# Indice

# 1 Introduction

This thesis work adopts a primarily **experimental** perspective, indeed: the goal is not to build a production-ready system nor to optimize an architecture with respect to latency, costs, or data-governance constraints. The purpose is instead to understand which choices have the greatest impact on retrieval quality and, consequently, on the answers produced by a Language Model in a RAG pipeline. The starting point is that the performance of a RAG system largely depends on retrieval and on how you make information queryable; for this reason, different RAG approaches are evaluated and compared: from simple solutions to more structured pipelines, discussing how these position themselves with respect to the fundamental criticalities of modern RAG. The analysis focuses on the families: Hierarchical RAG, Naive, and Adaptive. These approaches are particularly interesting when the document base is heterogeneous, as in Data Lakes: not only text, but also relational tables, metadata... which are hardly reducible, without excessive information loss, to a set of independent chunks; graph-based structures make it possible to model retrieval, with respect to non SQL-like queries, even for structured data such as these. From a methodological standpoint, the thesis proposes an experimental comparison by observing how performance changes as a function of:

- **indexing strategy** (flat index vs hierarchical / graph structures),
- **retrieval strategy** (single-shot top-$k$ vs communities/paths),
- **source integration mode** (unstructured text and structured tabular data),

The implementation uses multiple technologies: Azure AI Foundry as a baseline for a Naive RAG, LangChain and ElasticSearch to build more articulated variants (for example CoT and GraphRAG) and Neo4j to manage graph-based structures. These choices also follow the setup of the work: not to seek "the" solution, but to measure the tradeoff that exists between performance and retrieval.
Many criticalities attributed to generative models (hallucinations, incoherences, generic answers) often depend on the upstream problem: retrieving information that is pertinent to the question; the following chapters formalize the RAG paradigm, discuss the limits of naïve approaches, and present the experimental comparison among the considered methods, with particular emphasis on hierarchical and graph-based solutions.

# 2 Prerequisites

This chapter aims to provide the essential notions to correctly interpret the design choices and the technical components of the work. Some concepts will be introduced from scratch during the discussion, others recalled for coherence, but for the most part spending additional time, each time, on the necessary

definitions would weigh down the treatment and draw excessive attention away from the main topic; for this reason, we split the prerequisites into a dedicated section.

## 2.1 Inverted index

The *inverted index* (inverted index) is the key data structure of traditional search engines: it associates each term with the list of documents in which it appears and is entirely based on the idea that a query can be answered by a given set of documents that you can define as relevant because they possess an exact lexical overlap with respect to the request itself; indeed, in this way query evaluation avoids a full scan of the corpus and is reduced to combining the *posting lists*, that is, the lists of documents, associated with each query term. Each list typically contains document identifiers (docID) and, often, information useful for ranking such as the term frequency within the document. The inverted index therefore makes search efficient even on very large collections because it shifts the computational cost from reading documents to the *merge* operation among sorted lists; in the vast majority of search engines, such as Google for example, an intersection of posting lists is used, that is, a default AND between the query terms, precisely recalling the previous concept of lexical overlap.

## 2.2 Positional index and Phrase Queries

A *positional index* extends the inverted index by storing, for each term and for each document, also the positions (offsets) in which it appears. This additional information also enables different types of queries: *phrase queries*; these are conceptually simple, because you are not interested in exact lexical overlap independently of position, in the previous case the AND between query terms only imposes that they must appear in the reference documents but, potentially, also in a different order, but you want to preserve positional order as well. In fact, they require that terms appear adjacent and in the correct order, basing verification on comparing positional information: given a candidate document, one typically checks whether, for each pair of terms, the difference between relative positions is the same in the document and in the query. Phrase queries extend the very concept at the basis of so-called full - text search: exact lexical overlap is relevance, indeed they coincide with the case in which token - based search performs, in reality, better in general; on this concept and on the problems that derive from it we will return later.

## 2.3 Tokenization and Analyzers

In discussing Inverted and Positional Index we assumed that you already have those 'terms' to which you associate posting lists: in reality this often entails real pipelines that you apply to the raw incoming text, formally these are defined as *analyzers*. Among the most common transformations are *case-folding*,

*stopword* removal, and stemming or lemmatization procedures to bring morphological variants back to a more stable form, with the main goal of reducing text variability and increasing *recall*, while accepting situations in which semantically different words are reduced to the same syntactic root. The most common analyzers differ above all in their tokenization strategy: a *standard* analyzer splits text based on "word" boundaries according to the Unicode standard, while a *whitespace*-based analyzer simply splits on spaces, resulting in a simple but often too coarse approach especially in the presence of punctuation; there are other variants such as the simple, stop analyzer... but the two above are the most famous.

Although tokenization seems like a simple process, it is in fact particularly complex relative to requirements: when one talks about retrieval-oriented tokenizers, one typically exploits sets of simple rules, such as the Unicode or Whitespace ones discussed previously, different is the case of NLP models such as Transformers, which we will discuss later. In these cases, subword tokenization methods such as *Byte Pair Encoding* (BPE) and *WordPiece* are widespread. Without going excessively into detail, which is not the purpose of this introduction, the aforementioned methods split words into sub - units to handle rare vocabularies, even 'invented', and morphological variability in the general case, improving the model's generalization to the very variability of the text it may have to process. However, these schemes are often ill-suited to classic full-text search: they introduce a misalignment between the user's intuition (who formulates queries in words) and the unit actually indexed (subword), they reduce the interpretability of *highlight* and snippets and, above all, they increase the number of tokens per document, with a direct impact on index size and on the length of posting lists. It is interesting to note how the consequences are almost complementary in Transformers and in traditional retrieval systems. In Transformer models, subword segmentation (for example BPE) is advantageous because it drastically reduces vocabulary size: instead of storing a distinct token for each rare or morphologically complex word (e.g. `fortissimo`), the model can compose it from more frequent units (e.g. a root `fort` and a suffix `##issimo`, an exclusively intuitive example), optimizing lexical coverage at the same vocabulary size. In the case of an *inverted index*, instead, the goal is different: one wants the indexed unit to coincide as much as possible with the user's search unit and for terms to be *stable*. Systematically segmenting into subword tends to have the opposite effect: it increases the number of indexed terms, makes posting lists denser (because frequent subwords appear in many different words) and can worsen matching precision, since fragments shared by unrelated words generate noisy matches from the intersection. While, therefore, in Transformers subword decomposition is an excellent strategy to compress the vocabulary and generalize better, in classic full-text retrieval it risks turning the problem into matching on units that are too atomic, encouraging something that in the literature is referred to as the *Vocabulary Mismatch Problem*: the quality of results returned by a search engine depends on how the user formulates the query, or rather, on how aligned its lexical distribution is with respect to that of the documents, for

this reason one tries to keep tokenization as aligned as possible with what the user naturally expresses and segmentation into sub - units creates more problems than advantages; this does not mean that you cannot act on the query, for example via rewrites, it simply means that tokenizing like Transformers do is not a great idea.

## 2.4 Metrics in Full-Text Search

The documents obtained from posting lists are then ordered based on scores derived from the importance that the query tokens assume within the document; just as retrieval is based on lexical overlap, scores also take up this concept to build the so-called *ranking*. Two of the most widely used metrics in this context are *TF-IDF* and *BM25*, both founded on the idea that the informativeness of a term depends mutually on its local and global importance: if it is a term that is very widespread across all documents, then it is not a discriminant, otherwise if it is very widespread within the single document and little/medium in the others it is a discriminant.

**TF-IDF** TF-IDF (*Term Frequency – Inverse Document Frequency*) assigns a weight to query terms as a function of: the term frequency in the document (*TF*), capturing how representative that term is of it, and the term frequency in the entire collection (global) (*IDF*), which reduces the importance of excessively common words in the corpus, especially thematic ones, think of a corpus of medical documents for example. Conceptually, a document's score with respect to a query grows when the query terms appear often in the document, but above all when such terms are rare in the corpus, in a way fully aligned with what was mentioned before. TF-IDF is simple, effective, and constitutes a natural basis for lexical ranking, while suffering from the fact that the TF component can grow excessively for very long or repetitive documents: the longer a document is, the higher, on average, the TF of terms will be; this creates a bias in the assigned scores, which tend to systematically prefer longer documents and not more relevant ones.

**BM25** It adds to TF-IDF a non-linear normalization of Term Frequency: less harsh than the direct one with respect to document length, but with the same advantages. The idea is always the same: a document that has a good compromise between local and global frequency of the query terms is a relevant document. Length normalization simply aims to prevent longer documents from always being returned and to reduce the bias we discussed previously; this, as said, is applied to the TF term of TF - IDF, since IDF does not depend on document length.

With Chunking, discussed in the following chapters, one sees an indexing by 'local portions' of the document: generally this also allows you to have a structural normalization for TF-IDF without resorting to BM25, however the latter

remains the de facto standard since, depending on the strategy, you may not have all chunks of identical size.

## 2.5 Typical Problems of Full-Text Retrieval

We have already discussed the *Vocabulary Mismatch Problem* and all the criticalities of Full-Text search reduce to this; we can also study it in a different way, primarily as a function of lexical distribution. In fact, the complementary observation holds: a greater lexical overlap does not necessarily imply, in general, a greater semantic overlap; the presence of the same words in query and document does not guarantee that they are referring to the same meaning. This phenomenon is closely linked to domain *polysemy*, whereby a term can take on different senses depending on context, increasing ambiguity and degrading the precision of retrieval based on lexical overlap; conversely, in more *monosemic* domains, it is often observed that terms are more stable and less ambiguous, consequently lexical matching is more semantically reliable.

Already from here one can grasp a concept to which we will return multiple times in RAG: retrieval must be designed, tuned, and optimized as a function of the domain it belongs to, unless one is willing to settle for suboptimal performance. When it is possible to estimate the lexical distribution of the domain with good accuracy, one can often obtain high performance even with relatively simple approaches, avoiding introducing unnecessary complexity beyond traditional full-text search. This dependence on the domain becomes even more evident when one adopts *hybrid search*: combining lexical and dense signals requires appropriately weighting their respective scores and such weights are not universal but emerge above all from the characteristics of the corpus and the expected queries; essentially therefore, the choice between full-text, dense retrieval, or a combination of the two is strongly driven by the domain and by the distribution of queries.

## 2.6 Dense retrieval

*Dense retrieval* is a search paradigm in which documents and queries are represented vectorially in a continuous space and retrieval occurs through proximity mechanisms between the vectors defining queries and documents; one exploits similarity measures, mainly cosine-based: a metric that quantifies the angle between two vectors, the smaller it is the more the vectors you draw from the origin of the plane coincide and, therefore, the points in space representing queries or documents coincide in the same position and are similar. Operationally, the system computes such a vector (embedding) for the query, queries a vector index, and returns the $k$ documents with maximum similarity. The main advantage over classic full-text is the ability to capture *semantic* correspondences: a document can be relevant even without sharing exactly the same words as the query, structurally reducing the *Vocabulary Mismatch Problem*, indeed the dense representation focuses precisely on capturing the meaning of the text and

of the query, avoiding modeling it in a 'latent' manner with respect to lexical overlap.

## 2.7 Embeddings

When one speaks of *embedding* one refers to the aforementioned vectors; more formally: an embedding is a function that maps a discrete object (word, sentence, document) into a real vector $\mathbf{e} \in R^d$, with the goal of projecting into a space where its geometry reflects useful relations: nearby points should correspond to semantically similar contents, conversely distant ones. In this view, semantics emerges precisely with respect to the arrangement of points in space: similar concepts tend to form dense regions or *clusters*, while distant concepts are placed in areas that should be as separated as possible in space; the quality of this depends not only on the model, but above all on the training criterion. In modern retrieval systems one often uses *contrastive* training objectives: the idea is to "shape" the vector space so that a query ends up close to truly relevant documents and far from irrelevant ones. Concretely, given a positive example $(q, d^+)$ and a set of negatives $\{d_1^-, \ldots, d_m^-\}$, one obtains the embeddings $\mathbf{q}$, $\mathbf{d}^+$, and $\mathbf{d}_i^-$ and optimizes a loss that increases $s(\mathbf{q}, \mathbf{d}^+)$ and reduces $s(\mathbf{q}, \mathbf{d}_i^-)$, where $s(\cdot, \cdot)$ is the adopted similarity measure. Intuitively, the loss forces the query to "choose" the correct document among many alternatives, or rather, we are directly indicating what for us is semantically correlated and what is not.

In the following discussions we will explicitly recall the relationship between **anisotropy** and *contrastive* loss, therefore it is not useful to anticipate all details now; it is however important to highlight a key point: pre-training alone is rarely sufficient to obtain embeddings suitable for semantic retrieval. Models like BERT, for example, learn vector representations because they are trained to model grammatical, semantic, positional relationships... in language through so-called auxiliary tasks such as *masked language modeling*. However, the fact that a model can represent language well does not automatically imply that the produced embeddings are also geometrically well structured according to the previous needs: informally, pre-training teaches the model to represent the contextual meaning of language, namely that the same word can take on a different semantics depending on context (for example *apple* as a fruit or as a company), indirectly also teaching the meaning of words, however this does not automatically imply that the embedding space becomes well separated as we expect and as dense retrieval requires; indeed, the notion of "difference" that helps separate the space is something relative to the post-training phase and closely linked to the loss you use. This step is crucial to make querying via nearest neighbors effective and clarifies why a good language model does not necessarily coincide with a good retrieval model; after all, contrastive losses often correspond to later fine-tunings, where as an additional objective you also have that of modeling complex meaning nuances in your domain that a generalist pre-training does not allow you to.

## 2.8 Transformer

We did not say it directly, but reading a bit between the lines: embeddings, today, are all obtained from Transformer-based Deep Learning models; this does not necessarily imply that it has always been done this way, indeed: approaches such as Word2Vec or GloVe are not based on Transformers, but rather on the idea *tell me who you hang out with and I'll tell you who you are*, modeling contextual meaning in different ways. The big 'boost' in terms of the quality of the produced embeddings historically happened precisely with Transformer architectures: attention makes it possible to solve the aforementioned traditional approaches' problem of the 'staticity' of the produced embedding; GloVe, like Word2Vec, at inference provides static matrices such that you can obtain an embedding as a vector product between a OneHotEncoded representation of the single word and the embedding matrix itself, essentially a lookup, where the sentence embedding often coincided with an arithmetic mean of those of the single tokens; the problem is that, here, you have a single embedding for each word and independently of what is in its neighborhood: there is no contextual meaning, attention, instead, starts from a static embedding and modifies it as a function of the embeddings of all the other nearby tokens, producing an enhanced representation.

### 2.8.1 BERT

Discussing BERT is fundamental to better understand the Bi and Cross-Encoders that we will discuss in the following chapters, since the latter, although more sophisticated, are still based on the encoder-only architecture introduced precisely by BERT. *Bidirectional Encoder Representations from Transformers* is a model based on the homonymous Transformer architecture that uses exclusively the *stack* of **encoders**, processes a textual sequence via **self-attention** and produces, for each token, a contextual vector representation. The distinctive characteristic of BERT is its **bidirectional** nature: each token can integrate information coming both from the context on the left and from that on the right, because during training a *causal mask* is not applied as happens in generative models such as GPT; BERT, in fact, is not auto - regressive and the main advantage in NLP of using architectures of this type lies, precisely, in the interaction between each token. In the case of generative models this does not happen: there is the causality constraint, the token $T + 1$ is generated on the basis of the token $T$ produced at the previous step and, consequently, exploiting an attention mechanism that allows previous tokens to interact with future ones loses meaning simply because you do not know them; there are, then, considerations that could be made at the level of 'peeking' in training, but intuitively the concept remains solid even like this. Bidirectionality therefore does not depend on self-attention "in itself", but on the fact that it is not limited to seeing only the past and this is consistent with the objective of *understanding* rather than generation.

BERT's original pre-training is formulated as *multitask learning* and mainly combines two tasks: **Masked Language Modeling (MLM)** and **Next Sentence Prediction (NSP)**. In the MLM task one typically selects about 15% of the tokens and applies the 80/10/10 rule: in 80% of cases the token is replaced with `[MASK]`, in 10% with a random token and in the remaining 10% it remains unchanged. The sequence perturbed in this way is given as input to the encoder and the model must reconstruct the original token at the selected positions; the loss (cross-entropy) is computed *only* on such positions. Overall, training via MLM induces BERT to produce fairly rich embeddings: to correctly predict the masked tokens, the model must exploit syntactic and semantic dependencies present throughout the sentence, something that self-attention enables naturally; this makes BERT particularly effective in language understanding tasks, such as text classification and Named Entity Recognition (NER).

The NSP task, introduced in the original version, aims to model relationships between sentences: given an input of the form `[CLS] sentence_A [SEP] sentence_B [SEP]`, the model predicts whether the second sentence actually follows the first in the source text. The decision is made starting from the embedding of the special token `[CLS]`, which has no lexical meaning but, thanks to self-attention, can gather information from the entire sequence; how it does this is quite simple: it is a neutral (invented) token that does not appear in the training texts, it is placed at the beginning of the sequence precisely so that it can absorb the totality of the meaning of all subsequent tokens, thus the entirety of the sentence, thanks to attention itself. For this reason, `[CLS]` tends to become a "global" representation useful in sentence-level tasks. In subsequent implementations, such as RoBERTa, the NSP task was removed: the underlying idea is that obtaining a `[CLS]` *general purpose* embedding already well calibrated for many tasks is an objective that is too ambitious for pre-training alone, and that practical utility emerges above all after supervised *fine-tuning* on the specific task. BERT is conceived as a *general-purpose* model for language understanding, however the big advantages come when one specializes usage to a given task: note that by specializing one does not necessarily mean something relative to a given domain, such as fine-tunings for meaning nuances, but precisely an 'operational' task. There are various examples:

- *sentence classification* tasks (for example sentiment analysis), where a simple classification head is added that takes as input the final embedding of the special token `[CLS]`. Since the loss function directly supervises this representation, training pushes `[CLS]` to condense the information of the entire sequence into a single vector, useful for "global" decisions at the sentence level
- In *token-level* tasks such as *Named Entity Recognition*, instead, the objective is not to assign a label to the text as a whole, but to produce a prediction for each position. One therefore applies a classification head to each contextual embedding at the output of the encoder, obtaining a sequence of labels (for example `B-PER`, `I-ORG`, `O`, ...). In this scenario,

model quality depends above all on the ability to accurately represent the *local* context of each token, more than on the existence of a single summary vector.

- In retrieval instead, BERT is adapted with a fine-tuning that includes the aforementioned contrastive losses; architectures such as SBERT introduce this explicit tendency to optimize text representations so that they are comparable with a similarity metric (cosine or dot product). The textual embedding, obtained typically from `[CLS]` or via pooling over positions

To conclude, in my view, it is useful to give a general overview of what happens "inside" BERT during the construction of embeddings. At input, each token is transformed into an initial embedding via a matrix $W$, also learned during training: for each one, positional information is taken into account via a positional encoding strategy, dependent on the version of BERT that is considered; we avoid discussing positional encoding further because it is in fact the least interesting thing. The **Self-Attention** mechanism operates precisely on such initial embeddings: the key operation by which a contextual representation is built for each token by integrating information from the other tokens in the sequence via, essentially, appropriately weighted linear combinations. For each embedding three linear projections are computed: **Query** (**Q**), **Key** (**K**) and **Value** (**V**). Attention is obtained by comparing each query with all keys via dot product, then applying a softmax to obtain normalized weights:

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

The result is a weighted combination of the *values*: each token "decides" how much information to absorb from the others, assigning larger weights to the positions most relevant for the current context. Intuitively, the **Query** (**Q**) is "the question" that each token asks to the rest of the sequence: for the $i$-th token, $q_i$ asks "which other tokens are relevant to me?". The **Key** (**K**) is instead the "key" with which each token describes itself: the vector $k_j$ of token $j$ indicates "in what way I could be relevant to others". Finally, the **Value** (**V**) is the informative content that is actually aggregated: the vector $v_j$ contains the information that, weighted by attention, contributes to building the output; the contextually enhanced representation of each token embedding comes, in fact, from a linear combination of the attention weights, given by the softmax, with respect to the values $V$. **Multi-Head Attention** repeats the same calculation on multiple heads in parallel with different projections: different heads can specialize in capturing signals that are also different from simple semantics, for example syntactic dependencies or even more complex positional ones. The heads' outputs are concatenated and reprojected, the entire block is accompanied by *residual connections* and *layer normalization*, useful to stabilize optimization and to preserve the information flow across layers.
Alongside attention, each layer includes a **Feed-Forward Neural Network** (FFNN) *position-wise*, applied independently to each token: it is typically a two-layer MLP with a non-linearity that expands the dimensionality in the hidden

layer and then brings it back to the original one:

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

If attention realizes information *mixing* across tokens, the FFNN carries out the non-linear transformation that *models* and reorganizes the information for each token, increasing the expressive capacity of the model and making representations progressively more complete and informative. Also in this case, residuals and normalization facilitate composing many encoder layers without degrading or running into the classic training problems of Deep architectures.

## 2.9 Hybrid Search

In light of what has been discussed, it is easy to understand the idea of *hybrid search*: combining lexical signals (full-text) and semantic signals (dense retrieval) to exploit the complementarity between the problems of one and the advantages of the other. The idea, precisely, is that lexical models (for example BM25) excel when the query contains discriminative terms and when the user requires exact matches, while dense retrieval is more effective in handling synonymy, paraphrases and queries phrased differently from the document text. Operationally, in fact, a hybrid system can retrieve candidates from both channels and then merge them, computing a combined score; in both cases, fusion aims to obtain a ranking that structurally takes into account both the full-text score and the dense one, returning the best documents by compromise.

A common approach is to retrieve two top-$k$ lists (lexical and vector) and unify them with:

- **Reciprocal Rank Fusion (RRF)**: instead of directly combining scores, one combines *ranks*. If a document $d$ appears at position $r_1(d)$ in the lexical list and at position $r_2(d)$ in the dense list, the final score can be defined as

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

  where $k_0$ is a constant that dampens the effect of the very first positions. In this way, documents that are "good" in both the full-text and dense rankings are rewarded, obtaining a fairly robust fusion especially in the case where the single list scores are not comparable; RRF is powerful precisely because you do not have to worry a priori about normalizing scores to a common scale and this is also clear by looking at convex combination, where you would have one term much more preponderant than the other and that, regardless of the weight, would risk dominating in the definition of the ranking

- **Convex combination (weighted sum)**: one normalizes the scores of the two channels and computes an aggregated score via a weighted sum:

  $$s(d) = \alpha\, s_{\mathrm{lex}}(d) + (1-\alpha)\, s_{\mathrm{dense}}(d), \qquad \alpha \in [0,1].$$

The parameter $\alpha$ controls the tradeoff between lexical matching and semantic affinity. This strategy is very simple and effective, however it requires attention to normalization and to the choice of $\alpha$

In general, hybrid search is particularly useful in cases where one of the two components fails systematically: for example, a query with very specific terms (codes, proper names, acronyms) tends to favor lexical search, while "conceptual" or paraphrased queries benefit from the dense channel; fusion therefore aims to reduce the probability of *zero-hit* and to make retrieval quality better on average.

## 2.10 Approximate Nearest Neighbor (ANN) and HNSW

The *k-Nearest Neighbors* (KNN) search is the fundamental operation underlying *dense retrieval*: given a query embedding $\mathbf{q}$ and a set of chunk/document embeddings $\{\mathbf{d}_1, \ldots, \mathbf{d}_N\}$, the objective is to find the $k$ "closest" vectors according to a given metric, typically cosine similarity; the key assumption of dense retrieval, as said, is geometric: if the embedding space is well structured, proximity between points in space approximates relevance with respect to the query's *information need*, therefore the documents closest to $\mathbf{q}$ are good candidates to use as context in a retrieval process. However, exact KNN quickly becomes prohibitive on very large corpora: a brute-force search requires comparing $\mathbf{q}$ with all $N$ vectors, with cost linear in $N$, in addition to the cost of computing similarity with respect to the embedding dimensionality $d$; although often neglected it can become relevant.

In real retrieval scenarios (millions or billions of vectors, real-time latency constraints), this full scan is absolutely unmanageable: one therefore needs an index that allows one to drastically limit the number of comparisons, while still maintaining high *recall*. *Approximate Nearest Neighbor* (ANN) algorithms are born precisely to meet this need: they accept an approximation of the neighbors, abandoning the idea of solving the problem to optimality, but returning a set of *almost* optimal candidates in exchange for a significant reduction in the computational resources required by the procedure. Many approaches are based on building and populating a data structure that organizes vectors so as to explore, at query-time, only a small portion of the space, avoiding the full scan and optimizing the explicit trade-off between accuracy and performance that is typical of approximations.

Among the most widespread ANN methods, *HNSW* (*Hierarchical Navigable Small World*) is a de facto standard for dense search: it builds a proximity graph in which each node is a vector and edges connect "nearby" vectors; the structure is *hierarchical*: at higher levels the graph is sparser and is used for high-level navigation towards regions promising for the query, while at lower levels it is denser and makes it possible to make search more precise by reasoning locally; one often speaks of alternating between *coarse* and *fine search*. To be

precise: "sparser" means that higher levels contain far fewer nodes, indeed not all vectors appear at all levels, and node assignment to a level is probabilistic, with a node that can also appear at multiple levels. The hierarchy acts as a high-level summary of the space as a whole, useful to intuitively make "long jumps" towards the correct region, while the lower levels, more populated, make it possible to answer the query more precisely. At query-time, one starts from an *entry point* at the high levels and proceeds with a greedy search: one explores a limited set of candidates, iteratively updates the best neighbor and descends level by level as one gets closer to the most promising region, up to the base level where the final top-$k$ are selected. This scheme typically achieves high recall with very low latencies, making HNSW particularly suitable to the application conditions in real dense search cases. It is not surprising, indeed, how HNSW is extremely used also in the SDKs provided by the major cloud providers: for example, Azure AI Search exposes HNSW as the reference ANN algorithm for vector search, precisely because it offers the best trade-off.

## 2.11 PageRank

**PageRank** is a classic algorithm to estimate the importance (authoritativeness) of a node in a graph, originally introduced in the context of the web, where nodes represent pages and edges hyperlinks [Page et al., 1999]. The basic intuition is that a page is all the more authoritative the more it is "cited" by other pages and that a citation coming from an already authoritative page is worth more than one coming from a marginal page; in this sense, PageRank is primarily a *ranking* metric of results and not of semantic relevance with respect to a specific query. Formally, PageRank is interpreted via the *Random Surfer* model: a surfer who performs a random walk on the graph by randomly following outgoing links, but who with some probability $\lambda$ makes a jump (*teleport*) to a randomly chosen page; the stationary value associated with such a process is then assumed as a measure of node centrality [Langville and Meyer, 2004].

The interesting aspect is that, in graph structures, centrality/authoritativeness always depend on the properties that you encode via edges: in the case of the web they are links and PageRank represents how well a page is connected; if instead edges model structural or semantic relations, the meaning of "importance" changes, but the concept remains unchanged: PageRank quantifies the relevance of each node with respect to the specific graph structure and to the properties conveyed by edges.

## 2.12 Community Detection and Semantic Clustering

*Clustering* and *Community Detection* are very similar concepts, but not equivalent. Clustering, in the traditional sense, starts from a representation of points in a space and tries to group them into sets: with algorithms such as K-means Clustering, DBSCAN... the notion of "closeness" depends entirely on the representation and on the adopted metric, or rather, on what we called the 'geometry'

of the space when discussing embeddings; when one speaks of **semantic clustering**, in fact, there are no substantial differences from traditional clustering, except for the space in which you place points: if it is an embedding space, clustering means, therefore, identifying groups of chunks/documents that are *semantically aligned*, that is, that occupy nearby regions of the vector space. From here it is easy to understand how cluster quality depends critically on the embedding model and on how it structures the space.

*Community Detection*, instead, assumes that data are already represented as a graph and aims to identify sets of nodes **densely connected** internally and **weakly connected** externally; the difference with clustering does not lie, in fact, in the concept of 'group', but rather in the underlying data structure, namely the choice of whether to model possible interactions between nodes or not. Communities are often defined as graph "modules" and are the object of extensive study in the literature and especially in retrieval, in the sense that the Web can be represented as a graph where nodes are pages and edges links between them, as well as the extensive use in domains such as social networks, recommender systems... [Fortunato, 2010].

### 2.12.1 Louvain and Leiden

A very well-known algorithm is *Louvain*, which aims to maximize **modularity** (dense internally, sparse externally) via a greedy two-phase procedure repeated iteratively [Blondel et al., 2008]. In the first phase (*local moving*), one starts from a trivial partition in which each node is a community and, one at a time, each node is moved into the community of one of its neighbors if such a move produces an increase in modularity; the process continues until no further local improvements are obtained. In the second phase (*aggregation*), each found community is contracted into a *super-node* and the weights of the edges between super-nodes correspond to the sum of the weights of the edges between the respective original communities (internal edges become self-loops). The result is a smaller graph on which the same scheme is repeated, obtaining a hierarchical decomposition that tends to produce increasingly macroscopic communities at each level.

In the RAG pipelines that we will present later, we refer above all to **Leiden**: it arises as an improvement of Louvain, still keeping the idea of optimizing an objective function (typically modularity again), introducing, however, a *refinement* step that corrects a well-known criticality of Louvain, namely the possibility of obtaining communities that are not well connected; if you look entirely at the communities found by Louvain, you might see that these 'clusterize': communities formed by groups of nodes weakly connected to each other or, even, not connected, which goes against the fundamental assumption of maximizing internal community density, tending to a strongly connected subgraph in the optimal case indeed. Leiden therefore alternates the same *local moving* phases of nodes to improve the objective and *aggregation* into super-nodes, adding a

*refinement* that guarantees better connected communities and, repeating the process hierarchically, provides better guarantees on internal community connectivity than Louvain, as well as being, often, more computationally efficient [Traag et al., 2019]. To be more precise, the refinement phase intervenes within each community produced by local moving: instead of accepting the community as a single block, Leiden looks internally and 'splits' components or subsets that are not sufficiently cohesive; the nodes thus obtained can be reallocated into "child" sub-communities derived from the previous ones, in order to guarantee that each resulting group is well connected. Only after this structural cleanup does one proceed to contraction into super-nodes: thus aggregation occurs on communities that respect at least a certain level of internal connectivity, avoiding that poorly connected sub-groups are carried over to subsequent levels of the hierarchy.

## 2.13 Communities in Retrieval

From a retrieval standpoint, communities and clusters can be employed in an almost overlapping way: both define semantically coherent *regions* on which to route a query before doing a drill-down towards the components that constitute them (textual chunks, for example), recalling that the information need resides precisely in the latter.

The most common use, in fact, is *coarse-to-fine* retrieval: one restricts retrieval to only the chunks contained in the top communities by relevance with respect to the query, improving precision and reducing noise; we will return to this mechanism in the following chapters on RAG Architectures. Communities therefore act as a "thematic container" to reduce the search space and make it easier to connect information distributed within the single document or across different documents; even when community detection is technically different from clustering, in retrieval it is often understood as structured *clustering*: instead of grouping points only by closeness in vector space, one groups nodes based on how they are connected, still obtaining coherent sets with respect to the properties underlying the edges one models (semantic, positional...).

## 2.14 Generative Language Models

By *Language Model* (LM) one refers to models based on the Trasformer architecture, trained to estimate probability distributions over a vocabulary of 'tokens' referring to text or, as we have seen, sub-words to be concatenated; an LM can be generative or not, BERT is a non-generative model for example. As recalled in the section on Transformers, the reference architecture is the one introduced in [Vaswani et al., 2017], based on self-attention blocks, position-wise feed-forward, residual connections and layer normalization: although modern GPT-like generative models are **decoder-only**, the overall architecture is not that different from the **encoder-only** ones presented with BERT, except for the expected usage at inference of the model itself; the objective, now, is token

generation, indeed if in BERT the backbone is fundamental and the head is often changed as needed, here even in later fine-tunings the module you use to generate tokens remains identically the same. An LM can therefore be summarized precisely by looking at the probability distribution it learns over the vocabulary: given a context $x_{<t}$, the model produces a distribution $p(x_t \mid x_{<t})$ over possible tokens and, in the auto-regressive case, the probability of the whole sequence factorizes as $\prod_t p(x_t \mid x_{<t})$. The model does *not* "memorize" answers, but learns a function that associates a distribution over the next token to produce given, of course, a certain input sequence; what you expect the model to learn, therefore syntactic, grammatical, positional relationships... is the same as in non-generative LMs.

Generation, as said, happens in the *head* of the model: by this one means the output module that transforms the final hidden representation produced by the Transformer backbone into a prediction in the task domain; in the case of generative LMs, the *LM head* is typically a linear projection $W_{\text{out}}$ (often with bias) that maps the hidden state $h_t \in R^d$ into logits $z_t \in R^{|V|}$ over the vocabulary, followed by a softmax:

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

This head therefore explicitly implements the mapping operation towards the vocabulary and defines the distribution that the model uses at inference. There are different ways to exploit the learned distribution: always choosing the token with the highest probability (the softmax sums to 1), rather than introducing parameters, such as the *temperature* $T$, that avoid this structural bias of always choosing the best token, increasing the heterogeneity of the responses.

Attention works similarly to BERT, but this time it applies a **causal mask** constraining each position $t$ to "see" only the previous tokens $< t$; in this way, generation is **auto-regressive**: it happens one token at a time, by sampling. During training, this objective is implemented directly via *next-token prediction*: given a text $x_{1:T}$, one builds input/target pairs by *shifting* the sequence by one position, that is the model receives as input $x_{1:T-1}$ and must predict $x_{2:T}$ (equivalently, for each position $t$ it predicts $x_t$ given $x_{<t}$), optimizing a cross-entropy on the target tokens. The causal mask is therefore essential to avoid *peeking*: if self-attention were bidirectional, the representation at step $T-1$ could already "see" token $T$ during the forward pass, making the prediction task artificially easy and inducing the model to learn non-existent correspondences; at inference-time, after all, you do not have information about future tokens, so you would create a mismatch between training and usage and causality serves precisely this.

**LLM vs SLM** In the literature there is no universally accepted threshold separating *Small Language Models* (SLM) and *Large Language Models* (LLM): the distinction is largely relative to the state of the art. That said, a reasonable quantitative convention is to treat as **SLM** models on the order of **a few**

**billion parameters** (for example ∼1–8B), which remain deployable with more contained requirements and are often suitable for scenarios with lower resource availability [Subramanian et al., 2025]; many works and surveys use **LLM** to refer to so-called *foundational* models with **over ∼10B parameters** and often also on the order of tens or hundreds of billions [Subramanian et al., 2025]. In a more "industrial" perspective, one also finds operational definitions that place SLMs typically between **100M and 10B** parameters, while LLMs reach **hundreds of billions or trillions** [Chen, 2025], which, personally, I prefer.

## 2.15 Chain-of-Thought

*Chain-of-Thought* (CoT) is a family of prompting techniques that induces the model to simulate a sequence of intermediate steps before producing the final answer; in reality, the approach it is inspired by is very similar to the principle of *divide et impera*: breaking down a complex problem into simpler sub-problems and solving them in sequence; empirically step-by-step reasoning makes the Language Model much less error prone, especially effective for large models and complex tasks: after all, CoT itself coincides with what is commonly called the model's *reasoning*, that is, intermediate tokens it uses to decompose the problem and approach it, as said, step wise [Wei et al., 2022].

Operationally, CoT can be realized in different ways:

- **Few-Shot**: examples in the prompt in which, in addition to the answer, a possible chain of steps to follow is shown [Wei et al., 2022]
- **Zero-Shot**: minimalist instructions of the type "let's think step by step", inducing reasoning even without examples [Kojima et al., 2022]
- **Self-Consistency**: instead of using a single chain, one exploits multiple, let's call them, model 'runs', that is, multiple reasonings with temperature $> 0$, aggregating the most frequent or most coherent answer; this refers to concepts similar to majority vote and with strong integration in LLM ensembling, mitigating the variability of single generation [Wang et al., 2022]

If one looks at modern LLM training techniques, mainly the so-called post-trainings, one sees how models are directly trained on problems (mathematics, programming...) where one or more reasonings with which to build the solution are directly available: in modern models reasoning is not only induced by prompting, rather one tries to instill reasoning capabilities directly into the learned distribution and in such a way that CoT itself is more effective.

In this work, CoT was used above all as a mechanism of **retrieval adaptivity**: before generating the answer, the model develops a chain of reasoning by breaking the query down into simpler sub - questions to better interpret the request and estimate which evidence is necessary, which is already covered by

the retrieved context and, above all, what is still *missing* to satisfy the *information need*; thus, CoT does not only serve to "reason better" but to guide the search itself, making the pipeline, precisely, adaptive to the variability of query complexity, with a good increase in management costs.

## 2.16 LLM-as-a-Judge

Evaluating a search engine is not simple: in the previous sections we discussed the problems that exist in full-text retrieval, such as Vocabulary Mismatch, domain monosemy or polysemy... all characteristics reasonably mitigated by dense search; an aspect emerges that, here, becomes fundamental: the performance of a search engine depends predominantly on how users formulate queries, therefore you should fix a reference distribution to obtain metrics that are meaningful to the application context; this is not always possible and it is not even easy to do, and also building a ground truth is arbitrarily complex: if you reduce yourself to metrics such as Precision, Recall, F1-Score... you assume that, for each query you evaluate once the distribution is fixed, you can have a ground-truth based on all documents that answer, or not, the information need. In very large corpora, even if you were not directly evaluating chunking, you would, for each query, need labels with respect to all elements of the corpus: labeling costs explode, yet they are necessary to obtain, especially on Recall, meaningful evaluations.

To overcome criticalities of this kind, independently of the retrieval context, the *LLM-as-a-Judge* paradigm has become widespread: using a Language Model as an *evaluator* that, given an input and one or more candidate answers, produces a label following explicit evaluation criteria indicated via prompting. Essentially, the model's capabilities are not exploited only to generate an answer to the query, but also to *evaluate it*: the LLM can compare the output with a reference, or estimate its quality by assigning discrete labels or numerical scores referring to metrics indicated and explained in the prompt; in general deciding which among multiple candidate answers is the best. This approach has become popular above all for its simplicity: it enables fast and relatively inexpensive evaluations compared to human annotation, while still maintaining meaningfulness with respect to the produced evaluations. [Zheng et al., 2023, Liu et al., 2023].

Typical LLM-as-a-Judge patterns include:

- **Pairwise preference**: given (prompt, answer A, answer B), the judge chooses the best and/or provides a motivation; often more stable than an absolute score, after all you can exploit the explanation for subsequent audits and prompt engineering
- **Rubric-based scoring**: a rubric of criteria is provided (e.g. correctness, completeness, grounding, clarity) and a score is requested for each [Liu et al., 2023].

- **Reference-based grading**: the answer is compared with a gold answer; useful when you have a reliable reference and as a metric to understand the model's performance in judging, you do not always have gold answers though
- **Multi-judge / majority vote**: multiple judgments (prompt or model variations) are used and aggregated to reduce variance in responses; same concept as ensembling in ML, higher costs obviously

When it comes to evaluating a RAG pipeline, as we will explain later, judging with an LLM is very interesting in evaluating above all: **grounding**, that is whether the answer is actually supported by the retrieved evidence and does not contradict it, rather than **factuality**, that is the absence of hallucinations or statements not justified by context, and **completeness**, understood as covering the points required by the question without introducing digressions or superfluous content; moreover you could think of using it also to evaluate the quality of produced chunks, therefore as an evaluation for the chunking strategy. In short: the main advantage of LLMs, coinciding with strong general-purpose performance (zero - shotting), is seen also and especially by using them as evaluators.

### 2.16.1 Bias

Many advantages for sure, but just as many criticalities often "veiled" and difficult to interpret; to better explain what I mean: often Chinese models (e.g. Qwen, Kimi) are not good *judges* in general-purpose contexts, not so much due to performance limits, but due to *alignment* issues of judgment relative to the pre-training of these models. The judge function predominantly requires consistency in understanding and applying criteria: to put it informally, an LLM is as good a Judge as it is impartial, therefore, if training has taught it to respond by following certain policies, you automatically reduce impartiality and the quality of judgment; I previously gave the example of Chinese models precisely because they are quite emblematic of this: it is enough to try asking them for a judgment on the argumentative quality of two texts that talk about politically sensitive topics, such as Taiwan for example, to quickly realize what we are saying.

The most well-known and interesting problems of LLM judging, linking back to the previous intuition, in my view can be summarized in:

- **Style bias**: the LM may prefer longer answers, more confident in tone or better formatted even when they are less relevant with respect to the information need
- **Prompt Sensitivity**: here it makes no sense to do fine tuning, or rather, using the LM itself as a Judge imposes wanting to exploit general-purpose capabilities and fine tunings weigh down situations in which you would like, for example, to easily change different evaluative metrics; however, small changes in the prompt can alter judgments a lot and it is necessary

to provide stable evaluations; for this reason, one often speaks of 'setting the *temperature* to 0': this is a parameter that indicates, intuitively, the model's 'creativity', that is how much you want answers to differ given the same prompt. For judging, obviously, you would ideally want these to be as deterministic as possible, low variance in short

- **Score calibration**: numerical scores (e.g. 1–10) are often poorly calibrated and not comparable across prompts/models; these are quite dangerous, because they induce an illusion of precision: the LM answers, for example, 5 not because it realistically evaluates with respect to a scale it is aware of, but because it is the probability distribution over the tokens to generate, acquired during pre-training, that indicates that score; for this reason, in my view, when one uses an LM as a Judge it is always better to refer to categorical labels: the semantics of terms is learned and it is, for the model, easier to model a term-input association than an evaluation-input association, since numbers themselves do not have a well-defined semantics or one that can be connected to the prompt itself

A particularly interesting aspect is the strong analogy with the classic problems of *crowdsourcing*: when one wants to estimate a worker's reliability one typically introduces gold standards, majority vote, agreement measures and quality scores. LLM judging, indeed, re-proposes very similar issues and conceptually you could extend the aforementioned mitigations, very widely used in practice.

## 2.17 SDKs and Cloud Providers

Here we touch on a more 'technical' part related to the tools used to build the RAG pipelines discussed later; we always keep a conceptual tone, in the sense that discussing at a technical level 'implementation code', ultimately, adds nothing to the reader and unnecessarily weighs down the discussion. By *Cloud Provider* one means companies that primarily offer compute resources, storage and software services via distributed infrastructures (data centers), typically with the primary objective of guaranteeing **high availability** and service continuity: redundancy, fault tolerance, elastic scalability and, more generally, quality of service that is difficult to obtain on-premise. Even though, today, people talk much more about AI than Cloud, the latter still plays a fundamental role: the computational needs introduced by many of the tools presented for retrieval, such as embedding models, LLMs, indexing pipelines... are such as to make the cloud a prerequisite in many application contexts for all the aforementioned advantages.

Such Cloud Providers often provide so-called *Software Development Kits* (SDKs): they are collections of libraries, APIs and tools that facilitate access to the provider's services, allowing one to interact with them in terms of creation, deletion, monitoring... of resources. In our RAG application case, this abstraction translates into the possibility of quickly building an end-to-end pipeline using

predefined components for storage, ingestion, indexing, retrieval and integration with LLM/embedding endpoints, drastically reducing *time-to-first-result*; the main criticality, which we will discuss better later, concerns standardization itself and its tendency to constrain many choices that are critical for retrieval quality: pre-processing, chunking, ranking... mitigating, ultimately, much of the design simplicity that comes from the SDKs themselves.

## 2.18 LangChain

LangChain is a library designed to build LLM-centric applications, also offering an important abstraction level in managing LLM-based work units within larger products; the considerations I drew from working with LangChain focus, precisely, on what, in my view, is the main value: control; it allows one to explicitly define loaders, pre-processing transformations, advanced chunking strategies, embedding pipelines and the construction of vector indices, as well as to compose more articulated chains with intermediate steps (query rewriting, routing, multi-step retrieval...), making LangChain suitable for reproducing and extending RAG pipelines "managed" by providers, while maintaining the possibility to intervene on design-critical points. Essentially therefore, on the one hand cloud SDKs accelerate infrastructural integration, on the other LangChain accelerates experimental exploration and mainly when, precisely, you do not have to comply with all those constraints that derive from putting a product on the market.

## 2.19 ElasticSearch

**ElasticSearch** (ES) is a distributed query engine built on *Apache Lucene*, designed to index and query large volumes of data prioritizing scalability and low latency; it is widely adopted in industrial contexts, offering a very balanced compromise between performance and simplicity of use, for this reason I decided to adopt it also in my work. From a retrieval standpoint, ES inherits the setup of classic search engines based on *inverted index*: text is subjected to indexing pipelines defined via analyzers (tokenization, normalization, possible stemming...), with strong integration with the JSON format for documents. On this basis, ElasticSearch exploits ranking metrics that are extremely consolidated in practice and already discussed, such as BM25, enabling management of different query types: full-text, boolean, exact match... as well as vector ones according to the ANN mechanisms treated previously [Elastic, 2026].

## 2.20 Groq

Groq is an inference provider focused on **low latency** and high **throughput** in executing LLMs at competitive costs, exposing endpoints compatible with standard APIs and a fairly broad model pool to cover use cases also different from the classic QA one: general-purpose models, lighter variants for real-time,

Speech-to-Text.. The main advantage I found in usage is the simplicity of managing the API via Groq's Python library, generally resulting in less verbosity than solutions that are even more renowned such as OpenAI, or Google. In the work, Groq was used predominantly for *judging* and for generating the final answer starting from the context retrieved by retrieval.

# 3 Retrieval Augmented Generation

Large Language Models (LLMs) have made it possible to interact dynamically with a large amount of information encoded, thanks to pre - training, in the parameters of the model itself. When one speaks of **hallucinations**, one refers to a phenomenon in which the Language Model returns an answer by 'inventing' information that is, yes, often and willingly, semantically correlated with the request but is not supported by real evidence. The basic idea of RAG is simple: before answering, the system evaluates the user's request on a document corpus, then conditions generation on the basis of what the search returns, obtaining potentially more accurate and, above all, verifiable answers; there is a strong interaction, therefore, between retrieval and Language Models, indeed: in a certain sense, the latter serve only as a means of organizing the pertinent information, indeed with appropriate system prompts you can instruct them to use prior knowledge not to answer directly but to interpret information and build an answer based solely on the 'ground' provided by the search.

The importance of RAG is mainly traceable, in addition to the previous concept of mitigating hallucinations, to a strictly practical need: continuously training LMs is expensive, indeed, useless whenever information is characterized by a strong **dynamic** component; the fact that an LM can be trained on text and learn from it implies that these have a fairly natural extension to *Question Answering*, indeed the most widely used models today are called *Chat Completion Models* precisely for this, however training is not necessarily the best path to do so. In contexts, such as corporate ones for example, information typically evolves very quickly, consequently RAG focuses on designing a pipeline that first includes an efficient search over the document base, since if you can always provide a correct context you obtain good answers to user queries. Generally, it is thought that this happens independently not only of model training but also of the model itself, allowing better performance even with Small Language Models: we will return to this concept with the results, since, although very natural, it is not necessarily a correct intuition.

## 3.1 Semantic Collapse, Hubness and Chunk Dependency

In a *naïve* implementation, RAG typically proceeds like this:

1. documents are segmented into fixed-length *chunks*,

2. each chunk is transformed into an embedding vector and inserted into a vector index

3. at query time the question embedding is computed and the top-$k$ nearest neighbors are retrieved according to metrics such as, predominantly, *cosine similarity*, always following a Nearest Neighbor approach, or rather, more efficient approximate versions such as ANN but the point does not change

4. the retrieved chunks are passed to the LLM for the answer

This scheme works well in many applications, but tends to degrade when the knowledge base grows and becomes heterogeneous, in addition to peculiarities deriving from the lexical distribution of the case; it is precisely in these cases that the so-called *semantic collapse* emerges, that is the loss of useful separation between truly relevant contents and contents that are only semantically similar. Dense retrieval is based on the hypothesis that geometric similarity in embedding space is a good approximator of relevance with respect to the query (information need). However, when the index grows in size, retrieval accuracy can worsen significantly: Reimers and Gurevych show theoretically and empirically that, with large indices, dense representations can incur an increase in *false positives*, up to a tipping point in which full - text approaches (e.g. classic BM25) can become competitive or even superior [Reimers and Gurevych, 2021]. This behavior is perfectly consistent with more general aspects of the *curse of dimensionality*, where the notion of "closeness" in high-dimensional spaces loses importance compared to what one normally observes in a Euclidean plane for example [Chen et al., 2025a]. Consequently therefore, when you exploit a vector index you basically have a limit on embedding dimensionality, and thus on their representativeness, due to the *curse of dimensionality* problem and, as corpus size increases, you basically run into a *semantic collapse*, which helps to better understand, later, when we discuss Hierarchical approaches.

A second strictly correlated problem is *Hubness*: in high-dimensional spaces, some points, called hubs, tend to appear with anomalous frequency as neighbors of many other points, distorting nearest neighbor search [Feldbauer and Flexer, 2019]; in a RAG context like ours, hubs are essentially very "generic" chunks that lie in, let's say, 'central' positions of embedding space, often entering the top-$k$ regardless of the specificity of the question, reducing the quality of returned information. This hubness problem is strongly correlated with dense retrieval dependence on *chunking*: segmenting text determines the informative unit you will index, therefore your corpus, and consequently the performance advantage that semantic search adds to traditional methods; it seems trivial that embedding-based dense retrieval depends on what you use to build them, but precisely because it is a basic concept the chunking approach is fundamental. Chunks are almost never self-explanatory, indeed if too small they fragment context and break discourse, logical chains... while chunks that are too large increase noise, mix different concepts and much more easily reduce question - relevant text alignment. After all, the same information can be distributed across

distant parts of a document: chunking intrinsically makes you lose relations of this kind. Essentially therefore, part of RAG quality is "decided" even before retrieval, at the moment of defining chunks and metadata.

We will return more in depth to many of the previous concepts in the following sections.

## 3.2 Initial Proposal

The initial proposal envisioned a combination of three techniques:

1. Recursive Chunking as an alternative to Fixed to obtain a first "structural" segmentation of the document
2. Semantic clustering of chunks via embedding distance, with the objective of grouping semantically similar contents
3. LLM Stitching to rewrite the chunks of the semantic cluster in a more direct way, as well as to resolve errors deriving, for example, from preliminary OCR models

The work extended beyond the aforementioned formulation, maintaining much of the structure but integrating intuitions with respect to the supervisor's proposals; in particular: the analysis and use of graph structures (GraphRAG) integrates very well with semantic clustering, indeed to identify such subsets of chunks it is possible to use **Community Detection** algorithms such as Louvain/Leiden. The first step of the thesis nevertheless remains the analysis of chunking strategies alternative to fixed chunking, favoring *structure-aware* segmentations such as Recursive Chunking, extending also to *semantic-aware* solutions. A point that emerged during the study concerns, then, the impact of textual transformations on hybrid retrieval: we no longer adopt LLM stitching, since a rewrite, whether aggressive or not, can weaken the *full - text* component of search, decisive for a certain class of queries, precisely reducing lexical overlap between query and text, overall making the quality of obtained answers much lower for queries that are in fact simple; summaries and abstractions are, however, considered only as support to reduce the 'scope' of the search and, in any case, an integral part of the Hierarchical approach for which we give an intuition in the paragraph that follows.

The structure I built coincides with a *semantic chunk graph*, consistently with the supervisor's proposal: nodes represent chunks and edges encode both structural relations (e.g. adjacency) and semantic relations (e.g. embedding similarity); on this basis, we evaluate the *Clustering Community Based* component: the idea is to route the query primarily on an aggregated dimension given by a high-level summary of the community theme, and then to evaluate a subsequent drilldown on the chunks that compose it. This setup is consistent with the direction of hierarchical approaches in the RAG literature (e.g. RAPTOR), which

show the benefits of retrieval at different abstraction levels [Sarthi et al., 2024]; similarly, works such as GraphRAG highlight how a graph representation and community-level aggregation can support more "global" and multi-hop questions [Edge et al., 2024].

# 4 Chunking

Many aspects of chunking and the criticalities it introduces have already been discussed; it remains natural, however, to ask why it is really necessary. The first reason is structural: language models have a limited context window, which in practice almost always prevents providing an entire document (or, even more so, an entire document base) as input; consequently, decomposing text into more atomic units is not an "optional" choice, but a necessity. There is then a motivation tied to embeddings: if the goal is to represent text semantics in vectors, then content that is too long and rich in topics tends to mix different signals, since the specific information we would like to retrieve gets "diluted" in a more generic representation, conversely excessively short chunks risk losing global semantics and the context needed to interpret them correctly. Chunking, in fact, arises precisely as a compromise between these two extremes: preserving enough context to maintain coherence with respect to the reference document, but not so much as to lose specific information. Much of why, in practice, search is often a *Hybrid Search*, combining both dense and full - text search, stems precisely from guaranteeing greater retrieval specificity when lexical overlap allows it. In this sense, RAG is grounded on the assumption that large document collections cannot be included integrally in the model context; ideally, it would suffice to provide the entire corpus every time as context to the language model and obtain perfectly accurate answers. In practice, this approach is prohibitive mainly due to cost constraints (LMs are paid per input and generated token), making chunking the only viable alternative with respect to all constraints.

## 4.1 Sentence Transformers

Sentence Transformers are models designed to produce embeddings of textual spans, optimized for similarity and retrieval tasks: the typical architecture reuses a Transformer Encoder backbone (e.g. BERT/RoBERTa), even though today one also sees reuse of portions of Decoder architectures from generative Language Models, and adds a pooling mechanism (mean/max/CLS or variants) to compress token-level representations into a single fixed-dimensional vector. The resulting embeddings are projections into a space, typically high - dimensional, that aim to capture semantics "at the sentence level", so that texts with similar meaning have nearby vectors according to a metric (often cosine similarity) and semantically distant texts are well separated. Training happens with contrastive or ranking-oriented losses: in Siamese/bi-encoder configurations one uses, for example, cosine/MSE for similarity regression, triplet loss, contrastive loss and especially Multiple Negatives Ranking Loss (or InfoNCE), which pu-

shes positive pairs to get closer and separates negatives within the same batch. Through the aforementioned losses you solve a fundamental problem underlying the pre - training of Transformer architectures such as BERT: the produced embedding space is very squashed, to the point that cosine similarity differences that determine semantic difference or similarity between two texts are so small that, informally, 'everything looks the same as everything'; this is often referred to as **anisotropy** of the space.

The model one uses is very important relative to requirements: if the RAG pipeline lives in contexts with real - time constraints, then you cannot expect to use very large Sentence Transformers, or rather, those that occupy the top positions of the MTEB (benchmark) leaderboards, since they are very very slow in embedding construction, which goes entirely against high-traffic conditions. In our case, we used `BAAI/bge-large-en-v1.5`: a model with about $600M$ parameters, relatively slow but widely used among open - source ones for research projects and overall it guarantees a good tradeoff with respect to the constraints discussed previously. As the name suggests, the model is suitable only for English-language texts: we made this assumption in the choice of the testing datasets, although the real world does not necessarily put you in front of situations always of this type, indeed the use of Decoder portions from generative models as SentenceTransformer comes precisely from the fact that their pre - training makes them much more robust to language variability.

## 4.2 Strategies

The importance of chunking has already been treated at a conceptual level and, for the purposes of this section, the overview is sufficient: it is important to highlight, however, that there is no universally optimal strategy, since the goodness of the choice depends strongly on the domain and on the nature of the text. In technical documents, such as scientific papers for example, the explicit structure of titles, paragraphs, lists... is often reliable for segmenting while preserving useful signal, defining a fundamental semantic unit; conversely, in a novel semantic cohesion follows narrative dynamics (scenes, dialogues, characters, flashbacks) that make the previous boundaries too "mechanical" a way of chunking, risking breaking both structural and logical continuity. After all, texts coming from OCR or with a particular formatting are often subject to various conversion errors from commercial models and require a pre - processing phase before segmentation, since what these declare as chapter, section... could be a simple text in **bold** like this, therefore it is not always a viable alternative.

This is an important discussion point to frame the work done: the objective is to derive experimental results relative to procedures that are not built specifically for a single domain, even though to make a RAG pipeline competitive the choices you make must strictly be vertical to the particular context. Indeed, although there is no **gold - standard** I tried to build one, therefore approaches that regardless of the case guarantee reasonably decent performance and

the discussions we are going to make are to be understood as a starting point on which to base subsequent design choices and, precisely, more specific to the reference domain.

### 4.2.1 Fixed and Recursive Chunking

**Fixed Chunking** is, probably, the most famous strategy: it segments text into constant-length blocks (typically by number of characters or tokens), often with an overlap so as not to lose context between adjacent chunks; it is simple, fast and easy to parametrize, but it introduces evident structural problems because, despite overlap, nothing prevents you from producing chunks where sentences are interrupted and words split, with the risk of worsening both embeddings and retrieval due to low interpretability.

**Recursive Chunking**, as implemented in LangChain (e.g. `Recursive CharacterTextSplitter`), maintains the same fixed-dimensionality idea (`chunk_size` and `chunk_overlap`), but instead of cutting uniformly it first tries to split the text using a hierarchy of separators "from strongest to weakest" (e.g. double newlines → newline → spaces → fallback to characters), recursing iteratively until each segment fits the threshold: in this way it tends to preserve paragraphs and local units where possible, reducing the previous fragmentation of the Fixed approach. It nevertheless remains strongly heuristic: quality depends on the chosen separators and on text formatting (noisy OCR, LaTeX, markdown, etc.), still being able to produce unnatural cuts when it does not find "good" delimiters or when semantically relevant information exceeds the imposed sizes which, as said, remain fixed. However, Recursive Chunking retains the "good" simplicity that exists in Fixed Chunking (same parameters and similar costs), but structurally allows you to mitigate many of the criticalities at the basis of the Fixed approach; later we will refer precisely to this: although the strategy is strongly domain-dependent, often and willingly Recursive approaches are as close as it gets to a gold - standard.

### 4.2.2 Semantic Chunking

**Semantic Chunking** aims to segment a text following its conceptual coherence, rather than by fixed windows or structurality: a very recently used approach, and one I drew inspiration from, is the *sentence-by-sentence* one, where one computes embeddings of consecutive sentences (or short windows) and identifies a cut point when cosine similarity between adjacent portions drops below a threshold, interpreting it as a topic shift. This differentiates it from the previous approaches and makes it more robust to irregular layouts or content where structure does not coincide with semantic boundaries. There are many variants of Semantic Chunking, but almost all revolve around the same principle: segment where one observes a semantic discontinuity; the main problem of sentence-by-sentence approaches is predominantly given by defining the cut point and for this reason I built a semantic chunking variant that exploits statistical indica-

tors to make the approach more robust. From a design standpoint a rather delicate aspect emerged and that will be emblematic when we discuss results: if one progressively builds a chunk by "accumulating" sentences and representing it with the mean of already computed embeddings, then the representation often tends to collapse towards a poorly informative centroid, that is a 'Hub' embedding that encompasses the total semantic content of the document and that is always very similar to all subsequent textual content, making it harder to detect topic shifts and producing chunks by evaluating cuts only when preset maximum sizes are exceeded, practically collapsing into a fixed, in addition to the chunk itself could ideally converge to the totality of the document if such a size cap ideally were not there. The reason for this is related to the fact that using a mean of embeddings of the windows accumulated up to that point is less burdensome than evaluating a full recomputation: smaller textual portions, fewer computations, faster; this choice is precisely relative to the fact that the main inefficiency of the Semantic Chunking approach is computational complexity: the advantage that, intuitively, defining semantically cohesive units brings to retrieval is not always justified by evaluations on test sets or by the excessive increase in computation time and used resources, which makes chunking approaches that are not rule - based, as well as those that exploit LLMs, prohibitive on very large corpora [Qu et al., 2025].

The Semantic Chunking strategy I built introduces two measures to make it more stable:

- sentence pre-processing
- adaptive topic shift thresholds

After segmenting text into sentences and estimating their length in tokens, "too short" sentences are merged using statistical indicators: one computes the median of lengths and MAD (Median Absolute Deviation), i.e., the median of absolute deviations from the latter, merging sentences whose length is below the 'median + MAD' threshold; the latter is less sensitive to outliers than standard deviation, therefore it is suitable when lengths have anomalous values and this happens quite often with texts that derive, for example, from OCR, in addition the sentence splitting is always rule-based via common heuristics and is not perfect, indeed, it could generate arbitrarily short sentences; the objective is to avoid that too short units produce embeddings and, consequently, noisy breakpoints. For semantic signal, instead of embedding each sentence in isolation, 3-sentence windows are built (previous–current–next) and similarity comparison is between these: to recall overlap concepts, each window coincides with a neighborhood of the two adjacent sentences given a central reference; in simple words, this semantic approach wants to avoid that the SentenceTransformer has to infer the meaning of a sentence 'alone', by giving it context. To identify breakpoints an absolute threshold is not adopted, but an adaptive threshold derived from the distribution of distances between successive windows in the document; essentially, one computes a percentile of the distances (typi-

cally between the 75th and 90th): the percentile is the value such that a certain fraction of observations (e.g. 80%) is lower and, therefore, only the points where the distance exceeds a "high" threshold are marked as candidates, that is, the most marked differences with respect to the average of the text and that more plausibly represent topic shifts. In particular, the percentile is obviously not chosen a priori but relative to the coefficient of variation $CV = std/mean$: when distances are homogeneous (low CV) the threshold is raised using higher percentiles, while when there is greater irregularity (high CV) the threshold is lowered to intercept more topic changes. At the level of the reference text, this is equivalent to saying that the procedure adapts to the 'writing style': a document with a uniform trend, in the sense of gradual semantic transitions between concepts, tends to produce similar distances between consecutive windows and, therefore, it is convenient to "cut" only on truly marked differences since overall there is much homogeneity in the discourse; conversely, in more discontinuous texts lowering the threshold allows one to better separate topic shifts, avoiding considering two sub topics as a single chunk. Breakpoints are then filtered by imposing a minimum number of sentences per chunk and estimating token length of final segments; although many operations are done, in reality the performance bottleneck is not relative to statistical indicators, but to embedding computation itself and computationally you can consider it equivalent to traditional sentence-by-sentence.

## 4.3 Considerations on SQuAD and NQ Dataset

The SQuAD dataset (Stanford Question Answering Dataset) is a benchmark for *machine reading comprehension* in which, given a passage, the model must answer a question by extracting a textual span (that is a substring) from the context; it originates from selected Wikipedia articles and then annotated with question-answer pairs created via crowdsourcing, with answers referring to precise portions, precisely substrings, of the reference text. SQuAD was used as a base to measure retrieval statistics and compare the impact of the two different chunking strategies that are central to this work: fixed and semantic chunking. In SQuAD you have multiple questions associated with the same Wikipedia page and examples are built by grouping the entire 'context' for each question, in fact indirectly reconstructing the full page and evaluating the chunking strategy on the whole: it makes no sense to evaluate it only on the single textual portions obviously. Given the nature of the domain, therefore encyclopedic texts, queries maintain a certain lexical overlap with context (proper names, key terms, dates), while still including paraphrases occasionally.

A second dataset considered is Natural Questions (NQ), which differs from SQuAD mainly due to query nature. Questions in NQ derive from "realistic" queries performed on search engines, formulated by users who do not know in advance the content of the target document, differently from what happens in SQuAD; consequently, queries are less anchored to the text and lexical overlap is on average weaker than in SQuAD, a case in which the advantage of dense

search should excel. In this sense, NQ represents a benchmark more faithful to 'real' retrieval scenarios, where query–passage alignment is not guaranteed by annotators who focus, precisely, on single paragraphs; if we wanted, in a few words, to summarize what we said: informally NQ is a SQuAD that suffers more from the *vocabulary mismatch* problem, making the chunking phase potentially more critical not to break contexts needed for more complex semantic search.

In our experiments on SQuAD (1000 documents, 5825 queries, realistic estimates), performance is essentially saturated already at $K = 2$ (Hit@2 = 1.0 for both methods), with a very high Hit@1 and marginal differences between the two strategies (0.9888 for semantic vs 0.9906 for fixed). As shown in Table 1, chunk lengths produced by both strategies are practically identical and no clear winner emerges between the two approaches: the $\sim 150$ tokens were not chosen randomly, semantic chunking derives chunk length adaptively, as explained in the previous section, and we can exploit the average chunk length produced by the semantic strategy as the window size for the fixed one, making results perfectly comparable between the two. In a domain like this, the choice between *fixed* and *semantic chunking* has little impact on retrieval effectiveness. Shifting

Tabella 1: Retrieval Results on SQuAD (1000 Documents, 5825 Queries). Model: `baai/bge-large-en-v1.5`.

| Metric | 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 |

evaluation to NQ (still 1000 documents, 5000 queries, representative), the task becomes more discriminative, with an overall performance drop (Hit@1 $\sim 0.88$). However, even in this more complex scenario, an advantage of *semantic chunking* does not emerge: the *fixed* method is slightly superior for all considered $K$ (Table 3) and, overall, empirical results reinforce the trade-off discussed in [Qu et al., 2025]: *semantic chunking* introduces a non-negligible computational overhead, mainly due to embedding computation and breakpoint threshold estimation, without guaranteeing evident improvements in pure retrieval; obviously both NQ and SQuAD are simple datasets, just look at the metrics, however these tests reinforce experimental evidence that exists in very famous works such as those already cited and allow, in fact, to confirm the interesting considerations that are made: after all, if there is no clear delta in very simple tasks, then it does not mean that there cannot be a more marked advantage in more complex retrieval tasks but it will always be reasonably diluted and it is difficult that it can outweigh such an important increase in computational resources in the average case.

Tabella 2: Retrieval Results on Natural Questions (1000 Documents, 5000 Queries). Model: `baai/bge-large-en-v1.5`.

| Metric | 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 |

Still, the Wikipedia domain was chosen precisely to amplify the previous concepts tied to topic shifts, extending with NQ a further level of complexity, highlighting how the retrieval benefit of preserving semantic boundaries is not always automatic. Recalling the previous concept of **gold-standard**: a fixed-window strategy (*fixed chunking*), although more rudimentary, proves extremely competitive in the performance-cost tradeoff and in the case in which you do not have perfect knowledge of, for example, case lexical distribution, in my view adopting approaches such as Recursive Chunking is the most plausible path.

**Setting change.** As indicated previously, to guarantee comparability between methods one aligned the *Fixed* chunk size to that produced by the semantic method. To better highlight the previous concepts, we extended to a different setup: *Fixed Chunking* is constrained to a *rigid* and predefined window ($W = 256$); essentially, we no longer exploit the information derived from semantic chunking to calibrate the fixed window size. The objective becomes evaluating the ability of the semantic method to adapt segmentation to content with respect to a baseline with a fixed context budget.

Results in Table 3 show that, in this setting, *Semantic Chunking* maintains a consistent advantage over *Fixed Chunking* on all considered metrics. The main factor that explains the advantage of the semantic method is its ability to produce chunks that follow the informative boundaries of the text, adapting size based on topic-level dynamism of the document, indirectly with respect to its structure. In the case of NQ, where we avoided SQuAD since it is simpler and results would have been too similar to before, this translates into a larger average retrieved context (about 406 tokens, representative), which tends to preserve more cohesive thematic units until semantics itself allows it; conversely, the fixed window of *Fixed Chunking* ($W = 256$) forces much more aggressive fragmentation, which can break portions of text needed to correctly align query and content, reducing retrieval performance. Overall, these new results do not contradict the previous considerations, indeed, they extend them: one remains perfectly consistent with the observations of [Qu et al., 2025], underlining how the benefit of semantic chunking is not universal but tends to emerge when the text is particularly rich in distinct themes and fixed chunking would lose too

Tabella 3: Results on Natural Questions (1000 Doc, 5000 Query).

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

much in cohesion, however in the average case the performance gain on retrieval, although in some cases it can settle around an $\approx 10\%$ in F1-Score as in the cited works, does not necessarily change the negative balance in the performance-cost tradeoff of semantic chunking. It is interesting to note how fixed chunking, by exploiting window adaptivity, can achieve comparable performance in terms of chunk self-explainability: in this work I also evaluated a fixed-window chunking strategy where size was computed with respect to the same idea of statistical indicators of the proposed semantic chunking, however, in my view, much of this dynamism is anyway guaranteed structurally by approaches such as Recursive Chunking which overall have greater benefits and with fewer 'computations' to evaluate, hence reinforcing the previous **gold-standard** idea.

# 5 RAG Architectures

We have already introduced the fundamental idea of RAG: a paradigm in which generating an answer does not depend exclusively on the model's "internal" knowledge but is conditioned by contents retrieved from a preliminary retrieval phase; the LLM is in charge of interpreting and synthesizing information, not providing it directly. Roughly, a RAG system can be seen as a pipeline composed of two concatenated macro-phases: *retrieval* and *generation*, precisely; when one speaks of **RAG Architectures** one refers to how, implementation-wise, you decide to execute them, with particular emphasis above all on retrieval, nothing else.

Starting from this essential scheme, the fundamental elements that recur in any RAG architecture are:

- **Knowledge Base and Indexing**: how documents are acquired, normalized and made searchable; this phase includes chunking, tokenization, stopping, stemming, pattern matching, as well as the design of even complex OCR models
- **Embeddings and Similarity**: not so much vector search per se, today the algorithms are always the same and belong to the ANN family, rather design-wise one pushes on the model you use, fine tuning...
- **Ranking and Selection**: deciding which chunks to pass to the model and in what order; Ranking is a fundamental concept and one that is

very clear with search engines, today it also acquires in RAG a particular importance that we will discuss later

- **Prompt construction and context budget**: useful context is limited (token budget), therefore design-wise you must decide how many and which chunks to pass as context to the model not only at the Ranking level, but also what you can afford above all in terms of costs
- **Grounding and verifiability:** the practical objective of a RAG is not only to "answer", but to answer starting from evidence; the design of the flow must therefore reduce the ambiguity you have about 'where' the model takes the information it returns from

Next, the RAG architectures adopted and experimented with in the work are presented, following a progressive path: one starts from the most direct paradigm and then introduces variants that modify, predominantly, where and how retrieval happens and knowledge is organized. The following sections describe in detail the architectural choices for each approach, highlighting involved components, execution flows and practical implications, with a focus on the main advantages/disadvantages.

## 5.1 Naive RAG

The *Naive RAG* variant represents the most direct and "minimal" implementation of this paradigm: given a textual input, the system performs a search on one or more indices, selects the $k$ most relevant results and inserts them (often via simple concatenation) into the prompt context provided to the model; this is the same description we provided in the previous sections. This setup is extremely widespread and emblematic of the fundamental concept: the LLM does not have to "know" everything a priori, but can exploit an external, updatable knowledge base without resorting to further training. However, the simplicity of Naive RAG is also its main source of inefficiencies: small errors in upstream phases (in particular chunking and ranking) tend to propagate all the way to generation and it is often difficult to isolate cause - effect relationships; in reality, this is true in all RAG Architectures, therefore it is more correct to attribute the performance 'bottleneck' of this alternative to the fact that it uses a traditional Hybrid Search: you do not resort to Adaptive/Hierarchical mechanisms and you are, given the same chunking and ranking quality, exposed to the Hubness and Dimensionality problems discussed previously.

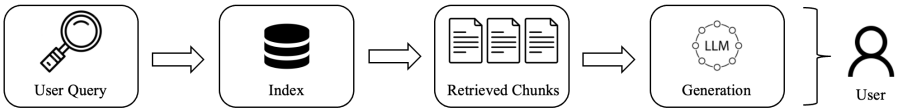


Figura 1: Naive RAG Architecture Schematic

Especially in today's literature, where there is still a lot of research on RAG, the Naive approach is viewed rather 'badly': this is a tendency tied to thinking that simple solutions are often not robust to real-world variability, in reality, especially when seeking a gold - standard, Naive RAG is often a baseline that is very hard to beat and if your corpus size is quite contained, it is probably the RAG Architecture closest to the concept of a 'to - go solution' and much of what we will discuss in the results hints at this concept here.

### 5.1.1 Azure SDK

In the carried-out work, Naive RAG was built leveraging both the Azure ecosystem and the LangChain library; the need for this comes from the desire to show two aspects:

- how simple it is, especially by leveraging SDKs from large providers such as Microsoft, to build a working RAG
- how, however, you are strongly constrained to choices that are often suboptimal on all the most critical design points

With Azure, a Naive RAG pipeline can be implemented relatively easily using Azure AI Search (ex Cognitive Search) as the retrieval engine and Azure Blob Storage as the database; conceptually, indeed, when you want to leverage SDKs from such providers, you have libraries that allow you to concatenate products belonging to different services to, in fact, realize the individual functionalities you are looking for. It is therefore important to discuss how Azure 'names' the fundamental components of a RAG pipeline, Naive or not in reality:

- **Blob Storage (Data Source)**: it represents the origin of data. Files (PDF, DOCX, HTML, texts, etc.) are uploaded into a container. Azure AI Search can be configured to read directly from the container via a *data source* that describes endpoint, access credentials and ingestion patterns. The text extraction phase, so-called *cracking*, is handled automatically and it is also possible to use Form Recognizing models if the data source has a standard template: in this thesis we do not discuss the importance of OCR in RAG, therefore results are obtained from text that is in fact already extracted, however it is a component as important as retrieval and, often, the main advantage of providers such as Microsoft is not given by the Language Models, but precisely by the OCR models they offer
- **Index**: it defines the schema of the search index. In it one specifies:
  - textual fields (chunk content, title, path, etc.)
  - metadata fields (date, author, category, ACL, document id, etc.)
  - the vector field (numeric array) for semantic search via embeddings
  - search settings (filters, facets, analyzer, and options for hybrid/semantic search)

In a proper RAG, the index is often optimized to quickly return chunks derived from the adopted strategy, with metadata useful to trace provenance

- **Skillset**: it is the component that describes the chain of *enrichment* applied to documents during ingestion. Essentially, it is a graph of "skills" that transform input into indexable output; by 'skill' one means a particular transformation over extracted text: although it looks complex, it is not conceptually different from tokenization, stopping or classic stemming done in search engines, independently of RAG or not. In a case like ours, the most relevant skills are:
  - *Parsing/extraction*: to obtain text and structure from the file;
  - *Split/Chunking*: to segment text into smaller units;
  - *Embedding*: to compute the vector associated with each chunk (integrated with an embedding endpoint).
  - *Entity Recognition*: to derive relevant real-world entities, as of the reference domain, often to support retrieval based on a Knowledge Graph

  In short, the skillset takes text and builds the object that can be indexed
- **Indexer**: it is the "engine" that connects data source, skillset and index. Its functioning can be summarized macroscopically as follows:
  - it reads documents from the data source (Blob)
  - it invokes skillsets for enrichment
  - it maps extracted information to the fields indicated in the index, namely those on which search will occur in practice
  - it writes results into the index
  - it can be scheduled or executed on-demand

The functioning is the same, broadly speaking, as what we already discussed, independently of whether you use Azure's SDK, or AWS, Google or you do 'manually' the vast majority of operations with LangChain; and the point is precisely here: implementation-wise, provider SDKs allow you to orchestrate the whole cycle (resource creation, updates, indexer run, query) with a very high level of abstraction that makes product construction and maintenance easier and faster; after all, despite people mostly talking about AI today, the advantages of Cloud Computing are exactly the same on putting the product you build into operation.

The advantage of SDKs becomes, in research projects like mine but also in domain - specific applications, rather negligible in the sense that: SDKs allow you to simply build RAG pipelines that have already been thought out by those who built the SDKs and it becomes more tedious than building everything

manually to go and define customized chunking strategies, such as algorithms or ranking models; in fact we recall the previous concept: the real advantage of cloud providers in RAG is given only and exclusively by service availability, not by the performance your product reaches or by implementation simplicity.

The Naive "managed" RAG pipeline on Azure implements all the previously discussed components: documents in the Blob container are indexed by an Indexer that performs cracking and applies a Skillset including Fixed Chunking and embedding computation with `text-embedding-ada-002`, one of Microsoft's models, then projecting the obtained chunk onto the fields that define the index. At runtime retrieval is hybrid (BM25 + dense ANN search on the vector field) and top chunks are passed to the model with a system prompt constraining usage only of information coming from contexts. As said, the same pipeline was emulated with LangChain, where chunking and transformations are fully customizable for overall more sophisticated pipelines.

**Considerations**

The described Azure pipeline therefore positions itself as an effective solution to quickly start a RAG system and manage infrastructural aspects (storage, indexing, orchestration). As said, precisely because it is built in a "guided" way, it has quite practical limitations, which we now describe better, when interventions on fundamental pipeline components are required. In particular:

- **Pre-processing rigidity**: many crucial choices (text normalization, table handling, structure preservation, deduplication) are constrained by the skills provided by the platform; when more specific needs emerge, implementation complexity increases significantly because it becomes necessary to introduce external components (e.g. functions or services on dedicated endpoints) and orchestrate them in the pipeline

- **Chunking**: it is relatively simple to modify chunk size or overlap within the limits of the provided SDKs; it is instead complex to implement advanced strategies (hierarchical, semantic, recursive chunking...): most providers encourage "standard" chunking, while more sophisticated strategies generally require defining external functions and building REST endpoints to call as needed; in Azure, for example, implementing chunking strategies other than Fixed generally requires building a *Custom Skills* within an Azure AI Search Skillset: in these cases, it is necessary to create an *Azure Function* that hosts the custom chunking logic (perhaps using libraries such as LangChain or Azure OpenAI embedding models) and register it as a Web Api Skill, recalling the previous concept. For Fixed the SDK abstracts all this; precisely in this sense we discussed earlier how, for real applications, using or not using SDKs has more or less the same level of complexity: obviously it does not mean that you cannot rely on a provider, simply that the provided tools focus predominantly on Naive RAG and do not cover the real complexity of needs

- **Reranking**: integrating a reranker also often introduces practical constraints; with provider SDKs one is typically limited to the exposed models, while adopting a custom reranker requires external orchestration (service/API), with impacts on latency, costs and complexity more generally

Providers such as Azure make it easy to implement an end-to-end RAG pipeline, but they do so by privileging a set of standardized choices. This is an advantage in terms of *time-to-first-result*, but it can become a constraint when the objective shifts to having a system optimized for the reference domain.

**Reranker** For *rerankers* we make a separate discussion, which serves to frame the underlying choice made in this work: not to consider them. A *reranker* is a component that, after a first retrieval phase (for example top-$k$ via BM25, vector or hybrid), recomputes result relevance order with a Transformer model, in reality conceptually similar to SentenceTransformers: a reranker evaluates *query–document* pairs with cross-encoder models producing a more accurate relevance score; in practice, the pair is merged into a single text and one exploits models with bidirectional attention such as encoder-only ones, the true peculiarity lies in producing a relevance score between the query and the document and the concept is the same as Contrastive Losses, that is teaching the model to separate what is relevant from what is not; modern approaches, as for SentenceTransformers, extend the idea also to LLM-based decoder-only models.

In my view, the reranker is often *overrated* in the general RAG context: in many cases it improves ranking metrics and this is an advantage, however the performance gain is not necessarily justified by introducing a second model downstream of retrieval rather than acting on the SentenceTransformer with, perhaps, a tuning more aligned to needs; if chunks are poorly built after all, a reranker can at most choose "the least bad" among already suboptimal candidates. The main inefficiency is, however, the same that in real contexts constrains you on the embedder: with a reranker you introduce additional latency that, if you can afford it, increasingly makes sense to spend on the SentenceTransformer since more relevant chunks at the base produce a structurally better ranking. In other words, the reranker can be useful as a refinement if constraints allow it, but it should rarely be the first tool to resort to in order to "fix" a RAG with weak retrieval; precisely in this sense, in my work I decided not to consider rerankers.

These considerations are supported by recent works that observe that reranking benefits do not grow monotonically when the "first stage" is already strong. For example, [Chen et al., 2025b] explicitly show diminishing returns: moving to more effective first-stage retrievers, the percentage improvement obtained from the reranker tends to decrease. Very similarly, [Jacob et al., 2024] challenge the "reranker always better" assumption, highlighting that, when scaling the number of documents to evaluate, rerankers can give initial improvements but then decline and even degrade quality beyond a certain threshold.

## 5.2 Hierarchical RAG

The *Hierarchical* approach extends the RAG paradigm by introducing a *multi-level* indexing and retrieval structure. When we talked about chunking we discussed how chunks are, essentially, local views of the document and, very often, especially in texts such as novels where there is a narrative and links also between text portions that are spatially far apart, one structurally loses this interaction property; indeed, one of the domains where Naive RAG approaches perform worse is precisely that of so-called MultiHop questions: queries that can be satisfied only by connecting information belonging to very distant portions of the same document or, even, of different documents. Useful information is therefore not uniformly distributed and is not always retrievable via local chunks: a hierarchical index allows one to represent the same content at different abstraction levels and has a very strong interaction with classic graph theory concepts, such as *Communities*.

Modern hierarchical approaches are in fact always based on the interaction among the same set of components:

- textual chunks as **base units**
- **clusters or communities** to semantically aggregate correlated text portions
- **summaries at different levels** these act conceptually as a text extension of *rollup* on structured data, with the objective therefore of representing the fundamental theme of the Cluster/Community and guiding the query towards a portion of embedding space that includes fewer relevant documents

Already from here one can grasp an important concept relative to fundamental RAG problems: the hierarchical approach has the big advantage of structurally mitigating the *semantic collapse* problem in search, precisely because summaries allow you to route the query referring to a much smaller set of chunks than at the low level; for example, if each Community comprises on average 10 chunks, then you have reduced search dimensionality by a factor 10: you evaluate the query on Community summaries and then you take as corpus only the set of chunks of the top communities for example, a simple and very widespread approach. By concatenating operations that coincide, as mentioned before, with *rollup* and *drilldown*, you ensure not only to reduce the size of the corpus on which you evaluate the query, but also a priori interactions between spatially distant chunks or cross - document.

Operationally, a Hierarchical RAG pipeline separates two problems:

1. **Hierarchy construction** (*index-time*): in addition to classic chunk indexing, "aggregated" nodes are built by choosing the rollup strategy (often summary based)

2. **Coarse-to-fine retrieval** (*query-time*): instead of selecting only local top-$k$ chunks, the query is first evaluated on the rolled - up dimension to maximize coverage and context, then one goes down in drilldown to maximize precision

As said, this approach is particularly performant in all those cases in which:

- the question requires **synthesis** or **multi-hop reasoning** single or cross document
- the collection is very large and one wants to mitigate *semantic collapse*

Obviously, much of the goodness of this approach boils down to how the rollup phase is designed: if one proceeds summary - based, as typically happens, noisy summaries limit the maximum precision achievable by subsequent drilldown.

### 5.2.1 RAPTOR RAG

RAPTOR (*Recursive Abstractive Processing for Tree-Organized Retrieval*) is an implementation of Hierarchical RAG in which the hierarchy takes the form of a **tree of summaries** built recursively; the peculiarity compared to other approaches, such as the one we will present later, lies precisely in using a Tree-based structure rather than a generic graph, for the rest the fundamental concepts discussed previously are all perfectly aligned with the idea of the approach. The objective is always the same: overcome the typical Naive RAG limit of retrieving only contiguous (flat) chunks, often not capturing the context needed for questions that require reasoning over spatially distant or multi-document parts.

**Tree construction (index-time)** The process can be described as a bottom-up pipeline:

1. **Initial segmentation**: the document is split into chunks (tree leaves) and an embedding is computed for each
2. **Semantic clustering**: chunks (or nodes at a given level) are grouped into clusters based on embedding similarity, with the objective of aggregating conceptually related contents; this approach belongs to the family of RAG strategies that directly use clustering algorithms and not community-based ones, but conceptually the idea remains the same and it is only a design choice
3. **Abstractive summary per cluster**: for each cluster a *summary* is generated via a Language Model; this summary becomes a "parent" node that represents group semantics, namely the rolled - up dimension. In this case, RAPTOR explicitly publishes the system prompts used to generate summaries: they exploit very simple templates, in my view it would require more prompt engineering for such a critical phase

4. **Recursion**: parent nodes are in turn embedded and (re-)clustered to build higher levels, until obtaining a small number of high-abstraction nodes

In this last case, it is important to underline that the rollup operation still has an associated degree of information loss: by combining them you gain in mitigation of all previous problems, however you could reach, beyond a certain threshold, performance comparable (if not worse) than Naive RAG. The biggest design difficulty of this approach lies precisely in the relationship between summaries and the target abstraction level.

**Retrieval and generation (query-time)** At query time, RAPTOR exploits the tree of representations; you can decide to do different things and even make retrieval itself even more sophisticated: not all queries are the same, some could be answered by simple full - text search, others gain a lot from semantic search, therefore the hierarchical structure is fine but it is not mandatory to use it. Typically, the most common approach wants to start from the highest-level dimension and go down in the tree at each step, always evaluating $top$ - $k$ nodes by similarity metrics, such as cosine, until obtaining a set of leaves (actual chunks) as a restricted document base on which to evaluate the query; nothing prevents you, however, from starting from intermediate abstraction levels and, therefore, basically relying on a lower degree of information loss in what is, fundamentally, a routing mechanism for the incoming query. In both scenarios, the hierarchy enables a context composition in which higher levels directly provide information about where and what you might find by descending to a lower level and links between concepts, while deeper levels provide the information need itself; obviously, these advantages are not free: system response time increases proportionally to tree complexity, however, if one reads between the lines, we did not discuss rerankers or models that further increase overall latency, therefore compared to commercial Naive RAGs that exploit tools such as rerankers the hierarchical approach has comparable response times.

A consideration that, in this case, makes absolute sense and that, later, we will also discuss with results is the following: sophisticated approaches like this surely have clear advantages, but they become evident when you can estimate the query type that, for the most part, you will have to answer at runtime. Search engine performance strongly depends on how the user formulates their need, therefore on query lexical distribution: for full - text searches, indeed, if this is strongly misaligned with the lexical distribution of documents the system will perform poorly and this is quite clear; the big advantage lies, indeed, in adopting generative models for query rewriting and it is one of the strongest mitigations to the Vocabulary Mismatch Problem. When, however, you have very simple queries that can be answered by retrieval systems much less complex than the hierarchical one, you typically see that: the more sophisticated the system you design, the worse it performs on simple queries, which are nevertheless always a very dense subpopulation of the total incoming queries. Indeed, what makes Naive RAG a hard baseline to beat, if you always look at the go - to

solution concept, lies precisely in the fact that, with the appropriate considerations, it can still satisfy the average information need well and the hierarchical structure gets lost in simple cases: for this reason, often adopting classifiers on the initial query type, indicating its difficulty, allows one to evaluate it directly on the document base at the deepest level of the hierarchy, namely the whole one, without reducing it dimensionally, precisely because the full - text part of the hybrid score is predominant.

### 5.2.2 Microsoft GraphRAG

Microsoft GraphRAG proposes another form of Hierarchical RAG, where the hierarchy is not a tree of summaries built only from clustering, but a **graph-based** structure centered on entities and relations extracted from text, therefore something structurally closer to the approach I will propose. The underlying hypothesis is always the same, however here one refers to one of the key concepts in modern retrieval and that, historically, made the fortune of search engines such as Google: Knowledge Graphs.

**Knowledge Graphs** A *Knowledge Graph* (KG) is a structured representation of knowledge in which information is modeled as a set of **entities** (nodes) and **relations** (edges) between them. Formally, a KG can be seen as a labeled graph that follows a certain way of building those labels: often via triples of the form *(subject, predicate, object)*, but more generally one speaks of **Ontologies** which could also have a different form than the previous one. This enables a transition from a purely textual corpus to an explicit description of "who is who" and, above all, "how things are connected", making content not only queryable at a semantic-lexical similarity level, but above all through logical connections. *Properties* represent the attributes and relations that *characterize* an entity, understood as a real-world object (people, places, organizations, concepts); in a KG, they do not only describe the entity, they also express links between them, often making it much easier to satisfy complex information needs.

Applicatively, it is now easy to understand how Knowledge Graphs answer a fundamental retrieval need: introducing a more stable representation level of text that allows one to 'navigate' information in a structured way, expand the query by adding contexts, dependencies, cause-effects, hierarchies and associations... even when such links do not emerge evidently with chunk-based retrieval; indeed, you can answer the query by navigating directly the graph if, as in the case we will present later, this is a chunk graph, however, as in *query rewriting*, you can also exploit it to resolve the ambiguity that typically exists in real search-engine queries, such as those in the NQ dataset discussed in previous chapters. As already mentioned, not all advantages come 'for free': KGs have management costs that are often prohibitive, especially when your domain is particularly complex and requires modeling many distinct entities and concepts; the two fundamental processes are *Entity Extraction* and *Entity Linking*: starting from a textual base, you need a way to automate both ac-

tual entity/property extraction and linking to objects already existing in the current KG; one therefore speaks of *disambiguation*, as well as normalization to the ontology of extracted information. Entity Extraction and Linking are two extremely complex processes, perhaps among the most complex in retrieval: there are NER models, also Transformer- or LLM-based, that can yield good performance on your domain, but often they are not competitive in zero - shot; to really understand not so much entities, but rather what defines a property relative to them in your domain, you need fine-tuned models trained on many labeled examples of *text* and *extracted pairs*, unless you want to obtain always suboptimal performance. After all, a search that relies entirely on KG navigation has performance that critically depends on Entity Extraction and Linking: if you do not have a good F1 - Score and you are, for example, skewed towards Recall rather than Precision, then you either make a given information in the text not searchable or you make searchable noisy, incomplete or even misleading information. The approach built later, in fact, is based on a chunk graph and not an entity graph: if you are looking for go - to solutions in RAG and you are not relying on a domain analysis, then doing NER or, more generally, populating a KG loses all meaning for the aforementioned reasons.

**Indexing and Communities** Microsoft GraphRAG's indexing pipeline can be schematized in three operational phases:

1. **Knowledge Graph construction**: the corpus is first segmented into *TextUnits* (chunks with possible overlap) that act as the fundamental extraction unit. On each TextUnit, an LLM performs *Entity and Relationship Extraction* producing a local subgraph with: entities described by *title*, *type* and one or more descriptions generated by the LM itself based on textual ground, directional relations between entity pairs, each with one or more textual descriptions. Subgraphs are then merged into a global graph with deterministic deduplication: entities with the same *title+type* are unified (accumulating descriptions), and similarly relations with the same (*source*, *target*) pair are aggregated. To reduce noise and redundancy, GraphRAG applies an LLM *summarization* phase that compresses the set of accumulated descriptions into a canonical description for each entity and relation, very similar to the previous rollup concepts

2. **Community detection and multi-level hierarchy**: once the Entities/Relationships graph is obtained, GraphRAG performs a community partition via *Hierarchical Leiden*, one of the most famous Community Detection algorithms, obtaining clusters of densely connected nodes and a recursive hierarchy that respects a certain size limit; this is the Hierarchical part of Microsoft's GraphRAG approach, after all, in the work we will present later, we will refer precisely to the Leiden algorithm

3. **Community reports**: for each community and each hierarchical level, GraphRAG generates *community reports* and a further compressed version (*shorthand*) that synthesize key concepts of central entities, salient

relations or, more generally, community semantics. In parallel, embeddings are computed for *TextUnits*, entity/relation descriptions and reports, enabling semantic retrieval both on unstructured text and on what the Language Model built

**Querying modes (query-time)** GraphRAG typically distinguishes two complementary strategies:

- **Local Search**: suitable for entity-centric or focused questions. One starts from entities relevant to the query and retrieves an informative neighborhood in the graph (related nodes, relations, descriptions) combining it with correlated textual chunks to enrich context; this is the part most similar to the previously explained drilled - down search
- **Global Search**: suitable for corpus-level synthesis questions. The answer is built in a way similar to a very famous RAG approach: HyDE (Hypothetical Document Embeddings), where, instead of doing retrieval using the query directly, the LLM first generates a "hypothetical document" that would answer the question and one uses that text embedding to search for the most similar real documents; in Microsoft GraphRAG the concept is very similar: the LLM generates a partial answer (*key-points*) using top - Communities summaries, also obtained by traversing different hierarchy levels, then produces a final answer by inferring on the initial query via the extracted *key-points*; note that, in the case of Global Search, one does not evaluate drilldown because one is interested in high-level information as a presupposition

Even if GraphRAG mainly presents Global and Local Search, the availability of a KG and a community hierarchy naturally enables the coarse-to-fine strategies discussed previously: selection of relevant communities, identification of candidate entities/relations and subsequent drill-down towards TextUnits as evidence, effectively obtaining a multi-step graph-guided search.

Microsoft GraphRAG is interesting because it pushes an LLM-centric index construction: the Language Model extracts entities and relations from chunks and synthesizes their descriptions, obtaining a "navigable" graph useful for retrieval; however this design also brings several practical problems. Beyond what we discussed about zero-shot, in the standard pipeline a true *Entity Resolution* cross-document phase does not emerge: merging is mainly deterministic (entities with same title+type are aggregated), therefore nominal variants, synonyms or ambiguities across documents may not map to the same "real" entity, fragmenting the graph and incentivizing duplications in Communities, which is not necessarily what you want; this is a problem not yet solved by the authors and that I was able to highlight by studying some of the open issues on the GitHub repository of the code.

All fundamental information was taken directly from studying the public code provided by the authors and the reference paper [Edge et al., 2024].

### 5.2.3 Semantic Chunk Graph

The proposed **Semantic Chunk Graph** approach is inspired by the two design alternatives discussed previously: RAPTOR and Microsoft GraphRAG. From Microsoft GraphRAG it takes the idea of leveraging a graph-based structure and providing coarse-to-fine search with rollup via communities (obtained with Leiden-type clustering), so as to satisfy the fundamental idea underlying hierarchical approaches. However, more similarly to RAPTOR, it does not introduce an explicit level of extracted entities and relations: graph nodes are directly textual chunks and edges represent semantic relations between chunks (e.g. cosine similarity), as well as spatial adjacency relations in the document; the motivation for this derives from the previous discussion on Language Models' zero - shot in graph construction. Hierarchical structures and refinement steps therefore operate on segmented text and its aggregations, keeping grounding on original contents. Regarding chunking strategy: I adopted, consistently with what was discussed in previous chapters, the strategy I find conceptually closest to the go - to solution idea, namely Recursive Chunking.

The concrete implementation of the **Semantic Chunk Graph** articulates into three main phases: graph construction and annotation at the intra-document chunk level, graph construction between communities to enable cross-document interactions and handling of the incoming query with hierarchical coarse-to-fine retrieval. The fundamental structure, as said, is that of a graph in which each node represents a textual chunk and edges encode two distinct signals: *semantic* relations (`SIMILAR_TO`), based on embedding similarity, and *structural* adjacency relations in the document (`NEXT`); the graph therefore remains anchored to original contents (nodes = segmented text) and allows one to apply structural analysis algorithms (PageRank and Community Detection) without directly passing through entity/relation extraction.

As discussed in the previous chapter on Chunking strategies, a fundamental problem lies in the fact that the very action of 'splitting' text, absolutely necessary for dense retrieval, does not allow many portions to "interact" directly, although they may be correlated: the relations modeled by `SIMILAR_TO` edges are functional precisely to favor interaction between semantically similar chunks and, indeed, they are evaluated among all chunks *of the same* document; to this we add `NEXT` edges, which act as a surcharge to model not only an affinity at the level of text meaning but structural relations, with the idea that `NEXT` should not "override" semantics, but guarantee a minimum connectivity and preserve document progression. `SIMILAR_TO` edges are weighted by cosine similarity between embeddings and, relative to this value, I used a threshold to avoid adding edges with too low weight and, therefore, semantically weak links, while `NEXT` edges receive a constant lower weight precisely because their role is more of an

auxiliary signal and the true protagonist must be the semantic signal. We can summarize the core intuition as:

- if two chunks are semantically close, the `SIMILAR_TO` edge dominates
- if a chunk is semantically distant, `NEXT` prevents its isolation and allows Community Detection algorithms to identify more cohesive communities and, overall, less *scattered* ones with, perhaps, trivial groups formed by few poorly connected chunks

On this structure, **PageRank** is first computed, with the objective of estimating the *relative* importance of chunks in the context of the single document: a chunk with high PageRank tends, indeed, to be more "central" in the graph of semantic interactions and adjacency, therefore it is used primarily as a representativeness measure to select, in practice, which chunks within each group to use in subsequent synthesis and indexing phases. Next, **community detection** is run via Leiden, an algorithm, like PageRank, discussed in the introduction to this work. Each community, in fact, can be interpreted as a document **subtopic**: the formal definition of a community can be reduced to its core as a set of elements with high internal edge density and, by modeling those as adjacency/semantic relations, they become assumable as local document themes, constituting the rollup unit on which to operate hierarchical coarse-to-fine search. The choice to perform clustering **per-document** is intentional and derives precisely from the previous nature of communities as sub - topics: allowing cross - document interaction of local portions could, for the most part, lead to spurious semantic relations, also because we recall that we use non fine-tuned embedding models and the chunk is a local view of the document; deriving cohesive groups internally to the same mitigates the previous spurious relations, in the sense that, although there may be a very large heterogeneity of topics, the fact that chunks belong to the same document still makes them belong to a single logical unit and semantic links are more reliable.

Once intra-document communities are obtained, the system leverages the previous **PageRank** metric to produce a textual *rollup* level: for each community it selects the most "representative" chunks, ordering them precisely by `pagerank`, and provides them to the model to generate a short summary of the sub-topic represented by the community itself. The result is:

- saved in Neo4j as a `Community` node and linked to the chunks composing the community itself, providing a hierarchical aggregation level
- indexed in Elasticsearch in a dedicated index together with the summary embedding vector, so as to guarantee hybrid search on them

These choices, precisely, enable the "coarse" component of searching on communities before returning to the chunk level. Beyond the low intra-document interaction among chunks, there is the complementary problem regarding the lack of *cross-document* interaction: with the structure built so far, distinct documents

still remain separated; this problem is addressed by building a subsequent **graph between communities** level based on similarity among summary vectors: for each community, its embedding is retrieved from the Elasticsearch index and a kNN search is executed to obtain nearby candidates (still communities); among these, only those that:

- belong to different documents
- exceed a cosine similarity threshold, thus an exact projection of the `SIMILAR_TO` mechanism among chunks

are kept. For each kept pair, a directed `COMM_SIMILAR_TO` edge is written between `Community` nodes, again weighted by cosine similarity. Using a threshold and a neighbor limit mainly serves computational efficiency: it is fine that communities are an aggregation level and therefore with lower dimensionality than the chunk level, however evaluating interactions among all community pairs still explodes, in fact, to $O(N^2)$ anyway and where $N$ is the number of chunks, since communities depend on a constant reduction factor. The objective, precisely, is not to connect "everything to everything", but to build relevant relations among thematically overlapping communities in different documents.

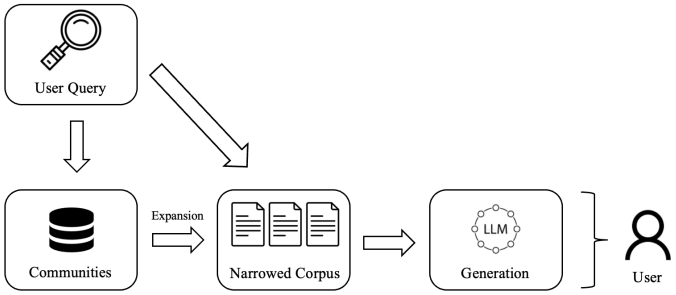


Figura 2: GraphRAG Architecture Schematic

In light of all this, we can discuss better how the incoming query is handled and fully understand the approach:

1. **Seed Communities Selection (Coarse Retrieval)**
   The query is compared against indexed community summaries via Hybrid Search and the result is a set of *seeds* that represent the macro-themes most plausibly relevant.

2. **Expansion in the community graph**
   Starting from seeds, the system explores `COMM_SIMILAR_TO` edges up to a limited number of `hop`s (a cap is set to 3, modifiable). Here too, I modeled a minimum similarity threshold on *each* edge traversed during expansion: a path is considered valid only if all `COMM_SIMILAR_TO` relations composing it have a score at least equal to the `min_sim` threshold. The idea is to avoid

reaching far communities by passing through weak links; after all, among all paths one chooses to introduce only new communities belonging to the $top - k$ paths by product of the similarities of the selected edges, deriving a metric that quantifies their goodness

3. **Fine-Grained Retrieval** The union between seed and expanded communities produces, by retrieving the chunks composing them, the set on which Fine-Grained search is evaluated; final search happens **only** on this subset via a hybrid retrieval on Elasticsearch

4. **Query Answering** Top-$k$ chunks, always by hybrid score of course, are retrieved with metadata (doc, position, text) and sent as context to the LLM; this receives the same prompt as the other approaches on answer generation, imposing: use only the context, explicitly admit insufficient information, and a short answer (maximum 3 sentences). This preserves grounding and reduces the tendency to introduce external knowledge, as already discussed

**Considerations**

The described graph-based architecture introduces an explicit hierarchical level that, in general, proves effective in reducing search space and mitigating fundamental issues such as Semantic Collapse, Hubness... however, it also presents some important structural criticalities, largely traceable to rollup level quality and, therefore, to community summary goodness.

The `Community` level acts, in fact, upstream of fine-grained retrieval as a coarse filtering component. If the summary does not correctly represent the sub-topic, however, or emphasizes marginal details with respect to the query intent, the procedure risks not selecting the correct seed communities and, consequently, excluding from the pool the truly relevant chunks. These criticalities show particularly well on *simple* or strongly *entity-centric* questions: although there is a very clear lexical match (proper names, acronyms, identifiers) and the full-text component of hybrid search at the chunk level would, alone, be sufficient, the hierarchical approach could risk not returning the correct context simply because the coarse component, dependent on summaries, could narrow the field by eliminating useful signal and not noise. Indeed, moving from chunk level to summary level, information is compressed and full-text evidence is largely diluted: a specific entity might not appear in the summary because it is deemed not central by the summarization, reducing the effectiveness of textual matching; there are cases, therefore, in which the pipeline is forced to "guess" the correct sub-topic first and only afterwards can it retrieve the chunk that contains the entity, making the approach more fragile compared to a classic retrieval-first on chunks. To mitigate this problem, the simplest way is what I introduced of expansion with *hops*: these must not be too many, in the sense that more hops correspond to more chunks on which you evaluate retrieval and, in the limit, tending to a null hierarchical reduction factor; expansion increases the chance

that chunks containing useful signal do *not* get lost in the coarse phase, but it is a purely 'statistical' mitigation. The real mechanism that would make the proposed approach more robust to the previous situation is always the same: *Entity Extraction*, adding, perhaps, a field where you can introduce an entity match component in retrieval, which, as already discussed, is not properly correct to do in our case with LLMs in zero-shot.

Nothing prevents, in theory, iterating the same scheme again and building additional hierarchical levels: for example, grouping communities into "supercommunities" and producing higher-level summaries, obtaining an even more aggressive rollup and a coarse-to-fine in multiple steps. However, such an extension would amplify even more the semantic compression problem: summaries of summaries tend to lose important information, especially that useful to the full-text component and to queries driven by more specific details. Abstraction would improve efficiency and mitigate even more "global" problems like the cited ones, but at the cost of further diluting useful signal and making the coarse component, probably, excessively abstract. For this reason, two-level integration (chunk $\rightarrow$ community, with cross-community links) already results in a natural compromise and, in my view, quite reasonable; any additional levels would require more robust representation strategies: better detail levels, structured summaries... since you cannot think of doing a summary of summaries, it is simply wrong for all the aforementioned reasons, and avoiding that aggregation becomes a semantic bottleneck is mandatory.

On the maintenance side, adding new documents does not introduce substantial criticalities for the intra-document part: this is a strength of the built approach, also oriented to the discussions we will later make on UI and Text Extraction. Indeed: chunking, edge construction, PageRank computation and clustering can be executed independently for each document and thus parallelized; same for generating community summaries: conceptually it is incremental and each new document produces new indexable communities separately. The most delicate point is the subsequent aggregation level among communities, namely building `COMM_SIMILAR_TO` edges via KNN on the embedding space of summaries: in an incremental context, where users can upload documents at different times, inserting new communities implies that they must be connected to the rest of the global graph, but also that some pre-existing communities might have to update their neighborhood, since new points in vector space can enter their top-$k$. Consequently, even by using KNN (which avoids $O(N^2)$), the problem remains of efficiently updating the graph, requiring at least: computing neighbors for new communities, possibly re-evaluating neighbors for a portion of existing communities or, in the limit, a periodic *ex-novo* recomputation of connections on the last level; this is the most onerous component of the architecture.

## 5.3 CoT RAG

The implemented *CoT RAG* approach extends Naive RAG by introducing an LLM **agent** that iteratively guides retrieval through, as the name suggests, a form of *Chain-of-Thought* materialized in a *scratchpad*. The idea is not (only) to make search step wise, but **adaptive**: instead of executing a single top-$k$ retrieval on the original query, the system alternates retrieval steps, updates of the 'search memory' given by the scratchpad and a decision to stop or to generate a new query aimed at filling missing information; adaptivity and the CoT concept lie precisely in this generation of sub-queries: the model focuses on searching only what it needs to satisfy the information need and adaptively with respect to already found information; this allows one to iteratively build much more precise queries and increase answer precision, with good flexibility with respect to initial query complexity in reality.

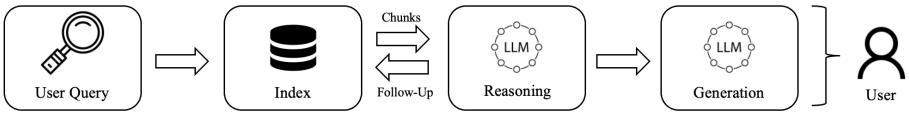


Figura 3: CoT RAG Architecture Schematic

In the literature, this family of methods has a strong interaction with approaches based on *interleaving* between reasoning and retrieval (e.g. IRCoT) or that formalize decomposition into sub-questions and the use of external tools (e.g. Self-Ask, ReAct), where the intermediate "state" acts as a guide for subsequent search; many of the design choices underlying the approach are inspired by rather recent works [Trivedi et al., 2023, Press et al., 2022, Yao et al., 2022], in particular: [Nye et al., 2021]. In the code, everything is orchestrated with LangChain (templates, parsing, answer construction) and Groq's endpoint to interact with the chosen model. An interesting aspect is that CoT RAG is agnostic to how you organize the corpus: you can, or not, use hierarchical structures and evaluate adaptive search on them, this depends on design choices; the produced pipeline is in fact *backend agnostic* and allows you to directly indicate whether you are working on a graph or not. We will return to this later in the considerations of this section.

### 5.3.1 Scratchpad

The *scratchpad*, as mentioned previously, is an **accumulative** textual memory that represents the current state of what the model "knows" in a *grounded* way; in short, the retrieval history up to that point. At each iteration, the LLM agent receives: the original question, the current scratchpad, a short history of recent steps (issued queries and retrieved doc-ids) and new information obtained from retrieval, with which to update the scratchpad and build queries for the next step. Since there are quite a few steps to manage, the most coherent way is

for the LM to produce a structured output (JSON) that facilitates extracting individual components:

- a **scratchpad update** (bullet-point facts derived *only* from evidence)
- a boolean **stop** decision (`enough`) if evidence is sufficient to satisfy the information need
- a description of **what is missing** (`missing`) if not
- a single, targeted **next query** for the next step

What I personally really like about this approach is the strong interaction with the true reason why Language Models are used extensively today: dynamism in Question Answering, natural handling of follow-ups and, more generally, the strong interaction component between user and model. In evaluating the approach, I often saved the scratchpad evolution for audit purposes and, setting aside small inaccuracies that you can fix with some prompt engineering, this adaptivity emerged clearly: search was "guided" towards increasingly targeted queries, as the system understood the current state of information and what was missing, refining subsequent queries.

### 5.3.2 Step-Back Prompting

*Step-Back Prompting* is a technique in which, before starting retrieval, the model is asked to rewrite the question at a higher abstraction level, producing a more generic query that captures *concept* and *intent*. The idea is always that classic rollup and drilldown of hierarchical approaches, in a more 'light' and simple-to-implement form.

The CoT RAG approach I presented above, in reality, derives from an initial variant that conceptually leveraged Step-Back Prompting precisely with adopting a *Planner*: one generates a `stepback_question` and an `intent`, then a planner produces a 2–4 step plan with sub-questions and initial query, updates the scratchpad and decides stop or follow-up. Compared to the previously described approach, here the plan is defined 'a priori': steps are built before retrieval and, therefore, before being able to actually access chunks, while the adopted approach directly relies on 'what is missing' with respect to actually collected information; the initial Planner more strongly constrains search direction. The underlying intuition comes precisely from Step-Back Prompting: the idea was to guide the model in building queries at different specificity levels, starting from an initial step-back and going down into more detail with subsequent queries. The adopted setup is, however, closer to the *adaptive* retrieval concept and, indeed, it was preferred; however, when we will later discuss **query drift**, the Planner version is much more robust to such a phenomenon, as well as generally more cost-efficient, losing, however, precisely on search adaptivity.

In the Plannerless approach Step-Back Prompting was not introduced explicitly,

since the benefit of "changing scale" in query formulation is, in principle, guaranteed structurally by adaptivity itself: CoT can guide both progressively more specific queries when information is missing, and generalizations when context is needed or when overly specific attempts fail. At inference, however, the most common and natural behavior is a sequence of increasingly specific queries; nonetheless, nothing prevents you from adding fallbacks where, for example, after $n$ steps without improvement one generates a more abstract (step-back) query to retrieve background or reorient search. CoT RAG is a very interesting approach precisely because the designer has a lot of freedom of choice and, after all, design-wise you have many alternatives with high potential: a structural advantage, also, of all LLM-centric products.

**Considerations**

In Naive RAG the query is evaluated *single-shot* and top-$k$ retrieval directly conditions generation. In CoT RAG, instead, the query becomes a **process**: the LLM uses the scratchpad to understand whether evidence is sufficient and, if not, produces subsequent more specific queries. The expected effect is an overall improvement on medium-high complexity queries thanks to CoT, as well as greater robustness to lexical variability: after all, writing more specific queries is structurally query rewriting. In this sense, the idea is not different from what was presented in previous chapters, but execution is: you cannot necessarily expect to use a fine-tuned model, unless you use multiple language models; it would have been interesting to split the agent into smaller sub-agents, with an SLM, for example, specialized in rewriting coherent with domain lexical distribution, however, not focusing on a specific domain, it is a path I did not take.

These adaptive capabilities typically require medium-large models: one needs strong text understanding, generalist knowledge, good reasoning abilities, synthesis, rewriting... and here comes the painful note: there are many advantages and, design-wise, the setup is not that much more complicated than Naive RAG, however inference cost grows not only because you use on average more expensive models, but above all because a single question can trigger many sub-steps in CoT; each step consumes tokens both for reasoning and for handling intermediate phases (scratchpad update, subsequent queries...) and, note, they are still generated tokens, therefore those of 'maximum cost'. Earlier we mentioned the *backend agnostic* concept, however the results we will present do not rely on hierarchical structures for corpus organization: in this way it is true that you do not mitigate fundamental retrieval problems, but on the other hand the cost you would have in managing such structures would add to an already very high inference component for the previously discussed reasons and, put very simply, you would have an extremely unbalanced performance-cost tradeoff, or rather, all advantages you can derive from combining Hierarchical + CoT would not be justified by what you spend to put them into practice; in real cases therefore, often with strong cost constraints, one would always choose Naive RAG alternatives as a better compromise.

**Query Drift** Beyond cost considerations, adaptivity hides an important **query drift** risk: if the controller misinterprets what is missing, it can generate queries that deviate from the original information need, accumulating "plausible" but irrelevant evidence; this is an informal way to emphasize even more what we said before: you need, on average, large models and, above all, *instruction-tuned*. One defines *instruction-tuned* Language Models as those that include a post-training phase where they are trained to respond coherently to user requests and expressed constraints, formulated via prompting, precisely reducing the same deviations that underlie query drift; if you teach/indicate to the model how to reason and how to formulate queries via prompting, basically you want it to commit as little drift as possible with respect to what you ask and, in CoT RAG, this is much more true than in all other architectures given the many intermediate steps in the retrieval process. This, in reality, is an important concept independently of RAG architecture: to ensure that the model uses only the context derived from retrieval, for example, one typically asks it explicitly as a system prompt (*Use only the provided context. If the context is insufficient, say so explicitly*) and if the model is not instruction-tuned, basically, RAG itself does not give you the same guarantees. In reality, what we are discussing is a known failure mode of reasoning+tool-use methods: convincing but wrong traces such as the agent's decisions, much discussed in practice in lines like ReAct [Yao et al., 2022]. There are no exact mitigation methods for this phenomenon, indeed the provided implementation includes mechanisms based on purely heuristic reasoning: the intuition I relied on is that drift, if it happens, is much more likely to arise after a certain number of adaptive retrieval steps, i.e., I assume that the information need of the initial query can be on average clear to the model and the error in query generation happens precisely when you have to estimate particularly specific information; this led me to the concept of history: if the previous intuitions hold, then the first queries designed by the model can be assumed to be coherent, consequently, by giving it a window of $2-3$ previous queries, the model always has a contextual anchor that acts as a 'reminder' of the correct way to model missing information, as a sort of few-shot-learning.

## 5.4 RAG Datasets

In the context of RAG pipelines, a practical limitation of evaluations is the absence of strong standardization and of "definitive" benchmarks: many resources available online are, in fact, *retrieval* or *ranking* benchmarks rather than end-to-end evaluations of a RAG pipeline; in this sense, indeed, the vast majority of datasets distribute text already pre-chunked, which does simplify evaluations but makes the metrics you derive much less realistic, since they already take for granted one of the most critical components of the pipeline, as well as preventing you from measuring the effects of the choices you make on strategy and hyperparameters. In general, then, the literature proposes numerous metrics for retrieval and ranking: Precision, Recall, F1, NDCG, MRR, etc.. in my view, these metrics are overall overrated in retrieval contexts and, in a RAG setting like mine, even more so: since there is no domain-specific scenario and

I adopt as plug-and-play solutions as possible (see SentenceTransformer), the suboptimality of the ranking I obtain is a known presupposition a priori and I do not need to measure it, moreover metrics like Precision and Recall tend to overestimate "mechanical" aspects of retrieval with respect to the real objective of the system. In a RAG, in fact, what primarily matters is satisfying the user's *information need* in a correct way and supported by context; for this reason, evaluation is formulated as judging against a gold answer: by verifying both informational alignment and correctness with respect to retrieved contexts, we also indirectly obtain a signal on retrieval quality; after all, if the answer is correct and constrained to grounded information, the system necessarily retrieved useful chunks among the top positions and thus obtained a good ranking.

At an implementation level, the judge receives: the question, the gold answer (typically concise), the model answer (typically more verbose) and the retrieved context; then, it assigns three judgments:

- `answer_correctness`, understood as overall correctness of the answer with respect to the question and to the information in the gold answer; it can be formulated differently or in more detail, but it must not introduce errors or contradictions
- `evidence_coverage`, understood as coverage of the *necessary* facts of the gold answer and interpretable as a sort of checklist of mandatory entities or details (names, dates, places, relations...). This comes from the fact that, generally, an answer can be overall correct, or rather, 'right' with respect to the asked question but not fully satisfy all information one expected to find; indeed, this provides us with a more retrieval-oriented component, linking back to the reasoning above
- `faithfulness_to_context`, understood as alignment between statements in context and in the answer, penalizing also the case in which the answer is correct but not supported by retrieval

A critical point I focused on in the prompt is mitigating bias towards short answers: since the gold answer is often a factual summary, the judge is instructed to *not penalize* longer answers if and only if additional information is relevant, supported by context and coherent with the reference; although one uses a (large, less bias prone) LLM as judge, we are talking about GPT-4o, it is still a guardrail that, in my view, remains necessary. The choice of GPT-4o and not of, perhaps, more state-of-the-art models such as Gemini 3, GPT 5.2, Claude Opus 4.6... stems from the fact that LLM judging is as much a 'new' practice as Language Models are and GPT-4o is one of the most used and tested judges in the literature, therefore overall it is a reliable alternative at comparable cost.

Although the prompt produces numerical values in $[0, 1]$, the scale is intentionally categorical (0.0, 0.5, 1.0): it is conceptually equivalent to discrete labels ("terrible", "good", "excellent"), but it spared me an additional mapping between

categories and numbers without changing evaluation logic; it is important to underline this, otherwise it might seem there is a discrepancy with the reasoning made in the introduction, where I implied that it is always better to have the judge produce categorical labels rather than numerical counterparts.

### 5.4.1 GraphRAG Bench

The `GraphRAG-Bench/GraphRAG-Bench` dataset, available on Hugging Face, is structured into two main configurations: *novel* and *medical*, each accompanied by a dedicated question file. The benchmark is explicitly aimed at comparing "traditional" RAG approaches and *graph-based* variants across the entire pipeline, from structure (graph) construction to retrieval, to generation, including heterogeneous tasks of increasing difficulty, such as: simple fact retrieval, multi-hop reasoning, summarization and creative generation.

I focused on the *Novel* portion of the dataset and the initial results we will present refer to this; query formulation is often markedly *entity-oriented*: many questions directly mention people, places or objects; this characteristic has an important practical consequence: lexical retrieval models (BM25) or hybrid ones (lexical + dense) are by default extremely effective, because they can latch onto highly discriminative terms already present in the question, reducing the need for true multi-hop reasoning. Essentially, lexical signal is often already sufficient to identify a set of relevant chunks in the general case and the dense component tends to act more as an aid than as a decisive factor. In my view, indeed, the dataset is not that good a benchmark: the actual complexity of some instances labeled as *Complex Reasoning*, namely the queries with maximum hardness in the dataset, is often answered with, for example, a CoT that stops at the very first step, therefore basically a classic Naive Hybrid Search: explicit presence of entities in the query is directly linked to this tendency, since entity match $\rightarrow$ relevant chunk $\rightarrow$ answer, without a clear advantage of adaptive or hierarchical strategies emerging. However, the entity-centric orientation is coherent with the benchmark's purpose: in a setting where the seed is easily identifiable, it becomes easier to measure how much a *GraphRAG* approach can exploit relations and structure to answer the information need; this, however, is true if the graph approach is entity-oriented, as Microsoft GraphRAG is: not our setting therefore.

The question therefore arises naturally: *why was it chosen as a benchmark if it seems inadequate?* Mainly for two reasons:

- it is a dataset where you can easily reconstruct the source text unchunked, therefore pipeline evaluation is influenced by the adopted strategy
- many of the fundamental considerations underlying the work are precisely confirmed by the quantitative results we derive here, particularly on Naive and CoT RAG

What I like about the benchmark, in particular, is precisely the possibility of directly accessing documents to chunk: you can intervene end-to-end, avoid having to rely on already chunked text or, even, having to build complex scraping pipelines to address the previous fundamental problem; without having the full text available it is not possible to observe, as we did in the following results, the importance of the chunking strategy itself. As mentioned before, this is a quality that not many benchmarks have, which makes it more suitable, compared to most, to derive meaningful statistics supporting the intuitions on which my work is based.

For *CoT RAG*, moreover, the dataset still proved useful above all for the prompt engineering phase and defining constraints on sub-question generation; by directly observing the *evidence* $\rightarrow$ *scratchpad* $\rightarrow$ *next_ query* cycle, indeed, it is possible to formalize rules that avoid "unnatural" follow-ups, for example queries too similar to previous ones, perhaps only reformulations, or avoid failing to use information derived from previous retrieval steps; in this sense, performance is strongly dependent on the *prompt* and, more generally, on how one instructs the model: since I do not intervene with ad hoc fine-tuning, the main contribution inevitably passes through prompt engineering. Even with a set of questions that are relatively "easy" compared to average retrieval complexity, qualitative observation of errors is sufficient to design effective *patches*, also leveraging observation of scratchpad evolution. A concrete example is the concept of *Entity Bridging*: in the CoT cycle I observed that the model did focus on missing information, but often generated subsequent queries that were very similar to each other and, above all, increased semantic coverage by little: in essence, rewriting often focused on paraphrases, or rather, once it identified what was missing the model was less inclined to use, if present, important information in the retrieved context such as entities. To mitigate this failure, I introduced instructions that explicitly constrain the model to use entities (proper names, places, objects) when these appear in the retrieved context: thus, the next query is not only more specific, but also more *grounded* with respect to what the system actually observed and, if you can, structuring search on entity interactions is very effective, just see the reasoning made earlier on GraphRAG or, more generally, on KGs; the model thus tries to fill missing information by directly leveraging as discriminative concepts as possible, which already makes full-text search highly precise and this dramatically improves the quality of sub-questions. Obviously, it is difficult to present quantitative results for this last claim: there are no metrics that define how good one query is compared to another, therefore considerations derive predominantly from a manual audit.

### 5.4.2 Results

For the results we are going to present, two models available on the Groq API were used: `moonshotai/kimi-k2-instruct-0905` and `llama-3.1-8b-instant`. The choice between these two simply derives from a compromise I wanted to reach in results: there are RAG approaches that are more dependent on the

quality of the used LM, such as CoT ones, and others that, instead, are reasonably agnostic to it. Kimi is an MoE (*Mixture of Experts*) model among the most famous and performant on major benchmarks, at the time of writing of this thesis it oscillates among the top-10 models on the market, and it is the 'Large' component of results; on the other hand, the 'Small' component is given by one of the Llama models, chosen mainly because, at comparable size, it is notoriously performant and widely used in research projects.

The comparison between Kimi and Llama was conducted on 324 questions from GraphRAG Bench, distributed across the types proposed by the dataset; overall there were about $2k$ queries and 324, from a preliminary quantitative analysis, guarantees representativeness: it does not make sense to test approaches on the totality of queries, it is sufficient that chosen samples are representative to derive statistics with good confidence.

| **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: Kimi - Llama comparison in CoT RAG.

Overall, judgments assigned to Kimi are systematically higher than those of Llama on all three judge (GPT-4o) metrics, with the most marked gap on `faithfulness_to_context`. On average, indeed, the difference $\Delta$ = Llama − Kimi is always negative; a coherent interpretation is that, when retrieval does not provide sufficient (or clearly decisive) context, Llama tends to incur more often in two alternative behaviors: *over-generation*, introducing plausible but evidence-unsupported details, which lowers `faithfulness_to_context`, or *under-answering*, more conservative and producing partial or vague answers, penalizing `answer_correctness` and `evidence_coverage`. Obviously, it is rather simple to understand how, in the ideal case, we would like to be in a reasonably intermediate state between these two behaviors: Llama's deficit is not only "how much" it answers, but *how* it handles uncertainty by adding non-grounded content or losing text-supported information. This observation is, in fact, both consistent with the more "strict" behavior of a judge with respect to alignment between statements and context, and with the qualitative difference between models: at equal correctness with respect to the gold answer, evaluation can drop significantly if the answer contains details not supported by retrieval, or if the judge interprets such details as unverifiable in context, this does not however remove Llama's tendency to introduce unsupported details and, despite instructions, to infer information not directly cited by text. This result is particularly relevant and ties directly to the concepts discussed in the CoT RAG chapter: in an adaptive setup, retrieval is no longer a single-shot operation, but

an iterative process in which the model must correctly interpret what is missing, transform it into a useful query and maintain coherence with the initial information need. Smaller models tend to be more error prone in these tasks: intermediate reasoning can degenerate more easily into *query drift*, since understanding or planning errors propagate step by step, accumulating plausible but irrelevant information; conversely, larger and better *instruction-tuned* models have on average greater capacity to maintain the grounding constraint and generate more stable follow-ups, reducing the probability that the "scratchpad → next_query" cycle becomes misaligned with the information need. In CoT RAG therefore, this corroborates how model choice is a fundamental component to achieve target quality-cost tradeoffs.

Breaking down by `question_type` makes the fundamental point clearer: Kimi's advantage is not tied only to "answer quality", but to a better text understanding in the generic sense; in a RAG setting, this translates into: ability to precisely understand the question, correctly interpret what retrieved chunks contain and avoid producing statements that are not traceable to context. This is reflected, as we discussed before, especially in `faithfulness_to_context`, where the gap remains systematically high across all categories: Kimi tends to maintain grounding more easily, while Llama more often introduces "reasonable" but unsupported details in chunks, degrading adherence to retrieval. The fact that the gap is marked also on `Fact Retrieval` is particularly interesting: in these cases complex multi-hop decomposition is not required, therefore the difference cannot be attributed only to the model's reasoning capability, but precisely to understanding the information need itself; essentially, even when necessary information is directly contained in evidence, Kimi seems more consistent in selecting and building the answer from the information that satisfies the query, while Llama is more inclined to complete the answer with inferences that are not requested or not present in context, penalizing faithfulness and, consequently, also `evidence_coverage`. In CoT RAG, then, it is not enough to "answer well" but one must **understand what is missing** and transform it into a truly useful `next_query`: step results suggest that Kimi, on average, triggers adaptivity more often, especially on `Contextual Summarize`, which is consistent with a model more capable of recognizing partial satisfaction of the information need and the need to search further before stopping. This property is exactly what one expects from an adaptive approach: quality does not derive only from the first top-$k$, but from the model's ability to formulate more targeted follow-ups; after all, the fact that it generates more steps does not mean follow-up question quality is lower, but it is perfectly explainable by what metrics tell you about text understanding: Llama declares `enough` prematurely, but because it does not understand the full missing semantics.

An important result that, in my opinion, emerges very clearly is that, in a RAG setting, relying on small models (SLMs) typically does not take you very far. The case of `Fact Retrieval` queries is particularly emblematic: these are questions where the information need is often localizable in a single passage

and, after all, in this dataset adaptivity tends to stop early precisely because the chunks recovered already by the first retrieval step are "enough" to answer; yet, although conditions seem theoretically favorable, clear differences in models' text understanding should not emerge given task simplicity, which however does not happen: Llama shows clear difficulties in the fundamental task of extracting from chunks and organizing for the user the information needed to answer the incoming query and here, note, we are no longer talking about a retrieval bottleneck. Indeed, key information is often already present in retrieved context, but the model still fails to leverage it: it selects it poorly, compresses it incorrectly, or distorts it. This downsizes a rather widespread intuition on RAG, namely that a perfect retrieval is "enough" to make an SLM competitive: in reality, final quality does not depend only on what you retrieve, but also on how well the model can read, integrate and 'stitch' those evidences; it is not rare, indeed, to observe the counterintuitive case in which an SLM with better retrieval produces less reliable or less preferable answers than an LLM that starts from a slightly worse retrieved context, but interprets it much better.

On the same benchmark, I also tested the Naive RAG approach and results are summarized in the following table.

| **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: Kimi–Llama comparison in Naive RAG

Results clearly highlight that the CoT RAG effect is not "universal", but strongly depends on model quality: understanding ability, follow-up formulation and adherence to the information need. In the case of **Kimi**, introducing the scratchpad yields a clear gain on `faithfulness_to_context` (from 0.828 to 0.881), while `answer_correctness` and `evidence_coverage` remain almost unchanged: direction is slightly negative and, in this sense, the model seems to adopt a more conservative behavior, preferring to generate slightly less complete answers but better anchored to retrieved context, rather than risking statements not directly supported. This pattern is consistent with the fundamental objective of RAG: not only producing correct answers, but guaranteeing that each statement is traceable to provided context. The small reductions in `answer_correctness` ($-0.012$) and `evidence_coverage` ($-0.015$), being relatively small compared to observed differences, could certainly be significant of a real tendency but it is also plausible that they are simple fluctuations, for this reason one can consider answer quality and coverage practically unchanged with good approximation. This behavior, after all, is perfectly consistent with a model that uses adaptivity mainly to *reduce unsupported statements*: the scratchpad makes more explicit the distinction between what was retrieved from previous retrieval phases and what, instead, are model deductions/inferences, improving overall answer

grounding; it is a very interesting result, in the sense that, as said, the principle underlying RAG itself is not only answering well, but in a context-supported way.

| **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: Intra-model variation moving from Naive RAG to CoT RAG

The overall picture is diametrically opposite for **Llama**: moving from Naive RAG to CoT RAG, all metrics worsen consistently, as summarized in Table 6. This does not necessarily mean CoT is not a good strategy, indeed: results confirm what we said earlier on the importance of model capabilities, for Llama CoT does not increase useful information, but increases error opportunities; more steps mean more generated queries, more evidence to integrate, more intermediate steps to manage and a less versatile model can more easily deviate from the initial question (*query drift*) or misinterpret retrieved information; after all, there is a very drastic drop both in `faithfulness_to_context` and in `answer_correctness`. An interesting way to see it is this: adaptivity acts as a *multiplier*, when the model is sufficiently capable and instruction-following, iterativity improves grounding; when it is not, it amplifies drift and degrades both context quality and final answer. This result corroborates even more the previous idea that adaptive retrieval typically requires medium-large models, however it highlights another important aspect: although, intuitively, Naive RAG might seem like a 'structural' baseline of CoT RAG, in the sense that, by not triggering adaptiveness, behavior degenerates to the classic variant, this is not directly supported by results; on average scratchpad, CoT instructions, reasoning... confuse SLMs, which perform worse than the Naive baseline.

It might seem strange that answer quality and semantic coverage, given by `answer_correctness` and `evidence_coverage`, do not improve with CoT for a large model like Kimi. In reality, in a setting like ours the key point is precisely that they *do not worsen*: introducing additivity (more LLM calls, more tokens, more points where drift can emerge) without degrading average quality compared to baseline is already a non-trivial result, because it indicates that the model is not "breaking" classic RAG behavior in cases where single-shot retrieval is sufficient, while still gaining clearly on `faithfulness_to_context`; to summarize to the bone: adaptivity, when activated, is useful to the model. I did expect, however, that compared to a Naive baseline overall answer quality would remain on average similar: adaptive steps are triggered few times (see mean steps) and, even if they were more often, we are still in a generic domain with *plug-and-play* components; as said, prompt engineering can mitigate failure modes and allow you to improve performance, but in general it does not make you competitive and this holds in general, not only in retrieval. If you are

looking for clear increases in `answer_correctness`, these typically require more "sophisticated" architectures and referring to the ensembling idea presented in the CoT RAG chapter: it is with models specialized in query rewriting, supervised fine-tunings to follow a specific reasoning scheme or decompositions into sub-agents with distinct roles, that you truly see how a CoT approach improves pipeline performance, solutions we do not adopt here. After all, the intuition is simple: when it was formulated, CoT was mostly induced with prompting, then one moved to post-training phases where models are directly taught how to reason, up to more recent implementations of *Reinforcement Learning*, and that is where consistent increment was obtained for a de facto standard, today, in LMs [Mukherjee et al., 2023, DeepSeek-AI et al., 2025]. For this reason, it is absolutely significant that performance does not degrade compared to the Naive baseline and it confirms the central point of the thesis: Naive RAG is surprisingly hard to beat and remains, in practice, the most robust *go-to* solution in a generic domain.

This last aspect about the "best go-to solution" is also confirmed by subsequent cost analyses. Adopted chunking is Recursive with a fixed window of 300 tokens and the model, for each retrieval step, is given top-5 chunks as context: adding to this scratchpad average length, mean adaptiveness steps, mean token number per generated follow-up query, we can obtain fairly precise estimates on cost increase compared to the classic baseline; after all, CoT is not only a *multiplier* of grounding, but also of costs!

| **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: Cost Analysis: Average Tokens - CoT RAG vs Classic

The most attentive will notice that there is no multiplicative factor on the `Total Tokens` entry: this comes from the fact that API costs of major LLMs, today, are dominated by output tokens; since, therefore, an input token does not weigh like a generated token, deriving a multiplicative factor from the ratio between the two would not necessarily be correct and meaningful of cost increase. Using Groq's pricing model costs, at the time of writing of this thesis, we obtain: an average cost per query of $C_{\text{CoT}} \approx 0.00606\$$ versus $C_{\text{Classic}} \approx 0.00325\$$ for **Kimi** (factor $\approx 1.87\times$), and $C_{\text{CoT}} \approx 0.00037\$$ versus $C_{\text{Classic}} \approx 0.00017\$$ for **Llama** (factor $\approx 2.22\times$). Looking at possible improvements to the CoT architecture, such as those cited earlier, these should not perturb costs excessively, remaining comparable to the current situation differently from performance: obviously,

costs grow a lot based on question difficulty, or rather, on the average number of times adaptiveness is evaluated, however, from these results, it seems that the direction of the tradeoff between possible pipeline performance increase and cost is, in fact, favorable especially for large models.

**Chunk Size** In the absence of a specific domain, various 'industrial' guidelines indicate $200 - 400$ token ranges as recommended; for previous results I therefore chose 300 tokens as chunk size [Singh et al., 2024, Coveo, 2024]. To better study how pipeline performance depends on this, I repeated the same experiments with, this time, a **600 token** configuration: doubling the baseline, chunks on average more self-contained and context generally less fragmented; the choice to double chunk size is, after all, quite consistent with experimental results in the literature, such as those presented by [Juvekar and Purwar, 2024], who propose a wider range between $512 - 1024$ tokens when on average more context is needed for generation.

| **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: Kimi–Llama comparison in CoT RAG (600 token)

Results with 600-token chunks confirm the pattern already emerged in previous discussions: Kimi maintains essentially stable performance, while Llama degrades markedly and, above all, systematically on all judge metrics, with a particularly large gap on `faithfulness_to_context`. Increasing chunk size, i.e., providing more and less fragmented context to the model, does not "help" Llama as one might intuitively expect, on the contrary, it amplifies difficulties in selecting and extracting important information from retrieved context and this is fully consistent with the previously exposed text understanding difficulties. Longer chunks introduce more competing signals (entities, events, descriptions...) and require a better ability to identify truly relevant chunks with respect to the question, maintain coherence in answer construction and, above all, synthesize more information and more details; 600-token chunking reduces fragmentation, but shifts the burden to the model: it requires greater semantic compression. The fact that Kimi holds this regime better, while Llama loses both in `answer_correctness` and in `evidence_coverage`, suggests that increased context becomes an advantage only when the model has sufficient capabilities to exploit it, otherwise greater length translates into additional noise and worse grounding; conceptually, the previous considerations are in fact directly grounded on the idea that more context is difficult to manage for an SLM: by itself, however, we are not isolating the adaptivity effect, therefore the additional text understanding capability we require from the model. Subsequent analyses, indeed, aim to answer the question: *is it adaptivity or the increase in*

*tokens that weighs on the SLM?*

Previous results reinforce, after all, one of the central nuances of this work: in a zero-shot and generic-domain setting, chunk size is not only a "universal" pipeline hyperparameter, but it interacts strongly with chosen model quality. A larger chunk can make retrieval more coherent for narrative texts perhaps, but it also makes more evident the difference between an LLM able to exploit long context and an SLM that struggles to isolate relevant information, even when that information is actually present in retrieved chunks. In the reference setting for CoT RAG, in fact, doubling chunk size yields a proportional cost increase, therefore the initially chosen 300 tokens are a much more sensible choice.

| **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: Kimi–Llama comparison in Naive RAG (600 token).

The cost increase, instead, is fully supported by the performance gain in Table 9: by increasing chunk size, fragmentation decreases and the probability that required information is contained in the same retrieved passage increases, chunks are more self-contained in short. This effect is particularly evident for **Kimi**, which shows a clear increase both in `answer_correctness` and in `evidence_coverage`: a more capable model can better exploit more self-contained chunks, having greater capacity to manage information. In reality, from analyzing **Llama**'s performance a benefit exists: although we discussed how Llama has less capability than Kimi in text understanding, increasing chunk size improves the quality of answers produced by the model across all fronts; this is not a result that contradicts previous discussions, on the contrary, it allows highlighting an important point: in Naive RAG, longer context can be advantageous because, basically, it reduces the need for adaptivity by having more information available, whereas in CoT RAG this becomes a problem relative to the fact that it requires *dynamic* understanding, namely the ability to maintain coherence across multiple steps, correctly update a state (scratchpad), recognize what is missing with respect to the information need and transform it into a useful `next_query` without introducing drift. Increasing tokens, indeed, is not necessarily a problem for an SLM as long as 'static' text understanding is required: adaptivity requires superior semantic understanding capabilities and not only the ability to strictly 'find' information in text, multiplying decision points and making the model more error prone. An LLM such as Kimi tends to benefit from adaptivity because it can control it, while an SLM such as Llama can worsen not because it "sees too many tokens", but because it struggles to manage process dynamics and this explains the negative covariance between performance increase in Naive as tokens increase and the performance drop in CoT for

Llama, answering the previously posed question.

We now present results on the graph-based approach; in theory, this type of pipeline captures overall corpus semantics well and allows different chunks, intra or cross - document, to interact and also answer questions characterized by contexts with document hops. In the case of GraphBench, however, results show rather clear criticalities: the hierarchical approach is *systematically* the worst among those tested, especially on `answer_correctness` and `evidence_coverage`. In Table 10 I report the estimated metrics for Kimi and 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: Kimi–Llama comparison in GraphRAG

The fact that it is systematically the worst does not necessarily mean the approach is not viable, indeed, the advantages of hierarchical pipelines are clear, however much depends on the tested query types and on the previously discussed concepts about summary information loss; indeed results are perfectly consistent with expectations.

The bottleneck is not retrieval in the strict sense, but *compression*: summaries at successive levels inevitably introduce detail loss and, worse, risk "rewriting" information (simplifications, generalizations, omissions...) and this is a fundamental problem when leveraging rollup modules; the impact is direct on `evidence_coverage`: even when the system retrieves nodes "close" to the query, the summarization chain makes it harder to obtain all evidence needed to cover the information need. Interpreting metrics better: despite context faithfulness being comparable to what other strategies obtain, correctness, in the sense of answer quality, drops drastically and this is due to the fact that the model receives chunks that are semantically similar, but belong to different contexts and tries to connect, perhaps, facts or events plausible with respect to the information need, basically producing something that remains anchored to context but that a judge considers practically on par with hallucinations. Here comes an interesting note: *does it really make sense to make different documents interact through expansion?* Conceptually yes, but the downside is precisely due to the fact that chunks still remain local portions of the document and thus explicit facts could be considered plausible with respect to the query but, as said, refer to a distinct universe; from here we already start to understand the fundamental concept that answers one of my research goals: cross-document interaction is an absolutely ambitious and important objective to pursue, however it becomes much less error prone when you do it by passing through an entity you recognize to be the same across two documents.

The other side of the coin is that GraphRAG Bench, by construction, contains many questions with clearly identifiable entities and an information need often solvable with a relatively direct match on original chunks: this is exactly the scenario in which Naive RAG (embedding top-$k$) is surprisingly competitive; the graph, instead, adds an abstraction level that does not bring a reasonable retrieval advantage in this case, often omitting entities in community summaries; as said, this does not mean this is necessarily a problem, indeed, for how we built it it is a feature: technically performance would become competitive with more ambiguous queries, recalling the concept that GraphRAG Bench is built mainly for entity-based KGs. An interesting detail is divergence on `faithfulness_to_context`: Kimi remains relatively high (0.771), while Llama collapses (0.509); this is consistent with the previously discussed behavior for other approaches: Llama is weaker in text understanding and basically you expect worse performance than Kimi, however we discussed how, if not forced to reason, it can still obtain acceptable performance at comparable context. In this case, faithfulness collapse is mainly tied to what we said earlier: chunks are not only different, but interaction yields a higher fraction compared to other approaches of cross-document chunks and, consequently, semantically relevant facts belonging to a very different context *encourage* Llama hallucinations, amplifying already non-perfect text understanding. Small models therefore tend to treat as factual information that seems relevant, with less reasoning capability with respect to the actual question intent and, as said, it is perfectly consistent with the intuition that came from CoT RAG; after all, in my view, this is a very common phenomenon on document bases of realistic size and reasonably heterogeneous, something my setup, as widely cited, simulates imperfectly.

To corroborate the previous considerations, I performed an **ambiguous** rewriting of a subset of 100 queries among the 350 previously tested: reasoning focuses predominantly on entities, building indirect descriptions that keep question intent and information need fixed, but without explicitly naming the entity itself; in this case, rewriting brings query form much closer to that of HotpotQA to that of NQ, therefore more aligned to a general-purpose search engine; this is, in fact, a regime more coherent with using graph-based pipelines: if the question does not explicitly "point" to the correct entity, semantic aggregation on the graph, given by community summaries, becomes, at least in principle, more advantageous than the previous setup.

Results on the subset of 100 rewritten queries are reported in Table 11 and show a clear and consistent improvement for both models on all considered metrics.
The most marked increase concerns `answer_correctness` and `evidence_coverage`, which confirms the previous intuition: compared to results in Table 10, coverage grows substantially, suggesting that the graph more often manages to build a useful *narrowed corpus* without a clear signal about entity presence. `faithfulness_to_context` also improves: it is an interesting result, because on the one hand it does not contradict previous considerations and, on the other,

| 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: GraphRAG standard - ambiguous comparison with respect to Table 10

it allows highlighting a further aspect that comes from this new setup; increased answer quality suggests that chunks returned by retrieval are semantically more aligned to the query than in the previous case and, in this situation, it seems both models tend to remain more anchored to provided context and build overall better answers. The phenomenon must however be interpreted better: in my view, in RAG models show a general bias towards retrieved context, this does not mean they tend to use it, this is exactly what you want, but more that they consider all chunks 'relevant' in the same way; consequently, even noisy portions or even distractors can strongly influence generation: in this sense, increased `faithfulness_to_context` does not necessarily imply a better ability to understand what is pertinent with respect to question intent, it rather indicates that, on average, context has become more useful and less ambiguous, making inferred information less prone to true hallucinations; indeed, it remains possible (and frequent) that the model infers plausible but not directly evidence-supported information, i.e., out of context with respect to the *information need*. This evidence directly recalls the theme of *rewriting* as a tool to produce queries more aligned with the architecture properties you choose and make, precisely, the constrained choice of *fixing* an expected distribution much much simpler.

It would have been natural to also evaluate a *hierarchical graph-based* approach in which the initial query would first be rewritten in a *step-back* form, as discussed in previous sections, used to select communities / macro-contexts and reduce information loss risk due to rollups; subsequently, the original, more specific query would be applied on a reasonably better *narrowed corpus* and we can say this on the basis of previous empirical results. Once again, we stress the core intuition: RAG architecture goodness is extremely dependent on retrieval and therefore directly on expected query form; the fact that an architecture in a given setup does not seem viable because it is underperforming with respect to other alternatives, in reality, does not mean you cannot make it competitive by adding upstream modules, such as *query rewriting*, which is the fundamental consideration of this section.

#### 5.4.3 HotpotQA

#### 5.4.4 HotpotQA

To extend the analysis to a more 'challenging' benchmark than the `Novel` portion of GraphRAG Bench, I used **HotpotQA** [Yang et al., 2018]: a Wikipedia

question answering dataset designed explicitly for *multi-hop reasoning*. Beyond providing a *gold answer* for each question, it includes sentence-level *supporting facts* and question types (e.g. *bridge* and *comparison*) that require combining evidence from multiple documents. The substantial difference with respect to GraphRAG-Bench is not so much the query form, indeed they remain similar also in HotpotQA with, often and willingly, entities and clear information needs, but rather the fact that the answer is not typically retrievable with simple reasoning: in *bridge* questions, for example, the cited entity acts as an *anchor* to identify a bridge entity in a first document and retrieve a second document containing the decisive fact and that, after all, is one of the prompt tunings we had discussed earlier with CoT; this is an important point: GraphRAG Bench ease is not only due to query form, after all to evaluate a retrieval system you must fix clear information needs, even if this remains a problem not to be underestimated, indeed at inference you might have very different and more ambiguous situations, but rather due to the fact that most questions are resolved already with the first retrieval step and adaptivity is not always evaluated: HotpotQA is different precisely in this sense. In *comparison* questions, for example, it is necessary to retrieve and compare information from two distinct pages and, in this setting, the retrieval component weighs substantially more end-to-end: the system must bring among the top results (and then into context) *all* required 'hops'.

As already discussed, the vast majority of benchmarks do not provide the 'raw' text to index and HotpotQA is no exception: for this reason, in building the dataset, I evaluated a scraping phase leveraging Wikipedia's API, chunking and indexing full text. I focused on the 'validation' portion of HotpotQA, fixing 400 examples: this includes only queries labeled 'hard', therefore the maximum difficulty level. Probably, this is the best moment to discuss a fundamental concept regarding some criticalities of my setup: building a dataset to test RAG architectures or chunking strategies is a very complex task, precisely because you would like to have, for each incoming query, a corpus that makes it progressively difficult to satisfy the information need based on the query complexity class and you can do this only if you try to 'fool' dense retrieval with a significant subset of distractors; these amplify the intrinsic ambiguity of the question and make the performance gap between different chunking strategies or RAG architectures clearer. The truth, especially at an industrial level, I have partially already explained: when you have a clear, specific domain in which you want to optimize, then typically you are also clear on what requests are on average made, how they are posed, what you need to optimize, document types... therefore, in domain-specific applications it is typically much simpler to have representative datasets that allow you to derive meaningful statistics for what your inference use is, differently from my experimental setup which, in this sense, is much more complex. When I say that the previous metrics are nevertheless meaningful, despite the corpus being characterized by the previous criticalities, I refer to the fact that: if you want to compare two chunking strategies, or RAG architectures, and you do it on the same corpus, although this has inefficiencies you know

that both architectures/strategies are subject to the same conditions; the metrics you obtain can be arbitrarily inflated, saturating, reduced gap compared to the real one... however the direction remains clear: if a chunking strategy or an architecture performs better on a certain corpus than another, then the metrics you obtain may not be meaningful with respect to the real inference case or the gap you expect may be more or less marked, but the direction of the advantage remains clear and the considerations that derive from it, which may be more or less marked in practice. An interesting analogy is the comparison between two ML models: if you do it on a dataset that is not or only partially representative, then the metrics you obtain may be over/underestimated with respect to real inference usage, however if one model is better than another under the same handicap it will also be so on a more representative dataset; in this sense it is the 'direction' that matters. HotpotQA ties in very well with these arguments, hence why I decided to use the distractors it provides: for each query you have not only *gold* documents containing the relevant information, but also a list of distractors; precisely the latter is necessary to build a more representative corpus, in the sense that, even if large, corpus size is not necessarily meaningful of retrieval difficulty. Having fixed, therefore, the target of 400 queries above, I imposed a balanced distribution constraint between gold documents and distractors, obtaining on average, for each query, scraping of 4 raw Wikipedia documents.

On HotpotQA I evaluated only **Naive RAG** and **CoT RAG** and not the graph-based approach: it would have been exposed to the same failure modes observed on GraphRAG Bench (information loss and summarization-induced distortion), with the aggravating factor that on HotpotQA evidence coverage is even more critical, indeed if you lose one hop, the answer becomes unrecoverable and, again, it is not strictly an architecture problem but a query form one. In Table 12, I report the results of the LLM evaluation on the 350 built examples.

| **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, Kimi–Llama comparison.

The observed pattern is consistent with what emerged on GraphRAG Bench, but "amplified" by higher average query complexity:

- the Kimi–Llama gap grows and remains marked across all metrics, especially on `faithfulness_to_context`;
- benchmark difficulty is reflected in a drop of `answer_correctness` compared to GraphRAG Bench (where many queries were solved single-shot),

because here answers need more context and reasoning capability over it

Breaking down by *bridge* and *comparison* makes even clearer *where* the small model struggles:

- on **bridge** questions (280 examples) the gap is very high across all metrics: here the model must identify a bridge entity and formulate a subsequent hop; planning or understanding errors propagate directly into coverage loss and also degrade correctness. After all, note how, here, adaptivity is triggered in the same way by both models, whereas before we had more steps for Kimi, a sign that adaptivity is clearer to both thanks to bridge intent
- on **comparison** questions (70 examples) the gap shrinks: it is a sensible pattern, because the question often explicitly mentions the two entities to compare, reducing query drift risk and making the operation more "mechanical" (retrieve two pages, extract attributes, compare). A clear advantage of Kimi on `faithfulness_to_context` still remains, a sign that Llama still tends to complete with facts not necessarily supported

As said, mean steps (1.697 vs 1.654) are very close: this indicates that, on HotpotQA, adaptivity is triggered with a similar frequency by both models, but this does not imply it is *managed* with the same quality; the point, indeed, is not "how many" hops you do, but *how useful* follow-up queries are and how much the model can maintain coherence with the initial information need, but considerations are essentially the same.

Compared to GraphRAG Bench results, HotpotQA is exactly the case in which CoT RAG should be more justified: the single-shot baseline has a lower probability of bringing into top-$k$ *all* required hops, whereas adaptive retrieval can correct course more easily. That said, conclusions remain compatible with previous ones: adaptivity is not a universal "shortcut" and, above all, it strongly depends on model capability and is, indeed, risky for small LMs. To make this point more concrete, I also measured the **Naive RAG** baseline and results are presented in the following table. The most relevant result is not only that Kimi remains systematically superior to Llama (a pattern now consolidated also on GraphRAG Bench), but that on HotpotQA Naive RAG is a *very strong* baseline already on its own: `answer_correctness` and `evidence_coverage` are practically aligned (0.729 vs 0.727), while `faithfulness_to_context` remains very high (0.897), suggesting that, in the built corpus, single-shot top-$k$ often manages to retrieve a sufficient amount of evidence to support a correct and well-grounded answer even on a similar but more complex benchmark.
Comparing these numbers with those of CoT RAG (Table 12), an important nuance emerges that replicates what was seen on GraphRAG Bench but in a multi-hop regime: for Kimi, scratchpad and adaptivity do not bring a clear net increase in correctness or coverage, which remain essentially stable, but they tend to improve *grounding*, with a slightly higher `faithfulness_to_context`.

| 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, Kimi–Llama comparison.

For **Llama**, instead, dynamics remain what we already observed: the model is more fragile in maintaining coherence and grounding when the process introduces additional reasoning steps; adaptivity increases drift opportunities and therefore does not guarantee, on average, an improvement over the single-shot baseline.

HotpotQA confirms, after all, two interesting aspects: adaptive retrieval is conceptually more sensible when the information need truly requires multi-hop, but the strength of the Naive baseline remains very clear even in this setting when retrieval is well set up and the model has good text understanding; for this reason, CoT RAG should be interpreted more as a targeted *upgrade* useful to improve grounding or recover missing hops, but not as a general-purpose replacement of the baseline.

## 5.5 Go-To Solution

The cross-cutting result that emerges from benchmarks is that, in a **generic-domain** and **general-purpose** context, the hardest pipeline to beat remains the simplest one: **Naive RAG**, with "correct" choices of sensible chunking, adequate top-$k$, a minimal but robust prompt... and a model that can understand context well. This is, in fact, the central objective of the thesis: showing that a simple but well set up approach is competitive and stable, while more sophisticated architectures tend to pay higher overheads, with respect to the cost-performance tradeoff, without a proportionate gain in the considered setting.

It is important, however, to emphasize a methodological point already discussed earlier: *there is no*, today, a de facto standard for evaluating RAG systems; unlike other contexts, such as evaluating LMs themselves, in RAG the end-to-end pipeline introduces a multiplicity of variables that are hard to control comparably: corpus choice and its "cleanliness", indexing schema, chunking and overlap, retriever and reranker, prompt and output formats, context budget, stopping criteria in adaptive settings and even the very definition of "correctness" in the presence of multi-evidence answers. Changing even a single one of these components can shift the cost-performance tradeoff significantly, making it complicated to build fully fair comparisons. In this sense, benchmarks used in my work reflect a choice you are forced to make a priori: *they are predominantly retrieval benchmarks* or, at least, benchmarks in which information need

is on average clear and, as said, often *entity-centric*; this means that questions do not cover the full variability of the real case: in a general-purpose system, indeed, the user can formulate ambiguous, underspecified queries, with implicit constraints or with intents that are hard to clearly understand. Moreover, "ground truth" can itself become intrinsically ambiguous: there is not always a single correct answer and much of answer goodness is at the discretion of who interprets them. The problem, however, is precisely *formal*: defining reliable labels for truly ambiguous information needs is extremely difficult, because it requires very precise annotations and, basically, strongly characterized by annotator tendencies, therefore with a not-so-clear signal, often, due to structural disagreement between different annotators, making it less clear what is being measured.

That said, choosing benchmarks with clearer information needs, like those used, is also **justified** with respect to the thesis objective: if the intent is to credibly derive architectural tradeoffs among approaches (Naive vs CoT vs Graph/hierarchical), one must reduce as much as possible sources of uncontrolled variability and fix a query set that allows more stable comparisons; to reprise the fundamental concept: *evaluating a retrieval system means fixing a priori the query type*. Indeed, if queries were too ambiguous, it would become almost impossible to distinguish whether a performance difference depends on a retriever limitation, on a judge mismatch with respect to the "gold answer", or on other hidden factors. This is why benchmarks like those adopted, although not fully representative of real-world variability, can still be sufficient to isolate structural phenomena: for example, the fact that summary hierarchy loses coverage, that adaptivity amplifies drift in small models or that Naive RAG is extremely competitive when the question is well specified reinforce this aspect.

Once a regime is fixed in which information need is sufficiently defined to make architectures comparable, Naive RAG emerges as a *go-to* solution, achieving a particularly favorable balance between quality and cost and, after all, in a more stable way to pipeline perturbations compared to other architectures (prompting, chunking, model). It is important to underline that Naive RAG is not "always" better, but it is the one for which, in the considered setting, properties emerge that make it preferable especially in practical industrial contexts; simplicity, cost efficiency and reasonable performance are exactly what one desires in a general-purpose system. This intuition then materializes in three key properties:

- **Robustness:** when the information need is entity-centric and solvable single-shot, Naive RAG is extremely competitive; in domains with a more ambiguous query distribution you might want, perhaps, to push more on the dense component and you can do so by changing the weight in the combined score or, even, by reasoning on retriever fine-tuning if you have a clear optimization context
- **Cost and simplicity:** it does not introduce iterative steps and therefore

minimizes token cost and failure points compared to other architectures. This makes it the natural choice for real workloads, where latency and cost matter at least as much as average quality.

- **Predictable behavior:** fewer components ⇒ fewer couplings; in my case, the hierarchical GraphRAG approach worsens precisely because it adds a transformation (summarization) that negatively interacts with query nature, similarly, CoT RAG becomes a useful multiplier only when the model is sufficiently capable to control iterative dynamics, otherwise it amplifies query drift

Operationally, if the objective is to build a reliable RAG system, the most sensible strategy, in my view, is the following:

1. start from a **Naive RAG** with **Recursive Chunking** as a *go-to* solution: simple, robust, inexpensive
2. optimize high-impact "low-level" hyperparameters, such as chunking, context top-$k$...
3. evaluate introducing **CoT/Adaptive Retrieval** only when workload analysis shows a real lack of context and only if, relative to cost constraints, one can use a model strong enough to control iterative dynamics, as well as evaluate domain specialization choices, if possible, for better follow-up queries
4. avoid chunk-based hierarchical architectures when expected query distribution is predominantly entity-centric; alternatively, in contexts with a clear domain, a KG-based approach could bring absolutely relevant advantages

In the considered generic-domain setting, the "go-to" solution is not the most complex one, but the most *robust* one and experimental results clearly show that beating a well-built Naive RAG is, in practice, surprisingly difficult.

## 5.6 Streamlit UI

It is possible to interact with all RAG architectures via a simple *Streamlit* UI available in the GitHub repository supporting the thesis (`https://github.com/ddemarchis11/TesiMagistrale_AdaptiveHierarchicalRAG`). The user can select the RAG pipeline via a multi-choice menu, as well as tune high-impact execution parameters such as: number of chunks delivered to the model by retrieval, number of CoT steps... providing greater interaction with more technical architecture components. As widely treated, the fundamental RAG concept is to foster source transparency, indeed the interface allows directly viewing retrieved chunks and, optionally, useful debug information to analyze retriever behavior and context quality provided to the model.

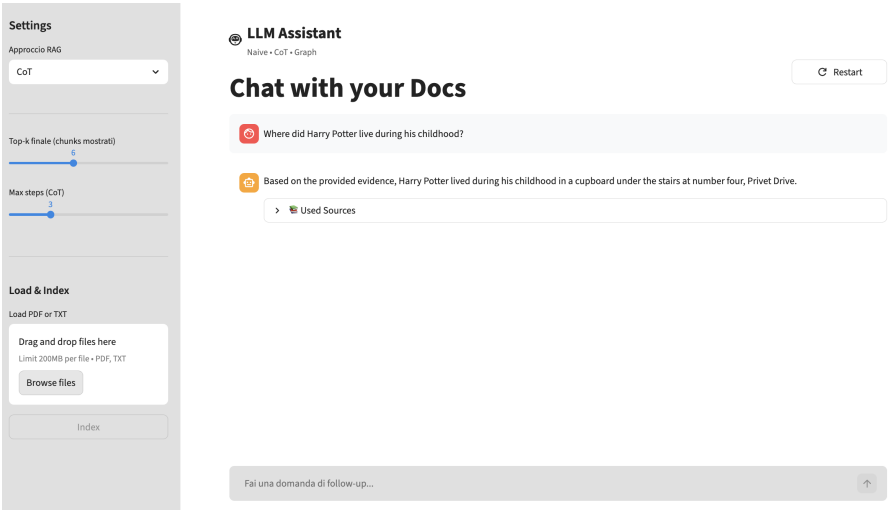


Figura 4: Streamlit UI Layout - CoT Architecture

### 5.6.1 Text Extraction

The UI also allows us to discuss a fundamental phase for 'real' RAG projects: **Text Extraction**; the user can upload files in `.pdf` or `.txt` format, which will then be indexed on Elasticsearch and thus made queryable. My idea was, precisely, to show how UIs should not only be QA "front-ends", but above all an entry point to autonomously upload documents themselves, relying on very simple formats that do not require excessive design effort. The importance of Text Extraction has already been discussed, especially in relation to chunking strategies based on document structure, and it makes little sense to repeat it again. For PDFs, extraction can be performed with fairly simple Python libraries such as `pypdf`, which offer direct methods to derive textual content; however, this "simple" solution is often insufficient when the objective is to preserve logical structure or, more generally, document layout: headings, paragraphs, columns, tables, footnotes... can be reordered or mixed with text by simple approaches such as `pypdf` and this stems from the fact that PDF is primarily a graphic layout-oriented format rather than content semantics [Atagong et al., 2025, Ramakrishnan et al., 2012].

Preserving structure is not a purely aesthetic detail: it directly determines indexed text quality. In PDFs with complex layouts (multi-column, tables, forms), a "plug-and-play" extraction tends to mix text and structural elements, producing chunks where useful content and noise (table rows, form fields, repeated headers...) overlap, degrading both retrieval (dense, full-text or hybrid) and generation. If structure is not maintained or reconstructed, it becomes difficult to apply targeted chunking (by sections, blocks or tables), which is often an ex-

tremely simple and efficient solution in domain-specific contexts (see manuals, legal benchmarks...). This naturally links extraction to OCR and *document understanding* techniques: having talked about Azure SDKs, indeed, to populate Storage you could leverage *Azure AI Document Intelligence*, the previous Form Recognizer), which includes *Read* models for OCR and *Layout* models for document structure analysis, with capabilities to extract not only text but also tables and other structural elements, offering possibilities both to use pretrained models on very common documents (receipts, balance sheets...) and to fine-tune them on your scenario [Microsoft, 2025, Microsoft, 2024].

In the repository supporting the project, more specifically under `ui_data/uploads`, there is also a document I used to test interaction with the platform: I refer to one of Microsoft's latest balance sheets; in documents of this kind formatting is absolutely crucial: layout, tables, structure... it is fundamental to have ways to keep them intact during indexing and, above all, make information queryable efficiently with respect to data type, an aspect we will discuss in the concluding part of the thesis. By trying the system on the balance sheet, looking not only at produced answers but also at retrieved chunks, you can already see with a few queries that: precise information contained inside tables is, often and willingly, misinterpreted by the model itself, while information contained in continuous text is organized much better. This is not a hallucination problem of the model, but precisely of chunking and Text Extraction: this phase is implemented in a very simple way and without complex OCR models, in this sense chunking remains agnostic to structures that can be 'nested' inside the text it splits, producing actual artifacts that directly affect answer quality decrease; for example: by not preserving table structure, a chunk could contain the reference metadata for the attribute and the actual value in two distinct chunks, completely decontextualizing the latter and, basically, this is one of the main failure mechanisms in this simple setup.

### 5.6.2 Incrementality

A fundamental aspect, already partially discussed, of proposed architectures concerns **incrementality** and, more generally, maintainability as corpus size changes. At an implementation level, all architectures support indexing both in **batch** mode, starting from a well-defined set of documents, and in **incremental** mode per document, coherent with the UI application scenario I wanted to model, where the user can upload documents at different times. For Naive and CoT RAG there is not much to say: the architecture idea does not directly leverage hierarchical structures to organize corpus, consequently dynamic addition of new documents and new chunks does not directly affect procedure goodness, even if you become progressively more subject to Hubness, Semantic Collapse... but this is not necessarily a disadvantage, precisely because, as discussed, you do nothing a priori to limit them. The opposite holds for the hierarchical approach, where a structure exists and incrementality can be hard to manage: recalling what was said in the relative section, each document can

indeed be processed autonomously, however the bottleneck emerges at the next level, namely in building and maintaining `COMM_SIMILAR_TO` edges among communities; the kNN approach on community summary embedding space requires not only connecting new communities to the already present global graph, but potentially also updating the neighborhood of previous communities: new points in vector space can in fact enter the top-$k$ of pre-existing nodes, partially modifying choices previously made by the algorithm. In the prototype, I did not implement **periodic trigger** mechanisms or automatic kNN recomputation for this last level: cross-community connections formed at a certain step and with a corpus size $N$ will be the same at $N + k$; more simply: I did not implement mitigations for this phenomenon, in the sense that the primary goal is not building a product but evaluating the architecture, therefore it seemed more correct to discuss possible mitigations theoretically.

If one were to move to production, indeed, there are different plausible strategies to manage this incrementality problem on hierarchical structures, each with a different compromise between cost and quality:

- **Local Update**
  At each insertion, top-$k$ neighbors are computed *only* for new communities and related edges are added. This solution is the one I chose, cheap yes but it does not mitigate graph degradation

- **Periodic Update**
  Insertions are accumulated until reaching a threshold $K$ (or a fixed time interval) and then a broader update policy is executed: typically the only thing that would make sense is a global recomputation, very costly but maximum quality and you try to mitigate cost growth by fixing a sufficiently large $K$; conceptually it is a periodic recomputation

- **Proximity Update**
  Conceptually, kNN recomputation on pre-existing communities must be done as a function of the fact that, with document additions, a given community's neighbors might change. It therefore makes sense to adopt an intermediate strategy between local and global update: when you add a new document and thus evaluate kNN for new communities, you propagate recomputation for a subset derived from the kNN operation over all new communities, namely the set of pre-existing neighbors distinct from new nodes to add, precisely because it is on these that you have ambiguity of 'neighborhood change'; you can decide whether to do a simple union, choose a subset adopting a relative metric, for example, from the kNN procedure itself for ranking, or other heuristics. In this sense, you have a good compromise: higher cost than Local Update but lower than a global one, and mitigation quality is intermediate in the same way

# 6 RAG for Structured Data

Up to this point, we have implicitly assumed that the reference corpus is composed of unstructured text: articles, novels, web pages... In practice, however, some databases, such as those in enterprise contexts for example, always have a different heterogeneity: documents are mixed, within them structured and unstructured data coexist and it is enough to take balance sheets as an example, where text and tables coexist. This section addresses the problem of how to extend RAG pipelines to such heterogeneity, with a specific focus on tabular data, which represents the most common problematic case for the architectures discussed in previous chapters.

The starting point is an observation already emerged in the Streamlit UI section: indexing Microsoft's balance sheet with the standard textual pipeline, information contained in tables was poorly retrieved and formulated not due to a Language Model problem, but rather due to an upstream problem in Text Extraction and chunking; on Text Extraction, in reality, we will not focus, in the sense that the reasoning and needs remain the same as those discussed previously and chunking is much more interesting here. Even assuming that you can preserve layout form, applying chunking strategies designed for continuous text: a chunk may contain the name of a balance sheet line item and its corresponding numeric value in separate positions, or even in distinct chunks, decontextualizing the information and making retrieval unreliable; this is not an edge case, but a structural behavior that derives from the impossibility of treating structured data as unstructured without intermediate transformations.

## 6.1 Data Lakes

A Data Lake is a centralized repository that stores raw data regardless of structure: textual documents, files of different extensions, images... The main characteristic is precisely versatility in managing different formats, indeed it is extremely used in corporate contexts for its flexibility compared to traditional Data Warehouses, which require a very rigid normalization phase before ingestion.

Relative to our main topic of RAG, Data Lakes introduce fairly evident architectural complexities: there is no single indexing strategy that works uniformly for all data types, so much so that, in the end, you are forced anyway to build normalization pipelines a posteriori, even just to extract text. The discussed chunking strategies, semantic search on embeddings, hybrid BM25 + dense retrieval... everything we discussed is designed for unstructured text and assumes that the fundamental unit is a token sequence; tabular data violates this assumption: information is not in the token sequence, but in the relationships between rows, columns and values; a bidimensional structure, therefore, which chunking strategies, whether Fixed, Recursive, Semantic... simply do not account for. What I will discuss in this section is precisely understanding *how* to

treat structured tabular data in a natively unstructured context such as RAG and what trade-offs the available alternatives entail.

## 6.2 Chunking and Tables

Before discussing strategies, it is useful to formalize the chunking problem for tabular data, which differs substantially from that for continuous text for two main factors: *granularity* and *relationality*. In text, the fundamental semantic unit is the sentence or paragraph and chunking aims to preserve such units. In a table, possible chunking units are hierarchically very different from each other: the single cell, the row with all its values, the column with its header, a block of thematically correlated rows, the entire table... in short, one clearly introduces an additional dimension of ambiguity on the object of chunking itself: this time, it is quite clear that the performance discriminant is not only the strategy, but the relationship between strategy and chosen unit. Going more into specifics:

- **Row-level chunking**: each row is indexed as an autonomous chunk, prefixing column names (headers) to their respective values. It is the finest granularity and offers maximum retrieval precision, provided, clearly, that the single row is interpretable on its own. It works well with tables with few columns and rows, precisely, semantically independent, while it performs worse when row meaning depends on adjacent, previous or following row context for example
- **Block-level chunking**: groups of thematically correlated rows are indexed together as a single unit. It requires a criterion to identify blocks, which can be structural or also semantic; for example: similarity between embeddings of adjacent rows has a logic analogous to textual Semantic Chunking. It increases available context for retrieval, at the expense of specificity
- **Table-level chunking**: the entire table is indexed as a single chunk; it is the simplest solution and preserves all internal relationality, but it produces units that are often too large to be retrieved with precision and difficult to represent in a single embedding. After all: full-text search is conceptually great for tables, however, recall, lexical overlap is agnostic to structure, therefore you could obtain tables with a very high rank and that contain the same search words but in distinct cells/positions, effectively changing meaning; for tables, lexical overlap is even less semantic relevance than for text

There is no universally optimal choice, of course: the right granularity depends on table structure, expected query types, model context constraints... in the absence of domain information, row-level chunking with prefixed headers represents a reasonable starting point, for the same reasons why Recursive Chunking was indicated as a go-to solution in the textual case: it is simple, controllable and does not introduce dependencies on additional components (embedder);

table-level is viable, but excessively burdensome, block-level is much more dependent on the semantic component than row-level.

Each alternative implies a different trade-off between context and specificity: for example, an isolated cell is too atomic to be interpreted without context, indeed the value `42.3M` means nothing without knowing which line item and which year it refers to, while the entire table, conversely, can exceed the context window more easily and mix very different signals in a single embedding. After all, there is a fundamental semantic problem: SentenceTransformers are born and optimized predominantly on textual data (sentences, paragraphs, query–document pairs, NLI, STS), where semantics is conveyed above all by word order, co-occurrences and linguistic dependencies. Tables, instead, encode information differently: the meaning of a cell is not (only) in its lexical content, but above all in the structural relations that tie it to other elements, rows/columns, units of measure, temporal granularities... consequently, applying a zero-shot SentenceTransformer to tabular representations often yields information loss: the model treats structured data as unstructured and it is simply not trained to understand the latent relations discussed previously, therefore you might observe a dense retrieval behavior very different from what you would expect.

## 6.3 De-Structuring

Having defined the chunking problem, strategies to treat tabular data in a RAG system bifurcate into two conceptually opposite directions, answering different questions and needs.

### 6.3.1 Text-to-SQL

The first direction gives up the idea of RAG as a pipeline exclusively for structured data and adopts an approach that preserves relational structure: given a starting query, expressed in natural language, the Language Model generates a corresponding SQL translation that is executed directly on the relational database; although Text-to-SQL seems very different from the RAG concept, it is in reality absolutely assimilable to a 'module' of it: in more sophisticated applications, we already discussed the importance of query rewriting and topic classification, indeed Text-to-SQL translation can be seen as a particular declination of both and you can always provide results obtained from the SQL engine as context supporting generation. In my view, it is a very interesting parallel, in the sense that any retrieval-oriented operation, whether structured or not, that you perform on the query to provide context is always alignable with a RAG pipeline. This approach has a fundamental structural advantage: traditional RAG on purely textual data could not, for example, effectively compute aggregations, statistics, counts... Text-to-SQL allows you to cover, with good complexity on query translation, also a part of information need you cannot cover by treating tables as text, or rather, that you cannot cover unless you have a way to query an aggregation engine.

The real bottleneck is not SQL generation per se: modern language models produce mostly correct SQL in zero-shot on simple schemas and relatively standard queries. The problem is *mapping to a common format*, namely schema canonicalization in realistic Data Lake contexts: in reality, one often has heterogeneous schemas, poorly descriptive or abbreviated column names, inconsistent types across different sources, denormalized tables... in these cases, the model does not have the elements to generate correct SQL without an explicit schema linking phase, which associates query terms to schema entities. Works such as DAIL-SQL [Gao et al., 2024] and DIN-SQL [Pourreza and Rafiei, 2023] address precisely this problem, decomposing generation into sub-problems: schema linking first, SQL generation then, with an approach that recalls the CoT RAG discussed in previous chapters. RESDSQL [Li et al., 2023] pushes further in this direction, explicitly separating the linking phase from generation and showing that this decomposition improves robustness on complex schemas.

The canonicalization limitation is not trivial in a real Data Lake and, after all, it is not always automatable without introducing some degree of error. However, I do not exaggerate if I say that building a RAG pipeline and introducing a Text-to-SQL model is the most efficient way you have to integrate structured data within RAG, although it is not a *go-to* solution: recalling previous concepts, when you do not have a specific domain to optimize on you try to make choices that are as correct as possible and to specialize later, in fact Data Lakes do not 'fall from the sky' and if you work on these you most likely also have a specific optimization context; in terms of performance-to-complexity tradeoff, Text-to-SQL provides an overall balanced ratio, very complex to design and maintain with respect to schema changes, but very efficient, although one could think of leveraging different indexing strategies, such as those we will present later.

### 6.3.2 Serialization

The second direction is, in reality, diametrically opposite: instead of preserving relational structure, we want to keep RAG as predominantly for unstructured data and thus transform tables into unstructured text, indexable with the same RAG pipelines used for purely textual documents. This approach, which I call *de-structuring*, has the advantage of keeping the pipeline uniform and not requiring excessively complex modifications to what you already do for textual content; it is an idea extremely anchored to the purpose of this thesis: in my generic-domain and, at this point, also generic-data case, the need is to be able to treat in a simple and reasonably robust way also data types different from natural text and for which chunking strategies typically fail; keeping strong the concepts of *go-to* solution, *time-to-first-result*... de-structuring allows you to reduce the problem of treating structured data to the problem of treating unstructured data and architecture reusability is a benefit I like a lot.

There are different table de-structuring techniques, which vary depending on how much you tend towards a complete translation of the table into natural language:

- **Structured Serialization** (Markdown, HTML, CSV)
  a table can be converted into a textual format that preserves at least visual structure; the most common case is conversion into Markdown: each row of the original table becomes a row of the Markdown table, with cells separated by the | character and a header row that makes columns explicit. This procedure is deterministic, does not introduce information loss and produces a representation that many Language Models can interpret in a fully reliable way. The main limitation, in reality, is the same we discussed about chunking strategies for tables: serialization does not solve embedding and lexical overlap problems discussed previously

- **Key-Value Serialization**
  each cell is represented as a `header: value` pair and rows are linearized by concatenating these pairs in sequence. Compared to Markdown it is more verbose, but it reduces any positional-interpretation ambiguity: each value explicitly includes its relative metadata (column name), making the chunk more self-explanatory which is, basically, an advantage; as said, Language Models typically interpret Markdown well thanks to | separators, but with more explicit structures you have higher guarantees. This strategy is particularly suitable for tables with few columns, where header repetition does not introduce excessive noise; the underlying problems are the same as Structured Serialization

- **LM-based Description**
  a Language Model is used to produce a textual description of the full table or of a subset, building it via descriptions of rows or blocks. The description can be content-oriented ("In 2023, revenue of the Cloud segment was 87.9 billion dollars, up 16% from the previous year") or structure-oriented ("The table reports revenues by business segment in years 2021, 2022 and 2023"). The advantage is that produced text is semantically dense and can be aligned with document lexical distribution if, for example, you also provide the LM with the text positionally close to tables and thus introductory to them. The main problem is, in reality, quite simple to understand: retrieval strongly depends on description quality, after all the description must always be at a conceptually more abstract level than individually contained information and, precisely, when you do rewritings with Language Models you structurally weaken the full-text component of retrieval. In my view, however, disadvantages are absolutely outweighed by advantages in my context: this is the most natural de-structuring, you truly treat the table as purely textual content, you can reuse RAG pipelines on unstructured data and you basically solve the embedding problem of tabular records

The choice is not trivial and critically depends, like all search engines after all, on expected query distribution; retrieval system performance is not an "absolute" property of indexing or of the adopted SentenceTransformer/LM, but emerges from the interaction of all design choices you make. A work coherent with what we are saying is that of Sui et al. [Sui et al., 2024], which systematically studies the impact of different serialization formats on LLM performance. The authors show that de-structuring into natural language tends to be competitive (or superior) when information need is predominantly semantic/descriptive, because it improves lexical alignment and makes dense retrieval more natural for how we have seen it; conversely, keeping a representation closer to table structure is preferable for queries with precise numeric constraints, where textual translation can introduce ambiguity or approximations, recalling the information loss concept we discussed earlier: with a rewrite you inevitably lose something. The result is consistent with the discussed trade-off: serializing into natural language favors retrieval but typically makes you lose precision, a structured serialization moves in the opposite direction.

As I said, rewriting into natural language is what is closest to a *go-to* solution and precisely because many RAG systems operate predominantly on semantic/descriptive requests: it is not because it is universally better, but because it tends to offer a good balance between robustness, model compatibility and ease of integration, especially when the primary objective is maximizing the probability of retrieving the correct context from variable queries that are not known a priori.

### 6.3.3 HybridQA

To quantitatively evaluate serialization impact on retrieval, I proposed the same setup adopted for comparing chunking strategies in previous chapters: the retriever is fixed (hybrid search: BM25 + dense with `baai/bge-large-en-v1.5`), the evaluation dataset is fixed and only the serialization strategy of the tabular component is varied.

As a benchmark we use **HybridQA** [Chen et al., 2020], a question answering dataset built on Wikipedia tables in which each cell can be linked to one or more textual passages extracted from and directly associated with the referenced entities' pages. It is important to understand dataset structure, since it reflects the nature of associated queries: on Wikipedia, a table listing a band's members, for example, can contain cells linked to the biographies of individual musicians, and the correct answer to a question may require crossing this structure–text boundary; each example in HybridQA consists of a question, a Wikipedia table and a set of textual passages associated with the cells themselves. In light of the results we want to obtain, it is necessary to isolate the portion of queries whose information need is directly contained in tables and not in the associated passages: I built a script that performs precisely a labeling of queries based on this, obtaining a subset of 1400 queries and 400 associated tables. The adopted

reference metric is Hit@k, consistently with the choice made in the Chunking Strategies chapter: it measures the fraction of queries for which at least one relevant chunk appears among the top $k$ results returned by retrieval and it is particularly suited to contexts where the primary objective is estimating how a certain chunking strategy affects context construction.

In my setup, as said previously, the most relevant comparison from the standpoint of the cost-performance tradeoff is between what I consider the two most viable alternatives to extend RAG also to structured data: **row-level chunking** and **LLM de-structuring**. Regarding the latter, precisely, there is a design choice to discuss on dataset construction: as I discussed in previous chapters, theoretically you would like a corpus as large and heterogeneous as possible in terms of distractors; in my setting, this would mean generating tens of thousands of descriptions and the costs, in terms of Language Model usage, would become absolutely prohibitive for me. However, indexing a small subset of tables and associated queries can still allow us to understand the viability of one strategy compared to another, since what interests us is metric 'direction' and not strictly the metric itself.

### 6.3.4 Results

We therefore present retrieval results obtained on the selected query subset of the HybridQA dataset, using:

- **sparse** retrieval (full-text BM25)
- **dense** retrieval (embeddings with `BAAI/bge-large-en-v1.5`)
- **hybrid** retrieval by combining the scores of the two approaches

Consistently with what was discussed in previous sections, the objective here is not "finding the universally best strategy", but observing how performance varies with respect to the representation you choose to de-structure the tables themselves. To obtain the

I built three distinct indices, each corresponding to a de-structuring choice, and evaluated hybrid search on these with respect to each of the 400 queries. In reality, I exploited fairly well-known concepts when working with SentenceTransformers: you cannot strictly rely on the fact that, without adequate context, the model can build a well-positioned vector in semantic space; consequently, the 3 indices differ not only by strategy, but above all by what you provide to the model to infer semantics:

- `hybridqa_tables_llm`: *table-level* indexing unit, obtained via *LM-based description* and with additional context given by intro + generated description. HybridQA is an excellent dataset for these tests, since it directly provides what, earlier, we mentioned as 'positionally close' text to the table, therefore introductory to it and which we call "intro"; this helps

precisely relative to what we said earlier: the quality of the vector for the LM-generated description also depends on what it can, or cannot, directly infer from text and introductions help to obtain semantically better positioned vectors

- `hybridqa_rows`: *row-level* indexing unit; for each row I indexed **table title + row**. In this case, I decided not to use the intro to avoid semantic collapse of embeddings: as we already discussed, public SentenceTransformers are not necessarily trained on embeddings of tabular row serializations, consequently, if all rows share the same introductory block, the effect is diametrically opposite to the previous one, since semantics given by the `key:value` record would be excessively diluted by introductions, much 'longer' and denser in information, producing vectors that are too close to each other in space

- `hybridqa_intros`: *intro-only* baseline, i.e. exclusively the positionally close text/introduction of the table, without tabular content; the idea is to deeply understand intro quality, since, by using them, if these are excessively informative then obtained metrics, especially on the generative LM approach of interest, could be inflated

| 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: Retrieval Evaluation on `table_only` queries

As a fundamental result we obtain that LM-based description does maximize dense retrieval, but penalizes the full-text component; in reality something we could reasonably expect, consistent with the discussions made. The `hybridqa_tables_llm` index achieves the best overall result in **dense** mode:

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

and it is clearly superior to the `intros` baseline and to the `rows` variant. This is consistent with what was discussed in the *de-structuring* section: rewriting into natural language produces a text more "aligned" to the type of semantics a SentenceTransformer captures well (*co-occurrences*, *natural lexicon*, *relations explicitly expressed linguistically*), and therefore it tends to improve separability in vector space. The downside is evident in **sparse**: `tables_llm` sparse has

Hit@1 0.312, lower than `rows` sparse 0.402; it is the direct consequence of the already anticipated trade-off: when I rewrite, I inevitably structurally weaken the full-text component (paraphrases, synonyms, detail compression), therefore I reduce lexical anchoring that BM25 exploits. Overall, as said, dense retrieval is superior for the LM-produced description compared to the row-based approach: this confirms and does not contradict the reasoning made previously, after all as SentenceTransformer I used a 'large' model that, in real applications, you would be less inclined to use and differences could be even more marked at inference; direction is therefore clear: there is a gain and it could also be more marked considering a smaller model, however it might not be reasonably justified by the loss on the full-text component and, indeed, much still depends on expected query type.

My expectation of "more context $\Rightarrow$ better dense" is respected and, indeed, it holds: `tables_llm` is the best in dense, however, from analyzing other strategies, one observes how the title alone cannot resolve the semantic ambiguity of record linearization, although evaluating additional strategies, still row-level, but with more context could cover inference edge cases where the latter fails.

Summarizing observed behavior:

- **Row-level** tends to favor **full-text**: it preserves "raw" labels and values, excellent for lexical matches and queries with precise constraints. Dense is good but less dominant as $k$ grows, precisely because row semantics can be poor (many numbers, perhaps)
- **LM-based** tends to favor **dense**: description creates better semantic signals and you can also reasonably control lexical alignment, although it tends to reduce full-text effectiveness, therefore the hybrid compromise may not be the best even with a very strong dense

This is exactly why, in heterogeneous Data Lake contexts, it makes sense to reason more on a *portfolio* of representations than on a single strategy, despite my tendency to consider generative rewriting as absolutely valid.

**Integration with RAG Architectures** The previously built indices, as already mentioned, make the built RAG architectures fully reusable: once you have indexed and connected the pipeline to the query engine, the architecture is fully agnostic and independent of index contents; reusability is the fundamental concept of why I chose to study de-structuring, although it does not mean you can use the exact same chunking strategies as before. Indeed, transforming structured data into unstructured text allows you to integrate much of them, however treating tabular data still requires an upstream step that does not exist in the purely textual case, primarily oriented to **table recognition** mechanisms and adopting a classic chunking strategy (still Fixed/Recursive/Semantic...) where, however, in my view it is rigorous to keep in a single chunk the generated

description/rows and the positionally close text, even at the cost of producing chunks that are reasonably larger than fixed size.

**Cost Considerations** As widely discussed in this work, choosing a better strategy always means analyzing *trade-offs* deriving from different indicators and not only performance goodness, after all on a setup that might not reflect real-case variability. The previous analysis allowed us to understand viability of individual strategies, however: much depends on expected query type and, not only, also on the cost you would have to sustain to adopt the strategy, which we discuss better here. In terms of costs, the row-based strategy is in fact absolutely efficient: it does not employ Language Model calls, it is very fast and depends, in reality, on the preprocessing you do on text and with competitive performance; the LM-based strategy, instead, has higher costs and latency, due to calls precisely to Language Models. As said, costs are always given by a token estimate you can make for input and generated tokens: in this case, one situation does not necessarily dominate the other, also because input includes positionally close text and not only the table itself, moreover you can put constraints on generated description 'size' but it does not necessarily make the strategy cost efficient. A performance gain, as said, exists and is clear, however it might not be justified by cost increase compared to the row-based strategy: after all, we are talking about an order of magnitude relative to inference costs of Large models, also because to the LM you are asking to provide a table description that still requires capability to analyze row-column links and semantic interaction with the provided introductory context and this serves to understand that, generally, using Small models could reduce description quality a lot and, therefore, dense search performance. Cost then also strongly depends on number of calls: approaches already discussed such as Microsoft GraphRAG, in my view, often turn out to be suboptimal precisely because using an LM at indexing time means making as many calls as corpus size and, in real contexts, this very easily approaches the order of hundreds of thousands of units, perhaps millions, quickly exploding costs, especially in enterprise mixed-text contexts (balance sheets, for example). Overall therefore, as mentioned earlier, the LM-based generative strategy, in my view, is the most stimulating and interesting in a research project, however pushing on more efficient ways to improve the dense component for the row-based approach could lead to more concrete advantages, in the average case, with respect to the cost-performance tradeoff.

# 7 Summary

We conclude by summarizing the most relevant results and considerations that emerged during the discussion of the work, comparing the main architectures discussed: *Naive RAG*, *CoT RAG* and *Hierarchical/Graph-based RAG*.

The fundamental result we arrived at is that: in a *zero-shot* and *generic-domain* setting, the hardest pipeline to beat is often still the simplest one, namely a well

set up *Naive RAG* (sensible chunking, adequate top-$k$...), paired with a Large model with strong text understanding capabilities; many more sophisticated architectures pay additional overhead and introduce extra criticalities without guaranteeing a proportional gain in the cost-performance trade-off, which is central in my analyses. On the GraphRAG Bench benchmark, *Naive RAG* is a strong baseline: it maintains high and well-balanced values on correctness and coverage, with excellent grounding as well; the same conclusion is also confirmed on HotpotQA: even though the task and the reasoning required to build answers are more complex, single-shot retrieval remains competitive with a good embedder and Recursive chunking. As widely discussed, performance of a RAG pipeline is strongly dependent on chunking *strategy* and *size*: moving to longer chunks (600 tokens), Naive RAG tends to improve, especially for LLMs, because it produces chunks that are on average more self-contained and makes it more likely that retrieval contains the evidence needed to satisfy the information need, at the cost of increased costs.

*CoT RAG* introduces an iterative retrieval guided by scratchpad and follow-up queries. Results show that the effect strongly depends on Language Model quality:

- for a Large model, adaptivity and, therefore, forcing the model to reason during retrieval, tends to improve *grounding*; indeed, on GraphRAG Bench one observes a clear increase in `faithfulness_to_context` against marginal variations on `answer_correctness` and `evidence_coverage`
- for a Small model, adaptivity is often *risky* due to weaker text understanding capabilities and, above all, to the absence of a true post-training phase where reasoning is taught, as in modern foundational models; query drift opportunities and errors in evidence usage increase, with a systematic degradation of metrics with respect to the Naive baseline

At the cost level, iterativity obviously multiplies tokens and calls: on average, spend per query grows by about $\sim 1.9\times$ for the "Large" model and $\sim 2.2\times$ for the "Small" model in the evaluated setup, making CoT RAG certainly a performance upgrade, but not necessarily justified by costs.

The hierarchical graph-based architecture is grounded on the objective of exploiting a *coarse-to-fine* retrieval via a summary-based *rollup* phase and subsequent *drilldown*. The most important structural limitation emerged during tests coincides with introducing an often excessive *information loss* during rollup, which directly reflects into drops in `evidence_coverage` and, consequently, in answer quality. In the tested setup on GraphRAG Bench, GraphRAG is systematically the worst approach on `answer_correctness` and `evidence_coverage`: the main interpretation is that the bottleneck is not retrieval per se, but the hierarchical compression itself; summaries, indeed, lose detail and rewriting information makes it harder to handle even very simple queries, such as entity-centric

ones. After all: linking back to the initial proposal, this is exactly why I abandoned the idea of LLM Stitching as support for semantic clustering.

Another fundamental insight concerns pipeline sensitivity to query form: as mentioned earlier, the graph-based architecture is extremely emblematic of this, precisely because semantic rollup allows better responses to queries with a higher degree of semantic/descriptive ambiguity, indeed the approach improves clearly on all metrics in this case, but performs poorly in all those cases, such as entity-centric queries, where the full-text component is the protagonist. This suggests, moreover, that *query rewriting* is a fundamental concept not only to improve inference performance of an architecture, assuming one knows expected query distribution, but also to make underperforming architectures competitive. Important considerations also emerged by analyzing possible extensions of the RAG paradigm to *mixed* structured data (text + tables), typical of enterprise Data Lakes. The core point is that tables violate the implicit assumption underlying RAG: information does not reside in the sole sequence of textual tokens, but in row–column–value relations; applying chunking and indexing strategies for text "as-is" produces excessively noisy chunks, with headers and values separated, incomplete records... making retrieval structurally unreliable.

The proposed division of tabular chunking strategies is grounded on a compromise between: granularity and relationality/context precision; in the absence of information on domain and query distribution, *row-level chunking* with explicit headers represents a reasonable starting point and, in fact, for reasons analogous to those discussed for Recursive Chunking in the textual case: simplicity, controllability and absence of dependencies on additional components. By also exploring more sophisticated alternatives, conceptual intuitions and adopted tests suggest that:

- a **Text-to-SQL** approach preserves structure and enables even very complex queries, such as aggregations (counts, statistics, group-by), which a purely textual RAG would not manage with the same simplicity; the practical bottleneck is not SQL syntax, but *schema linking* and canonicalization in realistic Data Lake scenarios which are very often characterized by heterogeneous and poorly descriptive schemas
- a **De-structuring / Serialization** approach transforms the table into a textual representation to reuse the whole unstructured RAG pipeline; the advantages of uniformity and *time-to-first-result* are clear, disadvantages are very similar to those discussed in retrieval: a trade-off between semantic alignment, useful to dense retrieval, and weakening of the full-text component

The experimental comparison on HybridQA (subset of `table_only` queries) confirms what I expected:

- **LM-based description** (`hybridqa_tables_llm`) maximizes **dense retrieval** performance (Hit@1=0.493, Hit@5=0.751), consistently with the

idea that textual rewriting provides a form more coherent with the semantics captured by a SentenceTransformer (trained on natural text and not on record serializations)

- **row-level chunking** (`hybridqa_rows`) better preserves the **sparse/full-text** component, keeping "raw" labels and values particularly suitable to more specific queries; the same rollup concept holds in the graph-based approach

Overall, results confirm that, as for RAG architectures, better choices can exist but are not always dominant; in highly heterogeneous data contexts, such as Data Lakes, it is beneficial to reason in terms of a *portfolio* of strategies: row-level for precision and lexical match, LM-based to maximize dense semantic alignment... possibly routed by query understanding modules (topic classification / query rewriting) already discussed in the thesis. After all, for structured data extension it is harder to find true *go-to* strategies, since the most interesting alternatives perform quite similarly overall, indeed: row-level is extremely cost-efficient, with no LM calls at indexing time, while LM-based strategies introduce costs and latency proportional to corpus size, making the approach more suitable when expected dense retrieval gains justify the costs you must manage.

**Final Takeaways**

- **Naive RAG is the go-to solution**: hard to beat in generic-domain and robust if well designed
- **CoT RAG is a targeted upgrade**: useful mainly to improve grounding and handle missing hops, but it requires capable models and introduces a significant cost overhead
- **Hierarchical/Graph-based is not "plug-and-play"**: summary-based compression introduces information loss and effectiveness strongly depends on query form; query rewriting can flip performance
- **For Structured Data, granularity is fundamental**: choosing the indexing unit (row/block/table) is an additional decision that increases chunking complexity compared to the textual case; without a representation that preserves row–column relations, you cannot expect grounding on tables
- **Text-to-SQL and De-structuring are complementary**: the former dominates on numeric constraints and aggregations, the latter maximizes reusability in mixed contexts; the choice (or combination) depends, once again, mainly on expected query distribution

# 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*.