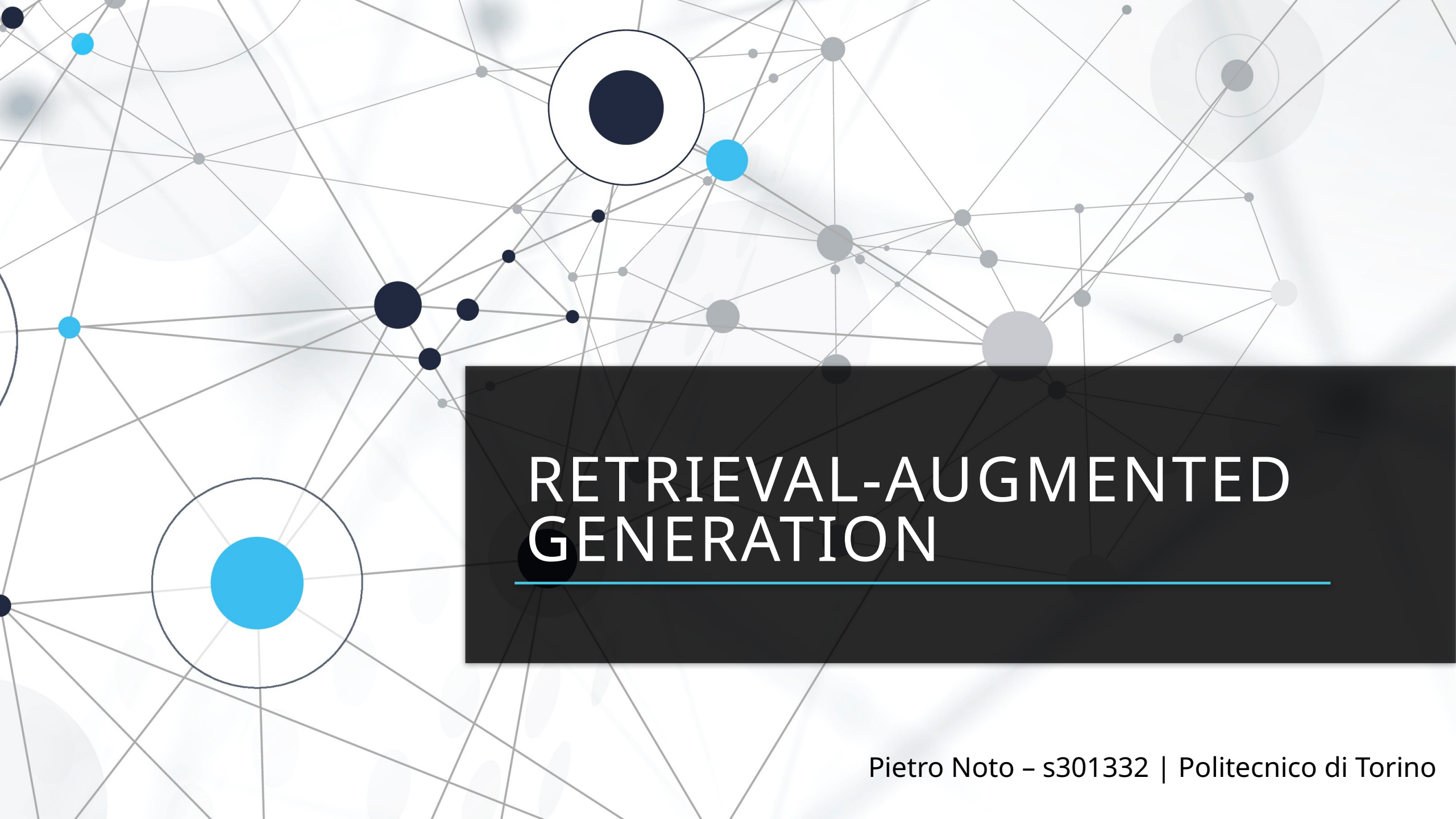


RETRIEVAL-AUGMENTED GENERATION

Pietro Noto – s301332 | Politecnico di Torino

PERCHÉ RAG?

Un Large-Language Model (LLM), essendo un modello Deep Learning, necessita di enormi quantità di dati per il training, recuperati da database di nozioni generali come Wikipedia Dump e Common Crawl.

LLM dimostra buone prestazioni in ambiti coperti durante il training, ma **non altrettanto in contesti specializzati**, come la medicina, giurisprudenza, biologia, ecc. proprio perché mancano i dati necessari durante il training.

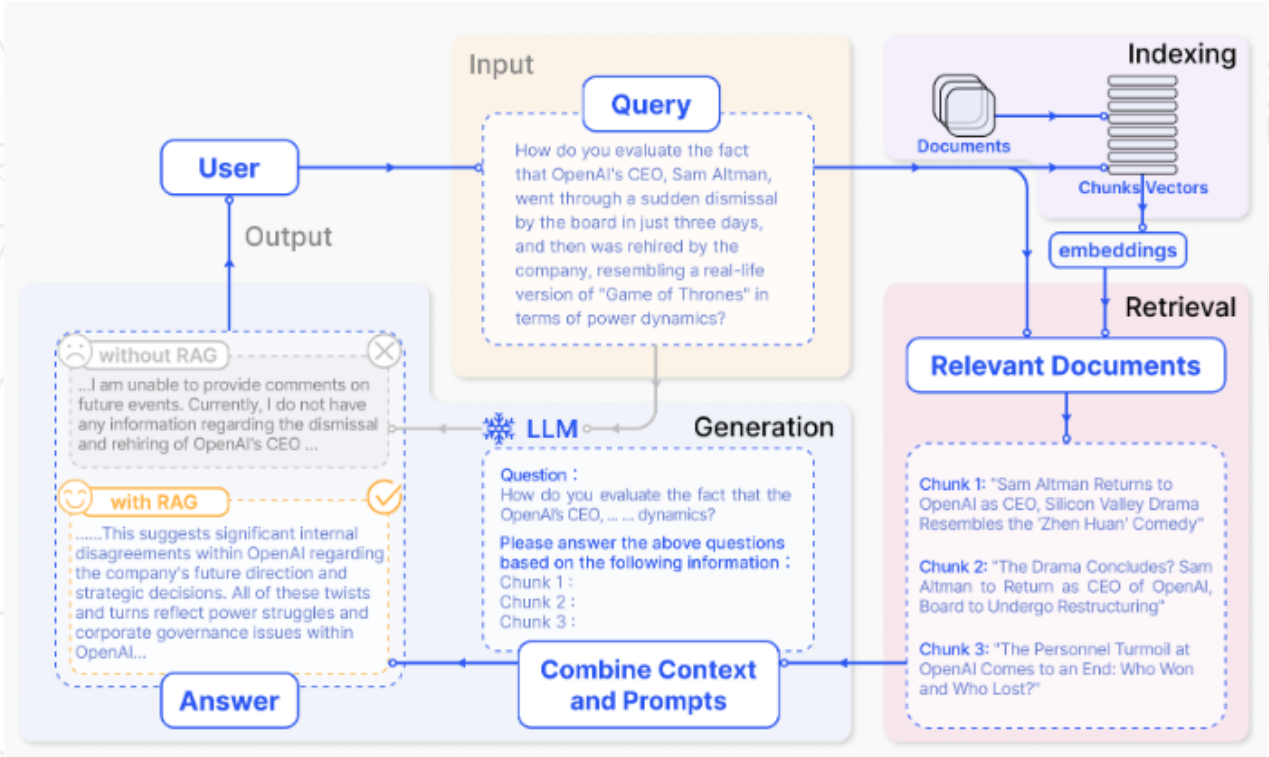
Se si interroga un LLM su un argomento che non sa, il risultato sarà un'**allucinazione**, cioè una risposta sbagliata ma che il modello crede sia corretta.

PERCHÉ RAG?

La soluzione a questo problema inizialmente era il fine-tuning del modello con i nuovi dati.

- ✔ Soluzione ben nota nel DL
- ✔ Approccio monolitico
- ✔ Garantisce buone prestazioni
- ✗ Serve conoscenza tecnica del modello
- ✗ Molto time-consuming
- ✗ Poco scalabile → nuovi dati, nuovo fine-tuning

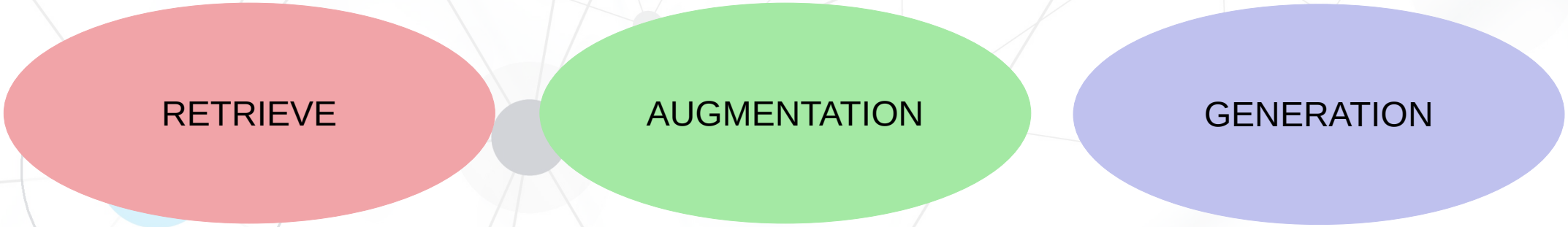
PERCHÉ RAG?



A causa dei lunghi tempi di training e delle conoscenze tecniche del modello, invece del fine-tuning, si usa un approccio diverso, proposto dal 2020, chiamato **Retrieval-Augmented Generation (RAG)**, e tuttora in evoluzione.

PERCHÉ RAG?

Per comprendere RAG occorre comprendere le tre parole chiave contenute nel suo nome:



RETRIEVE

L’idea del RAG è l’aggiunta di un nuovo modulo, detto Retriever, a un backbone esistente basato su un LLM.

Il **Retriever** si occupa di recuperare dati esterni all’LLM, di processarli e di inviare al modello il risultato dei dati processati, in modo che possa generare la risposta in linguaggio naturale.

In sostanza, il retriever fornisce al modello la conoscenza aggiuntiva necessaria per formulare la risposta.

RETRIEVE

In generale, il recupero (retrieve) dei dati può essere classificato nei seguenti tipi:

- **Sparso**: viene misurata rilevanza delle singole parole nei documenti, molto efficiente in caso di ricerche puntuali e specifiche *(BM25, basato su TF-IDF)*
- **Denso**: i documenti e la query dell'utente vengono convertiti in vettori di embedding e salvati in un database vettoriale (*es: QDrant*) e l'affinità viene misurata con diverse metriche *(somiglianza del coseno, FAISS, ecc.)*;
- **A grafo**: dai documenti vengono ricavate entità e relazioni e trasformati in un grafo *(GraphRAG, spiegate in seguito).*

RETRIEVE

I dati esterni possono presentarsi in forma:

- Non strutturata: testo semplice, come Wikipedia Dump → facile
- Semi-strutturata: ad esempio documenti medici, legali, in formato PDF → difficile, occorre trasformare tabelle in dati;
- Strutturata: nel caso di Knowledge Graph (KG) nelle GraphRAG

RETRIEVE

Pre-retrieve: indicizzazione

Prima del recupero vero e proprio è molto utile effettuare l'**indicizzazione** dei dati:

- Si può effettuare una pulizia del testo, normalizzandolo nel formato ASCII;
- I documenti vengono divisi in chunk, costituiti da un numero fisso di token (parole);
- Si possono costruire indici gerarchici oppure i KG

Possono vengono aggiunti dei metadati ai dati, per tenere traccia di informazioni come la data, che serve per filtrare dati vecchi, ad esempio.

RETRIEVE

Pre-retrieve: ottimizzazione della query

Molto spesso la query fornita dall'utente può essere ambigua, troppo generica e/o di non facile comprensione per il modello LLM, pertanto può essere ottimizzata:

- **Espansione**: il modello LLM moltiplica la query di partenza in più query affini ad essa, migliorando l'efficacia della risposta e riducendo le allucinazioni (es: "Ricette di primi italiani" può essere espansa in "Ricette di pasta italiane", "Ricette di spaghetti italiane", "Ricette di risotti italiani", ...);
- **Riscrittura**: la query viene riscritta in un linguaggio più digeribile dal modello (es: "Joe Biden" → "Chi è Joe Biden?")

RETRIEVE

Retrieve

- Nel caso di retriever denso, la query viene trasformata in vettori di embedding e viene calcolata la somiglianza con gli embedding dei documenti (es: somiglianza coseno, FAISS, ecc.);
- Nel caso di retriever sparso viene applicato un algoritmo, come BM25, basato su TF-IDF, che si dimostra più efficace degli algoritmi densi nel caso di query puntuali;
- Ad ogni modo, nella realtà viene usato un approccio ibrido che combina i due;
- Il caso di KG viene approfondito in seguito

AUGMENTATION

L’augmentation consiste nell’integrare il contesto recuperato in fase di retrieve con l’LLM.

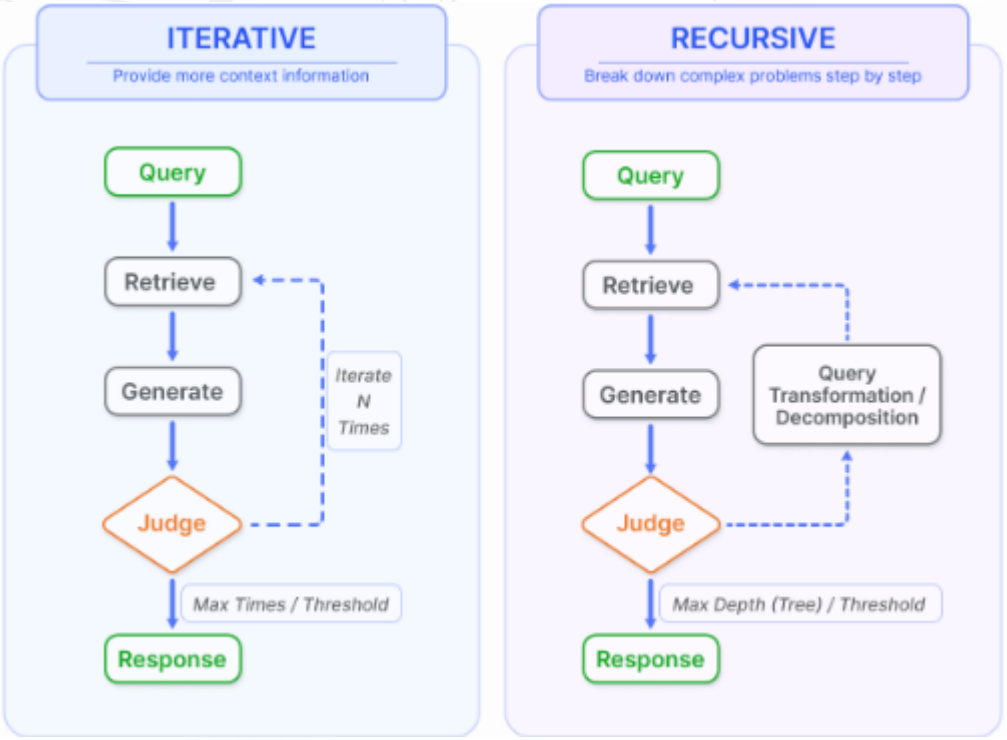
L’integrazione può essere effettuata a diversi livelli della catena RAG:

- Input: il contesto recuperato viene concatenato con la query dell’utente;
- Output: il contesto viene concatenato con il risultato della generazione;
- Intermedio: il contesto viene concatenato con i layer nascosti dell’LLM.

AUGMENTATION

La fase di retrieve descritta prevede un singolo recupero dei dati (**single-shot**) ma **si è dimostrato non ottimale**, pertanto si preferisce effettuare molteplici retrieve fino a quando non si ottiene un punteggio minimo di qualità.

- **Retrieve iterativo**: il recupero viene ripetuto tale e quale al più N volte;
- **Retrieve ricorsivo**: ad ogni iterazione la query viene espansa e/o riscritta, rendendo il problema corrente sempre più semplice (*divide et impera*).



GENERATION

La fase di generation consiste nella generazione dell’output in linguaggio naturale sulla base del contesto recuperato dal retriever.

- Viene effettuata dal modello LLM

GENERATION

Post-retrieve

- **Reranking** dei documenti: basandosi sui punteggi di rilevanza calcolati dall'LLM in fase di retrieve, viene ristabilito un ordine dei documenti recuperati per aumentare la pertinenza e la precisione delle risposte;
- **Selezione** del contesto: vengono scartati i documenti recuperati meno rilevanti;
- **Compressione** del contesto: viene ridotto il numero di token di ciascun chunk recuperato;

VALUTAZIONE

Punteggi:

- Rilevanza del contesto recuperato rispetto alla query;
- Fedeltà della risposta rispetto al contesto recuperato;
- Rilevanza della risposta rispetto alla query

VALUTAZIONE

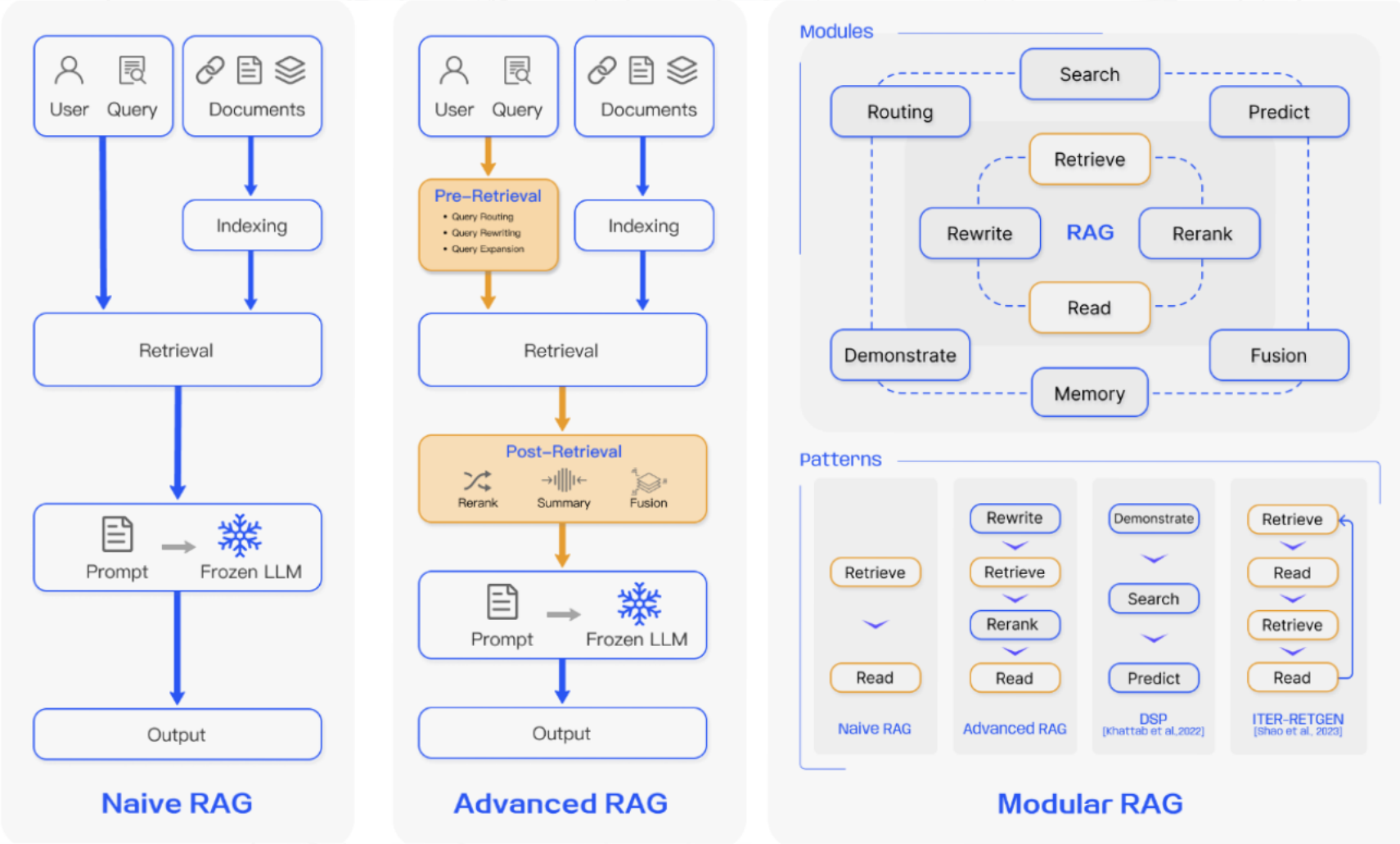
Competenze:

- Robustezza al rumore: scartare documenti pertinenti ma privi di informazioni rilevanti;
- Negative rejection: ammettere di non sapere quando il contesto recuperato non contiene le informazioni chieste;
- Coerenza nell’amalgamare le informazioni prese da diversi documenti;
- Robustezza controfattuale: riconoscere incoerenze all’interno di un documento (e scartarlo)

VALUTAZIONE

Evaluation Framework	Evaluation Targets	Evaluation Aspects	Quantitative Metrics
RGB†	Retrieval Quality Generation Quality	Noise Robustness Negative Rejection Information Integration Counterfactual Robustness	Accuracy EM Accuracy Accuracy
RECALL†	Generation Quality	Counterfactual Robustness	R-Rate (Reappearance Rate)
RAGAS‡	Retrieval Quality Generation Quality	Context Relevance Faithfulness Answer Relevance	* * Cosine Similarity
ARES‡	Retrieval Quality Generation Quality	Context Relevance Faithfulness Answer Relevance	Accuracy Accuracy Accuracy
TruLens‡	Retrieval Quality Generation Quality	Context Relevance Faithfulness Answer Relevance	* * *
CRUD†	Retrieval Quality Generation Quality	Creative Generation Knowledge-intensive QA Error Correction Summarization	BLEU ROUGE-L BertScore RAGQuestEval

EVOLUZIONE DEL RAG



EVOLUZIONE DEL RAG

- **RAG naive**: niente accorgimenti pre e post retrieve;
- **RAG avanzato**: introduce l'ottimizzazione della query prima del retrieve e il reranking dei documenti dopo;
- **RAG modulare**: introduce diversi moduli, che possono essere configurati (o disabilitati) e si adatta a molteplici domini;

EVOLUZIONE DEL RAG

RAG modulare

- **search**: consente la ricerca diretta delle informazioni da diverse fonti e database, adattandosi a molti contesti;
- **fusion**: permette più query input dopo l'espansione;
- **memory**: sfrutta la memoria del LLM per migliorare la fase di recupero;
- **routing**: sceglie il percorso elaborativo ottimale della query;
- **predict**: riduce il rumore e migliora l'accuratezza in fase di generazione, sempre facendo uso del LLM;
- **task adapter**: rende il sistema RAG versatile per diverse applicazioni downstream.

TRAINING DI SISTEMI RAG

- **Training-free**: non lo richiedono, come i sistemi basati su prompt engineering, che basano la qualità dell'output su una scrittura meticolosa della query;
- **Training indipendente**: il retriever e il generatore vengono allenati in maniera separata, senza interazione;
- **Training sequenziale**: viene prima allenato il retriever (oppure il generatore), quindi viene freezato e si allena l'altro componente;
- **Training end-to-end**: il retriever e il generatore vengono allenati entrambi, in sinergia, migliorando ulteriormente le prestazioni. Questa è la caratteristica fondamentale del paradigma **RAG 2.0**.

GRAPH RAG

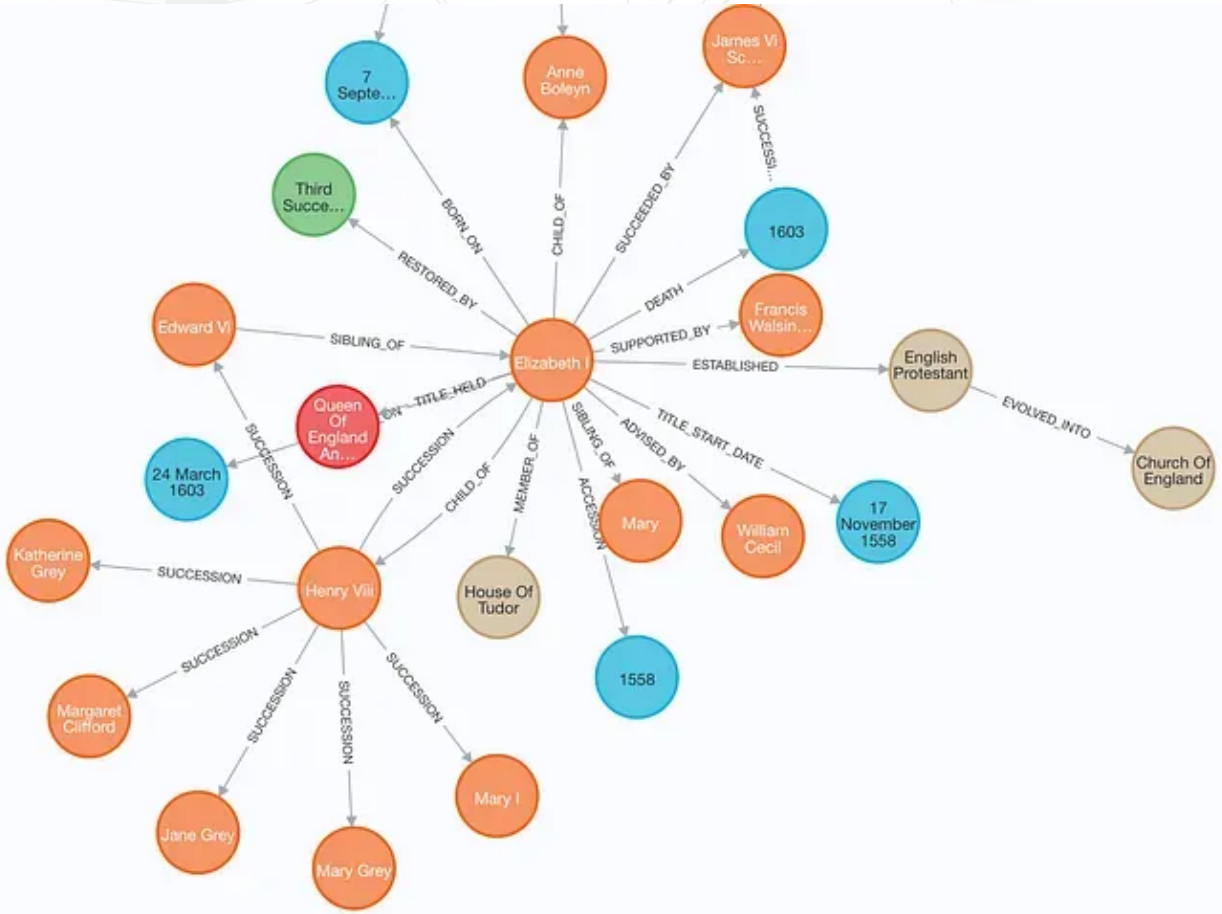
I Retriever vettoriali soffrono di una grande limitazione: **non sono capaci di rispondere a query “astratte”**, per le quali non è sufficiente il semplice recupero di informazioni da documenti ma che richiedono capacità di generalizzazione, di mettere insieme diversi concetti ed elaborare un responso, proprio come può un essere umano.

Ad esempio un sistema RAG basato su retriever vettoriale non riesce a dare la risposta a domande del tipo “*Chi furono i presidenti USA più influenti degli ultimi 50 anni?*” o “*Quali sono i vantaggi di questo prodotto rispetto all’altro?*”

GRAPH RAG

Soluzione: RAG basate su **grafi di conoscenza** (KG).

I documenti, invece di essere embeddati, vengono usati per estrarre **entità** e **relazioni** e creare un grafo. Così si creano **aree tematiche**, alle quali appartengono diverse entità collegate tra loro, e il sistema può comprendere tutto ciò che fa riferimento a una certo argomento.



GRAPH RAG

1) **Estrazione del grafo**: viene usato un modello LLM per identificare entità e relazioni (*es: Microsoft GraphRAG, che fa uso di modelli di OpenAI*);
2) **Salvataggio del grafo**: si usano database a grafi come ArangoDB o Neo4J, basato su AuraDB;
3) **Identificazione delle comunità**: gruppi di nodi più connessi di altri, aree tematiche;
4) **Report di comunità**: riassunto di ciascuna comunità che mette in risalto entità e relazioni;
5) **Map-reduce**: la query viene mappata in una lista di report di comunità; l'LLM calcola il punteggio di rilevanza di ciascuna comunità, si prendono solo le comunità più rilevanti per generare l'output finale.

RAG VS FINE-TUNING

- Minor rischio di allucinazioni
- Efficienza notevolmente superiore
- Non serve riallenare il modello se le informazioni non sono più aggiornate, basta solo integrarle in fase di retrieve
- Grande versatilità in molteplici ambiti

Tuttavia, il fine-tuning può essere utilizzato per migliorare le prestazioni del generatore, secondo il paradigma **RAG 2.0**.

ALCUNI TOOL UTILI

Langchain

Libreria Python che consente la creazione di pipeline LLM (anche con sistemi RAG) in maniera semplificata, tramite una sequenza di operazioni:

- Scelta del modello (open source: HugginFace, Llama, BLOOM, ecc.);
- Inserimento del prompt di input, che può subire espansione e/o riscrittura, tramite opportuni moduli di Langchain;
- Accesso a diversi tipi di dati esterni tramite i Loader, che li trasformano in chunk;
- Salvataggio dei documenti trasformati in database;
- Concatenazione del modello con altri componenti (per esempio più modelli LLM);

ALCUNI TOOL UTILI

Database

Modulo Python che implementa un database in cui vengono salvati i dati esterni trasformati, per consentire la fase di retrieve.

- Retrieve denso: database vettoriali come Qdrant, Weaviate e FAISS;
- Retrieve nelle GraphRAG: database a grafo come AuraDB (tool: Neo4J) e ArangoDB;