problemi strutturali evidenti poiché, nonostante l'overlap, nulla ti vieta di produrre chunk dove le

frasi vengono interrotte e le parole spezzate, con il rischio di peggiorare sia embedding che retrieval duplice la poca interpretabilità.

Il **Recursive Chunking**, come implementato in LangChain (ad es. `Recursive CharacterTextSplitter`), mantiene la stessa idea di dimensionalità fissa (`chunk_size` e `chunk_overlap`), ma invece di tagliare in modo uniforme prova prima a separare il testo usando una gerarchia di separatori "dal più forte al più debole" (ad es. doppi a capo → a capo → spazi → fallback su caratteri), ricorrendo in modo iterativo finché ogni segmento rientra nella soglia: in questo modo tende a preservare paragrafi e unità locali dove possibile, riducendo la frammentazione precedente dell'approccio Fixed. Rimane comunque fortemente euristico: la qualità dipende dai separatori scelti e dalla formattazione del testo (OCR rumoroso, LaTeX, markdown, ecc.), potendo ancora produrre tagli innaturali quando non trova delimitatori "buoni" o quando l'informazione semanticamente rilevante supera le dimensioni imposte che, come suddetto, rimangono fisse. Tuttavia, il Recursive Chunking conserva il "buono" della semplicità che c'è nel Fixed Chunking (stessi parametri e costi simili), ma strutturalmente ti permette di attenuare molte delle criticità alla base dell'approccio Fixed; successivamente faremo riferimento proprio a questo: sebbene la strategia sia fortemente dipendente dal dominio, spesso e volentieri approcci Recursive sono quanto di più vicino ci sia ad un gold - standard.

4.2.2 Semantic Chunking

Il **Semantic Chunking** mira a segmentare un testo seguendo la sua coerenza concettuale, anziché per finestre fisse o strutturalità: un approccio molto usato di recente, e a cui mi sono ispirato, è quello *sentence-by-sentence*, dove si calcolano gli embedding di frasi (o finestre brevi) consecutive e si identifica un punto di taglio quando la similarità coseno tra porzioni adiacenti scende sotto una soglia, interpretandolo come topic shift. Questo lo differenzia dagli approcci precedenti e lo rende più robusto ad impaginazioni irregolari o contenuti in cui la struttura non coincide con i confini semantici. Esistono molte varianti di Semantic Chunking, ma quasi tutte ruotano attorno allo stesso principio: segmentare dove si osserva una discontinuità semantica; il problema principale degli approcci sentence-by-sentence è prevalentemente dato dalla definizione del punto di taglio e per questo ho costruito una variante del chunking semantico che sfrutta indicatori statistici per rendere l'approccio più robusto. Progettualmente è sorto un aspetto abbastanza delicato e che sarà emblematico quando discuteremo dei risultati: se si costruisce progressivamente un chunk "accumulando" frasi e rappresentandolo con la media degli embedding già calcolati, allora la rappresentazione tende spesso a collassare verso un centroide poco informativo, ovvero un embedding 'Hub' che racchiude la totalità semantica del documento e che è sempre molto simile a tutto il contenuto testuale successivo, rendendo più difficile rilevare topic shift e producendo chunk valutando il taglio solo quando vengono eccedute le dimensioni massime prefissate, collassando praticamente ad un fixed, oltre che il chunk stesso potrebbe idealmente convergere alla totalità

del documento se tale cap sulle dimensioni idealmente non ci fosse. Il perché di questo è relativo al fatto che usare una media degli embeddings delle finestre accumulate fino a quel momento è meno oneroso che valutare un ricalcolo intero: porzioni testuali più piccole, meno calcoli, più veloce; questa scelta è proprio relativa al fatto che la principale inefficienza dell'approccio di Semantic Chunking è la complessità computazionale: il vantaggio che, intutivamente, la definizione di unità semanticamente coese porta al retrieval non sempre è giustificato dalle valutazioni sui test set o dall'incremento eccessivo che c'è del tempo di calcolo e delle risorse impiegate, il che rende proibitivi approcci di chunking non rule - based, come anche quelli che sfruttano LLM, su corpora molto grandi [Qu et al., 2025].

La strategia di Semantic Chunking che ho costruito introduce due accorgimenti per renderlo più stabile:

- pre-processing delle frasi
- soglie adattive per il topic shift

Dopo aver segmentato il testo in frasi e averne stimato la lunghezza in token, le frasi "troppo corte" vengono unite con usando indicatori statistici: si calcola la mediana delle lunghezze e la MAD (Median Absolute Deviation), cioè la mediana delle deviazioni assolute da quest'utlima, unendo frasi la cui lunghezza è sotto la soglia 'mediana + MAD'; quest'ultimo è meno sensibile agli outlier rispetto alla deviazione standard, quindi è adatto quando le lunghezze hanno valori anomali e questo succede abbastanza spesso con testi che derivano, ad esempio, da OCR, oltre che lo split su frasi è basato sempre tramite regole euristiche comuni e non è perfetto, anzi, potrebbe generare frasi arbitrariamente corte; l'obiettivo è evitare che unità troppo brevi producano embedding e, di conseguenza, breakpoint rumorosi. Per il segnale semantico, invece di embeddare ogni frase isolata, si costruiscono finestre di 3 frasi (precedente–corrente–successiva) ed il confronto di similarità è tra queste: per richiamare i concetti di overlap, ciascuna finestra coincide con un intorno delle due frasi adiacenti data una centrale di riferimento; in parole semplici, quest'approccio semantico vuole far sì di evitare che il SentenceTransformer debba infierire il significato di una frase 'da solo', dandogli contesto. Per individuare i breakpoint non si adotta una soglia assoluta, ma una soglia adattiva ricavata dalla distribuzione delle distanze tra finestre successive nel documento; sostanzialmente, si calcola un percentile delle distanze (tipicamente tra 75° e 90°): il percentile è quel valore tale che una certa quota delle osservazioni (es. l'80%) risulta inferiore e, quindi, vengono marcati come candidati solo i punti in cui la distanza supera una soglia "alta", cioè le differenze più marcate rispetto alla media del testo e che più plausibilmente rappresentano i topic shift. In particolare, il percentile non è ovviamente scelto a priori ma relativamente al coefficiente di variazione $CV = std/mean$: quando le distanze sono omogenee (CV basso) la soglia viene alzata usando percentili maggiori, mentre quando c'è maggiore irregolarità (CV alto) la soglia si abbassa per intercettare più cambi di argomento. A livello del

testo di riferimento, questo equivale a dire che la procedura si adatta allo 'stile di scrittura': un documento con andamento uniforme, ai sensi di transizioni semantiche graduali tra concetti, tende a produrre distanze simili tra finestre consecutive e, quindi, conviene "tagliare" solo sulle differenze davvero marcate poiché complessivamente c'è molta omogeneità nel discorso; al contrario, in testi più discontinui abbassare la soglia permette di separare meglio i topic shift, evitando di considerare due sub topics un chunk unico. I breakpoint vengono poi filtrati imponendo un numero minimo di frasi per chunk e stimando la lunghezza in token dei segmenti finali; sebbene si facciano molte operazioni, in realtà il bottleneck prestazionale non è relativo agli indicatori statistici, bensì al calcolo stesso degli embeddings e computazionalmente puoi considerarla equivalente al sentence-by-sentence tradizionale.

4.3 Considerazioni su SQuAD ed NQ Dataset

Il dataset SQuAD (Stanford Question Answering Dataset) è un benchmark per *machine reading comprehension* in cui, dato un brano, il modello deve rispondere a una domanda estraendo uno span testuale (cioè una sottostringa) dal contesto; nasce da articoli di Wikipedia selezionati e poi annotati con coppie question-answer create tramite crowdsourcing, con risposte relative a porzioni precise, appunto sottostringhe, del testo di riferimento. SQuAD è stato usato come base per misurare statistiche di retrieval e confrontare l'impatto delle due diverse strategie di chunking principe di questo lavoro: fixed e semantic chuning. In SQuAD hai diverse domande associate alla stessa pagina di Wikipedia e gli esempi vengono costruiti raggruppando tutto il 'contesto' per ogni domanda, difatto ricostruendo indirettamente la pagina completa e valutando la strategia di chunking sul complesso: non ha senso valutarla solo sulle singole porzioni testuali ovviamente. Data la natura del dominio, quindi testi enciclopedici, le query mantengono una certa sovrapposizione lessicale con il contesto (nomi propri, termini chiave, date), pur includendo parafrasi occasionalmente.

Un secondo dataset considerato è Natural Questions (NQ), che differisce da SQuAD prevalentemente per la natura delle query. Le domande in NQ derivano da interrogazioni "realistiche" effettuate su motori di ricerca, formulate da utenti che non conoscono a priori il contenuto del documento target, differentemente da quanto avviene in SQuAD; di conseguenza, le query sono meno ancorate al testo e la sovrapposizione lessicale è mediamente più debole rispetto a SQuAD, caso in cui dovrebbe eccellere il vantaggio della ricerca densa. In questo senso, NQ rappresenta un benchmark più fedele a scenari di retrieval 'reali', dove l'allineamento query–passage non è garantito da annotatori che si concentrano, proprio, sui singoli paragrafi; se volessimo, in poche parole, riassumere quanto abbiamo detto: in maniera informale NQ è uno SQuAD che soffre maggiormente del problema di *vocabulary mismatch*, rendendo la fase di chunking potenzialmente più critica per non spezzare contesti necessari ad una ricerca semantica più complessa.

Nei nostri esperimenti su SQuAD (1000 documenti, 5825 query, stime realistiche), le prestazioni risultano sostanzialmente sature già a $K = 2$ (Hit@2 = 1.0 per entrambi i metodi), con un Hit@1 molto elevato e differenze marginali tra le due strategie (0.9888 per il semantic vs 0.9906 per il fixed). Come mostrato in Tabella 1, le lunghezze dei chunk prodotti da entrambe le strategie sono praticamente identiche e non emerge un vincitore chiaro trai due approcci: i ~ 150 token non sono stati scelti casualmente, il semantic chunking ricava la lunghezza dei chunk in maniera adattiva, come spiegato nella sezione precedente, e possiamo sfruttare la lunghezza media dei chunk prodotti dalla strategia semantica come window size per la fixed, rendendo perfettamente comparabili i risultati tra le due. In un dominio come questo, la scelta tra *fixed* e *semantic chunking* incide poco sull'efficacia del retrieval. Spostando la valutazione su NQ

Tabella 1: Risultati di Retrieval su SQuAD (1000 Documenti, 5825 Query). Modello: `baai/bge-large-en-v1.5`.

Metrica	**Semantic Chunking**	**Fixed Token Chunking**
Hit@1	0.9888	**0.9906**
Hit@2	**1.0000**	**1.0000**
Hit@3	1.0000	1.0000
Hit@4	1.0000	1.0000
Hit@5	1.0000	1.0000

(sempre 1000 documenti, 5000 query, rappresentativo), il task diventa più discriminante, con un calo generale delle performance (Hit@1 ~ 0.88). Tuttavia, anche in questo scenario più complesso, non emerge un vantaggio del *semantic chunking*: il metodo *fixed* risulta leggermente superiore per tutti i K considerati (Tabella 3) e, nel complesso, i risultati empirici rafforzano il trade-off discusso in [Qu et al., 2025]: il *semantic chunking* introduce un overhead computazionale non trascurabile, dovuto principalmente al calcolo degli embedding e alla stima delle soglie di breakpoint, senza garantire miglioramenti evidenti in fase di retrieval puro; ovviamente sia NQ che SQuAD sono dataset semplici, basti guardare le metriche, tuttavia questi test rafforzano evidenze sperimentali che ci sono in lavori molto famosi come quelli già citati e permettono, difatto, di confermare le interessanti considerazioni che vengono fatte: d'altronde, se non c'è un delta chiaro in task molto semplici, allora non vuol dire che non possa esserci un vantaggio più marcato in task di retrieval più complessi ma risulterà sempre ragionevolmente diluito ed è difficile che possa sovrastare un aumento così importante delle risorse di calcolo nel caso medio.
Comunque, il dominio Wikpedia è stato scelto proprio per amplificare i concetti precedenti legati ai topic shift, estendendo con NQ un ulteriore livello di complessità, evidenziando come il beneficio al retrival di preservare i confini semantici non è sempre automatico. Richiamando al concetto precedente di **gold-standard**: una strategia a finestra fissa (*fixed chunking*), sebbene più rudimentale, si dimostra estremamente competitiva nel tradeoff performance-costo

Tabella 2: Risultati di Retrieval su Natural Questions (1000 Documenti, 5000 Query). Modello: `baai/bge-large-en-v1.5`.

Metrica	**Semantic Chunking**	**Fixed Token Chunking**
Hit@1	0.8768	**0.8816**
Hit@2	0.9728	0.9768
Hit@3	0.9860	0.9878
Hit@4	0.9904	0.9918
Hit@5	0.9924	0.9938

e nel caso in cui tu non abbia una perfetta conoscenza di, ad esempio, distribuzione lessicale del caso, a mio avviso, l'adozione ad approcci come il Recurisve Chunking sono la strada più plausibile.

Cambio di setting. Come indicato precedentemente, per garantire la comparabilità tra metodi si allineavano la dimensione dei chunk *Fixed* a quella prodotta dal metodo semantico. Per evidenziare meglio i concetti precedenti, abbiamo esteso ad un'impostazione diversa: il *Fixed Chunking* viene vincolato a una finestra *rigida* e predefinita ($W = 256$); sostanzialmente, non sfruttiamo più l'informazione derivante dal semantic chunking per calibrare la fixed window size. L'obiettivo diventa valutare la capacità del metodo semantico di adattare la segmentazione al contenuto rispetto a una baseline con budget di contesto fissato.

I risultati in Tabella 3 mostrano che, in questo setting, il *Semantic Chunking* mantiene un vantaggio consistente rispetto al *Fixed Chunking* su tutte le metriche considerate. Il principale fattore che spiega il vantaggio del metodo semantico è la sua capacità di produrre chunk che seguono i confini informativi del testo, adattando la grandezza in base dinamicità a livello di topic del documento, indirettamente rispetto alla sua struttura. Nel caso di NQ, dove abbiamo evitato SQuAD poiché più semplice ed i risultati sarebbero stati troppo simili a prima, ciò si traduce in un contesto medio recuperato più ampio (circa 406 token, rappresentativo), che tende a preservare unità tematiche più coese fino a che la semantica stessa lo permetta; al contrario, la finestra fissa del *Fixed Chunking* ($W = 256$) forza una frammentazione molto più aggressiva, che può spezzare porzioni di testo necessarie per allineare correttamente query e contenuto, riducendo le performance del retrieval. Nel complesso, questi nuovi risultati non smentiscono le considerazioni precedenti, anzi, le estendono: si rimane perfettamente coerenti con le osservazioni di [Qu et al., 2025], sottolineando come il beneficio del chunking semantico non è universale ma tende a emergere quando il testo è particolarmente ricco di tematiche distinte ed un chunking fisso perderebbe eccessivamente in coesione, tuttavia nel caso medio il gain prestazionale sul retrieval, benché in alcuni casi può assestarsi attorno ad un $\approx 10\%$ in F1-Score come sui lavori citati, non necessariamente cambia il bilanciamento

Tabella 3: Risultati su Natural Questions (1000 Doc, 5000 Query).

Metrica	Semantic Chunking	Fixed Chunking (W=256)
Hit@1	**0.8768**	0.8706
Hit@2	**0.9728**	0.9680
Hit@5	**0.9924**	0.9902

negativo al tradeoff performance-costo del chunking semantico. È interessante notare come il chunking fisso, sfruttando un'adattività della finestra, possa raggiungere performance comparabili in termine di auto-esplicabilità dei chunk: in questo lavoro ho valutato anche una strategia di chunking a finestra fissa ma dove la dimensione fosse calcolata rispetto alla stessa idea di indicatori statistici del semantic chunking proposto, tuttavia, a mio avviso, molta di questa dinamicità viene comunque garantita strutturalmente da approcci come il Recursive Chunking che nel complesso hanno maggiori benefici e con meno 'calcoli' da valutare, da cui si rafforza l'idea precedente di **gold-standard**.

5 Architetture RAG

Abbiamo già introddo l'idea fondamental del RAG: paradigma in cui la generazione di una risposta non dipende esclusivamente dalla conoscenza "interna" del modello ma viene condizionata da contenuti recuperati da una fase preliminare di retrieval; l'LLM si occupa di interpretare e sintetizzare l'informazione, non fornirla direttamente. Grossolanamente, un sistema RAG può essere visto come una pipeline composta da due macro-fasi concatenate: *retrieval* e *generazione* appunto; quando si parla di **Architetture RAG** si fa riferimento a come implementativamente decidi di eseguirle, con particolare enfasi soprattutto sul retrieval, nient'altro.

A partire da questo schema essenziale, gli elementi fondamentali che ricorrono in qualunque architettura RAG sono:

- **Knowledge Base ed Indicizzazione**: come i documenti vengono acquisiti, normalizzati e resi ricercabili; in questa fase rientrano chunking, tokenizzazione, stopping, stemming, pattern matching, piuttosto che la progettazione anche di complessi modelli di OCR
- **Embeddings e Similarità**: non tanto la ricerca vettoriale in sé, ad oggi gli algoritmi sono sempre gli stessi e della famiglia ANN, più che altro progettualmente si spinge sul modello che usi, fine tuning...
- **Ranking e Selezione**: decidere quali chunk passare al modello ed in che ordine; il Ranking è un concetto fondamentale e che si capisce molto bene con i motori di ricerca, oggi acquisisce anche nella RAG un'importanza particolare e che discuteremo successivamente

- **Costruzione del prompt e budget di contesto**: il contesto utile è limitato (token budget), quindi progettualmente devi decidere quanti e quali chunk passare come contesto al modello non solo a livello di Ranking, ma anche quanto puoi permetterti sopratutto a livello di costi
- **Grounding e verificabilità:** l'obiettivo pratico di un RAG non è solo "rispondere", ma rispondere a partire da evidenze; la progettazione del flusso deve quindi ridurre l'ambiguità che hai su 'dove' il modello prenda l'informazione che restituisce

Successivamente si presentano le architetture RAG adottate e sperimentate nel lavoro, seguendo un percorso progressivo: si parte dal paradigma più diretto e si introducono poi varianti che modificano, prevalentemente, dove e come avviene il retrieval e l'organizzazione della conoscenza. Le sezioni successive descrivono nel dettaglio le scelte architetturali per ciascun approccio, evidenziando componenti coinvolte, flussi di esecuzione e implicazioni pratiche, con un focus sui vantaggi/svantaggi principali.

5.1 Naive RAG

La variante *Naive RAG* rappresenta l'implementazione più diretta e "minimale" di questo paradigma: dato un input testuale, il sistema effettua una ricerca su uno o più indici, seleziona i k risultati più rilevanti e li inserisce (spesso tramite semplice concatenazione) nel contesto del prompt fornito al modello; questa è la stessa descrizione che abbiamo fornito nelle sezioni precedenti. Questa impostazione è estremamente diffusa ed emblematica del concetto fondamentale: l'LLM non deve "sapere" tutto a priori, ma può sfruttare una base di conoscenza esterna aggiornabile e senza ricorrere a training successivi. Tuttavia, la semplicità del Naive RAG è anche la sua principale sorgente di inefficienze: piccoli errori nelle fasi a monte (in particolare chunking e ranking) tendono a propagarsi fino alla generazione ed è spesso difficile isolare i rapporti causa - effetto; in realtà, questo è vero in tutte le Architetture RAG, quindi è più corretto attribuire il 'bottleneck' prestazionale di questa alternativa al fatto che utilizzi una Hybrid Search tradizionale: non ricorri a meccanismi Adattivi/Gerarchici e sei, a parità di bontà del chunking e ranking, esposto ai problemi su Hubness e Dimensionality discussi precedentemente.

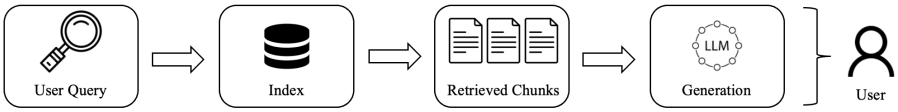

Figura 1: Schematizzazione Architettura Naive RAG

Sopratutto nella letteratura di oggi, dove sul RAG si fa ancora moltissima ricerca, l'approccio Naive è visto abbastanza 'male': questa è una tendenza relativa al pensare che soluzioni semplici spesso non siano robuste alla variabilità del

caso reale, in realtà, sopratutto quando si cerca un gold - standard, il Naive RAG è spesso una baseline molto difficile da battere e se le dimensioni del tuo corpora sono abbastanza contenute, probabilmente è l'Architettura RAG più vicina al concetto di 'soluzione to - go' e molto di quello che discuteremo nei risultati fa capolino a questo concetto qui.

5.1.1 Azure SDK

Nel lavoro svolto, il Naive RAG è stato costruito sfruttando sia l'ecosistema Azure che la libreria LangChain; la necessità di questo viene dalla volontà di mostrare due aspetti:

- quanto sia semplice, sopratutto sfruttando SDK di grossi provider come Microsoft, costruire un RAG funzionante
- quanto, però, sei fortemente vincolato a scelte spesso subottimali su tutti i punti progettuali più critici

Con Azure, una pipeline Naive RAG può essere implementata in modo relativamente semplice utilizzando Azure AI Search (ex Cognitive Search) come motore di retrieval e Azure Blob Storage come database; concettualmente infatti, quando vuoi sfruttare SDK di provider del genere, hai a disposizione librerie che ti permettono di concatenare prodotti appartenenti a diversi servizi per, difatto, realizzare le singole funzionalità che cerchi. È importante, quindi, discutere di come Azure 'chiami' le componenti fondamentali di una pipeline RAG, Naive o meno in realtà:

- **Blob Storage (Data Source)**: rappresenta l'origine dei dati. I file (PDF, DOCX, HTML, testi, ecc.) vengono caricati in un container. Azure AI Search può essere configurato per leggere direttamente dal container tramite una *data source* che descrive endpoint, credenziali di accesso e pattern di acquisizione. La fase di estrazione del testo, cosiddetta *cracking*, viene gestita automaticamente ed è possibile anche utilizzare modelli di Form Recognizing se la sorgente dei dati ha un template standard: in questa tesi non discutiamo dell'importanza dell'OCR nel RAG, quindi i risultati vengono ricavati da testo difatto già estratto, tuttavia è una componente tanto importante quanto il retrieval e, spesso, il grosso vantaggio dei provider come Microsoft non è dato dai Language Model, ma proprio dai modelli di OCR che offrono
- **Index**: definisce lo schema dell'indice di ricerca. In esso si specificano:
  - i campi testuali (contenuto del chunk, titolo, percorso, ecc.)
  - i campi di metadati (data, autore, categoria, ACL, id documento, ecc.)
  - il campo vettoriale (array numerico) per la ricerca semantica via embeddings

  - le impostazioni di ricerca (filtri, faccette, analyzer, e opzioni per ricerca ibrida/semantica)

  In un RAG che si rispetti, l'index è spesso ottimizzato per restituire rapidamente chunk che derivano dalla strategia adottata, con metadati utili a tracciare la provenienza

- **Skillset**: è il componente che descrive la catena di *enrichment* applicata ai documenti durante l'ingestion. Sostanzialmente, è un grafo di "skill" che trasformano l'input in output indicizzabile; per 'skill' si intende una particolare trasformazione sul testo estratto: sebbene sembri complesso, non è diverso concettualmente dalla tokenizzazione, stopping o stemming classica che si fà nei motori di ricerca, indipendentemente dal RAG o meno. In un caso come il nostro, le skill più rilevanti sono:

  - *Parsing/estrazione*: per ottenere testo e struttura dal file;
  - *Split/Chunking*: per segmentare il testo in unità più piccole;
  - *Embedding*: per calcolare il vettore associato a ciascun chunk (in integrazione con un endpoint di embedding).
  - *Entity Recognition*: ricavare entità rilevanti del mondo reale, come del dominio di riferimento, spesso per supportare retrieval che si basi su un Knowledge Graph

  Insomma, lo skillset prende il testo e costruisce l'oggetto che può essere indicizzato

- **Indexer**: è il "motore" che collega data source, skillset e index. Il funzionamento può essere riassunto macroscopicamente come segue:

  - legge i documenti dalla data source (Blob)
  - invoca gli skillset per enrichment
  - mappa le informazioni estratte ai campi indicati nell'indice, ovvero quelli su cui avverrà la ricerca all'atto pratico
  - scrive i risultati nell'index
  - può essere schedulato o eseguito on-demand

Il funzionamento è lo stesso, a grandi linee, di quello già discusso, indipendentemente dal fatto che utilizzi l'SDK di Azure, piuttosto che AWS, Google o che tu faccia 'manualmente' la stragrande maggioranza delle operazioni con LangChain; ed il punto è proprio qui: implementativamente, le SDK di provider consentono di orchestrare tutto il ciclo (creazione risorse, aggiornamenti, run dell'indexer, query) con un livello di astrazione molto alto che facilita e velocizza la costruzione del prodotto, nonché il mantenimento; d'altronde, nonostante si parli di AI per la maggiore ad oggi, i vantaggi del Cloud Computing sono esattamente gli stessi sulla messa in esercizio del prodotto che costruisci.

Il vantaggio delle SDK diventa, in progetti di ricerca come il mio ma anche in applicazioni domain - specific, abbastanza irrisorio ai sensi del fatto che: le SDK ti permettono di costruire semplicemente pipeline di RAG che sono già state pensate da chi ha costruito le SDK e diventa più tedioso che costruire tutto manualmente andare a definire strategie di chunking personalizzate, come algoritmi o modelli di ranking; difatto richiamiamo il concetto precedente: il vero vantaggio dei cloud providers nel RAG è dato solo ed esclusivamente dall'availability del servizio, non dalle performance che il tuo prodotto raggiunge o dalla semplicità di implementazione.

La pipeline RAG di Naive "gestita" su Azure implementa tutte le componenti precedentemente discusse: i documenti nel Blob container vengono indicizzati da un Indexer che esegue il cracking ed applica uno Skillset comprendente Fixed Chunking e calcolo degli embedding con `text-embedding-ada-002`, uno dei modelli di Microsoft, proiettando poi il chunk ottenuto sui fields che definiscono l'index. A runtime il retrieval è hybrid (BM25 + ricerca densa ANN sul campo vettoriale) e i top chunk vengono passati al modello con system prompt vincolante l'utilizzo unicamente di informazione proveniente dai contesti. Come suddetto, la stessa pipeline è stata emulata con LangChain, dove chunking e trasformazioni sono completamente personalizzabili per pipeline complessivamente più sofisticate.

Considerazioni

La pipeline Azure descritta si colloca, quindi, come una soluzione efficace per avviare rapidamente un sistema RAG e gestire gli aspetti infrastrutturali (storage, indicizzazione, orchestrazione). Come suddetto, proprio perché costruita in maniera "guidata", ha abbastanza limitazioni pratiche, che ora descriviamo meglio, quando si richiedono interventi su componenti fondamentali della pipeline. In particolare:

- **Rigidità del pre-processing**: molte scelte cruciali (normalizzazione del testo, gestione di tabelle, preservazione della struttura, deduplicazione) risultano vincolate alle skill fornite dalla piattaforma; quando emergono esigenze più specifiche, la complessità implementativa aumenta sensibilmente poiché diventa necessario introdurre componenti esterni (ad es. funzioni o servizi su endpoint dedicati) e orchestrarli nella pipeline

- **Chunking**: è relativamente semplice modificare dimensione del chunk o overlap entro i limiti delle SDK fornite; è invece complesso implementare strategie avanzate (chunking gerarchico, semantico, recursive...): la maggioranza dei provider incentiva un chunking "standard", mentre strategie più sofisticate richiedono generalmente la definizione di funzioni esterne e la costruzione di endpoint REST da chiamare all'occorrenza; in Azure, ad esempio, l'implementazione di strategie di chunking diverse dalla Fixed richiede generalmente la costruzione di una *Custom Skills* all'interno di uno Skillset Azure AI Search: in questi casi, è necessario creare una *Azure*

Function che ospiti la logica di chunking personalizzata (utilizzando magari librerie come LangChain o modelli di embedding di Azure OpenAI) e registrarla come un Web Api Skill, richiamando al concetto precedente. Per il Fixed la SDK astrae tutto questo; proprio in questo senso discutevamo prima di come, per applicazioni reali, usare o non usare SDK ha più o meno lo stesso livello di complessità: ovviamente non vuol dire che tu non possa appoggiarti ad un provider, semplicemente che gli strumenti messi a disposizione si concentrano prevalentemente su RAG Naive e non coprono la complessità reale delle esigenze

- **Reranking**: anche l'integrazione di un reranker introduce spesso vincoli pratici; con le SDK offerte dai provider si è tipicamente limitati ai modelli esposti dallo, mentre l'adozione di un reranker custom richiede orchestrazione esterna (servizio/API), con impatti su latenza, costi e complessità più in generale

Provider come Azure rendono agevole implementare una pipeline RAG end-to-end, ma lo fanno privilegiando un set di scelte standardizzate. Questo è un vantaggio in termini di *time-to-first-result*, ma può diventare un vincolo quando l'obiettivo si sposta all'avere un sistema ottimizzato sul dominio di riferimento.

Reranker Per i *reranker* facciamo un discorso a parte e che serve ad inquadrare la scelta di fondo che è stata fatta in questo lavoro: non considerarli. Il *reranker* è un componente che, dopo una prima fase di retrieval (ad esempio top-k via BM25, vettoriale o ibrida), ricalcola l'ordine di rilevanza dei risultati con un modello Transformer, in realtà, concettualmente simile ai SentenceTransformer: un reranker valuta coppie *query–documento* con modelli cross-encoder producendo un punteggio di pertinenza più accurato; in realtà, per la maggiore, la coppia viene unita in un unico testo e si sfruttano modelli con attention bidirezionale come gli encoder only, la vera particolarità sta nel fatto che si produca uno score di rilevanza tra la query ed il documento ed il concetto è lo stesso delle Contrastive Loss, ovvero insegnare al modello a separare ciò che è rilevante da ciò che non lo sia; approcci moderni, ma come per i SentenceTransformer, estendono l'idea anche a modelli LLM-based decoder only.

A mio avviso, il reranker è spesso *sopravvalutato* nel contesto generale di RAG: in molti casi migliora metriche di ranking e questo è un vantaggio, tuttavia il gain prestazionale non è per forza giustificato dall'introdurre un secondo modello a valle del retrieval piuttosto che agire sul SentenceTransformer con, magari, un tuning più allineato alle esigenze; se i chunk sono mal costruiti d'altronde, un reranker può al massimo scegliere "il meno peggio" tra candidati già sub-ottimali. L'inefficienza principale è, però, la stessa che in contesti reali ti vincola sull'embedder: con un reranker introduci ulteriore latenza che, se puoi spendere, ha sempre più senso farlo sul SentenceTransformer poiché chunk più rilevanti di base producono un ranking strutturalmente migliore. In altre parole, il reranker può essere utile come rifinitura se i vincoli lo permettono, ma

raramente dovrebbe essere il primo strumento a cui ricorrere per "aggiustare" un RAG con retrieval debole; proprio in questo senso, nel mio lavoro ho deciso di non considerare reranker.

Queste considerazioni sono supportate da lavori recenti che osservano che il beneficio del reranking non cresce in modo monotono quando il "primo stadio" è già forte. Ad esempio, [Chen et al., 2025b] mostrano esplicitamente diminishing returns: passando a first-stage retrievers più efficaci, la percentuale di miglioramento ottenuta dal reranker tende a ridursi. In maniera molto simile, [Jacob et al., 2024] mettono in discussione l'assunzione "reranker sempre meglio", evidenziando che, quando si scala il numero di documenti da valutare, i reranker possono dare miglioramenti iniziali ma poi calare e perfino degradare la qualità oltre una certa soglia.

5.2 Hierarchical RAG

L'approccio *Hierarchical* estende il paradigma RAG introducendo una struttura di indicizzazione e recupero *multi-livello*. Quando abbiamo parlato di chunking avevamo discusso di come questi siano, sostanzialmente, viste locali del documento e, molto spesso, soprattutto in testi come romanzi dove si ha una narrazione e collegamenti anche tra porzioni di testo spazialmente lontane, si perda strutturalmente questa proprietà di interazione; difatti, uno dei domini dove approcci di RAG Naive performano peggio è proprio quello delle cosiddette domande MultiHop: queries che possono essere soddisfatte solo collegando informazione che appartiene a porzioni molto distanti di uno stesso documento o, addirittura, di documenti diversi. L'informazione utile non è, quindi, uniformemente distribuita e né sempre recuperabile tramite chunk locali: un indice gerarchico consente di rappresentare lo stesso contenuto a diversi livelli di astrazione ed ha una fortissima interazione con i concetti classici della teoria dei grafi, come le *Communities*.

Gli approcci gerarchici moderni si basano, infatti, sempre sull'interazione tra lo stesso set di componenti:

- chunk testuali come **unità di base**
- **cluster o communities** per aggregare semanticamente porzioni correlate del testo
- **riassunti a diversi livelli** questi fungono concettualmente come estensione al testo di *rollup* su dati strutturati, con l'obiettivo quindi di rappresentare la tematica fondamentale del Cluster/Community e guidare la query verso una porzione dello spazio degli embedding che comprenda un numero minore di documenti rilevanti

Già da qui si capisce un concetto importante in relazione ai problemi fondamentali del RAG: l'approccio gerarchico ha il grosso vantaggio di mitigare strutturalmente il problema del *semantic collapse* nella ricerca, proprio perché i riassunti

ti permettono di instradare la query facendo riferimento ad un set di chunk molto più ridotto di quello a basso livello; ad esempio, se ogni Community è compresa mediamente da 10 Chunk, allora hai ridotto la dimensione di ricerca di un fattore 10: valuti la query sui riassunti di Community e, poi, prendi come corpora solamente l'insieme dei chunk delle top communities ad esempio, approccio semplice e molto diffuso. Concatenando operazioni che coincidono, come accennato prima, a *rollup* e *drilldown*, garantisci non solo di ridurre la dimensione del corpora su cui valuti la query, ma anche a priori interazioni tra chunk distanti spazialmente o cross - documento.

A livello operativo, una pipeline Hierarchical RAG separa due problemi:

1. **Costruzione della gerarchia** (*index-time*): oltre alla classica indicizzazione di chunk, si costruiscono nodi "aggregati" scegliendo la strategia di rollup (spesso summary based)

2. **Recupero coarse-to-fine** (*query-time*): invece di selezionare unicamente top-k chunk locali, la query viene dapprima valutata sulla dimensione rolled - up si per massimizzare copertura e contesto, poi si scende in drilldown per massimizzare la precisione

Come suddetto, questo approccio è particolarmente performante in tutti quei casi in cui:

- la domanda richiede **sintesi** o **reasoning multi-hop** single o cross document

- la collezione è molto grande e si vuole mitigare il *semantic collapse*

Ovviamente, molto della bontà di questo approccio si riduce al come venga progettata soprattutto la fase di rollup: se si procede summary - based, come tipicamente avviene, riassunti rumorosi limitano la massima precisione auspicabile dal drilldown successivo.

5.2.1 RAPTOR RAG

RAPTOR (*Recursive Abstractive Processing for Tree-Organized Retrieval*) è un'implementazione di Hierarchical RAG in cui la gerarchia assume la forma di un **albero di riassunti** costruito in modo ricorsivo; la particolarità rispetto ad altri approcci, come quello che presenteremo successivamente, sta proprio nell'utilizzo di una struttura Tree based piuttosto che un grafo generico, per il resto i concetti fondamentali discussi precedentemente sono tutti perfettamente allineati con l'idea dell'approccio. L'obiettivo è sempre lo stesso: superare il limite tipico del Naive RAG nel retrieval di soli chunk contigui (piatti), spesso non catturando il contesto necessario per domande che richiedono ragionamento su parti spazialmente distanti o multidocumento.

Costruzione dell'albero (index-time) Il processo può essere descritto come una pipeline bottom-up:

1. **Segmentazione iniziale**: il documento viene suddiviso in chunk (foglie dell'albero) e per ciascuno si calcola un embedding

2. **Clustering semantico**: i chunk (o nodi di un livello) vengono raggruppati in cluster basati sulla similarità degli embedding, con l'obiettivo di aggregare contenuti concettualmente affini; questo approccio appartiene alla famiglia di strategie RAG che utilizza direttamente algoritmi di Clustering e non Community - based, ma concettualmente l'idea rimane la stessa ed è unicamente una scelta progettuale

3. **Riassunto astrattivo per cluster**: per ogni cluster si genera un *summary* tramite un Language Model; questo summary diventa un nodo "padre" che rappresenta la semantica di gruppo, ovvero la dimensione rolled - up. In questo caso, RAPTOR pubblica esplicitamente i system prompt utilizzati per generare i riassunti: sfruttano template molto semplici, a mio avviso necessiterebbe di maggior prompt engineering per una fase così critica

4. **Ricorsione**: i nodi-padre vengono a loro volta embedded e (ri-)clusterizzati per costruire livelli superiori, fino a ottenere un numero ridotto di nodi ad alta astrazione

In quest'ultimo caso, è importante sottolineare che l'operazione di rollup comunque ha associato un certo grado di information loss: combinandole tra loro guadagni in mitigazione di tutte le problematiche precedenti, tuttavia potresti arrivare, oltre una certa soglia, a performare analogamente (se non peggio) a RAG Naive. La difficoltà progettuale maggiore di questo approccio sta, proprio, nel rapporto trai summaries ed il livello di astrazione target.

Recupero e generazione (query-time) A query time, RAPTOR sfrutta l'albero di rappresentazioni; puoi decidere di fare diverse cose e, addirittura, rendere ancor più sofisticato il retrieval stesso: le query non sono tutte uguali, alcune potrebbero essere risposte da una semplice full - text search, altre trovano grosso vantaggio nella ricerca semantica, quindi va bene la struttura gerarchica ma non è tassativo utilizzarla. Tipicamente, l'approccio più comune vuole partire dalla dimensione più ad alto livello e scendere nell'albero ad ogni passo, sempre valutando i *top - k* nodi per metrica di similarità, come la coseno, fino ad ottenere un parco di foglie (chunk effettivi) come base documentale ristretta su cui valutare la query; nulla ti vieta, però, di partire da livelli intermedi di astrazione e, quindi, di base basarti su un grado di information loss minore in quello che è, fondamentalmente, un meccanismo di routing della query in ingresso. In entrambi gli scenari, la gerarchia permette una composizione del contesto in cui i livelli alti danno direttamente informazione su dove e cosa potresti andare a trovare scendendo a livello più basso e collegamenti tra concetti, mentre i livelli

profondi forniscono l'information need stesso; ovviamente, questi vantaggi non sono gratis: il tempo di risposta del sistema aumenta proporzionalmente alla complessità dell'albero, tuttavia, se si legge tra le righe, non abbiamo discusso di reranker o modelli che ancor di più aumentano la latenza complessiva, quindi rispetto a Naive RAG commerciali che sfruttano strumenti come reranker l'approccio gerarchico ha tempi di risposta comparabili.

Una considerazione che, in questo caso, ha assolutamente senso fare e che, successivamente, discuteremo anche con i risultati è la seguente: approcci sofisticati come questo hanno sicuramente chiari vantaggi, ma che diventano evidenti quando puoi stimare la tipologia di query che, per la maggiore, ti troverai a dover rispondere a runtime. Le prestazioni di un motore di ricerca sono fortemente dipese da come l'utente stesso formula il suo bisogno, quindi dalla distribuzione lessicale della query: per ricerche full - text, infatti, se questa è fortemente disallineata rispetto alla distribuzione lessicale dei documenti il sistema performerà male ed è abbastanza chiaro; grosso vantaggio sta, infatti, nell'adozione di modelli generativi per query rewriting ed è una delle mitigazioni più forti al Vocabulary Mismatch Problem. Quando, però, hai query molto semplici e che possono essere risposte da sistemi di retrieval molto meno complessi di quello gerarchico, tipicamente vedi che: più è sofisticato il sistema che progetti e peggio performa nelle query semplici, che comunque sono sempre una sotto popolazione molto densa della totalità delle query in ingresso. Difatti, quello che rende il Naive RAG una baseline difficile da battere, se guardi sempre al concetto di go - to solution, sta proprio nel fatto che, con le dovute considerazioni, comunque riesce a soddisfare bene l'information need medio e la struttura gerarchica si perde nei casi semplici: per questo, adottare spesso classificatori sulla tipologia della query iniziale, indicanti la difficoltà della stessa, permette di poterla valutare direttamente sulla base documentale a livello più profondo della gerarchia, quella complessiva quindi, e senza ridurla dimensionalmente, proprio perché la parte full - text dello score ibrido è predominante.

5.2.2 Microsoft GraphRAG

Microsoft GraphRAG propone un'altra forma di Hierarchical RAG, in cui la gerarchia non è un albero di riassunti costruito solo da clustering, ma una struttura **a grafo** centrata su entità e relazioni estratte dal testo, quindi un qualcosa di strutturalmente più vicino all'approccio che andrò a proporre. L'ipotesi di fondo è sempre la stessa, tuttavia qui si fa riferimento ad uno dei concetti principe nel retrieval moderno e che, storicamente, ha fatto la fortuna di motori di ricerca come Google: i Knowledge Graphs.

Knowledge Graphs Un *Knowledge Graph* (KG) è una rappresentazione strutturata della conoscenza in cui le informazioni vengono modellate come un insieme di **entità** (nodi) e **relazioni** (archi) tra di esse. Formalmente, un KG può essere visto come un grafo etichettato che rispetta un determinato modo di costruire le suddette etichette: spesso tramite triple del tipo *(soggetto, predicato,*

oggetto), ma più generalmente si parla di **Ontologie** che potrebbero avere anche una forma diversa dalla precedente. Questo consente una transizione da un corpus puramente testuale a una descrizione esplicita di "chi è chi" e, sopratutto, "come le cose sono collegate", rendendo il contenuto non solo interrogabile a livello di similarità semantico-lessicale, ma soprattutto per connessioni logiche. Le *proprietà* rappresentano gli attributi e le relazioni che *caratterizzano* un'entità, intesa come un oggetto del mondo reale (persone, luoghi, organizzazioni, concetti); in un KG, esse non si limitano a descrivere l'entità, esprimono anche i legami tra le stesse, rendendo spesso molto più semplice soddisfare anche information need complessi.

Applicativamente, è facile ora capire come i Knowledge Graphs rispondano ad un'esigenza fondamentale del retrieval: introdurre un livello di rappresentazione più stabile del testo e che permetta di 'navigare' in maniera strutturata l'informazione, espandere la query aggiungendo contesti, dipendenze, cause-effetti, gerarchie e associazioni... anche quando tali collegamenti non emergono in modo evidente con un retrieval a chunk; infatti, puoi rispondere sì alla query navigando direttamente sul grafo se, come nel caso che presenteremo poi, questo è un grafo di chunk, tuttavia, come nel *query rewriting*, puoi anche sfruttarlo per risolvere l'ambiguità che tipicamente c'è in query reali a motori di ricerca, come quelle presenti nel dataset NQ discusso nei capitoli precedenti. Come già accennato, non tutti i vantaggi vengono 'gratis': i KG hanno, infatti, costi di gestione spesso e volentieri proibitivi, sopratutto quando il tuo dominio è particolarmente complesso e necessita la modellazione di tante entità e concetti distinti; i due processi fondamentali sono quelli di *Entity Extraction* ed *Entity Linking*: partendo da una base testuale, hai bisogno di un modo per automatizzare sia l'estrazione effettiva delle entities e delle proprietà, nonché il collegamento ad oggetti già esistenti nel KG corrente; si parla quindi di *disambiguazione* per l'appunto, oltre che di normalizzazioni all'ontologia dell'informazione che estrai. L'Entity Extraction e Linking sono due processi complicatissimi, forse trai più complicati nel retrieval: esistono modelli di NER, anche basati su Transformer o LLM, che ti permettono buone prestazioni sul tuo dominio, ma spesso e volentieri non sono competitivi in zero - shot; per capire davvero bene non tanto le entities, ma più cosa definisce una proprietà relativamente ad essa nel tuo dominio, hai bisogno di modelli fine - tuned allenati su tanti esempi etichettati di *testo* e *coppie estratte*, a meno che tu non voglia ottenere performance sempre subottimali. D'altronde, una ricerca che si fonda interamente sulla navigazione del KG ha prestazioni che dipendono criticamente da Entity Extraction e Linking: se non hai una buona F1 - Score e sei, ad esempio, sbilanciato sulla Recall piuttosto che sulla Precision, allora o rendi non ricercabile una data informazione contenuta nel testo o rendi ricercabile informazione rumorosa, incompleta o addirittura fuorviante. L'approccio costruito successivamente, infatti, si basa su un grafo di chunk e non di entities: se cerchi soluzioni go - to nel RAG e non ti basi su un dominio in analisi, allora fare NER o, più generalmente, popolare un KG perde totalmente di senso per i motivi suddetti.

Indexing e Communities La pipeline di indicizzazione di Microsoft GraphRAG può essere schematizzata in tre fasi operative:

1. **Costruzione del Knowledge Graph**: il corpus viene prima segmentato in *TextUnits* (chunk con eventuale overlap) che fungono da unità fondamentale di estrazione. Su ciascun TextUnit, un LLM esegue *Entity e Relationship Extraction* producendo un sottografo locale con: entità descritte da *title*, *type* e una o più descrizioni generate dal LM stesso sulla base del ground testuale, relazioni direzionali tra coppie di entità, ciascuna con una o più descrizioni testuali. I sottografi vengono poi fusi in un grafo globale con una deduplicazione deterministica: entità con lo stesso *title+type* vengono unificate (accumulando le descrizioni), e analogamente le relazioni con la stessa coppia (*source*, *target*) vengono aggregate. Per ridurre rumore e ridondanza, GraphRAG applica una fase LLM di *summarization* che comprime l'insieme delle descrizioni accumulate in una descrizione canonica per entità e per relazione, molto simile ai concetti precedenti di rollup

2. **Community detection e gerarchia multi-livello**: una volta ottenuto il grafo Entities/Relationships, GraphRAG esegue una partizione in *communities* tramite *Hierarchical Leiden*, uno degli algoritmi più famosi di Community Detection, ottenendo cluster di nodi densamente connessi e una gerarchia ricorsiva che rispetti un certo limite alle dimensioni; questa la parte Hierarchical dell'approccio Microsoft GraphRAG, d'altronde, nel lavoro che presenteremo successivamente, ci rifaremo proprio all'algoritmo di Leiden

3. **Community reports**: per ogni community e per ogni livello gerarchico, GraphRAG genera *community reports* e una versione ulteriormente compressa (*shorthand*) che sintetizzano concetti chiave di entità centrali, relazioni salienti o, più generalmente, la semantica della comunità. Parallelamente, vengono calcolati embedding per *TextUnits*, descrizioni di entità/relazioni e report, così da abilitare retrieval semantico sia sul testo non strutturato sia su ciò che ha costruito il Language Model

Modalità di interrogazione (query-time) GraphRAG distingue tipicamente due strategie complementari:

- **Local Search**: adatta a domande entity-centric o focalizzate. Si parte da entità rilevanti per la query e si recupera un intorno informativo nel grafo (nodi correlati, relazioni, descrizioni) combinandolo con i chunk testuali correlati per arricchire il contesto; questa è la parte più simile alla ricerca drilled - down precedentemente spiegata

- **Global Search**: adatta a domande di sintesi sul corpus. La risposta viene costruita in modo simile ad un approccio di RAG molto famoso: HyDE

(Hypotetical Document Embeddings), dove, invece di fare retrieval usando direttamente la query, l'LLM genera prima un "documento ipotetico" che risponderebbe alla domanda e si usa l'embedding di quel testo per cercare i documenti reali più simili; in Microsoft GraphRAG il concetto è molto simile: il LLM genera una risposta parziale (*key-points*) utilizzando i top - Communities summaries, ottenuti anche percorrendo diversi livelli della gerarchia, poi produce una risposta finale infierendo sulla query iniziale tramite i *key-points* estratti; si noti che, nel caso di Global Search, non si valuta drilldown perché ti interessano informazioni ad alto livello come presupposto

Anche se GraphRAG presenta principalmente Global e Local Search, la disponibilità di un KG e di una gerarchia di community permette naturalmente le strategie coarse-to-fine discusse precedentemente: selezione di community rilevanti, identificazione di entità/relazioni candidate e successivo drill-down verso TextUnits come evidenze, ottenendo di fatto una ricerca multi-step guidata dal grafo.

Microsoft GraphRAG è interessante poiché spinge su una costruzione LLM-centric dell'indice: il Language Model estrae entità e relazioni dai chunk e ne sintetizza le descrizioni, ottenendo un grafo "navigabile" e utile per retrieval; tuttavia questo design porta anche diversi problemi pratici. Oltre ciò che abbiamo discusso sul zero-shot, nella pipeline standard non emerge una vera e propria fase di *Entity Resolution* cross-document: il merge è principalmente deterministico (entità con stesso title+type vengono aggregate), quindi varianti nominali, sinonimi o ambiguità tra documenti possono non corrispondere alla stessa entità "reale", frammentando il grafo e incentivando duplicazioni nelle Communities, cosa che non necessariamente vuoi; questo è un problema non ancora risolto dagli autori e che ho potuto evidenziare studiando alcune delle issues aperte sulla repository GitHub del codice.

Tutte le informazioni fondamentali sono state prese direttamente dallo studio del codice pubblico fornito dagli autori e il paper di riferimento [Edge et al., 2024].

5.2.3 Semantic Chunk Graph

L'approccio di **Semantic Chunk Graph** proposto si ispira alle due alternative progettuali discusse precedentemente: RAPTOR e Microsoft GraphRAG. Da Microsoft GraphRAG riprende l'idea di sfruttare la struttura graph-based e di fornire una ricerca coarse-to-fine con rollup tramite communities (ottenute con clustering di tipo Leiden), così da soddisfare l'idea fondamentale alla base degli approcci gerarchici. Tuttavia, più similmente a RAPTOR, non introduce un livello esplicito di entità e relazioni estratte: i nodi del grafo sono direttamente i chunk testuali e gli archi rappresentano relazioni semantiche tra chunk (es. similarità coseno), piuttosto che relazioni spaziali di vicinanza nel documento; la motivazione di questo deriva dal discorso fatto precedentemente sul zero - shot

dei Language Models nella costruzione del grafo. Le strutture gerarchiche e i passaggi di raffinamento operano, quindi, sul testo segmentato e sulle sue aggregazioni, mantenendo il grounding sui contenuti originali. Riguardo la strategia di chunking: ho adottato, coerentemente a quanto discusso nei capitoli precedenti, la strategia che trovo concettualmente più vicina all'idea di soluzione go - to, ovvero il Recursive Chunking.

L'implementazione concreta del **Semantic Chunk Graph** si articola in tre fasi principali: costruzione e annotazione del grafo dei chunk a livello intra-documento, costruzione del grafo tra communities per abilitare interazioni cross-documento e gestione della query in ingresso con recupero gerarchico coarse-to-fine. La struttura fondamentale, come suddetto, è quella di un grafo in cui ogni nodo rappresenta un chunk testuale e gli archi codificano i due segnali distinti di: relazioni *semantiche* (`SIMILAR_TO`), basate su similarità tra embedding, e relazioni *strutturali* di adiacenza nel documento (`NEXT`); il grafo rimane, quindi, ancorato ai contenuti originali (nodi = testo segmentato) e consente di applicare algoritmi di analisi strutturale (PageRank e Community Detection) senza passare direttamente per l'estrazione di entità/relazioni.

Come discusso nel capitolo precedente sulle strategie di Chunking, un problema fondamentale risiede nel fatto che l'azione stessa di 'spezzare' il testo, assolutamente necessaria per il retrieval denso, non permette a molte porzioni di "interagire" direttamente, benché correlate magari: le relazioni modellate dagli archi `SIMILAR_TO` sono funzionali, proprio, a favorire l'interazione tra chunk semanticamente simili e, infatti, sono valutati tra tutti i chunk *dello stesso* documento; a questo aggiungiamo gli archi `NEXT`, che fungono da rincaro per modellare non solo un'attinenza a livello di significato del testo ma relazioni strutturali, con l'idea che `NEXT` non debba "sovrascrivere" la semantica, ma garantire una connettività minima e preservare la progressione del documento. Gli archi `SIMILAR_TO` sono pesati con la similarità coseno tra gli embeddings e, rispetto a questo valore, ho utilizzato una threshold per evitare di aggiungere archi con peso troppo basso e, quindi, collegamenti semanticamente deboli, mentre gli archi `NEXT` ricevono un peso costante più basso appunto perché il loro ruolo è più da segnale ausiliario ed il vero protagonista deve essere il segnale semantico. Possiamo riassumere l'intuizione fondamentale in:

- se due chunk sono semanticamente affini, l'arco `SIMILAR_TO` domina
- se un chunk è semanticamente distante, `NEXT` ne evita l'isolamento e permette agli algoritmi di Community Detection di identificare communities più coese e, complessivamente, meno *scattered* con, magari, gruppi banali formati da pochi chunk mal collegati

Su questa struttura viene dapprima calcolato il **PageRank**, con l'obiettivo di stimare l'importanza *relativa* dei chunk nel contesto del singolo documento: un chunk con PageRank elevato tende, infatti, a essere più "centrale" nel grafo delle interazioni semantiche e di adiacenza, quindi è utilizzato prevalentemente

come misura di rappresentatività per selezionare, all'atto pratico, quali chunk all'interno di ciascun gruppo utilizzare nelle fasi successive di sintesi e indicizzazione. Successivamente, viene eseguita la **community detection** tramite Leiden, algoritmo, come PageRank, discusso nell'introduzione a questo lavoro. Ogni community, in realtà, può essere interpretata come un **subtopic** del documento: la definizione formale di community può essere ridotta all'osso come insieme di elementi con alta densità di collegamenti interna e, modellando quest'ultimi relazioni di adiacenza/semantiche, diventano assumibili a tematiche locali del documento, costituenti l'unità di rollup su cui operare la ricerca gerarchica coarse-to-fine. La scelta di eseguire il clustering **per-documento** è intenzionale e deriva, proprio, dalla natura precedente di communties come sub - topics: far interagire porzioni locali cross - document potrebbe portare, per la maggiore, a relazioni semantiche fittizie, anche perché ricordiamo che utilizziamo modelli di embedding non fine-tuned ed il chunk è una vista locale del documento; ricavare gruppi coesi internamente allo stesso mitiga le precedenti relazioni fittizie, ai sensi del fatto che, nonostante ci possa essere anche una grandissima eterogeneità degli argomenti, il fatto che i chunk appartengano allo stesso documento comunque li rende appartenenti ad un'unica unità logica ed i collegamenti semantici sono più affidabili.

Una volta ottenute le communities intra-documento, il sistema sfrutta la metrica precedente di **PageRank** per produrre un livello di *rollup* testuale: per ogni community seleziona i chunk più "rappresentativi", ordinandoli proprio per `pagerank`, e li fornisce al modello per generare un breve summary del sub-topic rappresentante la community stessa. Il risultato viene:

- salvato in Neo4j come nodo `Community` e collegato ai chunk che compongono la community stessa, fornendo un livello di aggregazione generarchico
- indicizzato in Elasticsearch in un indice dedicato insieme al vettore embedding della sintesi, in maniera da garantire hybrid search su quest'ultimi

Queste scelte, appunto, abilitano proprio la componente "coarse" ricerca della ricerca sulle communities, prima di tornare al livello chunk. Oltre che la poca interazione intra-documento trai chunk, c'è il problema complementare riguardo la mancanza di interazione *cross-document*: con la struttura costruita finora, documenti distinti rimangono comunque separati; questo problema viene affrontato costruendo un livello successivo di **grafo tra communities** basato su similarità trai vettori dei summaries: per ciascuna community, si recupera il suo embedding dall'indice Elasticsearch e si esegue una ricerca kNN per ottenere candidati (sempre communities) vicini; tra questi, vengono mantenuti solo quelli che:

- appartengono a documenti diversi
- superano una soglia di similarità coseno, quindi una proiezione esatta del meccanismo dei `SIMILAR_TO` tra chunks

Per ogni coppia mantenuta, viene scritto un arco diretto `COMM_SIMILAR_TO` tra nodi `Community`, pesato sempre rispetto alla similarità coseno. L'uso di una soglia e di un limite ai vicini serve principalmente come efficientamento computazionale: va bene che le communities sono un livello di aggregazione e, quindi, con minore dimensionalità rispetto al chunk level, tuttavia valutare interazioni tra tutte le coppie di communities esplode, difatto, ad $O(N^2)$ comunque e dove N è il numero di chunk, essendo che le communities dipendono da un fattore di riduzione costante. L'obiettivo, appunto, non è connettere "tutto con tutto", ma costruire relazioni rilevanti tra communities tematicamente sovrapponibili in documenti differenti.

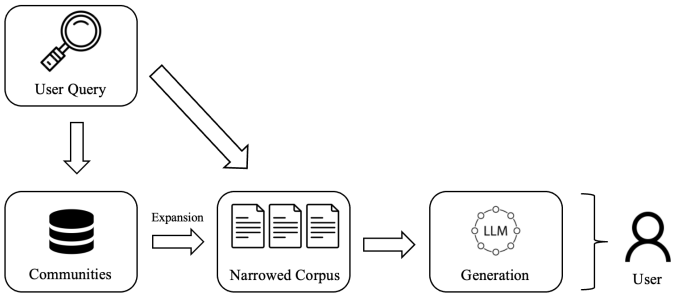

Figura 2: Schematizzazione Architettura GraphRAG

Alla luce di tutto questo, possiamo discutere meglio di come viene gestita la query in ingresso e capire completamente l'approccio:

1. **Seed Communities Selection (Coarse Retrieval)**
   La query viene confrontata con i summaries di community indicizzati tramite Hybrid Search ed il risultato è un insieme di *seed* che rappresentano i macro-temi più plausibilmente rilevanti.

2. **Espansione nel grafo delle communities**
   A partire dai seed, il sistema esplora archi `COMM_SIMILAR_TO` fino a un numero limitato di `hop` (impostato un tetto a 3, modificabile). Ho modellato, anche qui, una soglia minima sulla similarità di *ogni* arco attraversato durante l'espansione: un percorso viene considerato valido solo se tutte le relazioni `COMM_SIMILAR_TO` che lo compongono hanno uno score almeno pari alla threshold `min_sim`. L'idea è evitare di raggiungere communities lontane passando attraverso collegamenti deboli; d'altronde, tra tutti i percorsi si sceglie di introdurre solamente le nuove communities appartenenti ai $top - k$ percorsi per prodotto tra le similarità degli archi selezionati, ricavando una metrica che ne quantifica la bontà

3. **Retrieval Fine-Grained** L'unione tra seed ed espanse communities produce, ricavando i chunk che le compongono, il set su cui viene valutata la ricerca Fine-Grained; la ricerca finale, appunto, avviene **solo** su questo sottoinsieme tramite un retrieval ibrido su Elasticsearch

4. **Query Answering** I chunk top-k, sempre per hybrid score ovviamente, vengono recuperati con metadati (doc, posizione, testo) ed inviati come contesto al LLM; questo riceve sempre lo stesso prompt degli altri approcci sulla generazione della risposta, imponendo: uso del solo contesto, ammissione esplicita di insufficienza informativa, e risposta breve (massimo 3 frasi). Questo preserva il grounding e riduce la tendenza ad introdurre conoscenza esterna, come già discusso poi

Considerazioni

L'architettura graph based descritta introduce un livello gerarchico esplicito che, in generale, risulta efficace nel ridurre lo spazio di ricerca e mitigare problematiche fondamentali come Semantic Collapse, Hubness... tuttavia, presenta anche alcune criticità strutturali importanti, in larga parte riconducibili alla qualità del livello di *rollup* e, quindi, alla bontà dei summaries di community.

Il livello `Community` agisce, di fatto, a monte del retrieval fine-grained come componente coarse: il compito è di filtraggio semantico, appunto. Se il summary, però, non rappresenta correttamente il sub-topic, oppure enfatizza dettagli marginali rispetto all'intento della query, la procedura rischia di non selezionare le seed communities corrette e, conseguentemente, di escludere dal pool i chunk effettivamente rilevanti. Queste criticità si vedono particolarmente bene su domande *semplici* o fortemente *entity-centric*: benché esista una corrispondenza lessicale molto chiara (nomi propri, sigle, identificativi) e la componente full-text della hybrid search sul chunk level sarebbe, da sola, sufficiente, l'approccio gerarchico potrebbe rischiare di non restituire il contesto corretto per il semplice fatto che la componente coarse, dipendente dai rassunti, potrebbe restringere il campo eliminando segnale utile e non rumore. Infatti, passando dal chunk level al summary level, l'informazione viene compressa e l'evidenza full-text viene in gran parte diluita: una specifica entità potrebbe non comparire nel summary perché ritenuta non centrale dal riassunto, riducendo l'efficacia del match testuale; ci sono casi, quindi, in cui la pipeline è costretta a "indovinare" prima il sub-topic corretto e solo dopo può recuperare il chunk che contiene l'entità, rendendo l'approccio più fragile rispetto a una classica retrieval-first su chunk. Per mitigare questo problema, la maniera più semplice è quella che ho introdotto di expansion con *hops*: questi non devono essere troppi, ai sensi del fatto che più hop corrispondono a più chunk su cui valuti il retrieval e, al limite, tendenti ad un fattore di riduzione gerarchica nullo; l'espansione permette di avere più possibilità che i chunk contenenti segnale utile *non* si perdano nella fase coarse, ma è una mitigazione puramente 'statistica'. Il vero meccanismo che renderebbe l'approccio proposto più robusto alla situazione precedente è sempre lo stesso: *Entity Extraction*, aggiungendo, magari, un campo dove puoi introdurre una componente di match sulle entities nel retrieval, cosa che, come già discusso, non è propriamente corretto fare nel nostro caso con LLM zero-shot.

Nulla vieta, in linea teorica, di iterare nuovamente lo stesso schema e costrui-

re ulteriori livelli gerarchici: ad esempio, raggruppare communities in "super-communities" e produrre summaries di livello superiore, ottenendo un rollup ancora più aggressivo e un coarse-to-fine su più passi. Tuttavia, tale estensione amplificherebbe ancor di più il problema della compressione semantica: summaries di summaries tendono a perdere informazione importante, soprattutto quella utile alla componente full-text e alle query guidate da dettagli più specifici. L'astrazione migliorerebbe l'efficienza e mitigherebbe ancor di più problemi "globali" come quelli citati, ma al costo di diluire ulteriormente segnale utile e rendere, probabilmente, eccessivamente astratta la componente coarse. Per questo motivo, l'integrazione a due livelli (chunk $\rightarrow$ community, con collegamenti cross-community) risulta già un compromesso naturale e, a mio avviso, abbastanza ragionevole; eventuali livelli aggiuntivi richiederebbero strategie più robuste di rappresentazione: migliori livelli di dettaglio, summaries strutturati... in quanto non puoi pensare di fare un summary di summaries, è semplicemente sbagliato per tutti i motivi suddetti, ed evitare che l'aggregazione diventi un bottleneck semantico è tassativo.

Sul piano della manutenzione, l'aggiunta di nuovi documenti non introduce criticità sostanziali per la parte intra-documento: questa è una forza dell'approccio costruito, anche orientato ai discorsi che successivamente faremo su UI e Texi Extraction. Infatti: chunking, costruzione degli archi, calcolo di PageRank e clustering possono essere eseguiti in modo indipendente per ciascun documento e, quindi, parallelizzati; stesa cosa la generazione dei summaries per communities: concettualmente è incrementale ed ogni nuovo documento produce nuove communities indicizzabili separatamente. Il punto più delicato è il livello successivo di aggregazione tra communities, ovvero la costruzione degli archi `COMM_SIMILAR_TO` mediante KNN sullo spazio degli embedding dei summaries: in un contesto incrementale, dove l'utente può caricare documenti in momenti diversi, l'inserimento di nuove communities implica che esse debbano essere connesse al resto del grafo globale, ma anche che alcune communities pre-esistenti potrebbero dover aggiornare il proprio vicinato, poiché i nuovi punti nello spazio vettoriale possono entrare nei loro top-k. Di conseguenza, pur utilizzando KNN (che evita $O(N^2)$), rimane il problema di aggiornare efficacemente il grafo, richiedendo almeno di: calcolare i vicini delle nuove communities, eventualmente rivalutare i vicini di una parte delle communities esistenti o, al limite, un ricalcolo periodico *ex-novo* delle connessioni sull'ultimo livello; questa è la componente più onerosa dell'architettura.

5.3 CoT RAG

L'approccio di *CoT RAG* implementato estende il Naive RAG introducendo un **agente** LLM che guida iterativamente il retrieval tramite, come suggerisce il nome, una forma di *Chain-of-Thought* materializzata in uno *scratchpad*. L'idea non è (solo) rendere la ricerca step wise, ma **adattiva**: invece di eseguire un singolo retrieval top-k sulla query originale, il sistema alterna step di retrieval, aggiornamento della 'memoria di ricerca' data dallo scratchpad e decisione di

stop oppure generazione di una nuova query mirata a colmare l'informazione mancante; l'adattività ed il concetto CoT sta, proprio, in questa generazione di sub-queries: il modello si concentra nel cercare solamente ciò che gli serve per soddisfare l'information need ed adattivamente rispetto alle informazioni già trovate; questo permette di costruire, iterativamente, queries molto più puntuali ed aumentare la precisione della risposta, con buona flessibilità rispetto la complessità della query iniziale in realtà.

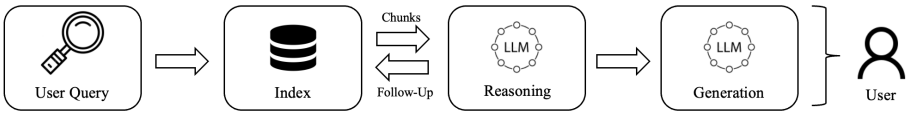

Figura 3: Schematizzazione Architettura CoT RAG

In letteratura, questa famiglia di metodi ha una forte interazione con approcci che si basano sull' *interleaving* tra reasoning e retrieval (ad es. IRCoT) o che formalizzano la decomposizione in sotto-domande e l'uso di strumenti esterni (ad es. Self-Ask, ReAct), dove lo "stato" intermedio funge da guida per la ricerca successiva; molto delle scelte progettuali alla base dell'approccio sono ispiratr a lavori abbastanza recenti [Trivedi et al., 2023, Press et al., 2022, Yao et al., 2022], in particolare: [Nye et al., 2021]. Nel codice, tutto viene orchestrato con LangChain (template, parsing, costruzione della risposta) e l'endpoint di Groq per interagire con il modello scelto. Un aspetto interessante è che il CoT RAG è agnostico a come organizzi il corpora: puoi, o meno, utilizzare strutture gerarchiche e valutare la ricerca adattiva su queste, ciò dipende dalle scelte progettuali; la pipeline prodotta è, infatti, *backend agnostic* e ti permette di indicare direttamente se stai lavorando su un grafo o meno. Ci torneremo successivamente nelle considerazioni di questa sezione.

5.3.1 Scratchpad

Lo *scratchpad*, come accennato precedentemente, è una memoria testuale **accumulativa** che rappresenta lo stato corrente di ciò che il modello "conosce" in modo *grounded*; insomma, la storia del retrieval fino a quel momento. Ad ogni iterazione, l'agente LLM riceve: la domanda originale, lo scratchpad corrente, una history breve degli step recenti (query emesse e doc-id recuperati) e le nuove informazioni ottenute dal retreival, con cui aggiornare lo scratchpad e costruire le query per il passo successivo. Essendoci abbastanza passaggi da gestire, la maniera più coerente è che il LM produca un output strutturato (JSON) con cui si facilita l'estrazione delle singole componenti:

- un **aggiornamento** dello scratchpad (fatti in bullet-point derivati *solo* dalle evidenze)
- una decisione booleana di **stop** (`enough`) se l'evidenza è sufficiente a soddisfare l'information need

- una descrizione di **cosa manca** (`missing`) se così non fosse
- una **next query** unica e mirata per il passo successivo

Quello che, personalmente, mi piace molto di quest'approccio è la forte interazione con il vero motivo per cui oggi si utilizzano estensivamente i Language Models: il dinamismo nel Question Answering, la gestione naturale dei follow-up e, più in generale, la forte componente d'interazione che c'è utente-modello. Valutando l'approccio, ho spesso salvato l'evoluzione dello scratchpad a fini di audit e, tralasciando piccole imprecisioni che puoi risolvere con un po' di prompt engineering, emergeva chiaramente proprio questa adattività: la ricerca veniva "guidata" verso query sempre più puntuali, man mano che il sistema comprendesse lo stato corrente dell'informazione e ciò che mancasse raffinando le query successive.

5.3.2 Step-Back Prompting

Lo *Step-Back Prompting* è una tecnica in cui, prima di avviare il retrieval, si chiede al modello di riscrivere la domanda ad un livello di astrazione superiore, producendo una query più generica che catturi *concetto* ed *intento*. L'idea è sempre quella di rollup e drilldown classica degli approcci gerarchici, in una forma più 'light' e semplice da realizzare.

L'approccio CoT RAG che ho presentato prima, in realtà, deriva da una variante iniziale che concettualmente sfruttava proprio lo Step-Back Prompting con l'adozione di un *Planner*: si genera una `stepback_question` e un `intent`, poi un planner produce un piano di 2–4 step con sub-questions e query iniziale, aggiorna lo scratchpad e decide stop o follow-up. Rispetto all'approccio descritto precedentemente, qui il piano viene definito 'a priori': gli step vengono costruiti prima della fase di retrieval e, quindi, prima di poter attingere effettivamente ai chunk, mentre l'approccio adottato si basa direttamente su 'cosa manca' rispetto all'informazione effettivamente raccolta; il Planner iniziale vincola maggiormente la direzione della ricerca. L'intuizione alla base viene, proprio, dallo Step-Back Prompting: l'idea era quella di guidare il modello nel costruire query a diversi livelli di specificità, partendo da una step-back iniziale e scendendo più nel dettaglio con query successive. L'impostazione adottata è, però, più vicina al concetto di retrieval *adaptive* e, infatti, è stata preferita; tuttavia, quando discuteremo successivamente di **query drift**, la versione con Planner è molto più robusta a tale fenomeno, nonché generalmente più cost-efficient, perdendo, però, proprio sull'adattività della ricerca.

Nell'approccio Plannerless non è stato introdotto esplicitamente lo Step-Back Prompting, poiché il beneficio di "cambiare scala" nella formulazione delle query è, in linea di principio, garantito strutturalmente dall'adattività stessa: la CoT può guidare sia query progressivamente più specifiche quando manca informazione, sia generalizzazioni quando serve contesto o quando i tentativi troppo

specifici falliscono. Ad inferenza, però, il comportamento più comune e naturale è una sequenza di query sempre più specifiche; tuttavia, nulla ti vieta di aggiungere dei fallback dove, ad esempio, dopo n step senza miglioramento si genera una query più astratta (step-back) per recuperare background o riorientare la ricerca. Il CoT RAG è un approccio interessantissimo proprio perché chi progetta ha molta libertà di scelta e, d'altronde, progettualmente hai tante alternative con molto potenziale: vantaggio strutturale, anch'esso, di tutti i prodotti LLM-centric.

Considerazioni

Nel Naive RAG la query è valutata *single-shot* ed il retrieval top-k è direttamente ciò che condiziona la generazione. Nel CoT RAG, invece, la query diventa un **processo**: l'LLM usa lo scratchpad per capire se l'evidenza è sufficiente e, in caso contrario, produce query successive più specifiche. L'effetto atteso è un miglioramento generale su query di medio-alta complessità grazie alla CoT, come maggior robustezza alla variabilità lessicale: d'altronde, la scrittura di query più specifiche è strutturalmente un query rewriting. In questo senso, non è differente l'idea rispetto a quella presentata nei capitoli precedenti, ma l'esecuzione: non puoi aspettarti necessariamente di utilizzare un modello fine-tuned, a meno che non utilizzi diversi language models; sarebbe stato interessante scomporre l'agente in sub-agents più piccoli, con uno SLM, ad esempio, specializzato in un rewriting coerente alla distribuzione lessicale del dominio, tuttavia, non concentrandomi su un dominio specifico, è una strada che non ho percorso.

Queste capacità adattive richiedono, tipicamente, modelli di dimensione mediograndе: serve una forte comprensione testuale, conoscenza generalista, buone abilità di reasoning, sintesi, riscrittura... e qui arriva la nota dolente: ci sono tantissimi vantaggi e, progettualmente, l'impostazione non è così più complicata del RAG Naive, tuttavia il costo ad inferenza cresce non solo perché usi modelli mediamente più costosi, ma soprattutto perché una singola domanda può innescare molti sub-step nella CoT; ogni step consuma token sia per il reasoning che per la gestione delle fasi intermedie (scratchpad update, query successive...) e, si noti, sono comunque token generati, quindi quelli di 'costo massimo'. Prima accennavamo al concetto di *backend agnostic*, tuttavia i risultati che presenteremo non si basano su strutture gerarchiche per l'organizzazione del corpus: in questo modo è vero che non mitighi le problematiche fondamentali del retrieval, ma d'altro canto il costo che avresti nella gestione di tali strutture si andrebbe a sommare ad una componente già molto alta ad inferenza per i motivi precedentemente discussi e, detta molto semplicemente, avresti un tradeoff performance-costo estremamente sbilanciato, o meglio, tutti i vantaggi che puoi derivare dalla combinazione Hierarchical + CoT non sarebbero giustificati da quanto spendi per metterli in pratica; in casi reali quindi, spesso con forti vincoli sui costi, si sceglierebbero sempre alternative di Naive RAG come miglior compromesso.

Query Drift Oltre che le considerazioni sui costi , l'adattività cela un rischio importante di **query drift**: se il controller interpreta male cosa manca, può generare query che deviano dal bisogno informativo originale, accumulando evidenze "plausibili" ma non pertinenti; questo è un modo informale per evidenziare ancor di più quello che dicevamo prima: hai bisogno mediamente di modelli grandi e, soprattutto, *instruction-tuned*. Si definiscono *instruction-tuned* Language Models che contano una fase di post-training dove vengono allenati a rispondere in maniera coerente alle richieste ed i vincoli esposti dell'utente, formulati tramite prompting, riducendo proprio le stesse deviazioni che sono alla base del query drift; se insegni/indichi al modello come ragionare e come formulare le query tramite prompting, di base vuoi che questo commetta meno drift possibili rispetto a ciò che gli chiedi e, nel CoT RAG, è molto più vero che in tutte le altre architetture dati i molti passi intermedi nel processo di retrieval. Questo, in realtà, è un concetto importante indipendentemente dall'architettura RAG: per garantire che il modello utilizzi solamente il contesto che ricava dal retrieval, ad esempio, tipicamente glielo si chiede proprio come system prompt (*Use only the provided context. If the context is insufficient, say so explicitly*) e se il modello non è instruction-tuned, di fondo, il RAG stesso non ti da le stesse garanzie. In realtà, quello che stiamo discutendo è un failure mode noto dei metodi reasoning+tool-use: tracce convincenti, ma errate come le decisioni prese dall'agente, molto discussi nella pratica in linee come ReAct [Yao et al., 2022]. Non ci sono metodi di mitigazione esatti a questo fenomeno, difatti l'implementazione fornita include meccanismi che si basano su ragionamenti puramente euristici: l'intuizione su cui mi sono basato è che il drift, se avviene, è molto più probabile nasca dopo un certo numero di step di adaptive retrieval, ovvero presuppongo che l'information need della query iniziale possa essere mediamente chiaro al modello e l'errore nella generazione di query avvenga, proprio, quando devi stimare informazioni particolarmente specifiche; questo mi ha portato al concetto di history: se valgono le intuizioni precedenti, allora le prime query progettate dal modello puoi supporre siano coerenti, di conseguenza, dandogli una finestra di $2-3$ query precedenti, il modello ha sempre un'ancora contestuale che funge da 'reminder' del modo corretto di modellare l'informazione mancante, come una sorta di few-shot-learning.

5.4 RAG Datasets

Nel contesto di pipeline RAG, un limite pratico alle valutazioni è l'assenza di una standardizzazione forte e di benchmark "definitivi": molte risorse disponibili online sono, di fatto, benchmark di *retrieval* o *ranking* più che valutazioni end-to-end di una pipeline RAG; ai sensi di questo, infatti, la stragrande maggioranza dei dataset distribuiscono già il testo pre-chunkato, semplificando sì le valutazioni ma rendendo le metriche che ne ricavi molto meno realistiche dando già 'per buona' una delle componenti più critiche della pipeline, oltre che impedendo di misurare gli effetti delle scelte che fai su strategia ed iperparametri. In generale, appunto, la letteratura propone numerose metriche per retrieval e ranking: Precision, Recall, F1, NDCG, MRR, ecc.. a mio avviso, queste metriche

sono complessivamente sopravvalutate in contesti di retrieval e, in un setting RAG come il mio, ancora di più: non essendoci uno scenario domain-specific ed adottando soluzioni il più possibilmente plug-and-play (vedi SentenceTransformer), la subottimalità del ranking che ottengo è un presupposto già noto a priori e non ho bisogno di misurarlo, oltre che metriche come Precision e Recall tendono a sovrastimare aspetti "meccanici" del recupero rispetto all'obiettivo reale del sistema. In un RAG, infatti, ciò che conta primariamente è soddisfare l'*information need* dell'utente in modo corretto e supportato dal contesto; per questo, la valutazione è formulata come judging rispetto ad una gold answer: verificando sia l'allineamento informativo, sia la correttezza rispetto ai contesti recuperati, indirettamente otteniamo anche un segnale sulla qualità del retrieval; d'altronde, se la risposta è corretta e vincolata alle informazioni grounded, il sistema ha necessariamente recuperato chunk utili trai primi posti e, quindi, ottenuto un buon ranking.

A livello implementativo, il judge riceve: la domanda, la gold answer (tipicamente concisa), la risposta del modello (tipicamente più verbosa) ed il contesto recuperato; poi, assegna tre giudizi:

- `answer_correctness`, inteso come correttezza complessiva della risposta rispetto alla domanda e alle informazioni della gold answer; può essere formulata diversamente o più in dettaglio, ma non deve introdurre errori o contraddizioni
- `evidence_coverage`, inteso come copertura dei fatti *necessari* della gold answer ed intendibile come una sorta checklist di entità o dettagli obbligatori (nomi, date, luoghi, relazioni...). Questa viene dal fatto che, generalmente, una risposta può essere complessivamente corretta, o meglio, 'giusta' rispetto alla domanda che è stata posta ma non soddisfare completamente tutte le informazioni che ci si aspettava di trovare; infatti, proprio questa ci fornisce una componente più retrieval-oriented, collegandoci ai ragionamenti di sopra
- `faithfulness_to_context`, inteso come allineamento tra le affermazioni nel contesto e nella risposta, penalizzando anche il caso in cui la risposta sia corretta ma non supportata dal retrieval

Un punto critico su cui mi sono concentrato nel prompt è la mitigazione del bias verso risposte brevi: poiché la gold answer è spesso un riassunto fattuale, il judge viene istruito a *non penalizzare* risposte più lunghe se e solo se le informazioni aggiuntive sono rilevanti, supportate dal contesto e coerenti con la reference; benché si utilizzi un modello LLM (grande, meno bias prone) come judge, parliamo di GPT-4o, comunque è un guardrail che, a mio avviso, rimane necessario. La scelta di GPT-4o e non di, magari, modelli più stato dell'arte come Gemini 3, GPT 5.2, Claude Opus 4.6... viene dal fatto che il LLM judging è tanto una pratica 'nuova' quanto lo sono i Language Models e GPT-4o è uno dei judge più usati e testati nella letteratura, quindi complessivamente è

un'alternativa affidabile a parità di costo.

Sebbene il prompt produca valori numerici in $[0, 1]$, la scala è intenzionalmente categoriale (0.0, 0.5, 1.0): è concettualmente equivalente a etichette discrete ("pessimo", "buono", "ottimo"), ma mi evitava una mappatura ulteriore tra categorie e numeri senza cambiare la logica della valutazione; è importante sottolineare questo, altrimenti potrebbe sembrare ci sia una discrepanza con i ragionamenti fatti nell'introduzione, dove intimavo di come sia sempre meglio far generare etichette categoriche, piuttosto che corrispettivi numerici, al judge.

5.4.1 GraphRAG Bench

Il dataset `GraphRAG-Bench/GraphRAG-Bench`, disponibile su Hugging Face, è strutturato in due configurazioni principali: *novel* e *medical*, ciascuna accompagnata da un file dedicato di domande. Il benchmark nasce con l'obiettivo esplicito di confrontare approcci RAG "tradizionali" e varianti *graph-based* lungo l'intera pipeline, dalla costruzione della struttura (grafo) al retrieval, fino alla generazione, includendo task eterogenei e a difficoltà crescente, come: semplice fact retrieval, reasoning multi-hop, summarization e creative generation.

Mi sono concentrato sulla porzione *Novel* del dataset e gli iniziali risultati che presenteremo fanno riferimento a questo; la formulazione delle query è spesso marcatamente *entity-oriented*: molte domande citano persone, luoghi o oggetti direttamente; questa caratteristica ha una conseguenza pratica importante: modelli di retrieval lessicale (BM25) o ibridi (lessicale + denso) risultano di base estremamente efficaci, poiché possono agganciarsi a termini altamente discriminanti già presenti nella domanda, riducendo la necessità di un vero reasoning multi-hop. Sostanzialmente, il segnale lessicale è spesso già sufficiente a individuare un insieme di chunk rilevanti nel caso generale e la componente densa tende a fungere più da ausilio che da fattore determinante. A mio avviso, infatti, il dataset non è un benchmark tanto buono: la complessità effettiva di alcune istanze etichettate come *Complex Reasoning*, ovvero le query con massima hardness nel dataset, spesso vengono risposte con, ad esempio, una CoT che si ferma al primissimo step, quindi di fondo un Hybrid Search Naive classico: la presenza esplicita di entities nella query è direttamente collegata a questa tendenza, essendo che match dell'entità $\rightarrow$ chunk rilevante $\rightarrow$ risposta, senza che emerga in modo netto il vantaggio di strategie adattive o gerarchice. Tuttavia, l'orientamento entity-centric è coerente con la finalità del benchmark: in un setting in cui il seed è facilmente individuabile, diventa più facile misurare quanto un approccio *GraphRAG* riesca a sfruttare relazioni e struttura per rispondere all'information need; questo, però, è vero se l'approccio graph è entity-oriented, come Microsoft GraphRAG: non il nostro setting quindi.

La domanda quindi sorge spontanea: *perché è stato scelto come benchmark se sembra inadeguato?* Principalmente per due motivi:

- è un dataset dove puoi agilmente ricostruire il testo di partenza non chunked, quindi la valutazione della pipeline è influenzata dalla strategia che adotti
- molte delle considerazioni fondamentali alla base del lavoro vengono proprio confermate dai risultati quantitativi che ricaviamo qui, in particolar modo su Naive e CoT RAG

Quello che mi piace del benchmark, in particolare, è proprio la possibilità di attingere direttamente ai documenti da chunkare: puoi intervenire end-to-end, evitare di doverti basare sul testo già chunked o, addirittura, dover costruire pipeline di scraping complesse per ovviare al problema fondamentale precedente; senza avere a disposizione direttamente la totalità del testo non è possibile osservare, come abbiamo fatto nei risultati che seguono, l'importanza della strategia di chunking stessa. Come accennato prima, questa è una qualità che non molti benchmark hanno, il che lo rende più adatto, rispetto alla maggior parte, a ricavare statistiche significative a supporto delle intuizioni su cui si basa il mio lavoro.

Per il *CoT RAG*, d'altronde, il dataset si è rivelato comunque utile soprattutto per la fase di prompt engineering e la definizione di vincoli sulla generazione delle sub-questions; osservando direttamente il ciclo *evidence* $\rightarrow$ *scratchpad* $\rightarrow$ *next_ query*, infatti, è possibile formalizzare regole che evitano follow-up "innaturali", ad esempio query troppo simili alle precedenti, magari solo riformulazioni, piuttosto che evitare di utilizzare l'informazione ricavata dagli step precedenti di retrieval; in questo senso, le prestazioni risultano fortemente dipendenti dal *prompt* e, più in generale, da come si istruisce il modello: non intervenendo con fine-tuning ad hoc, il contributo principale passa inevitabilmente dal prompt engineering. Anche in presenza di un set di domande relativamente "facili" rispetto alla complessità media del retrieval, l'osservazione qualitativa degli errori è sufficiente a progettare *patch* efficaci, sfruttando anche l'osservazione dell'evoluzione dello scratchpad. Un esempio concreto è il concetto di *Entity Bridging*: nel ciclo CoT ho osservato che il modello tendeva sì a concentrarsi sull'informazione mancante, ma spesso generava query successive molto simili tra loro e, soprattutto, che aumentavano di poco la copertura semantica: in sostanza, il rewriting si concentrava spesso su parafrasi, o meglio, identificato cosa mancasse il modello era meno tendente ad utilizzare, se presenti, informazioni importanti nel contesto retrieved come le entities. Per mitigare questo failure, ho introdotto istruzioni che vincolano esplicitamente il modello ad utilizzare le entità (nomi propri, luoghi, oggetti) quando queste compaiono nel contesto recuperato: così, la query successiva non è solo più specifica, ma anche più *grounded* rispetto a ciò che il sistema ha effettivamente osservato e, se puoi, strutturare la ricerca su interazioni tra entità è molto efficace, basta vedere i ragionamenti fatti in precedenza su GraphRAG o, più generalmente, sui KGs; il modello prova quindi a colmare l'informazione mancante sfruttando direttamente concetti il più possibilmente discriminanti, che rendono già la ricerca full-text altamente precisa e questo migliora drammaticamente la qualità delle sub-questions. Ovviamente, è

difficile presentare risultati quantitativi rispetto quest'ultima affermazione: non ci sono metriche che definiscono quanto una query sia buona rispetto ad un'altra, quindi le considerazioni derivano prevalentemente da un audit manuale.

5.4.2 Risultati

Per i risultati che andremo a presentare, si sono utilizzati due modelli disponibili sull'API di Groq: `moonshotai/kimi-k2-instruct-0905` e `llama-3.1-8b-instant`. La scelta tra questi due, semplicemente, deriva da un compromesso che volevo raggiungere sui risultati: ci sono approcci di RAG che sono più dipendenti dalla qualità del LM utilizzato, come quelli CoT, ed altri che, invece, sono ragionevolmente agnostici a questo. Kimi è un modello MoE (*Mixture of Experts*) tra i più famosi e performanti sui maggiori benchmark, al momento della scrittura di questa tesi oscilla trai top-10 modelli presenti sul mercato, ed è la componente 'Large' dei risultati; d'altra parte, la componente 'Small' è data da uno dei modelli Llama, scelto prevalentemente perché, a parità di size, è notoriamente performante e molto usato in progetti di ricerca.

Il confronto tra Kimi e Llama è stato condotto su 324 questions di GraphRAG Bench, distribuite rispetto alle tipologie proposte dal dataset; complessivamente si disponeva circa di $2k$ queries e 324, da un'analisi quantitativa preliminare, garantisce rappresentatività: non ha senso provare gli approcci sulla totalità della queries, basta che i sample scelti siano rappresentativi per ricavare statistiche con una buona confidenza.

Metric	**Kimi Mean**	**Llama Mean**	**Δ Mean**
answer_correctness	0.771	0.633	-0.138
faithfulness_to_context	0.881	0.675	-0.206
evidence_coverage	0.762	0.611	-0.150
CoT steps (mean)	1.715	1.573	-0.142

Tabella 4: Confronto Kimi - Llama in CoT RAG.

Nel complesso, i giudizi assegnati a Kimi risultano sistematicamente più alti rispetto a quelli di Llama su tutte e tre le metriche del judge (GPT-4o), con il gap più marcato su `faithfulness_to_context`. Mediamente, infatti, la differenza $\Delta = \text{Llama} - \text{Kimi}$ è sempre negativa; un'interpretazione coerente è che, quando il retrieval non fornisce un contesto sufficiente (o chiaramente risolutivo), Llama tenda a incorrere più spesso in due comportamenti alternativi: *over-generation*, introducendo dettagli plausibili ma non supportati dalle evidenze, che abbassa `faithfulness_to_context`, oppure *under-answering*, più conservativo e producendo risposte parziali o vaghe, penalizzando `answer_correctness` e `evidence_coverage`. Ovviamente, risulta abbastaza semplice capire come, nel caso ideale, vorremmo porci in uno stato ragionevolmente intermedio tra questi due comportamenti: il deficit di Llama non è solo "quanto" risponde, ma *come* gestisce l'incertezza aggiungendo contenuto non grounded o perdendo

informazione supportata dal testo. Questa osservazione, in realtà, è sia coerente con il comportamento più "rigidio" di un judge rispetto all'allineamento tra affermazioni e contesto, sia con la differenza qualitativa trai modelli: a parità di correttezza rispetto alla gold answer, la valutazione può ridursi sensibilmente se la risposta contiene dettagli non supportati dal retrieval, oppure se il judge interpreta tali dettagli come non verificabili nel contesto, ciò non toglie però la tendenza di Llama ad introdurre dettagli non supportati e, nonostante le indicazioni, ad inferire informazione non direttamente citata dal testo. Questo risultato è particolarmente rilevante e si collega direttamente ai concetti discussi nel capitolo sul CoT RAG: in un'impostazione adattiva, il retrieval non è più un'operazione single-shot, ma un processo iterativo in cui il modello deve interpretare correttamente ciò che manca, trasformarlo in una query utile e mantenere coerenza con l'information need iniziale. Modelli più piccoli tendono a essere più error prone in questi compiti: il reasoning intermedio può degenerare più facilmente in *query drift*, poiché errori di comprensione o di pianificazione si propagano step dopo step, accumulando informazioni plausibili ma non pertinenti alla domanda; al contrario, modelli più grandi e meglio *instruction-tuned* hanno in media maggiore capacità di mantenere il vincolo di grounding e di generare follow-up più stabili, riducendo la probabilità che il ciclo "scratchpad → next_query" si disallinei dall'information need. In CoT RAG quindi, questo corrobora quanto la scelta del modello non sia una componente fondamentale per raggiungere trade-off qualità-costo target.

La scomposizione per `question_type` rende più chiaro il punto fondamentale: il vantaggio di Kimi non è legato solo alla "qualità della risposta", ma a una migliore comprensione testuale nel senso generico; in un setting RAG, questo si traduce in: capacità di comprendere la domanda con precisione, interpretare correttamente cosa contengono i chunk recuperati ed evitare di produrre affermazioni che non siano tracciabili dal contesto. Questo si riflette, come discutevamo prima, soprattutto in `faithfulness_to_context`, dove il gap rimane sistematicamente elevato in tutte le categorie: Kimi tende a mantenere più facilmente il grounding, mentre Llama introduce più spesso dettagli "ragionevoli" ma non supportati nei chunk, degradando l'aderenza al retrieval. Il fatto che il divario sia marcato anche su `Fact Retrieval` è particolarmente interessante: in questi casi non è richiesta una decomposizione multi-hop complessa, quindi la differenza non può essere attribuita solo alla capacità di ragionamento del modello, ma proprio alla comprensione dell'information need stesso; sostanzialmente, anche quando l'informazione necessaria è direttamente contenuta nelle evidenze, Kimi sembra più consistente nel selezionare e costruire la risposta rispetto alle informazioni che soddisfano la query, mentre Llama è più incline a completare la risposta con inferenze non richieste o non presenti nel contesto, penalizzando faithfulness e, di conseguenza, anche `evidence_coverage`. Nel CoT RAG, poi, non basta "rispondere bene" ma bisogna **capire cosa manca** e trasformarlo in una `next_query` realmente utile: i risultati sugli step suggeriscono che Kimi, in media, 'triggera' più spesso l'adattività, soprattutto su `Contextual Summarize`, il che è coerente con un modello più capace di ricono-

scere un soddisfacimento parziale dell'information need e la necessità di cercare ulteriormente prima di fermarsi. Questa proprietà è esattamente ciò che ci si aspetta da un approccio adattivo: la qualità non deriva solo dal primo top-k, ma dalla capacità del modello di formulare follow-up più puntuali; d'altronde, il fatto che generi più step non vuol dire che la qualità delle follow-up questions sia minore, ma è perfettamente spiegabile dalle informazioni che le metriche ti danno sulla comprensione testuale: Llama dichiara `enough` prematuramente, ma perché non comprende la totalità della semantica mancante.

Un risultato importante e che, secondo me, emerge molto chiaramente è che, in un setting RAG, affidarsi a modelli piccoli (SLM) tipicamente non ti porta molto lontano. Il caso delle query `Fact Retrieval` è particolarmente emblematico: si tratta di domande in cui l'information need è spesso localizzabile in un singolo passaggio e, d'altronde, in questo dataset l'adattività tende a fermarsi presto proprio perché "bastano" i chunk recuperato già dal primo passo di retrieval per rispondere; eppure, sembrando le condizioni teoricamente favorevoli, non dovrebbero emergere differenze nette sulla comprensione testuali dei modelli data la semplicità del task, cosa che però non avviene: Llama mostra difficoltà palesi nel compito fondamentale di estrarre dai chunk ed organizzare all'utente l'informazione necessaria per rispondere alla query in ingresso e qui, si noti, non si parla più di bottleneck relativo al retrieval. Infatti, l'informazione chiave è spesso già presente nel contesto recuperato, ma il modello non riesce comunque a valorizzarla: la seleziona male, la comprime in modo scorretto, oppure la distorce. Questo ridimensiona un'intuizione piuttosto diffusa sul RAG, cioè che "basti" un retrieval perfetto per rendere competitivo anche uno SLM: in realtà, la qualità finale non dipende solo da cosa recuperi, ma anche da quanto bene il modello sa leggere, integrare e fare 'stitching' di quelle evidenze; non è raro, appunto, osservare il caso controintuitivo in cui uno SLM con retrieval migliore produce risposte meno affidabili o meno preferibili di un LLM che parte da un contesto recuperato leggermente peggiore, ma lo interpreta molto meglio.

Sullo stesso benchmark, ho testato anche l'approccio Naive RAG ed i risultati sono riassunti nella tabella che segue.

Metric	**Kimi Mean**	**Llama Mean**	**Δ Mean**
answer_correctness	0.783	0.711	-0.072
faithfulness_to_context	0.828	0.752	-0.077
evidence_coverage	0.777	0.672	-0.105

Tabella 5: Confronto Kimi–Llama in Naive RAG

I risultati evidenziano chiaramente che l'effetto del CoT RAG non è "universale", ma dipende fortemente dalla dalla qualità del modello: capacità di comprensione, formulazione dei follow-up e attinenza all'information need. Nel caso di **Kimi**, l'introduzione dello scratchpad produce un guadagno netto sulla `faithfulness_to_context` (da 0.828 a 0.881), mentre `answer_correctness`

ed `evidence_coverage` rimangono quasi invariate: la direzione è leggermente negativa ed, in questo senso, il modello sembra adottare un comportamento più conservativo, preferendo generare risposte leggermente meno complete ma meglio ancorate al contesto recuperato, piuttosto che rischiare affermazioni non direttamente supportate. Questo pattern è coerente con l'obiettivo fondamentale del RAG: non solo produrre risposte corrette, ma garantire che ogni affermazione sia tracciabile rispetto al contesto fornito. La lievi riduzioni in `answer_correctness` (-0.012) e `evidence_coverage` (-0.015), essendo relativamente piccole rispetto alle differenze osservate, potrebbero sicuramente essere significative di una tendenza reale ma è anche plausibile che siano semplici fluttuazioni, per questo si può considerare praticamente invariata la qualità e copertura delle risposte con buona approssimazione. Questo comportamento, d'altronde, è perfettamente coerente con un modello che usa l'adattività principalmente per *ridurre affermazioni non supportate*: lo scratchpad rende più esplicita la distinzione tra ciò che è stato recuperato dalle fasi precedenti di retrieval e ciò che, invece, sono deduzioni/inferenze del modello, migliorando quindi il grounding complessivo della risposta; è un risultato molto interessante, ai sensi del fatto che, come suddetto, il principio alla base del RAG stesso è non solo rispondere bene, ma in maniera supportata dal contesto.

Metric	**Δ Kimi** (CoT−Classic)	**Δ Llama** (CoT−Classic)
answer_correctness	−0.012	−0.078
faithfulness_to_context	+0.053	−0.077
evidence_coverage	−0.015	−0.061

Tabella 6: Variazione *intra-modello* passando da RAG Naive a CoT RAG

Il quadro complessivo è diametralmente opposto per **Llama**: passando dal Naive RAG al CoT RAG, tutte le metriche peggiorano in modo consistente, come riassunto in Tabella 6. Questo non necessariamente vuol dire che il CoT non sia una buona strategia, anzi: i risultati confermano quanto dicevamo precedentemente sull'importanza delle capacità del modello, per Llama il CoT non aumenta l'informazione utile, ma aumenta le opportunità di errore; più step significano più query generate, più evidenze da integrare, più passaggi intermedi da gestire e un modello meno versatile può deviare più facilmente dalla domanda iniziale (*query drift*) o interpretare male le informazioni ricavate; d'altronde, c'è un calo molto drastico sia in `faithfulness_to_context` che in `answer_correctness`. Una maniera interessante per vederla è qusta: l'adattività agisce come un *moltiplicatore*, quando il modello è sufficientemente capace e instruction-following, l'iteratività migliora il grounding; quando non lo è, amplifica la deriva e degrada sia la qualità del contesto sia la risposta finale. Questo risultato corrobora ancor di più l'idea precedebte che l'adaptive retrieval richieda tipicamente modelli medio-grandi, tuttavia evidenzia un altro aspetto importante: sebbene, intuitivamente, il Naive RAG possa sembrare una baseline 'strutturale' del CoT RAG, ai sensi del fatto che, non triggerando l'adaptiveness, il comportamento degenera alla variante classica, questo non è supportato dai risultati in maniera

diretta; mediamente lo scratchpad, le istruzioni CoT, il reasoning... confonde modelli SLM, che performano peggio della baseline Naive.

Potrebbe sembrare strano il fatto che la qualità delle risposte e la copertura semantica, date da `answer_correctness`, e `evidence_coverage`, non migliorino con il CoT per un modello grande come Kimi. In realtà, in un setting come il nostro il punto chiave è proprio che *non peggiorano*: introdurre additività (più chiamate LLM, più token, più punti in cui può emergere drift) senza degradare la qualità media rispetto alla baseline è già un risultato non banale, perché indica che il modello non sta "rompendo" il comportamento del RAG classico nei casi in cui il retrieval single-shot è sufficiente, guadagnando però in modo netto sulla `faithfulness_to_context`; per riassumere all'osso: l'adattività, quando attivata, è utile al modello. Mi aspettavo, tuttavia, che rispetto a una baseline Naive la qualità complessiva della risposta rimanesse mediamente simile: gli step adattivi vengono valutati poche volte (si guardi i medium steps) e, anche se lo fossero più spesso, siamo comunque in un dominio generico con componenti *plug-and-play*; come suddetto, il prompt engineering può mitigare failure mode e permetterti di migliorare le performance, ma generalmente non ti rende competitivo e questo vale in generale, non solo nel retrieval. Se cerchi incrementi netti di `answer_correctness`, questi richiedono tipicamente architetture più "sofisticate" e facenti riferimento all'idea di ensembling presentata nel capitolo sul CoT RAG: è con modelli specializzati al query rewriting, fine-tuning supervisionati per seguire uno schema di reasoning specifico o decomposizioni in sub-agents con ruoli distinti, che vedi davvero quanto un approccio CoT migliori le performance della pipeline, soluzioni che qui non adottiamo. D'altronde, l'intuizione è semplice: quando fu formulata, la CoT veniva prevalentemente indotta con il prompting, poi si passò a fasi di post-training dove ai modelli viene direttamente insegnato come ragionare, fino ad implementazioni più recenti di *Reinforcement Learning*, ed è lì che si ebbe l'incremento consistente per uno standard de facto, ad oggi, nei LM [Mukherjee et al., 2023, DeepSeek-AI et al., 2025]. Per questo, dicevo, è assolutamente significativo che le performance non degradino rispetto alla baseline Naive e conferma il punto centrale della tesi: il Naive RAG è sorprendentemente difficile da battere e rimane, nella pratica, la soluzione *go-to* più robusta in un generic domain.

Quest'ultimo aspetto sulla "migliore soluzione go-to" è confermato anche da analisi successive sui costi. Il chunking adottato è di tipo Recursive a finestra fissa di 300 token ed al modello, per ogni step di retrieval, vengono consegnati i top-5 chunk come contesto: aggiungendo a questo la lunghezza media dello scratchpad, il numero medio di step di adaptiveness valutati, il numero medio di token per follow-up query generata, possiamo ottenere stime abbastanza precise sull'aumento dei costi rispetto alla baseline classica; d'altronde, il CoT non è solo *moltiplicatore* di grounding, ma anche di costi!
I più attenti si accorgeranno che non c'è un fattore moltiplicativo sulla voce `Total Tokens`: questo viene dal fatto che i costi delle API dei maggiori LLM, ad oggi, sono dominate dai token di output; essendo che, quindi, un token di in-

Model	Metric	CoT	Classic	Factor
Kimi	Input tokens	5259	3060	1.7×
	Output tokens	267	62	4.3×
	Total tokens	5526	3122	
Llama	Input tokens	5382	3060	1.8×
	Output tokens	1261	173	7.3×
	Total tokens	6643	3234	

Tabella 7: Analisi dei Costi: Average Tokens - CoT RAG vs Classic

put non pesa come un token prodotto, ricavare dal rapporto trai due un fattore moltiplicativo non sarebbe stato necessariamente corretto e significativo dell'aumento dei costi. Usando i costi del pricing model di Groq, al momento della scrittura di questa tesi, otteniamo: un costo medio per query di $C_{\text{CoT}} \approx 0.00606\$$ contro $C_{\text{Classic}} \approx 0.00325\$$ per **Kimi** (fattore $\approx 1.87\times$), e $C_{\text{CoT}} \approx 0.00037\$$ contro $C_{\text{Classic}} \approx 0.00017\$$ per **Llama** (fattore $\approx 2.22\times$). Guardando a possibili miglioramenti dell'architettura CoT, come quelli citati precedentemente, questi non dovrebbero perturbare eccessivamente i costi, rimanendo confrontabili alla situazione attuale differentemente dalle performance: ovviamente, i costi crescono molto in base alla difficoltà delle domande, o meglio, al numero medio di volte che viene valutata l'adattività, tuttavia, da questi risultati, sembra come la direzione del tradeoff tra incremento possibile delle performance della pipeline e costo sia, in realtà, favorevole sopratutto per modelli di grandi dimensioni.

Chunk Size In assenza di un dominio specifico, diverse linee guida 'industriali' indicano intervalli di $200 - 400$ token come raccomandati; per i risultati precedenti ho, appunto, scelto 300 token come chunk size [Singh et al., 2024, Coveo, 2024]. Per studiare meglio quanto le performance della pipeline dipendano da quest'ultimo, ho ripetuto gli stessi esperimenti con, stavolta, una configurazione di **600 token**: raddoppio della baseline, chunk mediamente più auto-contenuti e contesto generalmente meno frammentato; la scelta di raddoppiare il chunk size è, d'altronde, abbastanza coerente a risultati sperimentali in letteratura, come quelli presentati da [Juvekar and Purwar, 2024], che propongono un range più ampio tra $512 - 1024$ token quando serve mediamente più contesto alla generazione.

Metric	Kimi Mean	Llama Mean	Δ Mean
answer_correctness	0.785	0.624	-0.161
faithfulness_to_context	0.878	0.661	-0.217
evidence_coverage	0.776	0.604	-0.172

Tabella 8: Confronto Kimi–Llama in CoT RAG (600 token)

I risultati con chunk da 600 token confermano il pattern già emerso nelle di-

scussioni precedenti: Kimi mantiene prestazioni sostanzialmente stabili, mentre Llama degrada in modo marcato e, soprattutto, sistematico su tutte le metriche del judge, con un gap particolarmente ampio su `faithfulness_to_context`. Aumentare il chunk size, cioè fornire più contesto e meno frammentato al modello, non "aiuta" Llama come ci si potrebbe aspettare intuitivamente, al contrario, amplifica le difficoltà nel selezionare ed estrarre l'informazione importante dal contesto recuperato e questo è assolutamente coerente con le difficoltà di comprensione testuale precedentemente esposte. Chunk più lunghi introducono più segnali, chiamiamoli, concorrenti (entità, eventi, descrizioni...) e richiedono una migliore capacità di identificare i chunk realmente rilevanti rispetto alla domanda, mantenere coerenza nella costruzione della risposta e, soprattutto, sintetizzare meglio più informazioni e più dettagli; il chunking a 600 token riduce la frammentazione, ma sposta il carico sul modello: serve maggiore compressione semantica. Il fatto che Kimi regga meglio questo regime, mentre Llama perda sia in `answer_correctness` sia in `evidence_coverage`, suggerisce che l'aumento di contesto diventa un vantaggio solo quando il modello ha capacità sufficienti per sfruttarlo, altrimenti la maggiore lunghezza si traduce in rumore aggiuntivo e in peggior grounding; concettualmente, le considerazioni precedenti, in realtà, si basano direttamente sull'idea che più contesto sia difficile da gestire per uno SLM: di per sé, però, non stiamo isolando l'effetto dell'adattività, quindi dell'ulteriore capacità di comprensione testuale che richiediamo al modello. Le analisi successive, infatti, vogliono rispondere alla domanda: *è l'adattività o l'aumento di token che pesa allo SLM?*

I risultati precedenti rinforzano, d'altronde, una delle sfumature centrali di questo lavoro: in un setting zero-shot e generic-domain, il chunk size non è un solamente un iperparametro "universale" della pipeline, ma interagisce fortemente con la qualità del modello scelto. Un chunk più grande può rendere il retrieval più coerente per testi narrativi magari, ma rende anche più evidente la differenza tra un LLM in grado di sfruttare contesto lungo e uno SLM che fatica ad isolare le informazioni rilevanti, anche quando queste sono effettivamente presente nei chunk recuperati. Nel setting di riferimento per CoT RAG, in realtà, raddoppiando il chunk size hai un incremento proporzionale dei costi, quindi i 300 token scelti inizialmente sono una scelta molto più sensata.

Metric	**Kimi Mean**	**Llama Mean**	**Δ Mean**
answer_correctness	0.824	0.729	-0.095
faithfulness_to_context	0.841	0.762	-0.079
evidence_coverage	0.816	0.697	-0.119

Tabella 9: Confronto Kimi–Llama in Naive RAG (600 token).

L'incremento dei costi, invece, è assolutamente supportato dal gain sulle performance in Tabella 9: aumentando la dimensione del chunk, diminuisce la frammentazione e cresce la probabilità che l'informazione richiesta sia contenu-

ta nello stesso passaggio recuperato, i chunk sono più auto-contenuti insomma. Questo effetto è particolarmente evidente per **Kimi**, che mostra un incremento netto sia in `answer_correctness` sia in `evidence_coverage`: un modello più capace riesce a sfruttare meglio chunk più auto-contenuti, avendo maggiore capacità di gestire l'informazione. In realtà, dall'analisi delle performance di **Llama** un beneficio esiste: sebbene avevamo discusso di come Llama abbia una minore capacità di Kimi nella comprensione testuale, la crescita del chunk size migliore la qualità delle risposte prodotte dal modello su tutti i fronti; questo non è un risultato che contraddice le discussioni precedenti, anzi, permette di evidenziare un punto importante: nel Naive RAG, un contesto più lungo può essere vantaggioso perché, di base, riduce il bisogno di adattività disponendo di più informazione, mentre nel CoT RAG questo diventa un problema in relazione al fatto che occorre una comprensione *dinamica*, cioè la capacità di mantenere coerenza lungo più step, aggiornare correttamente uno stato (scratchpad), riconoscere cosa manca rispetto all'information need e trasformarlo in una `next_query` utile senza introdurre drift. L'aumento dei token, infatti, non è un problema necessariamente per uno SLM fin tanto che sia necessario 'statismo' nella comprensione testuale: l'adattività richiede capacità superiori di comprensione semantica e non solo di 'trovare' strettamente l'informazione nel testo, moltiplicando i punti decisionali e rende il modello più error prone. Un LLM come Kimi tende a beneficiare dell'adattività perché riesce a controllarla, mentre un SLM come Llama può peggiorare non perché "vede troppi token", ma perché fatica a gestire la dinamica del processo e questo spiega la covarianza negativa tra incremento di performance nel Naive a all'aumentare dei token e decremento prestazionale nel CoT per Llama, risponendo alla domanda posta precedentemente.

Presentiamo, ora, i risultati sull'approccio graph-based; in teoria, questo tipo di pipeline cattura molto bene la semantica complessiva del corpus e permette a diversi chunk, intra o cross - document, di interagire e rispondere anche a domande caratterizzate da contesti con hop documentali. Nel caso di GraphBench, però, i risultati mostrano criticità abbastanza nette: l'approccio gerarchico è *sistematicamente* il peggiore tra quelli testati, soprattutto su `answer_correctness` ed `evidence_coverage`. In Tabella 10 riporto le metriche stimate per Kimi e Llama.

Metric	**Kimi Mean**	**Llama Mean**	**Δ Mean**
answer_correctness	0.345	0.342	-0.003
faithfulness_to_context	0.771	0.509	-0.262
evidence_coverage	0.331	0.322	-0.009

Tabella 10: confronto Kimi–Llama in GraphRAG

Il fatto che sia sistematicamente il peggiore non necessariamente vuol dire che l'approccio non sia viabile, anzi, i vantaggi di pipeline gerarchiche sono chiare, tuttavia molto dipende dalla tipologia di queries testata ed i concetti, prece-

dentemente discussi, sull'information loss dei summaries; difatto i risultati sono perfettamente coerenti con l'aspettativa.

Il collo di bottiglia non è il retrieval in senso stretto, ma la *compressione*: i riassunti a livelli successivi introducono inevitabilmente perdita di dettaglio e, peggio, rischiano di "riscrivere" l'informazione (semplificazioni, generalizzazioni, omissioni...) e questo è un problema fondamentale quando si sfruttano moduli di rollup; l'impatto è diretto sull'`evidence_coverage`: anche quando il sistema recupera nodi "vicini" alla query, la catena di summarization rende più difficile ottenere tutte le evidenze necessarie a coprire l'information need. Interpretando meglio le metriche: nonostante l'attinenza al contesto sia comparabile a quella ottenuta dalle altre strategie, la correttezza, ai sensi di qualità della risposta, cala drasticamete e ciò è dipeso dal fatto che il modello riceve chunk sì semanticamente simili, ma appartenenti a contesti diversi e cerca di collegare, magari, fatti o avvenimenti plausibili rispetto all'information need, di fondo ottenendo un qualcosa che rimane ancorato al contesto ma che un judge considera praticamente alla pari di allucinazioni. Qui viene una nota interessante: *ha veramente senso far interagire tra di loro documenti diversi con l'espansione?* Concettualmente sì, ma il rovescio della medaglia è dato proprio dal fatto che i chunk rimangono comunque porzioni locali del documento e, quindi, i fatti espliciti potrebbero essere considerati plausibili rispetto alla query ma, come suddetto, riferenti ad un universo distinto; da qui iniziamo già a capire il concetto fondamentale e che risponde ad uno dei miei obiettivi di ricerca: l'interazione cross-document è un obiettivo assolutamente ambizioso ed importante da perseguire, tuttavia diventa molto meno error prone quando lo fai passando tramite una entity che riconosci essere la stessa tra due documenti.

L'altra faccia della medaglia è che GraphRAG Bench, per costruzione, contiene molte domande con entità chiaramente identificabili e un information need spesso risolvibile con un match relativamente diretto sui chunk originali: questo è esattamente lo scenario in cui il Naive RAG (top-k su embedding) è sorprendentemente competitivo; il grafo, invece, aggiunge un livello di astrazione che non porta ragionevole vantaggio al retrieval in questo caso, spesso omettendo le entities nei community summaries; come suddetto, non vuol dire che necessariamente questo sia un problema, anzi, per come l'abbiamo costruito è una feature: tecnicamente le performance diventerebbero competitive con query più ambigue, richiamando il concetto che GraphRAG Bench è costruito prevalentemente per KGs con entities. Un dettaglio interessante è la divergenza su `faithfulness_to_context`: Kimi rimane relativamente alto (0.771), mentre Llama collassa (0.509); questo è coerente con il comportamento discusso in precedenza per gli altri approcci: Llama è più carente nella comprensione testuale e di base ti aspetti performance peggiori di Kimi, tuttavia avevamo discusso di come, se non forzato a ragionare, possa comunque ottenere performance accettabili a parità di contesto. In questo caso, il collasso della faithfulness è principalmente relativo a quello che dicevamo precedentemente: chunk non solo diversi, ma l'interazione porta ad ottenere una percentuale maggiore rispetto

agli altri approcci di chunk cross-document e, di conseguenza, fatti semanticamente rilevanti ma appartenenti ad un contesto molto diverso *favoriscono* le allucinazioni di Llama, amplificando una comprensione testuale già non perfetta. Modelli piccoli tendono, quindi, a considerare fattuali informazioni che sembrano rilevanti, con meno capacità di ragionamento rispetto all'intento effettivo della domanda e, come suddetto, è perfettamente coerente con l'intuizione che veniva dal CoT RAG; d'altronde, questo, a mio avviso, è un fenomeno molto comune su basi documentali di grandezza reale e ragionevolmente eterogenee, cosa che il mio setup, come ampiamente citato, simula in maniera imperfetta.

Per corroborare le considerazioni precedenti, ho effettuato una riscrittura **ambiguous** di un sottoinsieme di 100 queries tra le 350 testate in precedenza: il ragionamento si concentra prevalentemente su entità, costruendone descrizioni indirette che mantengano fisso l'intento della domanda e l'information need, ma senza esplicitare direttamente la entity stessa; in questo caso, la riscrittura avvicina di molto la forma delle queries di HotpotQA a quella di NQ, quindi più allineate ad un motore di ricerca general purpose; questo è, di fatto, un regime più coerente con l'uso di pipeline graph-based: se la domanda non "punta" esplicitamente all'entità corretta, l'aggregazione semantica sul grafo, data dai community summaries, diventa, almeno in principio, più vantaggiosa rispetto al setup precedente.

I risultati sul sottoinsieme di 100 query riscritte sono riportati in Tabella 11 e mostrano un miglioramento netto e consistente per entrambi i modelli su tutte le metriche considerate.

Metric	**Kimi (amb)**	**Δ Kimi**	**Llama (amb)**	**Δ Llama**
answer_correctness	0.609	+0.264	0.485	+0.143
faithfulness_to_context	0.886	+0.115	0.584	+0.075
evidence_coverage	0.599	+0.268	0.465	+0.143

Tabella 11: Confronto GraphRAG standard - ambiguous rispetto a Tabella 10

L'incremento più marcato riguarda `answer_correctness` ed `evidence_coverage`, il che conferma l'intuizione precedente: rispetto ai risultati in Tabella 10, la copertura cresce in modo sostanziale, suggerendo che il grafo riesce più spesso a costruire un *narrowed corpus* utile senza un segnale chiaro sulla presenza della entity. Anche la `faithfulness_to_context` migliora: è un risultato interessante, poiché da un lato non contraddice le considerazioni precedenti e, dall'altro, permette di evidenziare un ulteriore aspetto che viene da questo nuovo setup; l'incremento sulla qualità della risposta suggerisce che i chunk restituiti dal retrieval sono semanticamente più allineati alla query rispetto al caso precedente ed, in questa situazione, sembra che entrambi i modelli tendano a rimanere più ancorati al contesto fornito e a costruire risposte complessivamente migliori. Il fenomeno va però interpretato meglio: a mio avviso, nel RAG i modelli mostrano un bias generale verso il contesto recuperato, ciò non vuol dire che sono

tendenti ad utilizzarlo, questo è esattamente quello che vuoi, ma più sul fatto che considerano tutti i chunk 'rilevanti' allo stesso modo; di conseguenza, anche porzioni rumorose o, addirittura, distrattori possono influenzare molto la generazione: proprio in questo senso, l'aumento di `faithfulness_to_context` non implica necessariamente una migliore capacità di comprendere cosa sia pertinente rispetto all'intento della domanda, indica piuttosto che, mediamente, il contesto è diventato più utile e meno ambiguo, rendendo le informazioni che il modello infierisce meno tendenti ad allucinazioni vere e proprie; difatti, rimane comunque possibile (e frequente) che il modello infierisca informazioni plausibili ma non supportate in modo diretto dalle evidenze rilevanti, ossia fuori contesto rispetto all'*information need*. Questa evidenza richiama direttamente il tema del *rewriting* come strumento per produrre query più allineate alle proprietà dell'architettura che scegli e rendere, appunto, la scelta vincolata di *fissare* una distribuzione attesa molto molto più semplice.

Sarebbe stato naturale valutare anche un approccio *hierarchical graph-based* in cui la query iniziale venisse prima riscritta in una forma *step-back*, come discusso nelle sezioni precedenti, usata per selezionare comunità / macro-contesti e ridurre il rischio di information loss dovuta ai rollup; successivamente, la query iniziale, più specifica, verrebbe applicata su un *narrowed corpus* ragionevolmente migliore e questo possiamo dirlo sulla base dei risultati empirici precedenti. Ancora una volta, rimarchiamo l'intuizione fondamentale: la bontà di un'architettura RAG è estremamente dipendente dal retrieval e direttamente, quindi, dalla forma delle queries attesa; il fatto che un'architettura in un dato setup non sembri viabile poiché in underperformance rispetto ad altre alternative, in realtà, non vuol dire che tu non possa renderla competitiva aggiungendo moduli a priori, come il *query rewriting* insomma, che è la considerazione fondamentale di questa sezione.

5.4.3 HotpotQA

Per estendere l'analisi ad un benchmark più 'challenging' rispetto alla porzione `Novel` di GraphRAG Bench, ho utilizzato **HotpotQA** [Yang et al., 2018]: dataset di question answering su Wikipedia progettato esplicitamente per il *multi-hop reasoning*. Oltre a fornire una *gold answer* per ogni domanda, include *supporting facts* a livello di frase e tipologie di quesiti (es. *bridge* e *comparison*) che richiedono di combinare evidenze da più documenti. La differenza sostanziale rispetto a GraphRAG-Bench non è tanto la forma della query, infatti rimangono simili anche in HotpotQA con, spesso e volentieri, entities ed information need chiari, bensì il fatto che la risposta non è tipicamente recuperabile con un reasoning semplice: nelle domande *bridge*, ad esempio, l'entità citata funge da *ancora* per individuare un'entità ponte in un primo documento e recuperare un secondo documento contenente il fatto risolutivo e che, d'altronde, è uno dei prompt tunings che avevamo discusso precedentemente con il CoT; questo è un punto importante: la facilità di GraphRAG Bench non è unicamente data dalla forma delle queries, d'altronde per poter valutare un sistema di retrieval devi fissare

information need chiari, anche se questo rimane un problema da non trascurare, infatti ad inferenza potresti avere situazioni molto diverse e con maggiore ambiguità, ma più che altro dal fatto che la maggior parte viene risolta già con il primo step di retrieval e l'adattività non viene valutata sempre: HotpotQA è diverso proprio in questo senso. Nelle domande *comparison*, per esempio, è necessario recuperare e confrontare informazioni da due pagine distinte ed, in questo setting quindi, la componente di retrieval pesa in modo sostanzialmente più forte end-to-end: il sistema deve portare tra i primi risultati (e poi nel contesto) *tutti* gli 'hop' richiesti.

Come già discusso, la stragrande maggioranza dei benchmark non propone il testo 'raw' da indicizzare ed HotpotQA non è da meno: per questo, nella costruzione del dataset, ho valutato una fase di scraping sfruttando l'API di Wikipedia, chunkando ed indicizzando il testo completo. Mi sono concentrato sulla porzione 'validation' di HotpotQA, fissando 400 esempi: questo è comprensivo solo delle queries labeled 'hard', quindi il massimo grado di difficoltà. Probabilmente, questo è il momento migliore per discutere di un concetto fondamentale riguardo alcune criticità del mio setup: la costruzione di un dataset per testare architetture RAG o strategie di chunking è un compito molto complesso, proprio perché vorresti avere, per ogni query in ingresso, un corpus che renda progressivamente difficile soddisfare l'information need sulla base della classe di complessità della query e questo puoi farlo solo se cerchi di 'ingannare' il retrieval denso con un un sottoinsieme significativo di distrattori; questi amplificano l'ambiguità intrinseca della domanda e rendono più chiaro il gap perstazionale tra diverse strategie di chunking o architetture RAG. La verità, soprattutto a livello industriale, di fondo l'ho parzialmente già spiegata: quando hai un dominio chiaro, specifico, nel quale vuoi ottimizzare, allora tipicamente hai anche chiaro quali sono mediamente le richieste che vengono fatte, in che modo vengono poste, cosa ti serve per ottimizzare, le tipologie di documento... quindi, in applicazioni domain-specific tipicamente è molto più semplice avere dataset rappresentativi e che ti permettono di ricavare statistiche significative per quello che è il tuo uso ad inferenza, differentemente dal mio setup sperimentale che, in questo senso, è molto più complesso. Quando dico che le metriche precedenti sono comunque signficative, nonostante il corpus sia caratterizzato dalle criticità precedenti, faccio riferimento al fatto che: se vuoi comparare due strategie di chunking, piuttosto che architetture RAG, e lo fai su uno stesso corpus, benché questo abbia delle inefficienze sai che entrambe le architetture/strategie sono sottoposte alle stesse condizioni; le metriche che ottieni possono essere arbitrariamente gonfiate, saturanti, gap ridotto rispetto a quello reale... tuttavia rimane chiara la direzione: se una strategia di chunking o un'architettura performa meglio su un certo corpus rispetto ad un'altra, allora le metriche che ottieni possono non essere significative rispetto al caso reale ad inferenza o il gap che ti aspetti può essere più o meno marcato, ma rimane chiara la direzione del vantaggio e le considerazioni che ne derivano, quali possono essere più o meno marcate all'effettivo. Un'analogia interessante è quella del confronto tra due modelli di ML: se lo fai su un dataset non o parzialmente rappresentativo, allora le

metriche che ottieni possono essere sovra/sottostimate rispetto all'utilizzo reale ad inferenza, tuttavia se un modello è migliore di un altro a parità di handicap lo sarà anche su un dataset più rappresentativo; in questo senso è la 'direzione' che conta. HotpotQA è un benchmark che si collega benissimo con questi discorsi, da cui capiamo il perché ho deciso di usufruire dei dei distrattori che fornisce: per ogni query hai non solo documenti *gold* contenenti l'informazione rilevante, ma anche una lista di distrattori; proprio quest'ultima è necessaria per costruire un corpus più rappresentativo, ai sensi del fatto che, seppur grande, le dimensioni di un corpus non sono necessariamente significative della difficoltà del retrieval. Fissato, quindi, il target di 400 query sopracitato, ho imposto un vincolo di distribuzione equilibrata tra documenti gold e distrattori, ottenendo in media, per ciascuna query, lo scraping di 4 documenti Wikipedia grezzi.

Su HotpotQA ho valutato solo **Naive RAG** e **CoT RAG** e non l'approccio graph-based: questo sarebbe stato esposto agli stessi failure mode osservati su GraphRAG Bench (perdita informativa e distorsione dovuta a summarization), con l'aggravante che su HotpotQA la copertura delle evidenze è ancora più critica, difatti se perdi uno degli hop, la risposta diventa irrecuperabile e, rimarchiamo ancora, non è strettamente un problema di architettura ma di forma delle query. In Tabella 12, riporto i risultati della valutazione LLM sui 350 esempi costruiti.

Metric	**Kimi Mean**	**Llama Mean**	**Δ Mean**
answer_correctness	0.731	0.556	-0.176
faithfulness_to_context	0.903	0.667	-0.236
evidence_coverage	0.720	0.541	-0.179
CoT steps (mean)	1.697	1.654	-0.043

Tabella 12: HotpotQA in CoT RAG, confronto Kimi–Llama.

Il pattern osservato è coerente con quanto emerso su GraphRAG Bench, ma "amplificato" dalla maggiore complessità media delle query:

- il gap Kimi–Llama cresce e rimane marcato su tutte le metriche, soprattutto su `faithfulness_to_context`;
- la difficoltà del benchmark si riflette in un calo di `answer_correctness` rispetto a GraphRAG Bench (dove molte query si risolvevano single-shot), perché qui le risposte hanno bisogno di maggiore contesto e capacità di reasoning sullo stesso

La scomposizione per *bridge* e *comparison* rende ancora più chiaro *dove* il modello piccolo fatica:

- sulle domande **bridge** (280 esempi) il divario è molto alto su tutte le metriche: qui il modello deve identificare un'entità ponte e formulare un hop

successivo; errori di pianificazione o comprensione si propagano direttamente in perdita di coverage e degradano anche la correttezza. D'altronde, si noti come, qui, l'adattività viene valutata allo stesso modo da entrambi i modelli, prima invece avevamo maggiori step per Kimi, sintomo del fatto che l'adattività è più chiara ad entrambi grazie all'intento bridge

- sulle domande **comparison** (70 esempi) il gap si riduce: è un pattern sensato, perché spesso la domanda esplicita già le due entità da confrontare, riducendo il rischio di query drift e rendendo l'operazione più "meccanica" (recupera due pagine, estrai attributi, confronta). Rimane comunque un vantaggio netto di Kimi su `faithfulness_to_context`, segnale che Llama tende ancora a completare con fatti non necessariamente supportati

Come suddetto, le medie sugli step (1.697 vs 1.654) sono molto vicine: questo indica che, su HotpotQA, l'adattività viene attivata con una frequenza simile da entrambi i modelli, ma ciò non implica che venga *gestita* con la stessa qualità; il punto, infatti, non è "quanti" hop fai, ma *quanto sono utili* le follow-up queries e quanto il modello riesce a mantenere coerenza con l'information need iniziale, ma le considerazioni sono sostanzialmente le stesse.

Rispetto ai risultati di GraphRAG Bench, HotpotQA è esattamente il caso in cui il CoT RAG dovrebbe essere più giustificato: la baseline single-shot ha meno probabilità di portare nel top-k *tutti* gli hop necessari, mentre l'adaptive retrieval può correggere il tiro facilmente. Detto questo, le conclusioni rimangono compatibili con quelle precedenti: l'adattività non è una "scorciatoia" universale e, soprattutto, dipende fortemente dalla capacità del modello ed è, infatti, rischioso per LMs di piccole dimensioni. Per rendere questo punto più concreto, ho misurato anche la baseline **Naive RAG** ed i risultati sono presentati nella tabella che segue. Il risultato più rilevante non è solo che Kimi rimane sistematicamente superiore a Llama (pattern ormai consolidato anche su GraphRAG Bench), ma che su HotpotQA il Naive RAG è una baseline *molto forte* già di per sé: `answer_correctness` ed `evidence_coverage` risultano praticamente allineate (0.729 vs 0.727), mentre `faithfulness_to_context` rimane molto alta (0.897), seggerendo che, nel corpus costruito, il top-k single-shot riesce spesso a recuperare una quantità di evidenze sufficiente per sostenere una risposta corretta e ben ancorata al contesto anche su un benchmark simile, ma più complesso.

Metric	**Kimi Mean**	**Llama Mean**	**Δ Mean**
answer_correctness	0.729	0.546	-0.183
faithfulness_to_context	0.897	0.657	-0.240
evidence_coverage	0.727	0.531	-0.196

Tabella 13: HotpotQA in Naive RAG, confronto Kimi–Llama.

Confrontando questi numeri con quelli del CoT RAG (Tabella 12), emerge una sfumatura importante che replica quanto visto su GraphRAG Bench ma in un re-

gime multi-hop: per Kimi lo scratchpad e l'adattività non portano un incremento netto della correttezza o della coverage, che rimangono sostanzialmente stabili, ma tendono a migliorare il *grounding*, con una `faithfulness_to_context` leggermente più alta. Per **Llama**, invece, la dinamica rimane quella già osservata: il modello è più fragile nel mantenere coerenza e grounding quando il processo introduce passaggi di reasoning aggiuntivi; l'adattività aumenta le opportunità di drift e quindi non garantisce, in media, un miglioramento rispetto alla baseline single-shot.

HotpotQA conferma, d'altronde, due aspetti interessanti: l'adaptive retrieval è concettualmente più sensato quando l'information need richiede davvero multi-hop, ma la forza della baseline Naive rimane molto chiara anche in questo setting quando il retrieval è ben impostato e il modello ha buona comprensione testuale; per questo motivo, il CoT RAG va interpretato più come un *upgrade mirato* ed utile per migliorare grounding o recuperare hop mancanti, tuttavia non come una sostituzione general-purpose della baseline.

5.5 Soluzione Go-To

Il risultato trasversale che emerge dai benchmark è che, in un contesto **generic-domain** e **general-purpose**, la pipeline più difficile da battere rimane quella più semplice: **Naive RAG**, con scelte "corrette" di chunking sensato, top-k adeguato, prompt minimale ma robusto... ed un modello che sappia comprendere bene il contesto. Questo è, di fatto, l'obiettivo centrale della tesi: mostrare che un approccio semplice ma ben impostato è competitivo e stabile, mentre le architetture più sofisticate tendono a pagare overhead maggiori, in riferimento al tradeoff costo-performance, senza un guadagno proporzionato nel setting considerato.

È importante, tuttavia, rimarcare un punto metodologico discusso già precedentemente: *non esiste*, ad oggi, uno standard de facto per la valutazione di sistemi RAG; a differenza di altri contesti, come quelli della valutazione stessa di LMs, nel RAG la pipeline end-to-end introduce una molteplicità di variabili difficilmente controllabili in modo comparabile: scelta del corpus e sua "pulizia", schema di indicizzazione, chunking e overlap, retriever e reranker, prompt e formati di output, budget di contesto, criteri di arresto in setting adattivi e perfino la definizione stessa di "correttezza" in presenza di risposte multi-evidenza. Cambiare anche una sola di queste componenti può spostare sensibilmente il tradeoff costo-performance, rendendo complicato costruire confronti pienamente fair. In questo senso, i benchmark utilizzati nel mio lavoro riflettono una scelta che sei costretto a fare a priori: *sono prevalentemente benchmark di retrieval* o, quantomeno, benchmark in cui l'information need è mediamente chiaro e, come suddetto, spesso *entity-centric*; questo significa che le domande non coprono tutta la variabilità del caso reale: in un sistema general-purpose, infatti, l'utente può formulare query ambigue, underspecified, con vincoli impliciti o con intenti difficili da comprendere chiaramente. Inoltre, il "ground truth"

può diventare anch'esso intrinsecamente ambiguo: non esiste sempre una singola risposta corretta e molto della bontà delle risposte è a discrezione di chi le interpreta. Il problema, però, è di carattere proprio *formale*: definire etichette affidabili per information need realmente ambigui è estremamente difficile, perché richiede annotazioni molto precise e, di base, fortemente caratterizzate dalle tendenze dell'annotare, quindi con un segnale non chiarissimo relativo, spesso, a disagreement strutturale tra annotatori diversi, rendendo meno chiaro cosa si stia misurando.

Detto questo, la scelta di benchmark con information need più chiari, come quelli usati, è anche **giustificata** rispetto all'obiettivo della tesi: se l'intento è ricavare in modo credibile i tradeoff architetturali tra approcci (Naive vs CoT vs Graph/gerarchico), occorre ridurre il più possibile le fonti di variabilità non controllata e fissare un set di query che permetta comparazioni più stabili; per riprendere il concetto fondamentale: *valutare un sistema di retrieval vuol dire fissare a priori la tipologia di query.* Difatti, se le query fossero troppo ambigue, diventerebbe quasi impossibile distinguere se una differenza di performance architetturale dipenda da un limite del retriever, da un mismatch delle valutazioni del judge rispetto alla "gold answer", oppure da altri fattori nascosti. Questo è il motivo per cui benchmark come quelli adottati, pur non essendo totalmente rappresentativi della variabilità del mondo reale, possono risultare comunque sufficienti per isolare fenomeni strutturali: ad esempio, il fatto che la gerarchia a summary perda coverage, che l'adattività amplifichi drift nei modelli piccoli o che il Naive RAG sia estremamente competitivo quando la domanda è ben specificata rafforzano questo aspetto.

Una volta fissato un regime in cui l'information need è sufficientemente definito da rendere confrontabili le architetture, il Naive RAG emerge come soluzione *go-to*, ottenendo un equilibrio particolarmente favorevole tra qualità e costo e, d'altronde, in modo più stabile alle perturbazioni della pipeline rispetto alle altre architetture (prompting, chunking, modello). Importante sottolineare che il Naive RAG non è "sempre" migliore, ma è quello per cui, nel setting considerato, emergono proprietà che lo rendono preferibile soprattutto in contesti industriali pratici; semplicità, cost efficiency e performance ragionevoli sono esattamente ciò che si desidera in un sistema general-purpose. Questa intuizione si concretizza, poi, in tre proprietà chiave:

- **Robustezza:** quando l'information need è entity-centric e risolvibile single-shot, il Naive RAG è estremamente competitivo; in domini con una distribuzione più ambigua di query vorresti, magari, spingere di più sulla componente densa e puoi farlo cambiando il peso nello score combinato o, addirittura, ragionando su fine - tuning del retriever nel caso in cui disponi di un contesto chiaro di ottimizzazione
- **Costo e semplicità:** non introduce step iterativi e, quindi, minimizza il costo per token e punti di failure rispetto ad altre architetture. Questo lo

rende la scelta naturale per workload reali, dove latenza e costo contano almeno quanto la qualità media.

- **Comportamento prevedibile:** meno componenti ⇒ meno accoppiamenti; nel mio caso, l'approccio GraphRAG gerarchico peggiora proprio perché aggiunge una trasformazione (summarization) che interagisce negativamente con la natura delle query, analogamente, il CoT RAG diventa un moltiplicatore utile solo quando il modello è sufficientemente capace da controllare la dinamica iterativa, altrimenti amplifica query drift

Operativamente, se l'obiettivo è costruire un sistema RAG affidabile, la strategia più sensata, a mio avviso, è la seguente:

1. partire da un **Naive RAG** con **Recursive Chunking** come soluzione *go-to*: semplice, robusto, economico
2. ottimizzare gli iperparametri "low-level" ad alto impatto, come chunking, top-k di contesto...
3. valutare l'introduzione di **CoT/Adaptive Retrieval** solo quando l'analisi del workload mostra una reale carenza di contesto e solo se si può disporre, in relazione ai vincoli sui costi, di un modello abbastanza forte da controllare la dinamica iterativa, oltre che valutare scelte di specializzazione sul dominio, se possibile, per migliori follow-up queries
4. evitare architetture gerarchiche chunk-based quando la distribuzione attesa delle query è prevalentemente entity-centric; alternativamente, in contesti con un dominio chiaro, l'approccio KG-based potrebbe portare vantaggi assolutamente rilevanti

Nel setting generic-domain considerato, la soluzione "go-to" non è quella più complessa, ma quella più *robusta* ed i risultati sperimentali mostrano chiaramente che battere un Naive RAG ben costruito è, nella pratica, sorprendentemente difficile.

5.6 UI Streamlit

È possibile interagire con tutte le architetture RAG tramite una semplice UI *Streamlit* disponibile nella repository GitHub a supporto della tesi (`https://github.com/ddemarchis11/TesiMagistrale_AdaptiveHierarchicalRAG`). L'utente può selezionare la pipeline RAG tramite un menu multi-choice, oltre che tarare parametri di esecuzione ad alto impatto come: numero di chunk consegnati al modello dal retrieval, numero di step CoT... dando una maggiore interazione con le componenti più tecniche dell'architettura. Come ampiamente trattato, il concetto fondamentale del RAG è favorire trasparenza rispetto alle fonti utilizzate, difatti l'interfaccia permette di visionare direttamente i chunk recuperati e, opzionalmente, informazioni di debug utili per analizzare il comportamento del retriever e la qualità del contesto fornito al modello.

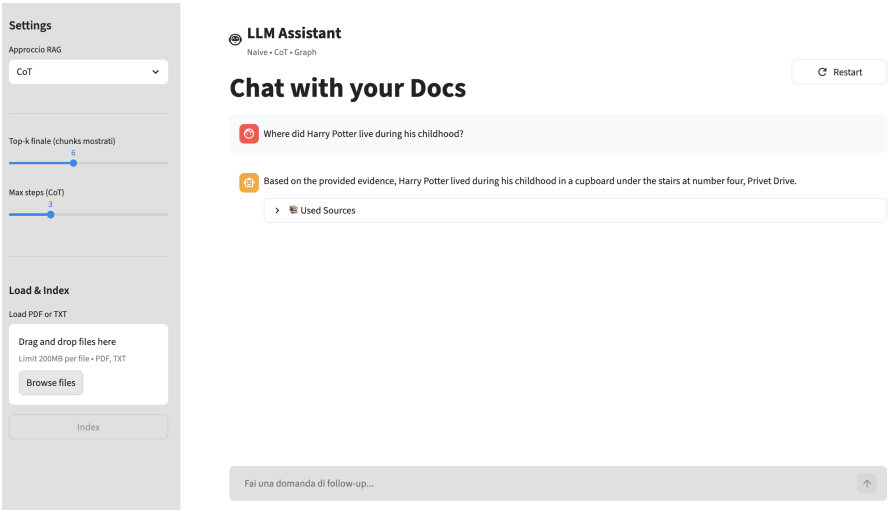

Figura 4: Layout UI Streamlit - CoT Architecture

5.6.1 Text Extraction

La UI ci permette, d'altronde, di discutere di una fase fondamentale per progetti di RAG 'reali': la **Text Extraction**; l'utente può caricare file in formato `.pdf` o `.txt`, questi verranno, poi, indicizzati su Elasticsearch e resi, difatto, interrogabili. La mia idea era, appunto, mostrare come le UI non debbano soltanto essere "front-end" di QA, ma soprattutto un punto di ingresso per caricare i documenti stessi in modo autonomo, basandomi su formati molto semplici e che non richiedono eccessivo effort in progettazione. L'importanza della Text Extraction è stata già discussa, soprattutto in relazione a strategie di Chunking basate sulla struttura del documento e non ha molto senso ripeterci ancora. Per i PDF, l'estrazione può essere eseguita con librerie Python abbastanza semplici come `pypdf`, che offrono metodi diretti per ricavare il contenuto testuale; tuttavia, questa soluzione "semplice" è spesso insufficiente quando l'obiettivo è preservare la struttura logica o, più generalmente, il layout del documento: intestazioni, paragrafi, colonne, tabelle, note a piè di pagina... possono venire riordinate o mescolate con il testo da approcci semplici come `pypdf` e questo viene dal fatto che il PDF è principalmente un formato orientato al layout grafico più che alla semantica del contenuto [Atagong et al., 2025, Ramakrishnan et al., 2012].

Preservare la struttura non è un dettaglio puramente estetico: determina direttamente la qualità del testo indicizzato. Nei PDF con layout complessi (multicolonna, tabelle, moduli), un'estrazione "plug-and-play" tende a mescolare testo ed elementi strutturali, producendo chunk in cui contenuto utile e rumore (righe di tabella, campi di form, intestazioni ripetute...) si sovrappongono, degradando sia il retrieval (dense, full-text o ibrido) sia la generazione. Se la struttura

non viene mantenuta o ricostruita, diventa infatti difficile applicare chunking mirato (per sezioni, blocchi o tabelle), che spesso è una soluzione estremamente semplice ed efficiente in contesti domain specific (vedi manuali, benchmark legali...). Questo collega naturalmente l'estrazione a tecniche OCR e *document understanding*: avendo parlato delle SDK di Azure, infatti, per popolare lo Storage potresti sfruttare *Azure AI Document Intelligence*, il precedente Form Recognizer), che include modelli *Read* per OCR e modelli *Layout* per l'analisi della struttura del documento, con capacità di estrarre non solo testo ma anche tabelle ed altri elementi strutturali, offrendo possibilità sia di utilizzare modelli pre-trained su documenti molto comuni (ricevute, bilanci...) che di fine-tunings degli stessi sul tuo scenario [Microsoft, 2025, Microsoft, 2024].

Nella repository a supporto del progetto, più specificatamente sotto `ui_data/uploads`, si trova anche un documento che ho utilizzato per testare l'interazione con la piattaforma: mi riferisco ad uno degli ultimi balance sheet di Microsoft; in documenti di questo tipo è tassativamente importante la formattazione: layout, tabelle, struttura... è fondamentale avere modi per mantenerle integre durante l'indicizzazione e, soprattutto, rendere l'informazione interrogabile in maniera efficiente rispetto alla tipologia del dato, aspetto che discuteremo nella parte conclusiva della tesi. Provando il sistema sul balance sheet, guardando non solo alle risposte prodotte ma anche ai chunk retreieved, vedi già con poche query che: informazioni puntuali contenute all'interno di tabelle vengono, spesso e volentieri, male interpretate dal modello stesso, mentre informazioni contenute in testo continuo vengono organizzate molto meglio. Questo non è un problema di allucinazione del modello, ma proprio di chunking e Text Extraction: quest'ultima fase è implementata in maniera molto semplice e senza modelli di OCR complessi, in questo senso il chunking rimane agnostico alle strutture che possono trovarsi 'annidate' all'interno del testo che spezza, producendo veri e propri artefatti che influiscono direttamente sul decremento della qualità delle risposte; ad esempio: non mantenendo la struttura della tabella, un chunk potrebbe contenere il metadato di riferimento per l'attributo e il valore effettivo in due chunk distinti, andando completamente a decontestualizzare quest'ultimo e, di fondo, questa è uno dei failure mechanisms principali in questo setup semplice.

5.6.2 Incrementalità

Un aspetto fondamentale, già parzialmente discusso, delle architetture proposte riguarda l'**incrementalità** e, più in generale, la manutenibilità al variare delle dimensioni del corpus. A livello implementativo, tutte le architetture supportano un'indicizzazione sia in modalità **batch**, partendo da un insieme ben definito di documenti, sia in modalità **incrementale** per singolo documento, coerente con lo scenario applicativo della UI che ho voluto modellare, in cui l'utente può caricare documenti in momenti differenti. Per Naive e CoT RAG non c'è molto da dire: l'idea dell'architettura non sfrutta direttamente strutture gerarchiche con cui organizza il corpus, di conseguenza l'aggiunta dinamica di nuovi documenti e nuovi chunk non affligge direttamente la bontà della procedura, anche se

diventi progressivamente più soggetto ad Hubness, Semantic Collapse... ma questo non necessariamente è uno svantaggio, appunto perché, come discusso, non fai nulla a priori per limitarli. Discorso opposto per l'approccio gerarchico, dove una struttura c'è e l'incrementalità può essere difficile da gestire: richiamando quanto detto nella sezione relativa, ogni documento può essere sì processato autonomamente, tuttavia il collo di bottiglia emerge nel livello successivo, ovvero nella costruzione e mantenimento degli archi `COMM_SIMILAR_TO` tra communities; l'approccio KNN sullo spazio degli embedding dei summaries richiede non solo di connettere le nuove communities al grafo globale già presente, ma potenzialmente di aggiornare anche il vicinato di communities precedenti: nuovi punti nello spazio vettoriale possono infatti entrare nei top-k di nodi pre-esistenti, modificando parzialmente le scelte precedentemente fatte dall'algoritmo. Nel prototipo, non ho implementato meccanismi di **trigger periodico** o di ricalcolo automatico del KNN per questo ultimo livello: le connessioni cross-community che vengono formate ad un certo passo e con un corpus di dimensione N saranno le stesse ad $N + k$; più semplicemente: non ho implementato meccanismi di mitigazione per questo fenomeno, ai sensi del fatto che l'obiettivo primario non è quello di costruire un prodotto ma di valutare l'architettura, quindi mi sembrava più corretto discutere a livello teorico di possibili mitigazioni.

Se si passasse alla produzione, infatti, esistono diverse strategie plausibili per gestire tale problema di incrementalità su strutture gerarchiche, ognuna con un compromesso diverso tra costo e qualità:

- **Local Update**
  Ad ogni inserimento, si calcolano i top-k vicini *solo* per le nuove communities e si aggiungono i relativi archi. Questa soluzione è quella che ho scelto, economica sì ma non mitiga il degrado del grafo

- **Periodic Update**
  Si accumulano inserimenti fino a raggiungere una soglia K (o un intervallo temporale fisso) e poi si esegue una politica di aggiornamento più ampia: tipicamente l'unica cosa che avrebbe senso è un ricalcolo globale, molto oneroso ma massima qualità e cerchi di mitigare la crescita sui costi fissando un K abbastanza ampio; concettualmente è un ricalcolo periodico appunto

- **Proximity Update**
  Concettualmente, il ricalcolo di kNN sulle communities pre-esistenti devi farlo in funzione del fatto che, con l'aggiunta di documenti, i vicini di una certa community potrebbero cambiare. Allora, ha senso adottare una strategia intermedia tra local e global update: quando aggiungi un nuovo documento e, di conseguenza, valuti kNN per le nuove communities, propaghi il ricalcolo per un sottoinsieme derivante dall'operazione di kNN su tutte le nuove communities, ovvero l'insieme dei vicini pre-esistenti distinti ai nuovi nodi da aggiungere, appunto perché è su queste che hai l'ambiguità di 'cambiamento del vicinato'; puoi decidere se fare una sem-

plice unione, scegliere un sottoinsieme adottando una metrica relativa, ad esempio, dalla procedura di kNN stessa per ranking, piuttosto che altre euristiche. In questo senso, hai un buon compromesso tra maggiore costo rispetto al Local Update, ma più contenuto di un global, e qualità della mitigazione è intermedia allo stesso modo

6 RAG per Structured Data

Fino a questo punto, abbiamo assunto implicitamente che il corpus di riferimento sia composta da testo non strutturato: articoli, romanzi, pagine web... Nella pratica, però, alcuni database, come quelli in contesti enterprise ad esempio, hanno sempre un'eterogeneità diversa: i documenti sono misti, allinterno convivono sia structured che unstructured data e basta prendere bilanci come esempio, dove convivono testo e tabelle. Questa sezione affronta il problema di come estendere le pipeline RAG a tale eterogeneità e con un'attenzione specifica ai dati tabellari, che rappresentano il caso più comune problematico per le architetture discusse nei capitoli precedenti.

Il punto di partenza è un'osservazione già emersa nella sezione sulla UI Streamlit: indicizzando il balance sheet di Microsoft con la pipeline testuale standard, le informazioni contenute nelle tabelle venivano mal recuperate e formulate non per un problema del Language Model, ma più che altro per un problema a monte nella Text Extraction e nel chunking; sulla Text Extraction, in realtà, non ci concentreremo, ai sensi del fatto che i ragionamenti e le necessità rimangono le stesse discusse precedentemente ed il chunking è molto più interessante qui. Supposto anche che tu riesca a mantenere la forma del layout, applicando strategie di chunking pensate per testo continuo: un chunk può contenere il nome di una voce di bilancio e il valore numerico corrispondente in posizioni separate, o addirittura in chunk distinti, decontestualizzando l'informazione e rendendo il retrieval inaffidabile; questo non è un caso limite, bensì un comportamento strutturale che deriva dall'impossibilità di trattare dati strutturati come non strutturati senza trasformazioni intermedie.

6.1 Data Lakes

Un Data Lake è un repository centralizzato che archivia dati grezzi indipendentemente dalla struttura: documenti testuali, file di estensioni diverse, immagini... La caratteristica principale è, proprio, la versatilità nella gestione di formati diversi, difatti è estremamente utilizzato in contesti aziendali per la sua flessibilità rispetto ai Data Warehouse tradizionali, che richiedono una fase di normalizzazione molto rigida prima dell'ingestion.

Relativamente al nostro main topic di RAG, i Data Lakes introducono complessità architetturali abbastanza palesi: non esiste un'unica strategia di indicizzazione che funzioni uniformemente per tutte le tipologie di dato, tanto che,

alla fin fine, sei costretto comunque a costruire delle pipeline di normalizzazione a posteriori, anche solo per estrarre il testo. Le strategie di chunking discusse, la ricerca semantica su embedding, il retrieval ibrido BM25 + denso... tutto ciò che abbiamo discusso, è progettato per testo non strutturato e presuppone che l'unità fondamentale sia una sequenza di token; i dati tabellari violano questa assunzione: l'informazione non è nella sequenza dei token, ma nelle relazioni tra righe, colonne e valori; una struttura bidimensionale, quindi, di cui strategie di chunking, siano queste Fixed, Recursive, Semantiche... semplicemente non tengono conto. Quello che andrò a discutere in questa sezione è, proprio, capire *come* trattare il dato strutturato tabellare in un contesto nativamente unstructured come quello RAG e quali siano i trade-off delle alternative disponibili.

6.2 Chunking e Tabelle

Prima di discutere le strategie, è utile formalizzare il problema del chunking per dati tabellari, che differisce in maniera sostanziale da quello per testo continuo per due principali fattori: *granularità* e *relazionalità*. Nel testo, l'unità semantica fondamentale è la frase o il paragrafo ed il chunking mira a preservare tali unità. In una tabella, le possibili unità di chunking sono gerarchicamente molto diverse tra loro: la singola cella, la riga con tutti i suoi valori, la colonna con il suo header, un blocco di righe tematicamente correlate, l'intera tabella... insomma, si introduce chiaramente un'ulteriore dimensione sull'ambiguità dell'oggetto stesso del chunking: stavolta, è abbastanza chiaro che la discriminante prestazionale non è solamente la strategia, ma la relazione tra strategia e unità scelta. Scendendo più nello specifico:

- **Row-level chunking**: ogni riga viene indicizzata come chunk autonomo, anteponendo i nomi delle colonne (header) come prefisso ai relativi valori. È la granularità più fine e offre la massima precisione nel retrieval, purché, sia chiaro, la singola riga sia interpretabile da sola. Funziona bene con tabelle con poche colonne e righe, appunto, semanticamente indipendenti, mentre rende peggio quando il significato di una riga dipende dal contesto delle righe adiacenti, precedenti o successive ad esempio

- **Block-level chunking**: gruppi di righe tematicamente correlate vengono indicizzati insieme come unità singola. Richiede un criterio per identificare i blocchi, che può essere strutturale o anche semantico; ad esempio: la similarità tra embedding di righe adiacenti, ha una logica analoga al Semantic Chunking testuale. Aumenta il contesto disponibile per il retrieval, a scapito della specificità

- **Table-level chunking**: l'intera tabella viene indicizzata come un singolo chunk; è la soluzione più semplice e preserva tutta la relazionalità interna, ma produce unità spesso troppo grandi per essere recuperate con precisione e difficili da rappresentare in un singolo embedding. D'altronde: la

ricerca full-text è concettualmente ottima per le tabelle, tuttavia, ricordiamo, la sovrapposizione lessicale è agnostica alla struttura, quindi potresti ottenere tabelle con un rank altissimo e che contengono le stesse parole di ricerca ma in celle/posizioni distinte, cambiandone difatto il significato; per le tabelle, sovrapposizione lessicale è ancor meno rilevanza semantica che per testo

Non esiste una scelta universalmente ottima, ovviamente: la granularità giusta dipende dalla struttura della tabella, dalla tipologia delle query attesem dai vincoli di contesto del modello... in assenza di informazioni sul dominio, in realtà, il row-level chunking con header preposti rappresenta un punto di partenza ragionevole, per le stesse ragioni per cui il Recursive Chunking è stato indicato come soluzione go-to nel caso testuale: è semplice, controllabile e non introduce dipendenze da componenti aggiuntive (embedder); il table-level è viabile, ma eccessivamente oneroso, il blocking-level è molto più dipendente dalla componente semantica che il row-level.

Ciascuna alternativa implica un diverso trade-off tra contesto e specificità: ad esempio, una cella isolata è troppo atomica per essere interpretata senza contesto, difatti il valore `42.3M` non significa nulla senza sapere a quale voce e a quale anno si riferisce, mentre l'intera tabella, viceversa, può eccedere la finestra di contesto più facilmente e mescolare segnali molto diversi in un unico embedding. D'altronde, c'è un problema fondamentale sulla semantica: i SentenceTransformer nascono e vengono ottimizzati prevalentemente su dati testuali (frasi, paragrafi, coppie query–documento, NLI, STS), dove la semantica è veicolata soprattutto da ordine delle parole, co-occorrenze e dipendenze linguistiche. Le tabelle, invece, codificano informazione in modo diverso: il significato di una cella non è (solo) nel suo contenuto lessicale, ma soprattutto nelle relazioni strutturali che la legano agli altri elementi, righe/colonne, unità di misura, granularità temporali... conseguentemente quindi, applicare un SentenceTransformer zero-shot a rappresentazioni tabellari comporta spesso una perdita di informazione: il modello tratta dato strutturato come non strutturato e, semplicemente, non è allenato a comprendere le relazioni latenti discusse precedentemente, quindi potresti vedere un retrieval denso molto diverso da quello che ti aspetteresti.

6.3 De-Strutturazione

Definito il problema del chunking, le strategie per trattare i dati tabellari in un sistema RAG si biforcano in due direzioni concettualmente opposte, rispondendo a domande e bisogni diversi.

6.3.1 Text-to-SQL

La prima direzione rinuncia all'idea di RAG come pipeline unicamente per i dati strutturati e adotta un approccio che preserva la struttura relazionale: data una query di partenza, espressa in linguaggio naturale, il Language Model genera una traduzione SQL corrispondente che viene eseguita direttamente sul

database relazionale; sebbene il Text-to-SQL sembri tanto differente dal concetto di RAG, in realtà è assolutamente assimilabile ad un 'modulo' dello stesso: in applicazioni più sofisticate, abbiamo già discusso dell'importanza di query rewritings e topic classification, difatto la traduzione Text-to-SQL può essere vista come una declinazione particolare di entrambe e puoi fornire i risultati ottenuti dal motore SQL sempre come contesto a supporto della generazione. A mio avviso, è un parallelismo molto interessante, ai sensi del fatto che qualunque operazione orientata al retrieval, sia questo strutturato o meno, che fai sulla query per fornire contesto è sempre allineabile ad una pipeline RAG. Questo approccio ha un vantaggio strutturale fondamentale: il RAG tradizionale su dati puramente testuali non potrebbe, ad esempio, caloclare efficacemente aggregazioni, statistiche, conteggi... il Text-to-SQL ti permette di coprire, con buona complessità sulla traduzione della query, anche una parte di information need che non puoi fare trattando le tabelle come testo, o meglio, che non puoi fare a meno che non hai modo di interrogare un motore di aggregazione.

Il collo di bottiglia reale non è la generazione SQL in sé: i modelli di linguaggio moderni producono SQL prevalentemente corretto in zero-shot su schemi semplici e query relativamente standard. Il problema è il *mapping a un formato comune*, ovvero la canonizzazione dello schema in contesti realistici di Data Lake: la realtà vuole che, spesso, si abbiano schemi eterogenei, nomi di colonne poco descrittivi o abbreviati, tipi inconsistenti tra sorgenti diverse, tabelle denormalizzate... in questi casi, il modello non ha gli elementi per generare un SQL corretto senza una fase di schema linking esplicita, che associa i termini della query alle entità dello schema. Lavori come DAIL-SQL [Gao et al., 2024] e DIN-SQL [Pourreza and Rafiei, 2023] affrontano proprio questo problema, scomponendo la generazione in sotto-problemi: schema linking prima, generazione SQL poi, con un approccio che richiama il CoT RAG discusso nei capitoli precedenti. RESDSQL [Li et al., 2023] spinge ulteriormente in questa direzione, separando esplicitamente la fase di linking dalla generazione e mostrando che questa decomposizione migliora la robustezza su schemi complessi.

La limitazione della canonizzazione non è banale i un un Data Lake reale e, d'altronde, non è sempre automatizzabile senza introdurre un certo grado di errore. Tuttavia, non mi sbilancio se dico che costruire una pipeline di RAG ed introdurre un modello Text-to-SQL è la maniera più efficiente che hai per integrare structured data all'interno del RAG, sebbene non sia una soluzione *go-to*: richiamando i concetti precedenti, quando non hai un dominio specifico in cui ottimizzare cerchi di prendere delle scelte che siano il più possibilmente corrette e da specializzare successivamente, difatto i Data Lake non 'piovono dal cielo' e se lavori su questi molto probabilmente hai anche un contesto di ottimizzazione specifico; a livello di tradeoff tra performance e complessità dell'approccio, il Text-to-SQL fornisce un rapporto complessivamente bilanciato, molto complesso da progettare e manutenere rispetto ai cambiamenti di schema, ma molto efficiente, sebbene si potrebbe pensare di sfruttare strategie di indicizzazione diverse, come quelle che esporremo successivamente.

6.3.2 Serializzazione

La seconda direzione è, in realtà, diametralmente opposta: invece di preservare la struttura relazionale, vogliamo mantenere il fatto che il RAG sia prevalentemente per unstructured data e, quindi, trasformare le tabelle in testo non strutturato, indicizzabile con le stesse pipeline RAG usate per i documenti puramente testuali. Questo approccio, che chiamo *de-strutturazione*, ha il vantaggio di mantenere la pipeline uniforme e di non richiedere modifiche eccessivamente complesse a quello che fai già per il contenuto testuale; è un'idea estremamente ancorata allo scopo di questa tesi: nel mio caso di generic domain e, a questo punto, anche generic data, la necessità è quella di poter trattare in maniera semplice e ragionevolmente robusta anche tipologie di dato differenti dal testo naturale e per cui strategie di chunking tipicamente falliscono; mantenendo forti i concetti di soluzione *go-to*, *time-to-first-result*... de-strutturare ti permette di ridurre il problema di trattare structured data al problema di trattare unstructured data e la riutilizzabilità delle architetture è un vantaggio che mi piace e non poco.

Ci sono diverse tecniche di de-strutturazione della tabella, che variano in funzione di quanto sei tendente ad una completa traduzione della stessa in linguaggio naturale:

- **Serializzazione Strutturata** (Markdown, HTML, CSV)
  una tabella può essere convertita in un formato testuale che ne preservi quantomeno la struttura visiva; il caso più comune è la conversione in Markdown: ogni riga della tabella originale diventa una riga della tabella Markdown, con le celle separate dal carattere `|` e una riga di intestazione che esplicita le colonne. Questo procedimento è deterministico, non introduce perdita di informazione e produce una rappresentazione che molti Language Model riescono a interpretare in modo del tutto affidabile. Il limite principale, in realtà, è lo stesso che abbiamo discusso sulle strategie di chunking per le tabelle: la serializzazione non ti risolve le problematiche di embedding e sulla sovrapposizione lessicale discusse precedentemente

- **Serializzazione Key-Value**
  ogni cella viene rappresentata come una coppia `header: valore` e le righe sono linearizzate concatenando in sequenza tali coppie. Rispetto al Markdown è più verbosa, ma riduce qualunque ambiguità possibile legata all'interpretazione posizionale: ogni valore include esplicitamente il metadato relativo (nome della colonna), rendendo il chunk più autoesplicativo e che, di base, è un vantaggio; come suddetto, i Language Models tipicamente interpretano bene il Markdown grazie ai separatori |, ma con strutture più esplicite hai maggiore garanzia. Questa strategia risulta particolarmente adatta a tabelle con poche colonne, dove la ripetizione degli header non introduce un rumore eccessivo; i problemi alla base sono gli stessi della Serializzazione Strutturata

- **Descrizione LM based**
  un Language Model viene utilizzato per produrre una descrizione testuale della tabella complessiva o di un suo sottoinsieme, costruendola tramite descrizioni di righe o blocchi. La descrizione può essere orientata al contenuto ("Nel 2023, il fatturato del segmento Cloud è stato pari a 87.9 miliardi di dollari, in crescita del 16% rispetto all'anno precedente") o alla struttura ("La tabella riporta i ricavi per segmento di business negli anni 2021, 2022 e 2023"). Il vantaggio è che il testo prodotto è semanticamente denso e allineabile con la distribuzione lessicale del documento se, ad esempio, dai al LM anche il contenuto testuale posizionalmente vicino alle tabelle e, quindi, introduttivo per le stesse. Il problema principale è, in realtà, abbastanza semplice da capire: il retrieval è fortemente dipendente dalla qualità delle descrizione prodotte, d'altronde la descrizione deve sempre essere ad un livello concettualmente più astratto dell'informazione singolarmente contenuta e, appunto, quando fai rewritings con Language Models indebolisci strutturalmente la componente full-text del retrieval. A mio avviso, però, gli svantaggi sono assolutamente surclassati dai vantaggi nel mio contesto: questa è la de-strutturazione più naturale, tratti proprio la tabella come contenuto puramente testuale, puoi riutilizzare le pipeline di RAG su unstructured data e risolvi di base il problema di embedding dei record tabellari

La scelta non è banale e dipende in maniera critica, come tutti i motori di ricerca d'altronde, dalla distribuzione delle query attese; le prestazioni di un sistema di retrieval non sono una proprietà "assoluta" dell'indicizzazione o del SentenceTransformer/LM adottato, ma emergono dall'interazione di tutte le scelte progetturali che fai.Un lavoro coerente con quanto stiamo dicendo è quello di Sui et al. [Sui et al., 2024], che studia sistematicamente l'impatto di diversi formati di serializzazione sulle performance degli LLM. Gli autori mostrano che la de-strutturazione in linguaggio naturale tende a essere competitiva (o superiore) quando l'information need è prevalentemente semantico/descrittivo, perché migliora l'allineamento lessicale e rende più naturale il retrieval denso per come lo abbiamo visto; viceversa, mantenere una rappresentazione più vicina alla struttura tabellare risulta preferibile per interrogazioni con vincoli numerici precisi, dove la traduzione testuale può introdurre ambiguità o approssimazioni in riferimento al concetto di information loss che discutevamo precedentemente: con una riscrittura perdi per forza qualcosa. Il risultato è coerente con il trade-off discusso: serializzare in linguaggio naturale favorisce il retrieval ma tipicamente ti fa perdere in precisione, una serializzazione strutturata si muove in direzione opposta.

Come dicevo, la riscrittura in linguaggio naturale è quello che di più vicino hai ad una soluzione *go-to* e proprio perché molti sistemi RAG operano prevalentemente su richieste semantico/descrittive: non è perché sia universalmente migliore, ma perché tende a offrire un buon equilibrio tra robustezza, compatibilità con i modelli e facilità di integrazione, soprattutto quando l'obiettivo pri-

mario è massimizzare la probabilità di recuperare il contesto corretto a partire da query variabili e non note a priori.

6.3.3 HybridQA

Per valutare quantitativamente l'impatto della serializzazione sul retrieval, ho proposto la stessa impostazione adottata per il confronto tra strategie di chunking nei capitoli precedenti: si fissa il retriever (hybrid search: BM25 + denso con `baai/bge-large-en-v1.5`), si fissa il dataset di valutazione e si fa variare esclusivamente la strategia di serializzazione della componente tabellare.

Come benchmark utilizziamo **HybridQA** [Chen et al., 2020], un dataset di question answering costruito su tabelle di Wikipedia in cui ogni cella può essere collegata a uno o più passaggi testuali estratti dalle pagine delle entità referenziate e direttamente associabili. È importante comprendere la struttura del dataset, in quanto riflette la natura delle queries associate: su Wikipedia, una tabella che elenca i componenti di una band, ad esempio, può contenere celle collegate alle biografie dei singoli musicisti, e la risposta corretta a una domanda potrebbe richiedere di attraversare questo confine struttura–testo; ogni esempio in HybridQA è composto da una domanda, una tabella di Wikipedia e un insieme di passaggi testuali associati alle celle stesse. Alla luce dei risultati che vogliamo ottenere, è necessario isolare quella porzione di query il cui information need è direttamente contenuto nelle tabelle e non nei passaggi ad essi associati: ho costruito uno script che effettua, proprio, un labeling delle query in funzione di questo, andando ad ottenere un sottoinsieme di 1400 query e 400 tabelle associate. La metrica di riferimento adottata è Hit@k, coerentemente con la scelta operata nel capitolo sulle Chunking Strategies: questa misura la frazione di query per cui almeno un chunk rilevante compare tra i primi k risultati restituiti dal retrieval ed è particolarmente adatta a contesti in cui l'obiettivo primario è stimare quanto una certa strategia di chunking influisca nella costruzione del contesto.

Nel mio setup, come detto precedentemente, il confronto più rilevante dal punto di vista del tradeoff costo-performance è tra quelle che considero le due alternative più viabili per estendere il RAG anche a structured data: **row-level chunking** e **de-strutturazione con LLM**. Riguardo quest'ultimo, proprio, c'è da discutere di una scelta progettuale sulla costruzione del dataset: come ho discusso nei precedenti capitoli, teoricamente vorresti un corpus il più possibilmente ampio ed eterogeneo a livello di distrattori; nel mio setting, questo vorrebbe dire generare decine di migliaia di descrizioni ed i costi, a livello di Language Model, per me diventerebbero assolutamente proibitivi. Tuttavia, indicizzare un sottoinsieme piccolo di tabelle e le query associate può comunque permetterci di comprendere la viabilità di una strategia rispetto ad un'altra, in quanto quello che ci interessa è la 'direzione' delle metriche e non strettamente la metrica in sé.

6.3.4 Risultati

Presentiamo, quindi, i risultati di retrieval ottenuti sul sottoinsieme di query selezionate del dataset HybridQA, utilizzando:

- retrieval **sparse** (full-text BM25)
- retrieval **dense** (embedding con `BAAI/bge-large-en-v1.5`)
- retrieval **ibrido** combinando gli score dei due approcci

Coerentemente con quanto discusso nelle sezioni precedenti, l'obiettivo qui non è "trovare la strategia universalmente migliore", ma osservare come varino le performance rispetto alla rappresentazione che scegli per de-strutturare le tabelle stesse. Per ottenere i

Ho costruito tre indici distinti, ciascuno corrispondente a una scelta di destrutturazione e valutato su questi l'hybrid search rispetto a ciascuna delle 400 query. In realtà, ho sfruttato concetti abbastanza noti quando si lavora con i SentenceTransformer: non puoi strettamente basarti sul fatto che, senza adeguato contesto, il modello sappia costruire un vettore ben posizionato nello spazio semantico; di conseguenza, i 3 indici si differenziano non solo per la strategia, ma soprattutto sulla scelta di cosa fornisci al modello per infierire la semantica:

- `hybridqa_tables_llm`: unità di indicizzazione *table-level*, ottenuta tramite *descrizione LM-based* e con contesto aggiuntivo dato da intro + descrizione generata. HybridQA è un dataset assolutamente ottimo per questi test, essendo che fornisce direttamente quello che, prima, avevamo accennato come testo 'posizionalmente vicino' alla tabella, quindi introduttivo per la stessa e che chiamiamo "intro"; questo aiuta, proprio, in relazione a ciò che dicevamo precedentemente: la qualità del vettore della descrizione generata dal LM dipende anche e soprattutto da cosa riesce direttamente, o meno, ad infierire dal testo e introduzioni aiutano ad ottenere vettori semanticamente meglio posizionati
- `hybridqa_rows`: unità di indicizzazione *row-level*; per ogni riga ho indicizzato **titolo della tabella + riga**. In questo caso, ho deciso di non utilizzare la intro per evitare il collasso semantico degli embedding: come abbiamo già discusso, i SentenceTransformer pubblici non sono necessariamente allenati su embedding di serializzazioni di righe tabellari, di conseguenza, se tutte le righe condividono lo stesso blocco introduttivo, l'effetto è diametralmente inverso a quello precedente, poiché la semantica data dal record `key:value` sarebbe eccessivamente diluita dalle introduzioni, molto più 'lunghe' e dense di informazioni, andando a produrre vettori troppo vicini tra loro nello spazio
- `hybridqa_intros`: baseline *solo intro*, cioè esclusivamente testo posizionalmente vicino/introduzione della tabella, senza contenuto tabellare; l'idea è quella di capire a fondo la qualità delle intro, in quanto, utilizzandole,

se queste sono eccessivamente informative allora le metriche ottenute, soprattutto sull'approccio generativo con LM di interesse, potrebbero essere gonfiate

Index	Mode	N	Hit@1	Hit@3	Hit@5	MRR
hybridqa_tables_llm	sparse	1463	0.312	0.458	0.528	0.414
hybridqa_tables_llm	dense	1463	0.493	0.683	0.751	0.615
hybridqa_tables_llm	hybrid	1463	0.422	0.597	0.670	0.534
hybridqa_rows	sparse	1463	0.402	0.455	0.471	0.442
hybridqa_rows	dense	1463	0.486	0.543	0.571	0.533
hybridqa_rows	hybrid	1463	0.501	0.560	0.593	0.546
hybridqa_intros	sparse	1463	0.185	0.292	0.340	0.261
hybridqa_intros	dense	1463	0.353	0.528	0.599	0.470
hybridqa_intros	hybrid	1463	0.265	0.396	0.465	0.357

Tabella 14: Valutazione Retrieval su query `table_only`

Come risultato fondamentale otteniamo che la descrizione LM-based massimizza sì il dense retrieval, ma penalizza la componente full-text; in realtà qualcosa che, ragionevolmente, potevamo aspettarci coerentemente alle discussioni fatte. L'indice `hybridqa_tables_llm` ottiene il miglior risultato in assoluto in modalità **dense**:

$$\text{Hit@1} = 0.493, \ \text{Hit@5} = 0.751$$

ed è nettamente superiore alla baseline `intros` e alla variante `rows`. Questo è coerente con quanto discusso nella sezione sulla *de-strutturazione* appunto: la riscrittura in linguaggio naturale produce un testo più "allineato" al tipo di semantica che un SentenceTransformer cattura bene (*co-occorrenze*, *lessico naturale*, *relazioni esplicitate linguisticamente*), e quindi tende a migliorare la separabilità nello spazio vettoriale. Il rovescio della medaglia è evidente nel **sparse**: `tables_llm` sparse ha Hit@1 0.312, inferiore a `rows` sparse 0.402; è la conseguenza diretta del trade-off già anticipato: quando riscrivo, introduco inevitabilmente un indebolimento strutturale della componente full-text (parafrasi, sinonimi, compressione di dettagli), quindi riduco l'ancoraggio lessicale che BM25 sfrutta. Complessivamente, come suddetto, il retrieval denso è superiore per la descrizione prodotta dal LM rispetto all'approccio row based: questo conferma smentisce i ragionamenti fatti precedentemente, d'altronde come SentenceTransformer ho utilizzato un modello 'grosso' e che, in applicazioni reali, saresti meno tendente ad utilizzare ed le differenze potrebbero essere ancora più marcate ad inferenza; la direzione, quindi, è chiara: un gain c'è e potrebbe essere anche più marcato considerando un modello di minori dimensioni, tuttavia potrebbe non essere ragionevolmente giustificato dalla perdita sulla componente full-text e, infatti, ancora molto dipende dalla tipologia di query attesa.

La mia aspettativa di "più contesto $\Rightarrow$ dense migliore" è rispettata ed, infatti, si verifica: `tables_llm` è il migliore in dense, tuttavia, dall'analisi sulle

altre strategie, si osserva come il titolo da solo non riesca a risolvere l'ambiguità semantica che c'è sulla linearizzazione dei record, sebbene valutare ulteriori strategie, sempre row-level, ma con maggiore contesto potrebbero coprire edge cases ad inferenza dove quest'ultima fallisce.

Volendo riassumere il comportamento osservato:

- **Row-level** tende a favorire il **full-text**: mantiene label e valori "crudi", ottimi per match lessicali e query con vincoli precisi. Il dense è buono ma meno dominante al crescere di k, appunto perché la semantica della riga può essere povera (molti numeri, magari)
- **LM-based** tende a favorire il **dense**: la descrizione crea segnali semantici migliori e puoi anche controllare ragionevolmente l'allineamento lessicale, sebbene tenda a ridurre l'efficacia del full-text, quindi il compromesso ibrido può non essere il migliore anche con un dense molto forte

Questo è esattamente il motivo per cui, in contesti come quelli di Data Lake eterogenei, conviene ragionare più su un *portafoglio* di rappresentazioni che di singola strategia, nonostante la mia tendenza sia quella di considerare la riscrittura generativa come assolutamente valida.

Integrazione con le Architetture RAG La costruzione precedentemente fatta degli indici rende, come già accennato, totalmente riutilizzabili le architetture di RAG costruite: una volta che hai indicizzato e collegato la pipeline al query engine, l'architettura è totalmente agnostica ed indipendente dal contenuto degli indici; la riutilizzabilità è il concetto fondamentale del perché ho scelto di studiare la de-strutturazione, sebbene non voglia dire che puoi utilizzare le stesse esatte strategie di chunking precedente. Infatti, trasformare dati strutturati in testo non strutturato ti permette di integrare gran parte di queste, tuttavia trattare dati tabellari richiede comunque un passaggio a monte che non esiste nel caso puramente testuale, orientato principalmente su meccanismi di **table recognition** ed adozione di una strategia di chunking classica (sempre Fixed/Recursive/Semantic...) dove, però, a mio avviso è rigoroso mantenere in un unico chunk descrizione generata/rows e testo posizionalmente vicino, anche a costo di produrre chunk ragionevolmente più grandi del size fissato.

Considerazione sui Costi Come già ampiamente discusso in questo lavoro, scegliere una strategia migliore comporta sempre analizzare *compromessi* derivanti da diversi indicatori e non unicamente la bontà prestazionale, d'altronde su un setup che potrebbe non rispecchiare la variabilità del caso reale. L'analisi precedente ha permesso di comprendere la viabilità delle singole strategie, tuttavia: molto dipende dalla tipologia di query attesa e, non solo, anche dal costo stesso che dovresti sostenere nell'adozione della strategia, cosa che discutiamo meglio qui. A livello di costi, la strategia row-based, in realtà, è assolutamente

efficiente: non impiega chiamate a Language Model, è molto veloce e dipendente, in realtà, dal preprocessamento che fai sul testo e con performance competitive; la strategia LM-based, invece, presenta costi e latenza maggiori, dovuti alle chiamate, proprio, ai Language Model stessi. Come suddetto, i costi sono sempre dati da una stima che puoi realizzare dei token in input e generati: in questo caso, non necessariamente una situazione domina l'altra, anche perché in input viene dato testo posizionalmente vicino e non solo la tabella stessa, nonché puoi mettere dei vincoli sulla 'grandezza' della descrizione generata ma non necessariamente rende cost efficient la strategia. Un gain prestazionale, come suddetto, c'è ed è chiaro, tuttavia potrebbe non essere giustificato dall'aumento dei costi rispetto alla strategia row-based: d'altronde, parliamo di un ordine di grandezza relativo ai costi ad inferenza di modelli Large, anche perché al LM stai chiedendo di fornire una descrizione della tabella che comprenda comunque capacità di analisi dei legami riga-colonna ed interazione semantica con il contesto introduttivo fornito e questo serve a capire che, generalmente, usare modelli Small potrebbe ridurre di molto la qualità delle descrizioni e, quindi, le performance della ricerca densa. Il costo, poi, dipende anche fortemente dalla quantità di chiamate: approcci già discussi come Microsoft GraphRAG, a mio avviso, molte volte si dimostrano scelte subottime proprio perché sfruttare un LM nella fase di indicizzazione vuol dire fare tante chiamate quanto è grande la dimensione del corpus e, in contesti reali, questa si avvicina molto facilmente all'ordine di centinaia di migliaia di unità, milioni magari, facendo esplodere ben presto i costi, soprattutto in contesti enterprise di 'mixed text' (bilanci, ad esempio). Complessivamente quindi, come accennato in precedenza, la strategia generativa LM-based, a mio avviso, è quella più stimolante ed interessante in un progetto di ricerca, tuttavia spingere su modi più efficienti di migliorare la componente densa per la row-based potrebbe portare a vantaggi più concreti, nel caso medio, rispetto al tradeoff costo-performance.

7 Sommario

Concludiamo riassumendo i risultati e le considerazioni più rilevanti emerse durante la discussione del lavoro, mettendo a confronto le principali architetture discusse: *Naive RAG*, *CoT RAG* e *Hierarchical/Graph-based RAG*.

Il risultato fondamentale a cui siamo arrivati è che: in un setting *zero-shot* e *generic-domain*, la pipeline più difficile da battere rimane spesso quella più semplice, ovvero un *Naive RAG* ben impostato (chunking sensato, top-k adeguato...), abbinato ad un modello Large con ottime capacità di comprensione testuale; molte architetture più sofisticate pagano overhead e criticità aggiuntive senza garantire un guadagno proporzionato nel trade-off costo-performance, fondamentale nelle mie analisi. Nel benchmark GraphRAG Bench, *Naive RAG* risulta una baseline forte: mantiene valori alti e bilanciati su correttezza e copertura, con ottimo grounding d'altronde; la stessa conclusione si conferma anche su HotpotQA: nonostante il task ed il reasoning richiesto per costruire le risposta

sia pià complesso, il single-shot retrieval resta competitivo con un buon embedder ed un chunking Recursive. Come ampiamente discusso, le prestazioni di una pipeline RAG sono fortemente dipendenti da *strategia* e *size* di chunking: passando a chunk più lunghi (600 token), il Naive RAG tende a migliorare, soprattutto per LLM, poiché produce chunk mediamente più auto-contenuti e rende più probabile che il retrieval contenga le evidenze necessarie a soddisfare l'information need, con un compromesso sull'aumento dei costi.

Il *CoT RAG* introduce un retrieval iterativo guidato da scratchpad e follow-up queries. I risultati mostrano che l'effetto dipende fortemente dalla qualità del Language Model:

- per un modello Large, l'adattività e, quindi, forzare il modello a ragionare durante il retrieval, tende a migliorare il *grounding*; infatti, su GraphRAG Bench si osserva un aumento netto di `faithfulness_to_context` a fronte di variazioni marginali su `answer_correctness` e `evidence_coverage`
- per un modello Small, l'adattività è spesso *rischiosa* in relazione alle minori capacità di comprensione testuale e, soprattutto, all'assenza di una vera e propria fase di post-training dove il reasoning viene insegnato, come i foundational modes moderni; aumentano le opportunità di query drift e di errori nell'utilizzo delle evidenze, con un degrado sistematico delle metriche rispetto alla baseline Naive

A livello di costi, l'iteratività, ovviamente, moltiplica token e chiamate: mediamente la spesa per query cresce di circa $\sim 1.9\times$ per il modello "Large" e $\sim 2.2\times$ per il modello "Small" nel setup valutato, rendendo il CoT RAG sicuramente un upgrade a livello prestazionale, tuttavia non necessariamente giustificato dai costi.

L'architettura gerarchica graph-based si fonda sull'obiettivo di sfruttare un retrieval *coarse-to-fine* tramite una fase di *rollup* summary-based e successivo *drilldown*. Il limite strutturale più importante emerso durante i test coincide con l'introduzione di un'*information loss* spesso eccessiva durante la fase di rollup, il che si riflette direttamente in cali di `evidence_coverage` e, di conseguenza, di qualità della risposta. Nel setup testato su GraphRAG Bench, GraphRAG risulta sistematicamente l'approccio peggiore in `answer_correctness` ed `evidence_coverage`: l'interpretazione principale è che il collo di bottiglia non sia il retrieval in sé, ma proprio la compressione gerarchica stessa; i summaries, appunto, perdono dettaglio e la riscrittura dell'informazione rende più difficile trattare queries anche molto semplici, come quelle entity centric. D'altronde: collegandoci con la proposta iniziale, questo è esattamente il motivo per cui ho abbandonato l'idea di LLM Stitching come appoggio alla clusterizzazione semantica.

Un altro degli insight fondamentali riguarda la sensibilità delle pipeline alla

forma delle queries: come accennato prima, l'architettura graph-based è estremamente embletica di questo, ai sensi proprio del fatto che il rollup semantico ti permette di rispondere meglio a query con maggior grado di ambiguità semantico/descrittiva, difatti l'approccio migliora nettamente su tutte le metriche in questo caso, ma pessimamente in tutti quei casi, come queries entity centric, dove la componente full-text è la protagonista. Questo suggerisce, d'altronde, che il *query rewriting* è un concetto fondamentale non solo per migliorare le performance ad inferenza di un'architettura, supposto si conosca la distribuzione attesa delle query, ma anche per rendere competitive architetture in underperformance. Considerazioni importanti le abbiamo ricavate anche analizzando possibili estensioni del paradigma RAG a structured data *mixed* (testo + tabelle), tipici di Data Lake enterprise. Il punto centrale è che le tabelle violano l'assunzione implicita alla base del RAG: l'informazione non risiede nella sola sequenza di token testuali, ma nelle relazioni riga–colonna–valore; applicare "as-is" strategie di chunking ed indicizzazione per il testo produce chunk eccessivamente rumorosi, con header e valori separati, record incompleti... rendendo il retrieval strutturalmente inaffidabile.

La divisione che ho proposto delle strategie di chunking tabellare si basa un compromesso tra: granularità e relazionalità/precisione del contesto; in assenza di informazioni sul dominio e sulla distribuzione delle query, il *row-level chunking* con header espliciti rappresenta un punto di partenza ragionevole ed, in realtà, per ragioni analoghe a quelle discusse per il Recursive Chunking nel caso testuale: semplicità, controllabilità e assenza di dipendenze da componenti aggiuntive. Esplorando anche alternative più sofisticate, le intuizioni concettuali ed i test adottati suggeriscono che:

- un approccio **Text-to-SQL** preserva la struttura e abilita interrogazioni anche molto complesse, come aggregazioni (conteggi, statistiche, group-by), che un RAG puramente testuale non riuscirebbe a gestire con la stessa semplicità; il bottleneck pratico non è la sintassi SQL, ma lo *schema linking* e la canonizzazione in scenari realistici di Data Lake che, molto spesso, sono caratterizzati da schemi eterogenei e poco descrittivi
- un approccio di **De-strutturazione / Serializzazione** trasforma la tabella in una rappresentazione testuale per riutilizzare l'intera pipeline RAG unstructured; i vantaggi di uniformità e *time-to-first-result* sono chiari, gli svantaggi sono molto simili a quelli discussi nel retrieval: trade-off tra allineamento semantico, utile al dense retrieval, ed indebolimento della componente full-text

Il confronto sperimentale su HybridQA (sottoinsieme di query `table_only`) conferma quanto mi aspettassi:

- la **descrizione LM-based** (`hybridqa_tables_llm`) massimizza le performance in **dense retrieval** (Hit@1=0.493, Hit@5=0.751), coerentemente con l'idea che la riscrittura testuale garantisca una forma più coerente

alla semantica catturata da un SentenceTransformer (allenato su testo naturale e non su serializzazioni di record)

- il **row-level chunking** (`hybridqa_rows`) preserva meglio la componente **sparse/full-text**, mantenendo label e valori "raw" particolarmente adatti a query più specifiche; stesso concetto di rollup semantico nell'approccio graph-based

Nel complesso, i risultati confermano che, come per le architetture RAG, possono esistere scelte migliori ma non sempre dominanti; in contesti con alta eterogeneità del dato, come i Data Lakes, conviene ragionare in termini di *portafoglio* di strategie: row-level per precisione e match lessicale, LM-based per massimizzare l'allineamento semantico nel denso... eventualmente instradate da moduli di query understanding (topic classification / query rewriting) già discussi nel resto della tesi. D'altronde, per l'estensione a structured data è più complesso trovare vere e proprie strategie *go-to*, essendo che le alternative più interessanti performano davvero similmente nel complesso, infatti: il row-level è estremamente cost-efficient, non avendo nessuna chiamata a LM in indicizzazione, mentre le strategie LM-based introducono costi e latenza proporzionali alla dimensione del corpus, rendendo l'approccio più adatto quando il guadagno atteso sul dense retrieval giustifica i costi che devi gestire .

Takeaway Finali

- **Naive RAG è la soluzione go-to**: difficile da battere in generic-domain e robusta se ben progettata
- **CoT RAG è un upgrade mirato**: utile soprattutto per migliorare il grounding e gestire hop mancanti, ma richiede modelli capaci e introduce un overhead di costo significativo
- **Hierarchical/Graph-based non è "plug-and-play"**: la compressione summary-based introduce information loss e l'efficacia dipende fortemente dalla forma delle query; il query rewriting può ribaltare le performance
- **Per Structured Data, la granularità è fondamentale**: scegliere l'unità di indicizzazione (riga/blocco/tabella) è una decisione ulteriore che aumenta la complessità del chunking rispetto al caso testuale; senza una rappresentazione che preservi le relazioni riga–colonna, non puoi aspettarti grounding su tabelle
- **Text-to-SQL e De-strutturazione sono complementari**: il primo domina su vincoli numerici e aggregazioni, la seconda massimizza riusabilità in contesti mixed; la scelta (o combinazione) dipende, ancora una volta, principalmente dalla distribuzione attesa delle query

Riferimenti bibliografici

[Atagong et al., 2025] Atagong, S. D., Tonnang, H., Senagi, K., Wamalwa, M., Agboka, K. M., and Odindi, J. (2025). A review on knowledge and information extraction from pdf documents and storage approaches. *Frontiers in Artificial Intelligence*, 8:1466092.

[Blondel et al., 2008] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008.

[Chen et al., 2025a] Chen, H. et al. (2025a). Interpreting the curse of dimensionality from distance concentration and manifold effect. *arXiv preprint*.

[Chen, 2025] Chen, M. (2025). What are small language models (slms)? how do they work? `https://www.oracle.com/it/artificial-intelligence/small-language-models/`. Accessed: 2026-02-06.

[Chen et al., 2020] Chen, W., Zha, H., Chen, Z., Xiong, W., Wang, H., and Wang, W. Y. (2020). Hybridqa: A dataset of multi-hop question answering over tabular and textual data. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1026–1036, Online. Association for Computational Linguistics.

[Chen et al., 2025b] Chen, Z., Pradeep, R., and Lin, J. (2025b). Accelerating listwise reranking: Reproducing and enhancing first. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '25)*, Padua, Italy. ACM.

[Coveo, 2024] Coveo (2024). Chunking strategy / passage retrieval best practices (fixed-size chunking). La documentazione riporta una media di 300 token per chunk (min 200, max 400) per fixed-size chunking in contesti di passage retrieval.

[DeepSeek-AI et al., 2025] DeepSeek-AI et al. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*. Dimostra come il post-training con RL ("Large-Scale Reinforcement Learning") incentivi il modello a generare autonomamente tracce di ragionamento (CoT) per risolvere problemi complessi.

[Edge et al., 2024] Edge, D., Trinh, H., Cheng, Y., Bradley, J., Chao, A., Mody, A., Truitt, S., and Larson, J. (2024). From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*.

[Elastic, 2026] Elastic (2026). Elasticsearch reference: Elasticsearch. Accessed: 2026-02-17.

[Feldbauer and Flexer, 2019] Feldbauer, R. and Flexer, A. (2019). A comprehensive empirical comparison of hubness reduction in high-dimensional spaces. *Knowledge and Information Systems*.

[Fortunato, 2010] Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3–5):75–174.

[Gao et al., 2024] Gao, D., Wang, H., Li, Y., Sun, X., Qian, Y., Ding, B., and Zhou, J. (2024). Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment (PVLDB)*, 17(5):1132–1145.

[Jacob et al., 2024] Jacob, M., Lindgren, E., Zaharia, M., Carbin, M., Khattab, O., and Drozdov, A. (2024). Drowning in documents: Consequences of scaling reranker inference. *arXiv preprint*.

[Juvekar and Purwar, 2024] Juvekar, K. and Purwar, A. (2024). Introducing a new hyper-parameter for rag: Context window utilization. *arXiv preprint arXiv:2407.19794*.

[Kojima et al., 2022] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*.

[Langville and Meyer, 2004] Langville, A. N. and Meyer, C. D. (2004). Deeper inside PageRank. *Internet Mathematics*, 1(3):335–380.

[Li et al., 2023] Li, H., Zhang, J., Li, C., and Chen, H. (2023). RESDSQL: decoupling schema linking and skeleton parsing for text-to-sql. In *Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2023)*, pages 13067–13075. AAAI Press.

[Liu et al., 2023] Liu, Y., Iter, D., Xu, Y., Wang, S., Xu, R., and Zhu, C. (2023). G-eval: Nlg evaluation using gpt-4 with better human alignment. *arXiv preprint arXiv:2303.16634*.

[Microsoft, 2024] Microsoft (2024). Document layout analysis (layout model) — azure ai document intelligence. `https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/prebuilt/layout?view=doc-intel-4.0.0`. Accessed: 2026-02-11.

[Microsoft, 2025] Microsoft (2025). Read model ocr data extraction — azure ai document intelligence. `https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/prebuilt/read?view=doc-intel-4.0.0`. Accessed: 2026-02-11.

[Mukherjee et al., 2023] Mukherjee, S., Mitra, A., Jawahar, G., Agarwal, S., Iyyer, M., and Awadallah, A. (2023). Orca: Progressive learning from complex explanation traces of gpt-4. *arXiv preprint arXiv:2306.02707*. Introduce l'idea di "Explanation Tuning": addestrare un modello piccolo usando le tracce di

ragionamento (CoT) di un modello più grande, rendendo il ragionamento una capacità appresa tramite SFT.

[Nye et al., 2021] Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. (2021). Show your work: Scratchpads for intermediate computation with language models.

[Page et al., 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab.

[Pourreza and Rafiei, 2023] Pourreza, M. and Rafiei, D. (2023). Din-sql: Decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*.

[Press et al., 2022] Press, O., Smith, N. A., and Lewis, M. (2022). Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*. Introduces Self-Ask prompting and shows how to plug in search.

[Qu et al., 2025] Qu, R., Tu, R., and Bao, F. S. (2025). Is semantic chunking worth the computational cost? In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 2155–2177, Albuquerque, New Mexico. Association for Computational Linguistics.

[Ramakrishnan et al., 2012] Ramakrishnan, C., Patnia, A., Hovy, E., and Burns, G. A. P. C. (2012). Layout-aware text extraction from full-text pdf of scientific articles. *Source Code for Biology and Medicine*, 7(1):7. Accessed: 2026-02-11.

[Reimers and Gurevych, 2021] Reimers, N. and Gurevych, I. (2021). The curse of dense low-dimensional information retrieval for large index sizes. In *Proceedings of ACL-IJCNLP (Short Papers)*.

[Sarthi et al., 2024] Sarthi, P. et al. (2024). Raptor: Recursive abstractive processing for tree-organized retrieval. In *ICLR*.

[Singh et al., 2024] Singh, S., Pecora, C., and Khanuja, M. (2024). Amazon bedrock knowledge bases now supports advanced parsing, chunking, and query reformulation giving greater control of accuracy in rag based applications. Nel post viene indicato un valore raccomandato di 300 per il parametro *max token size for a chunk*.

[Subramanian et al., 2025] Subramanian, S., Elango, V., and Gungor, M. (2025). Small language models (slms) can still pack a punch: A survey. *arXiv preprint arXiv:2501.05465*.

[Sui et al., 2024] Sui, Y., Zhou, M., Zhou, M., Han, S., and Zhang, D. (2024). Table meets llm: Can large language models understand structured table data? a benchmark and empirical study. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining (WSDM '24)*.

[Traag et al., 2019] Traag, V. A., Waltman, L., and van Eck, N. J. (2019). From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*.

[Trivedi et al., 2023] Trivedi, H., Balasubramanian, N., Khot, T., and Sabharwal, A. (2023). Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509*.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[Wang et al., 2022] Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., and Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

[Wei et al., 2022] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E. H., Le, Q. V., and Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

[Yang et al., 2018] Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. (2018). Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380. Association for Computational Linguistics.

[Yao et al., 2022] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

[Zheng et al., 2023] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. (2023). Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*.