RAG Tutorial - Weizenbaum 7.3.25

Author: Dr. Mathis Börner

Welcome to this tutorial on Retrieval-Augmented Generation (RAG)! In recent years, Large Language Models (LLMs) have become increasingly powerful, but they still have limitations when it comes to accessing factual information and staying up-to-date with the latest knowledge. RAG addresses these challenges by combining the strengths of LLMs with external data retrieval, ensuring more accurate and context-rich responses.

In this tutorial, you'll learn about the fundamental concepts behind RAG, the key components of a RAG pipeline, and some advanced patterns that extend its capabilities. We will walk through everything from preparing documents and building embeddings, to retrieving relevant data, to generating answers that are grounded in the latest and most reliable sources.

Above is a high-level illustration of a typical RAG pipeline. It shows how a user's question is routed, translated (if needed), combined with relevant documents, and ultimately used to generate a fact-based answer.

Throughout this tutorial, we'll explore:

- Fundamentals – LLMs, in-context learning, what RAG is, and how it compares to other approaches like fine-tuning or long-context models.
- Components – The building blocks of a RAG system, including document preparation, indexing & retrieval, generation, query translation, routing, and query construction.
- Advanced Patterns – Extensions to the RAG framework, such as GraphRAG, AgenticRAG, and Deep Research.

By the end of this guide, you will have a solid understanding of how RAG works, the benefits it provides, and how to implement your own RAG-powered applications. Let's get started!

Large Language Models

Over the past decade, large language models (LLMs) have emerged as a transformative force in the field of artificial intelligence. Their development began with advances in machine learning techniques, particularly the introduction of the transformer architecture in 2017. This architecture revolutionized the way models process and generate human language, using self-attention mechanisms to understand the context and relationships within text. As a result, LLMs can now handle complex language tasks such as translation, summarization, and conversation with remarkable accuracy. Their impact is widespread and profound: businesses are using them to automate customer service through chatbots, content creators are using them to write articles and scripts, and researchers are using them to analyze and generate insights from vast amounts of text data. The ability of these models to understand and generate human-like text has opened up new possibilities across multiple industries, making them an indispensable tool in the digital age.

Today, many people automatically associate "LLMs" with generative models such as GPT-3/4, but the term is not always limited to such models, and early LLMs in particular were not all generative models. To better understand the different types of models and how they are used, it is worth taking a closer look at transformer architectures and the different developments that have resulted from them.

Transformer Architectures

The transformer model's unique ability to process and generate human language through (self-)attention mechanisms has made it the backbone of today's LLMs.

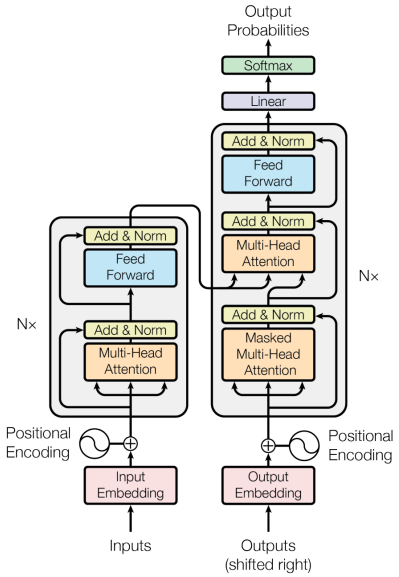


Figure 1: The Transformer - model architecture.

Figure 1: ... [Vaswani et al.'24]

The figure above is one of the most popular illustrations in all of machine learning. It is taken from the paper "Attention is All You Need" by Vaswani et al.. It introduced the Transformer architecture, which is the basis for all subsequent developments in the field. The architecture consists of two components: the *encoder* part (left half) and the *decoder* part (right half). Therefore, this architecture is referred to as an *encoder-decoder* architecture.

Encoder-Decoder Models

Encoder-decoder models, such as the original Transformer architecture, are designed for sequence-to-sequence tasks. These tasks include machine translation, where an input sequence in one language is transformed into an output sequence in another language. The encoder processes the input sequence and compresses the information into a context vector. The decoder generates the output sequence token by token based on the context vector provided by the encoder and the previously generated tokens.

Encoder-only models

Encoder-only models, such as BERT, only use the encoder part of the transformer architecture. In the encoder-decoder architecture, the encoder embeds the content of the input sequence into a single context vector. This context vector can be used directly as a contextual embedding vector (many of the text embedding models in use today are encoder-only models, often based on BERT), or as a starting point for tasks that require understanding and interpreting text, such as text classification or named entity recognition.

Decoder-only models

Decoder-only models, such as GPT (Generative Pre-trained Transformer), use only the decoder part of the Transformer architecture. The decoder generates the output token by token. In the encoder-decoder architecture, the decoder generates the output based on the context vector from the encoder and the tokens already generated. In decoder-only models, there is no context vector, so the output depends only on the tokens already generated. To further control the output, the decoder's generation is seeded with additional text, called a *prompt*, which forms the beginning of the output.

Self-supervised Learning and Scaling

One of the key features of the Transformer architecture, especially the decoder-only models, is that the training can be scaled fairly easily. Scaling architecture and training has been challenging in the past, with problems like vanishing/exploding gradients or the issue that scaling the model did not improve the results. Another big challenge when increasing the model size is that enough data is needed to train such large models. For training, labeled data is required. Here, another pivotal development took over: self-supervised learning. The basic idea is to use a training task where the labels needed in the training are trivial to obtain. For generative LLMs, this self-supervised task is predicting the next part of the text, this allows that any text can be used for

training. So, a race to scale the model size and collect more data to train the model began and continues to this day:

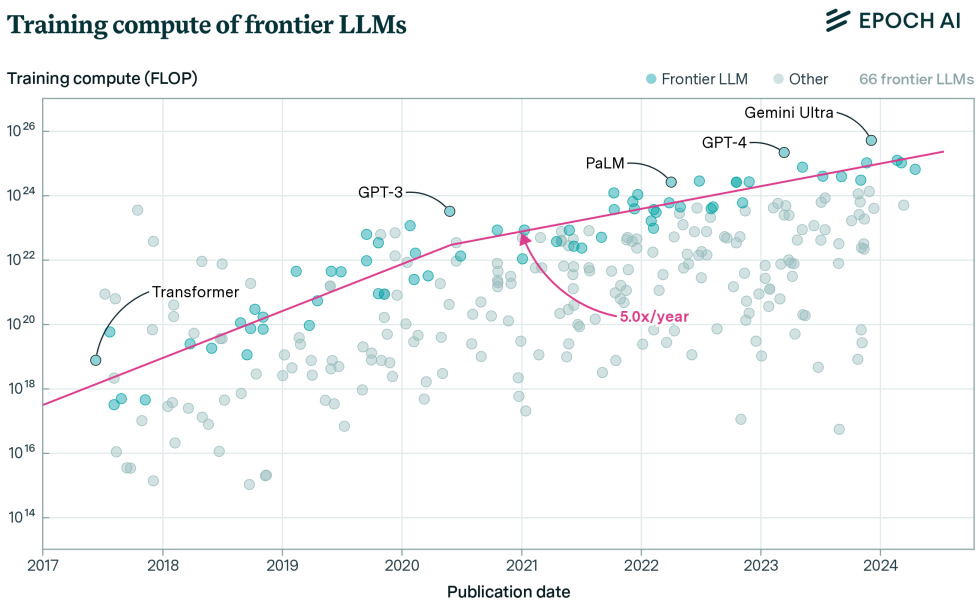


Figure 2: Training compute used for frontier models vs. time. [epoch.ai]

It is important to note that training nowaday consists of several stages. The details of training and how it relates to fine-tuning are discussed in the next section. It quickly showed that not only scaling models and training is doable, but it also showed that performance continued to improve.

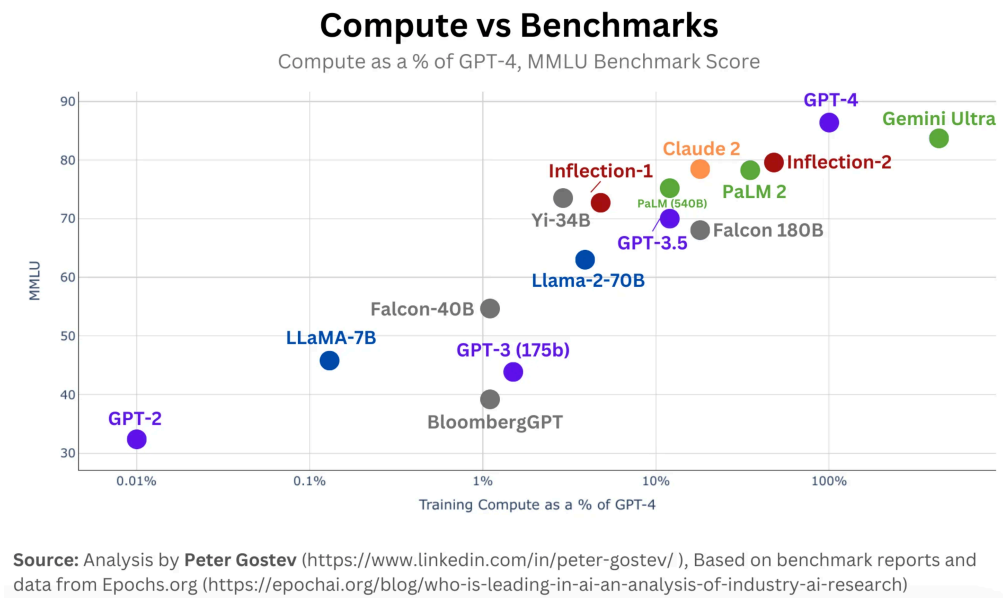


Figure 3: Training Compute vs. MMLU Performance. [epoch.ai]

Evaluating models is not straightforward, especially when the goal is to measure the general capabilities of a model. The figure above uses the Massive Multitask Language Understanding (MMLU) benchmark. It is one of the most popular benchmarks for generative LLMs, but it is important to recognize that there is no single benchmark for judging the quality of a model.

Trainings Pipeline

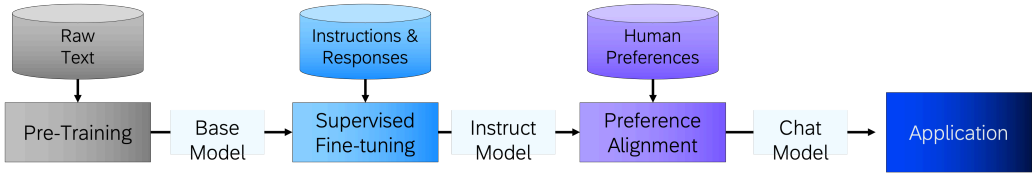


Figure 4: Canonical LLM trainings pipeline.

Pretraining

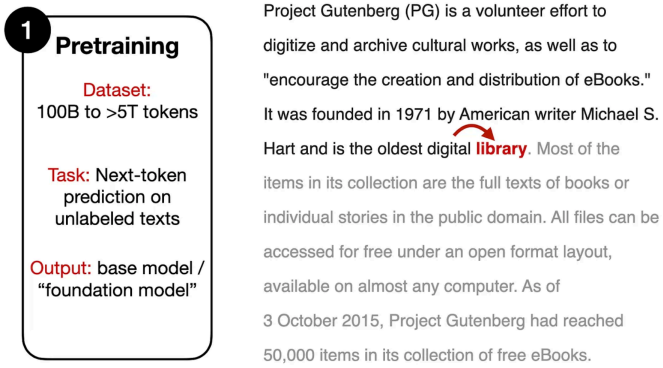


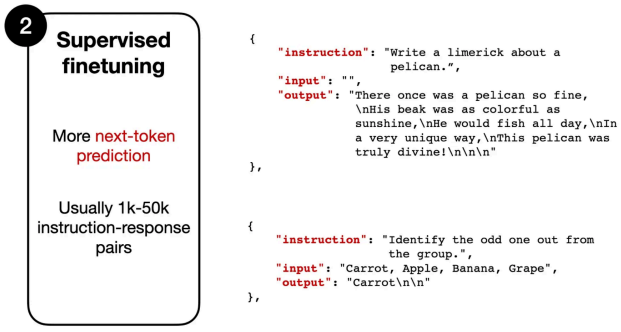
Figure 5: LLM pretraining illustrated. [S. Raschka "LLM Training: RLHF and Its Alternatives"]

During pretraining, the language model learns from an extensive and diverse collection of unlabeled text data. The model is typically trained using a self-supervised next-token prediction objective, where it predicts the following word in a sequence based on the context provided by the preceding words. This process enables the model to capture the statistical structure of the language, building a broad understanding of syntax, semantics, and world knowledge. Working with vast amounts of text allows the model to internalize language patterns without the need for manually labeled data, forming the foundational representation upon which further refinements are based.

```python
from transformers import pipeline, set_seed
generator = pipeline('text-generation', model='gpt2')
set_seed(42)
gens = generator("Hello, I'm a language model,", max_length=30,
num_return_sequences=5)
for i, g in enumerate(gens):
    print(f'Generation {i+1}: "{g["generated_text"]}"')
# Output:
# Generation 1: "Hello, I'm a language model, and my project will
get better with time, but I think there are a lot more things that
can help you"
# Generation 2: "Hello, I'm a language model, not a language model,
so if I don't have a problem, I can fix it by creating new words"
# Generation 3: "Hello, I'm a language model, and I'm trying to
learn some stuff. I'll try to do some basic programming and just
learn better ways"
# Generation 4: "Hello, I'm a language model, but I don't believe in
grammar. This will work for every language model. You can define it
very quickly"
# Generation 5: "Hello, I'm a language model, a model of how things
should be, and then we look at different things as well." I'd like
to"
```

Radford et al.'19 "Language Models are Unsupervised Multitask Learners" (GPT-2 paper)

Supervised Fine-Tuning

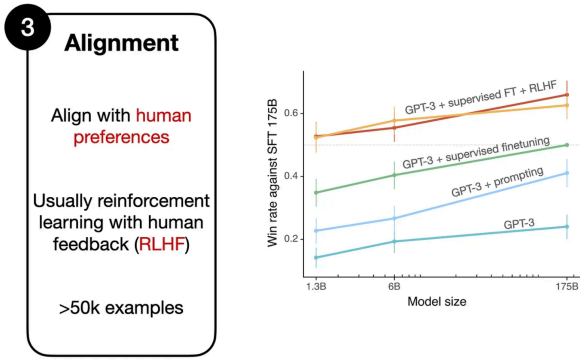


Finetuning the pretrained model on instruction data.

Figure 6: LLM supervised training illustrated. [S. Raschka "LLM Training: RLHF and Its Alternatives"]

The supervised fine-tuning phase refines the pretrained model by training it on curated instruction-output pairs, where the model learns to follow specific prompts or instructions more accurately. This step involves leveraging datasets that have been manually annotated or verified to ensure that the responses align with desired outputs for given tasks. Through fine-tuning., the model adapts to generate responses that are not only grammatically and contextually sound but also precisely tailored to meet the expectations of end users. This supervised approach significantly enhances the model's capability to produce useful and task-relevant content.

Alignment/Preference Training (RLHF)



Annotated graph from InstructGPT paper, https://arxiv.org/abs/2203.02155

Figure 7: LLM alignment training effect on human preference. [S. Raschka "LLM Training: RLHF and Its Alternatives"]

In the alignment or preference training phase, reinforcement learning with human feedback (RLHF) is employed to further calibrate the model's behavior according to human preferences, particularly in terms of helpfulness and safety. This stage builds upon the previously finetuned model by incorporating feedback from human evaluators who assess and rank the model's outputs during interactions. The feedback is then used to adjust the model's decision-making process through reinforcement learning techniques, ensuring that the final model is better aligned with the expectations and ethical considerations of its users. This step is critical for refining the model's performance in real-world

applications, ultimately leading to more reliable and contextually appropriate responses.

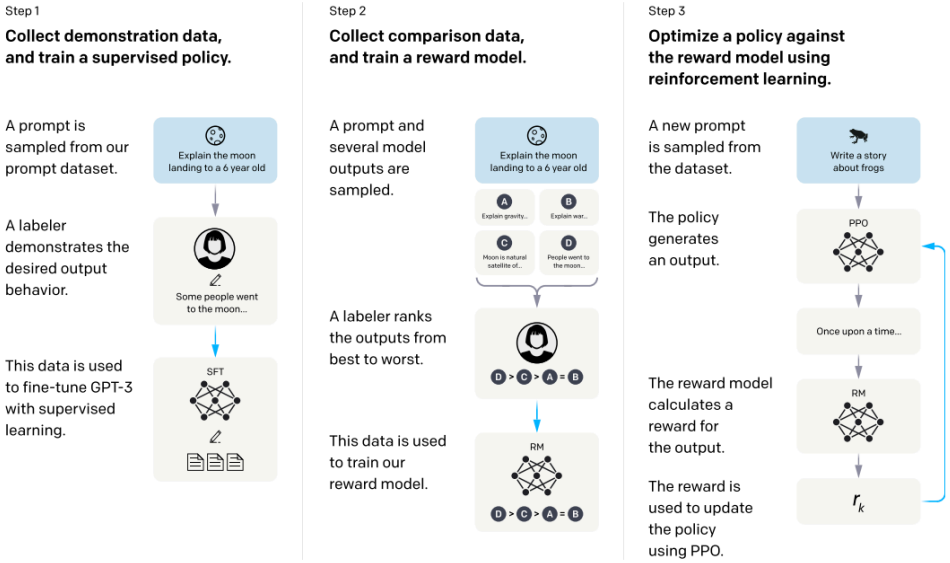


Figure 8: InstructGPT RLHF Steps. [S. Raschka "LLM Training: RLHF and Its Alternatives"]

- Key Papers & Research
  - "Neural Machine Translation by Jointly Learning to Align and Translate" (2014) by Bahdanau, Cho, and Bengio: *Introduces the attention mechanism for improving sequence modeling in recurrent neural networks (RNNs).*
  - "Attention Is All You Need" (2017) by Vaswani et al.: *Presents the transformer architecture, which uses self-attention mechanisms to handle long-range dependencies in text.*
  - "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (2018) by Devlin et al.: *Describes the BERT model, which uses bidirectional transformers for language understanding tasks.*
  - GPT Series (GPT-1, GPT-2, GPT-3, InstructGPT, GPT-4)
- "Understanding Large Language Models" by Sebastian Raschka
- "The Illustrated Transformer" by Jay Alammar: *Visual explanation of the attention mechanism and the transformer architecture.*
- "The Transformer Family Version 2.0" by Lilian Weng: *Overview of the evolution of the transformer architecture and its most popular variants*

- "[1hr Talk] Intro to Large Language Models" by Andrej Karpathy: *Introduction talk on LLMs giving a comprehension overview covering the current state of LLMs, the future and some security topics.*
- epochai.org: *Research institute investigating key trends and questions around AI and providing rich data and analyses of trends and trajectories.*

In-Context Learning (ICL): A Paradigm Shift for LLMs

In-Context Learning (ICL) is a technique that allows large language models (LLMs) to **learn and perform new tasks at inference time by providing examples in the prompt**, without any updates to the model's weights. Instead of training a model with gradient descent on new data, the user simply supplies a prompt containing a few **input-output demonstrations** of the task, followed by a new query. The model uses these examples to **condition its predictions** and generate the appropriate output for the query as if it had "learned" from the prompt context. This means the model is leveraging patterns it internalized during pre-training and **adapting on the fly** to the task described in the prompt. Crucially, this adaptation is ephemeral – once the prompt is processed, the model doesn't retain those examples or alter its parameters (hence "in-context"). In essence, ICL lets a user *program* the model's behavior through natural language examples, a major shift from the traditional train-then-deploy paradigm.

ICL remained impractical for smaller models, but it became **feasible and prominent around 2020** with the advent of very large LLMs like OpenAI's GPT-3. In mid-2020, the landmark paper Brown et al.'20 "Language Models are Few-Shot Learners" introduced GPT-3 (175 billion parameters) and highlighted ICL as an **emergent capability** of scale.

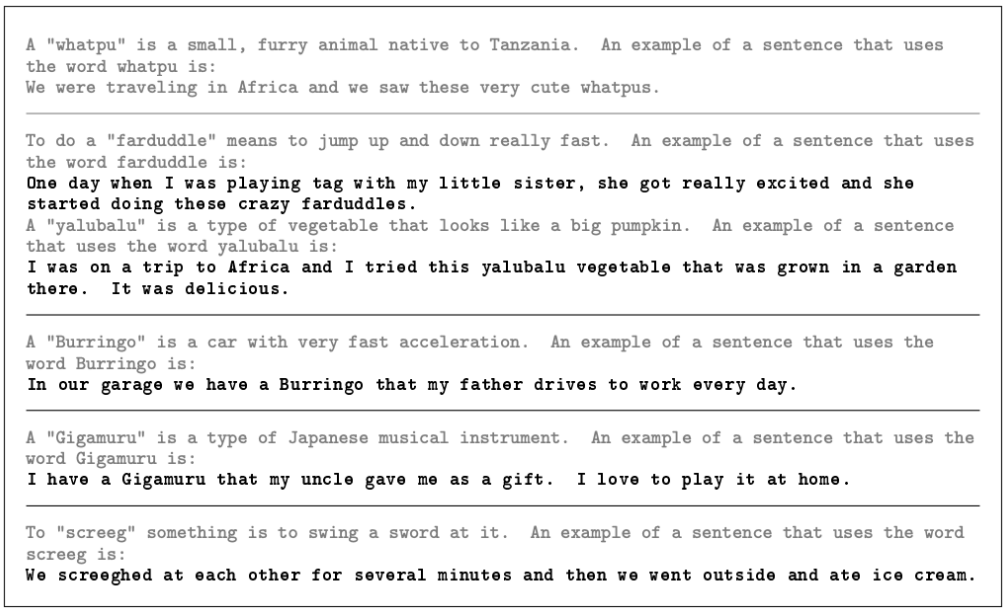


Figure 1: GPT-3 completions for the few-shot task of using a new word in a sentence.[Brown et al.'20]

GPT-3 demonstrated that a sufficiently large model could perform tasks *zero-shot* (with just an instruction) or *few-shot* (with a handful of examples) nearly as well as fine-tuned models on those tasks. Researchers observed that **larger models make increasingly efficient use of in-context information**, meaning the bigger the model, the better it gets at learning from examples given in the prompt. This was a pivotal finding – it showed that simply by scaling up model size and training on vast data, an LLM could *generalize* to tasks it had never explicitly been trained on, provided those tasks were framed appropriately in its input. Thus, around 2020–2021, ICL moved from a curiosity to a practical technique, as companies and researchers realized that *prompting* could unlock many capabilities of these massive models without additional training.

A Paradigm Shift for LLMs

The emergence of in-context learning has been **pivotal in enabling new types of applications** with LLMs. By allowing models to be quickly repurposed for different tasks through prompting, ICL offers several key benefits:

- **Task Flexibility without Fine-Tuning:** Developers can get a single LLM to perform a wide range of tasks (translation, summarization, Q&A, coding, etc.) by just changing the prompt examples, instead of training a new model for each task. This dramatically speeds up prototyping and deployment of NLP applications.
- **Rapid Adaptation to New Instructions:** ICL means an LLM can adapt its behavior on the fly. For example, given a few examples of sentiment-labeled sentences in the prompt, the model can immediately classify sentiment for a new sentence. If you then give it examples of a different task (like math problems), it can switch to that task – all in one model, *without retraining*.
- **Leveraging Pre-trained Knowledge:** Because the model retains its broad knowledge from pre-training, ICL taps into that foundation. The prompt's examples help the model *retrieve* the relevant concepts or patterns from its latent knowledge. In practical terms, this means an LLM can handle niche tasks or domains as long as you illustrate the task with a few well-chosen examples and instructions.
- **Fewer Labeled Examples Needed:** Unlike traditional supervised learning which might require thousands of labeled examples and a training process, ICL works with just a handful of examples provided at inference time. This lowers the barrier to experimenting with AI on new problems – you often only need to **write down a couple of demonstrations** of the desired input-output behavior.

Overall, ICL has transformed how we build with LLMs. It blurs the line between using a model and teaching it: users can **steer the model's output by crafting a good prompt**, a practice now known as *prompt engineering*. In fact, many advanced prompting techniques (like *chain-of-thought prompting*, where the model is guided to reason step-by-step) build on the foundation of in-context learning. The rise of ICL is a major reason we see LLMs being applied in creative ways – from writing code based on a few examples, to conversational agents that adjust style/tone based on example dialogues, to domain-specific assistants powered by a general model with appropriate prompting.

Example: Few-Shot In-Context Learning in Practice

To make the concept of ICL more concrete, let's walk through a simple example. Suppose we want an LLM to perform sentiment analysis on product reviews. We don't train a special sentiment classifier; instead, we will **provide a few examples of reviews with their sentiment labels in the prompt** (few-shot prompting), and then ask the model to label a new review.

For instance, consider the following prompt constructed with two demo examples (one positive, one negative) and then a new review:

```
prompt = """Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.

Poor English input: Thank you for picking me as your designer. I'd
appreciate it.
Good English output: Thank you for choosing me as your designer. I
appreciate it.

Poor English input: The mentioned changes have done. or I did the
alteration that you requested. or I changed things you wanted and
did the modifications.
Good English output: The requested changes have been made. or I made
the alteration that you requested. or I changed things you wanted
and made the modifications.

Poor English input: {input}
Good English output:"""
```

When this prompt is fed to a larger language model, the model should infer from the examples that the task is correct sentencs. It should then produce a continuation of the prompt, for example:

```
[Poor English input: I'd be more than happy to work with you in
another project.]
Good English output: I'd be more than happy to work with you on
another project.
```

Running this using GPT-2 wiht a bunch of examples:

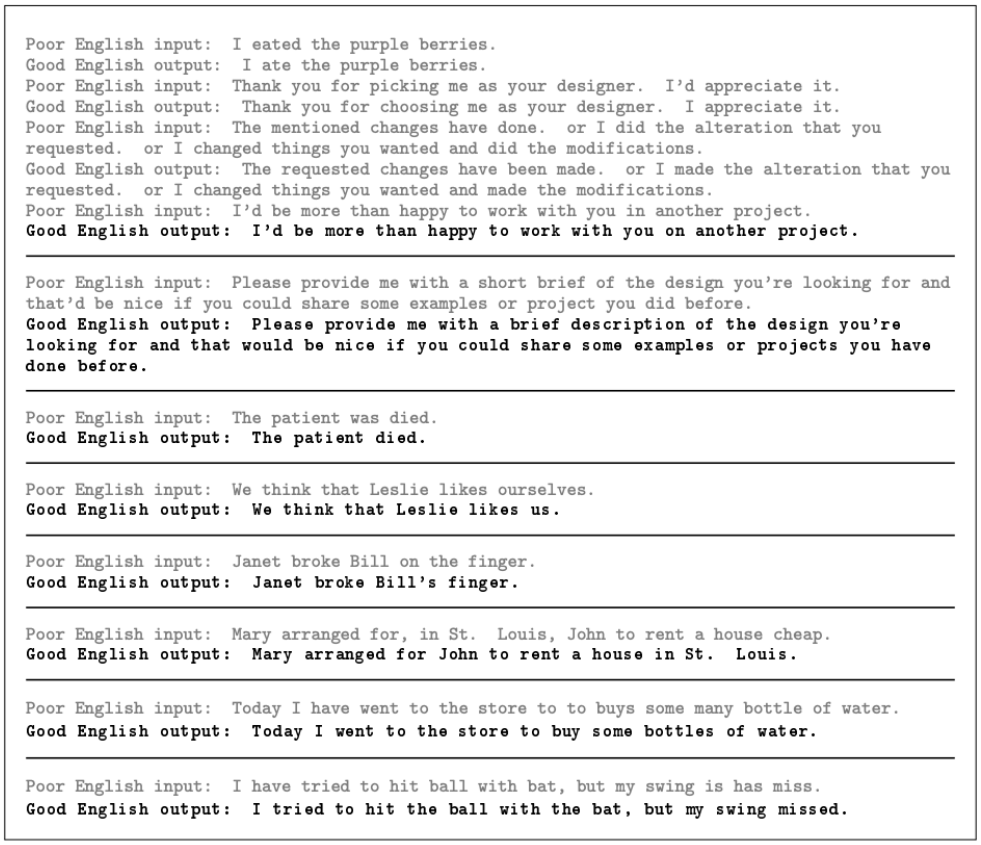
```
examples = [
    ("I'd be more than happy to work with you in another project.",
"I'd be more than happy to work with you on another project."),
    ("She don't like apples.", "She doesn't like apples."),
    ("He do his homework everyday.", "He does his homework every
day."),
    ("Me and him went to the store.", "He and I went to the
store."),
    ("I have saw that movie last week.", "I saw that movie last
week."),
    ("They was going to the party.", "They were going to the
party."),
    ("I ain't got no money.", "I don't have any money."),
    ("Her like cooking and dancing.", "She likes cooking and
dancing."),
    ("We was excited about the trip.", "We were excited about the
trip."),
    ("This is the most quickest route.", "This is the quickest
route."),
    ("I can't hardly wait for the event.", "I can hardly wait for
the event.")
]

from transformers import pipeline, set_seed
generator = pipeline('text-generation', model='gpt2')
set_seed(42)
gens = generator(prompt, max_length=256,
num_return_sequences=n_completions, )

for i, (in_, out_) in enumerate(examples):
    gen = generator(prompt.format(input=in_), max_length=256,
num_return_sequences=1, )[0]
    lines = gen["generated_text"].split("\n")
    print(f'Example {i+1}:')
    print('\n'.join([f"> {lines[i]}" for i in range(len(lines)) if
in_ in lines[i] or in_ in lines[i-1]]))
# Output:
# Example 1:
# > Poor English input: I'd be more than happy to work with you in
another project.
# > Good English output: The requested changes have been made. or I
might become involved in your project. or I might become associated
with you in your next project.
# Example 2:
# > Poor English input: She don't like apples.
# > Good English output: I'll just leave out her. she like apples.
```

```
# Example 3:
# > Poor English input: He do his homework everyday.
# > Good English output: He do his homework everyday.
# ...
```

shows that *small* LLMs fail with this task. GPT-2 seems not to understand the task it is supposed to do.



Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.
Poor English input: Thank you for picking me as your designer. I'd appreciate it.
Good English output: Thank you for choosing me as your designer. I appreciate it.
Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
Good English output: The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.
Poor English input: I'd be more than happy to work with you in another project.
Good English output: I'd be more than happy to work with you on another project.

Poor English input: Please provide me with a short brief of the design you're looking for and that'd be nice if you could share some examples or project you did before.
Good English output: Please provide me with a brief description of the design you're looking for and that would be nice if you could share some examples or projects you have done before.

Poor English input: The patient was died.
Good English output: The patient died.

Poor English input: We think that Leslie likes ourselves.
Good English output: We think that Leslie likes us.

Poor English input: Janet broke Bill on the finger.
Good English output: Janet broke Bill's finger.

Poor English input: Mary arranged for, in St. Louis, John to rent a house cheap.
Good English output: Mary arranged for John to rent a house in St. Louis.

Poor English input: Today I have went to the store to to buys some many bottle of water.
Good English output: Today I went to the store to buy some bottles of water.

Poor English input: I have tried to hit ball with bat, but my swing is has miss.
Good English output: I tried to hit the ball with the bat, but my swing missed.

Figure 2: Continous GPT-3 completions for the few-shot task of correcting English grammar.[Brown et al.'20]

In the example above show GPT-3's capability to . **Why did this work?** Because the model recognized the pattern from the prompt: it was the task by example and was given a schema to follow. This is in-context learning in action – the model **learned the task from the context** without any explicit training step. We could easily swap in different examples (or even a different task description) to get the model to do something else, all by editing the prompt.

In practice, the effectiveness of ICL improves with model size and quality. As shown, smaller models like GPT-2 may not reliably follow the pattern with just a couple of examples, but larger models (GPT-3, GPT-4, etc.) excel at this, often matching or surpassing task-specific models on many benchmarks. The example above is simple, but more complex uses of ICL can include providing few-shot examples for translation, grammar correction, question answering, and more – even complex reasoning tasks can be guided via ICL combined with advanced prompting strategies.

RAG

In-context learning leverages information *provided in the prompt* and the model's pre-trained knowledge to perform tasks. However, there are scenarios where the model's internal knowledge might not be enough – for example, when asked about very recent events or obscure facts that weren't in its training data. The prompt also has a limited length, which restricts how much context or reference we can pack into it. This is where **Retrieval-Augmented Generation (RAG)** comes into play. RAG is a technique that combines ICL with information retrieval: first relevant external data (from a database or knowledge base) is retrieved and then this data is used as additional context for generation. In other words, RAG **augments the prompt with retrieved knowledge** so that the LLM can produce more accurate and up-to-date responses. This approach helps address some limitations of pure ICL (like knowledge cut-off and hallucinations) by grounding the model's output in real documents.

The term "RAG" - Retrieval Augmented Generation was coined by Patrick Lewis, lead author of the 2020 paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". Lewis et al. used the followin architecture:

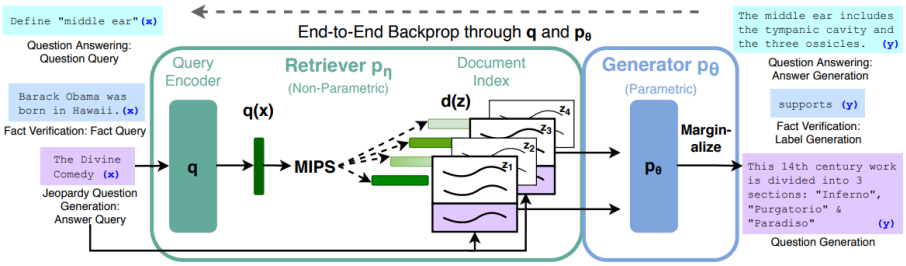


Figure 1: Lewis et al. combined a pre-trained retriever (Query Encoder + Document Index) with a pre-trained seq2seq model (Generator) and fine-tune end-to-end.[Lewis et al.'20]

The paper introduced a hybrid approach that integrated a pre-trained sequence-to-sequence generator with a dense retrieval system. It treated retrieved documents as latent variables, marginalizing over them at either the sequence or token level. This strategy allowed the model to incorporate external knowledge dynamically while reducing hallucinations and enabling

easy updates to its knowledge base. The system achieved state-of-the-art resulöts on open-domain QA and Jepoardy Question generation task.

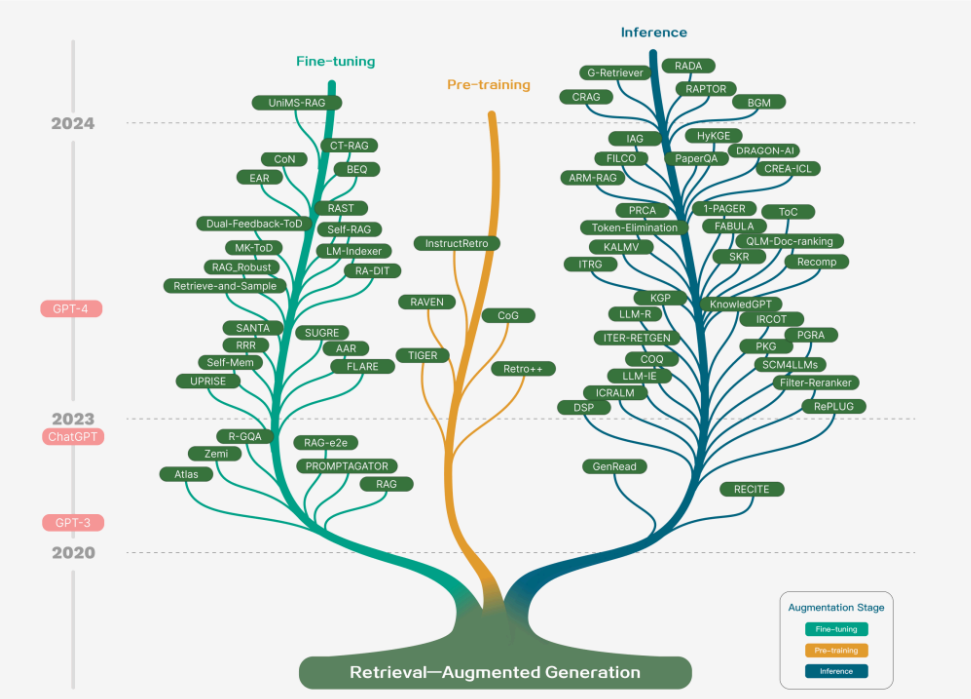


Figure 2: Technology tree of RAG research. The stages of involving RAG mainly include pre-training, fine-tuning, and inference. With the emergence of LLMs, research on RAG initially focused on leveraging the powerful in context learning abilities of LLMs, primarily concentrating on the inference stage. Subsequent research has delved deeper, gradually integrating more with the fine-tuning of LLMs. [Gao et al.'23]

The technology tree in the survey illustrates how RAG techniques have evolved alongside large language models across three main stages—pre-training, fine-tuning, and inference. The “pre-training” trunk represents early methods that incorporate retrieval directly into large language model training, thereby grounding the model in external knowledge from the outset. The “fine-tuning” branch highlights approaches where retrieval is introduced or refined after an initial large model has been trained, tailoring it to specific tasks or domains by merging parametric knowledge with retrieved context. Finally, the “inference” branch shows systems that rely on retrieval dynamically at runtime—these methods augment model outputs on-the-fly by pulling in relevant documents or facts just before generation. Together, these three branches underscore the

versatility of RAG: it can be embedded in the training pipeline itself, layered on top of an existing model, or flexibly activated whenever new information is needed for more accurate and transparent responses. Nowadays most popular techniques in practice are techniques from the inference branch. These techniques utilize a RAG system engineered around the LLM.

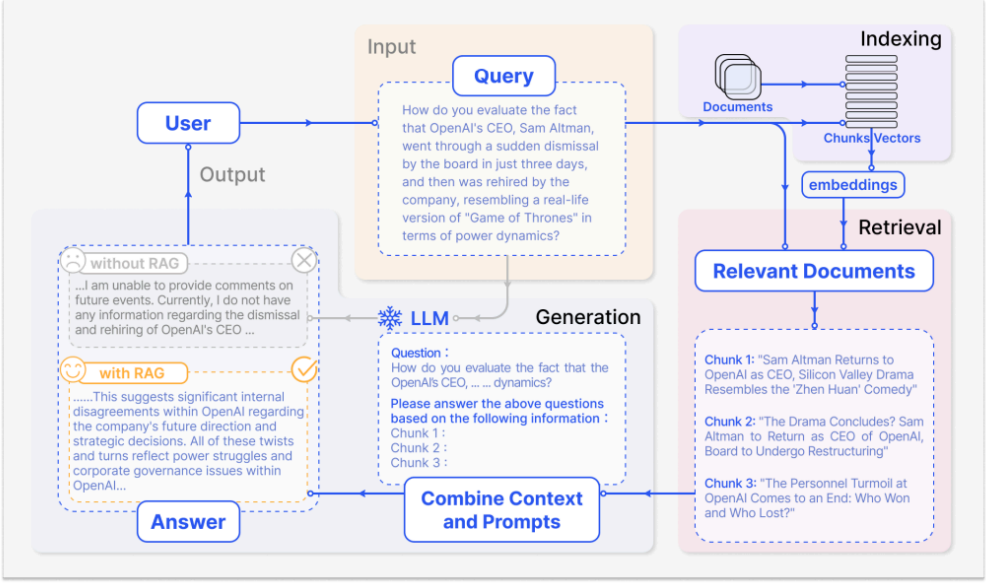


Figure 3: A representative instance of the RAG process applied to question answering. [Gao et al.'23]

A prototypical basic RAG pipeline is show in Fig.3 In this RAG pipeline, you start by indexing your documents: they're broken into smaller chunks (e.g., paragraphs), turned into vector embeddings, and stored in a searchable database. When a user's question (the "query") comes in, it's also converted into an embedding and matched against the stored chunks. The top-matching chunks—those most likely to be relevant—are retrieved and fed, alongside the user's original query, into the large language model. The model then "reads" this augmented prompt and generates an answer, drawing on both its internal knowledge and the retrieved information. This simple "retrieve-then-read" structure helps the model produce answers grounded in the latest or domain-specific data.

RAG extends beyond document retrieval to encompass diverse knowledge sources including real-time internet searches, databases, and Wikipedia. Any knowledge repository that *supports natural language queries* can serve as a

retrieval source for RAG systems. Regardless of the information source, the fundamental objective remains consistent: **to acquire context that maximizes relevance while minimizing redundancy**.

From Counting to Context: The Evolution of Embedding Models

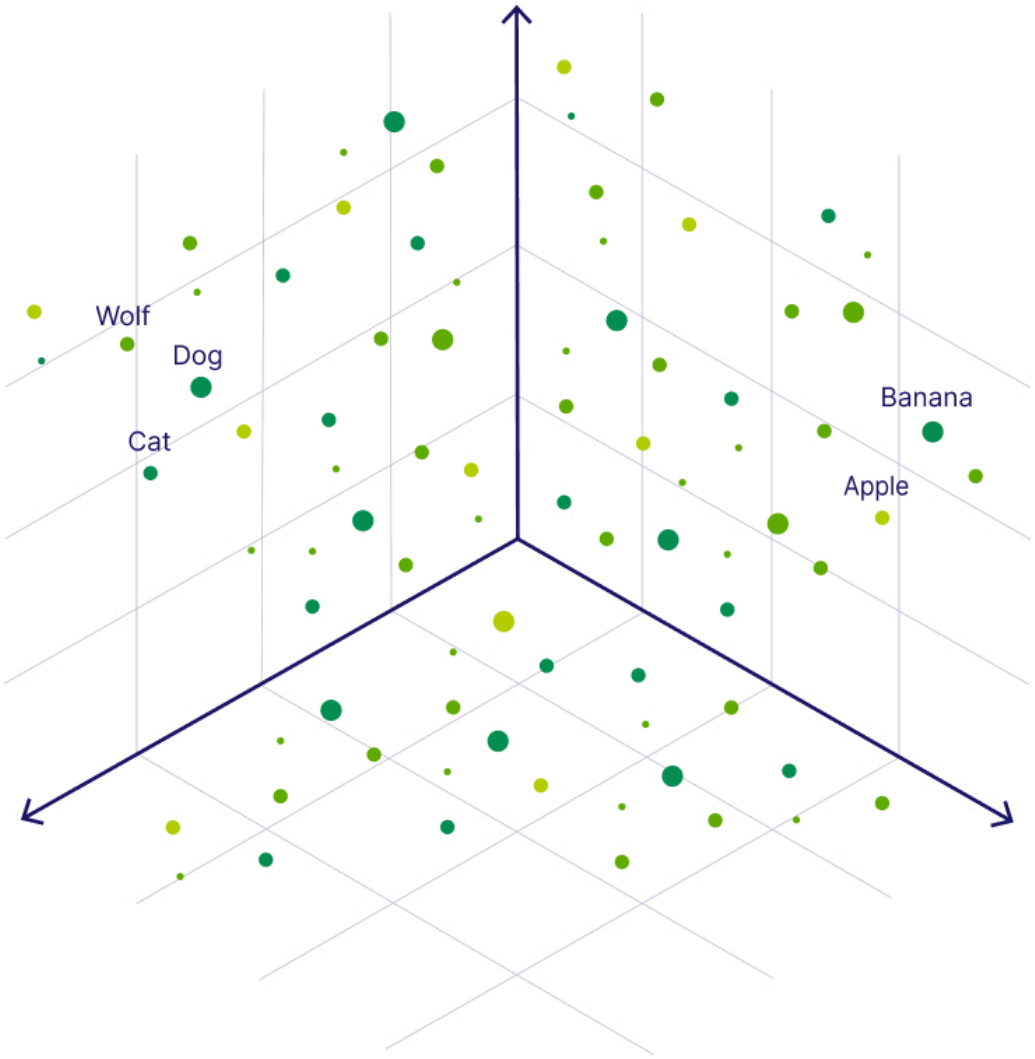


Figure 1: Embedding Space

Beginning with early count-based representations, tracing the progression to predictive neural models like Word2Vec and GloVe, and leads finally to today's context-aware architectures. This evolution—from simplistic one-hot vectors to dynamic, context-sensitive embeddings—has not only redefined how language is represented but also driven innovations across various AI applications.

The representation of language in a form that computers can process has long been a challenge. In the early days of natural language processing (NLP), words were treated as discrete symbols, isolated in high-dimensional spaces. However, as understanding deepened, researchers realized that capturing the nuances of meaning required moving beyond these rigid representations. This historical progression—from static, count-based models to the sophisticated, context-driven embeddings of today—provides essential insights into both the challenges and breakthroughs that have shaped modern AI.

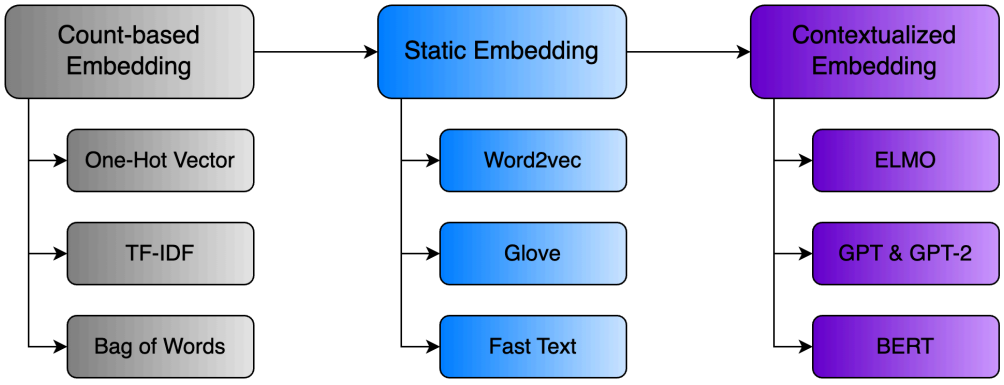


Figure 2: Overview of the evolution of text embedding models.

Early Representations of Language

In the initial phase of NLP, words were represented using simple one-hot encodings—a method where each word corresponds to a unique, sparse vector. While this approach was straightforward, it lacked the ability to capture semantic relationships between words. To address this, the bag-of-words model was introduced. This method counted word frequencies, creating high-dimensional vector spaces that, despite their computational challenges, began to hint at underlying linguistic patterns.

Building on these ideas, the concept of distributional semantics emerged with the realization that "a word is characterized by the company it keeps." Techniques like Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA) used word co-occurrence statistics to uncover latent structures in language, setting the stage for future innovations.

One-Hot Encoding

One-hot encoding is the most straightforward method for representing words, where each word is mapped to a binary vector with a single `1` in the position corresponding to its index in the vocabulary and `0` in all other positions. This approach offers a clear and interpretable representation of words, making it simple to implement and understand. However, its main drawbacks are the resulting high dimensionality and the fact that it treats every word as completely independent, failing to capture any semantic similarity or contextual relationships between different words.

```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

# Sample data
corpus = ['dog', 'cat', 'dog', 'fish']

# Reshape data to fit the model
corpus = np.array(corpus).reshape(-1, 1)

# One-hot encode the data
onehot_encoder = OneHotEncoder(sparse=False)
onehot_encoded = onehot_encoder.fit_transform(corpus)

print(onehot_encoded)
# Output:
# > [[0. 1. 0.]
# >  [1. 0. 0.]
# >  [0. 1. 0.]
# >  [0. 0. 1.]]
```

Bag of Words (BoW)

The Bag of Words model takes a step forward by representing text documents as collections of word counts, rather than isolated symbols. In this approach, each document is transformed into a vector that reflects the frequency of each word from the vocabulary appearing in the text. While BoW can capture the importance of words based on their occurrence, it does so at the expense of word order and syntactic structure, which means that the nuanced meaning arising from word combinations is lost.

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

print(X.toarray())
# Output:
# > [[0 1 1 1 0 0 1 0 1]
# >  [0 2 0 1 0 1 1 0 1]
# >  [1 0 0 1 1 0 1 1 1]
# >  [0 1 1 1 0 0 1 0 1]]
print(vectorizer.get_feature_names_out())
# Output:
# > ['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third'
'this']
```

TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical technique that evaluates the importance of a word within a document relative to its occurrence across a larger corpus. By multiplying the term frequency (how often a word appears in a document) with the inverse document frequency (which diminishes the weight of common words across all documents), TF-IDF emphasizes words that are more unique and relevant to the specific text. This method strikes a balance between highlighting frequently occurring terms and down-weighting ubiquitous words, making it an effective tool for tasks such as information retrieval and document classification, even though it does not capture the deeper contextual or semantic nuances that more advanced embedding methods provide.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample data
corpus = [
    'This is the first document.',
    'This document is the second document.',
    'And this is the third one.',
    'Is this the first document?',
]

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

print(X.toarray())
# Output:
# > [[0.         0.46979139 0.58028582 0.38408524 0.         0.
0.38408524 0.         0.38408524]
# >  [0.         0.6876236  0.         0.28108867 0.
0.53864762 0.28108867 0.         0.28108867]
# >  [0.51184851 0.         0.         0.26710379 0.51184851 0.
0.26710379 0.51184851 0.26710379]
# >  [0.         0.46979139 0.58028582 0.38408524 0.         0.
0.38408524 0.         0.38408524]]
print(vectorizer.get_feature_names_out())
# Output:
# > ['and' 'document' 'first' 'is' 'one' 'second' 'the' 'third'
'this']
```

The Neural Revolution in Language Modeling

The early 2000s witnessed a paradigm shift with the introduction of neural probabilistic language models. Pioneered by researchers such as Bengio and colleagues, these models replaced rigid count-based methods with dynamic neural networks that learned distributed representations for words. This shift culminated in the development of word embeddings—dense vector representations that encapsulated richer semantic relationships.

The breakthrough moment came with the advent of Word2Vec in 2013. Utilizing techniques such as the Continuous Bag-of-Words (CBOW) and skip-gram models, Word2Vec demonstrated that embeddings could be learned efficiently

by predicting contextual words. Soon after, GloVe further refined these ideas by integrating global co-occurrence statistics, providing yet another perspective on capturing word meaning.

Proposed Image: An infographic showing the architectures of Word2Vec and GloVe, with visualizations of how words are mapped into a dense vector space. This image could include a simplified diagram of the neural network components that drive these models.

Word2Vec

Word2Vec revolutionized text representation by introducing neural networks to learn dense, continuous vector representations for words. By employing architectures like Continuous Bag-of-Words (CBOW) and skip-gram, Word2Vec captures the context in which words appear, effectively mapping semantically similar words to nearby points in the vector space. This model marked a significant departure from sparse, high-dimensional representations and has paved the way for more sophisticated embedding techniques by focusing on learning from the context in which words occur, rather than relying solely on their frequency.

```python
from gensim.models import Word2Vec

# Sample data
sentences = [
    ['this', 'is', 'the', 'first', 'document'],
    ['this', 'document', 'is', 'the', 'second', 'document'],
    ['and', 'this', 'is', 'the', 'third', 'one'],
    ['is', 'this', 'the', 'first', 'document']
]

# Initialize the Word2Vec model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1,
workers=4)

# Train the model
model.train(sentences, total_examples=len(sentences), epochs=10)

# Get vector for a word
print(model.wv['document'])
# Output:
# > [-5.3622725e-04  2.3643136e-04  5.1033497e-03   ...  6.3925339e-
03]
```

Glove

GloVe, or Global Vectors for Word Representation, builds on the idea of context by combining local context information with global statistical data derived from word co-occurrence matrices. Instead of solely relying on a neural network's predictive power like Word2Vec, GloVe leverages ratios of word co-occurrence probabilities to directly capture the semantic relationships between words. This method effectively balances both local and global information, providing word embeddings that not only reflect direct contextual similarities but also encode broader corpus-wide patterns of word usage.

```
import gensim.downloader as api

# Download pre-trained GloVe model (choose the size you need - 50,
100, 200, or 300 dimensions)
glove_vectors = api.load("glove-wiki-gigaword-100")  # Example: 100-
dimensional GloVe

# Get word vectors (embeddings)
word1 = "king"
word2 = "queen"
vector1 = glove_vectors[word1]
vector2 = glove_vectors[word2]

# Compute cosine similarity between the two word vectors
similarity = glove_vectors.similarity(word1, word2)

print(f"Word vectors for '{word1}': {vector1}")
# Output:
# > Word vectors for 'king': [-0.32307  -0.87616   0.21977  ...
-0.98878 ]
print(f"Word vectors for '{word2}': {vector2}")
# Output:
# > Word vectors for 'queen': [-0.50045  -0.70826   0.55388  ...
-0.56075 ]
print(f"Cosine similarity between '{word1}' and '{word2}':
{similarity}")
# Output:
# > Cosine similarity between 'king' and 'queen': 0.7507690191268921
```

Fast-Text

FastText extends the capabilities of traditional word embedding models by incorporating subword information, treating each word as a bag of character n-grams. This approach allows the model to generate meaningful representations even for words that are rare or entirely absent from the training data by constructing embeddings from the smaller subword units. As a result, FastText excels in handling morphological variations and out-of-vocabulary words, making it particularly valuable for languages with rich morphology or for applications where the vocabulary is constantly evolving.

```
from gensim.models import FastText

# Sample data
sentences = [
    ['this', 'is', 'the', 'first', 'document'],
    ['this', 'document', 'is', 'the', 'second', 'document'],
    ['and', 'this', 'is', 'the', 'third', 'one'],
    ['is', 'this', 'the', 'first', 'document']
]

# Initialize the FastText model
model = FastText(sentences, vector_size=100, window=5, min_count=1,
workers=4)

# Train the model
model.train(sentences, total_examples=len(sentences), epochs=10)

# Get vector for a word
print(model.wv['document'])
# Output
# > [ 7.28658633e-04  4.16975323e-04 -9.04823013e-04  ...
4.49019681e-05]
```

The Contextual Turn

Despite their success, static embeddings such as those produced by Word2Vec and GloVe had inherent limitations. Each word was assigned a single vector, regardless of its varying meanings in different contexts—a significant drawback for polysemous words. This challenge paved the way for the development of contextualized word embeddings.

Models like ELMo introduced deep bidirectional LSTMs that dynamically generated embeddings based on the entire sentence context, allowing the same word to acquire different representations depending on its usage. Building on this concept, Transformer-based architectures, notably BERT, replaced LSTMs with attention mechanisms. This innovation not only enhanced parallel processing capabilities but also improved the overall understanding of context, overcoming many of the limitations of earlier models.

Elmo

ELMo (Embeddings from Language Models) marks a significant shift toward context-dependent word representations by generating embeddings based on the entire sentence context. Using deep bidirectional LSTMs, ELMo produces dynamic representations that capture the complex syntactic and semantic nuances of words, allowing the same word to have different embeddings depending on its surrounding context. This contextual approach overcomes one of the major limitations of static embeddings, making ELMo particularly powerful for tasks where word sense disambiguation and nuanced understanding are crucial.

```
import tensorflow as tf
import tensorflow_hub as hub

# Load pre-trained ELMo model from TensorFlow Hub
elmo = hub.load("https://tfhub.dev/google/elmo/3")

# Sample data
sentences = ["This is the first document.", "This document is the
second document."]

def elmo_vectors(sentences):
    embeddings = elmo.signatures['default'](tf.constant(sentences))
['elmo']
    return embeddings

# Get ELMo embeddings
elmo_embeddings = elmo_vectors(sentences)
print(elmo_embeddings)
# Output:
# > tf.Tensor([[[-0.51420665 -0.46363437  0.06954873 ...
-0.10347158]]], shape=(2, 6, 1024), dtype=float32)
```

BERT

BERT (Bidirectional Encoder Representations from Transformers) further advances the field by leveraging the Transformer architecture to produce deeply contextualized embeddings. Unlike previous models that processed text in a single direction, BERT's bidirectional attention mechanism enables it to

consider both the left and right contexts simultaneously when encoding each token. Pre-trained on vast amounts of text using techniques like masked language modeling and next sentence prediction, BERT's embeddings capture rich linguistic information that can be fine-tuned for a wide array of NLP tasks, resulting in state-of-the-art performance in areas like question answering and sentiment analysis.

```
from transformers import BertTokenizer, BertModel
import torch

# Load pre-trained BERT model and tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')

# Sample data
sentence = "This is the first document."

# Tokenize input
inputs = tokenizer(sentence, return_tensors='pt')

# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
# Output:
# > tensor([[[-0.1793,  0.2468,  0.2588,  ...,  0.4188]]])
```

GPT

GPT (Generative Pre-trained Transformer) takes a slightly different approach by using a unidirectional transformer model focused on generating text. Although GPT processes tokens from left to right, its generative nature allows it to create coherent and contextually appropriate sequences of text. The embeddings produced by GPT are tailored for tasks that involve language generation, such as text completion and dialogue systems. Despite being unidirectional, GPT's pre-training on large-scale corpora enables it to capture significant contextual patterns, albeit with a different emphasis compared to bidirectional models like BERT.

```python
from transformers import GPT2Tokenizer, GPT2Model
import torch

# Load pre-trained GPT-2 model and tokenizer
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2Model.from_pretrained('gpt2')

# Sample data
sentence = "This is the first document."

# Tokenize input
inputs = tokenizer(sentence, return_tensors='pt')

# Get embeddings
with torch.no_grad():
    outputs = model(**inputs)
    embeddings = outputs.last_hidden_state

print(embeddings)
# Output:
# > tensor([[[ 0.0530, -0.0137, -0.2393,  ..., -0.0594]]])
```

From Token to Text Embeddings:

Converting token-level embeddings into a single representation for an entire sentence or document is a crucial step in many NLP applications. This process involves aggregation techniques such as pooling (either mean or max pooling) or simply using the special classification token (CLS) available in some models. Each method comes with trade-offs: pooling can smooth over the variability in token importance, while relying on a dedicated token like CLS might not capture all nuances present in the text. The choice of strategy depends on the specific task at hand and the desired balance between preserving detailed token-level information and achieving a compact representation that captures the overall meaning.

```
def max_pooling(model_output, attention_mask):
    token_embeddings = model_output.last_hidden_state
    attention_mask =
attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
    token_embeddings[attention_mask == 0] = -1e9  # Mask padding
    return torch.max(token_embeddings, 1)[0]

def mean_poooling(model_output, attention_mask):
    token_embeddings = model_output.last_hidden_state
    attention_mask =
attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
    sum_embeddings = torch.sum(token_embeddings * attention_mask, 1)
    sum_mask = torch.clamp(attention_mask.sum(1), min=1e-9)
    return sum_embeddings / sum_mask

# For BERT models, the CLS token is the first token in the sequence
def cls_token(model_output, *args):
    return model_output.last_hidden_state[:, 0, :]
```

Modern Embedding Models and Their Applications

Today's embedding models are predominantly Transformer-based, and they continue to push the boundaries of what is possible in language representation. The practical applications of these embedding models are vast. Embedding models transform text or other modalities into high-dimensional vectors where semantically similar texts are placed closer together in the vector space. This transformation is key for retrieval because it allows the system to:

- **Represent Semantics:** Capture the meaning behind sentences rather than just matching keywords.
- **Efficiently Compute Similarity:** Quickly compute the similarity between a query and a large corpus of documents using mathematical operations.

Popular embedding models include those from the Sentence Transformers library (like `all-MiniLM-L6-v2`), BERT, and others. These models output dense vector representations that encapsulate the contextual meaning of the input

text. The most popular leaderboard for embedding models is Massive Text Embedding Benchmark (MTEB).

Understanding Similarity in the Context of Embeddings

Similarity in embedding spaces refers to how close or related two text representations are. The closer the vectors, the more semantically similar the texts. This measure is crucial in RAG because when a user poses a query, the system needs to identify documents whose embeddings are similar to that of the query. Once these similar documents are retrieved, they serve as additional context for generating an answer.

Cosine Similarity: A Closer Look

One of the most popular metrics for measuring similarity between two vectors is **cosine similarity**. It calculates the cosine of the angle between two non-zero vectors:

$$\text{cosine\_similarity}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}$$

Cosine similarity has a few nice properties:

- **Scale Invariance:** It is independent of the vector magnitude, meaning that only the orientation (or direction) of the vectors matters.
- **Efficient:** Cosine similarity can be computed quickly even for high-dimensional vectors.
- **Common Use:** It is widely adopted in many information retrieval and NLP tasks due to its effectiveness in capturing semantic similarity.

Let's walk through some Python examples to see how cosine similarity works and how embedding models can be applied in a RAG-like scenario. Below is a simple function using NumPy to calculate the cosine similarity between two vectors:

```python
import numpy as np

def cosine_similarity(vec1, vec2):
    """Calculate cosine similarity between two vectors."""
    dot_product = np.dot(vec1, vec2)
    norm_a = np.linalg.norm(vec1)
    norm_b = np.linalg.norm(vec2)
    return dot_product / (norm_a * norm_b)

# Example vectors
vec_a = np.array([1, 2, 3])
vec_b = np.array([4, 5, 6])

similarity = cosine_similarity(vec_a, vec_b)
print("Cosine Similarity:", similarity)
# Output
# > Cosine Similarity: 0.9746318461970762
```

This code computes the cosine similarity between two simple vectors, illustrating the core mathematical idea behind comparing text embeddings. Now, let's see how to use an embedding model to compute the similarity between sentences:

```
from sentence_transformers import SentenceTransformer
import numpy as np

# Load a pre-trained sentence transformer model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Define some example sentences
sentences = [
    "The cat sat on the mat.",
    "A feline rested on a rug.",
    "The weather is sunny today."
]

# Generate embeddings for the sentences
embeddings = model.encode(sentences)

# Calculate cosine similarity between each pair of sentences
def compute_similarity_matrix(embeddings):
    num_sentences = len(embeddings)
    similarity_matrix = np.zeros((num_sentences, num_sentences))
    for i in range(num_sentences):
        for j in range(num_sentences):
            similarity_matrix[i][j] =
cosine_similarity(embeddings[i], embeddings[j])
    return similarity_matrix

similarity_matrix = compute_similarity_matrix(embeddings)
print("Cosine Similarity Matrix:\n", similarity_matrix)
# Output:
# > Cosine Similarity Matrix:
# > [[ 1.00000000e+00  5.52950740e-01 -1.47880986e-04]
# > [ 5.52950740e-01  1.00000012e+00  9.15080011e-02]
# > [-1.47880986e-04  9.15080011e-02  1.00000000e+00]]
```

In this example:

- We load a pre-trained embedding model.
- The model transforms each sentence into a vector.
- We compute a similarity matrix, where each element (i, j) represents the cosine similarity between the i-th and j-th sentence.

- "An intuitive introduction to text embeddings"

Long-Context LLMs vs. RAG

In recent years, the rise of long context models has generated significant excitement across the AI community. Pioneered by models such as Gemini—with its ability to handle contexts of up to 1 million tokens—these architectures challenge traditional limitations. Originally, transformer networks employed short context windows, typically around 500 tokens, later expanded in models like GPT-3.5 Turbo to 4,000 tokens. Today, many models support contexts of up to 128,000 tokens, with several, especially from Google, extending this capability to two million tokens or more. However, while these advances offer a seemingly straightforward alternative to retrieval-augmented generation (RAG), the underlying challenges associated with long-context processing remain.

Long Context Models

Long context models are designed to process and reason over extended sequences. Their potential is evident in applications where maintaining a vast context is beneficial. With the ability to ingest extensive text, these models can, in principle, retrieve specific information directly from the input without the need for separate retrieval steps. Nonetheless, this capability hinges on the scaling of the attention mechanism in transformer architectures, which grows quadratically with the sequence length. Consequently, this demands significant computational resources and may result in longer response times.

One of the key challenges with long-context models is ensuring that they accurately locate and recall the relevant piece of information buried within voluminous text. The pivotal question is whether these models can reliably identify the appropriate fact within extensive contexts and produce precise answers based on that information.

The "Needle in the Haystack" Benchmarks

The "Needle in the Haystack" benchmarks were introduced to test the ability of long context models to retrieve a specific piece of information hidden within an expansive prompt. In these experiments, a detailed context is provided with a crucial fact embedded somewhere within it. The model's task is to find and reference this fact when queried.

Here is a simple variant of the "Needle in the Haystack" benchmark:

```
import random
import re
from openai import OpenAI

def assemble_snippet(essays, anchor_phrase, relative_anchor_pos,
target_length):
    random.shuffle(essays)
    sentences = re.split(r'(?<=[.!?])\s+', " ".join(essays))
    if not sentences:
        return ""

    l = r = random.randint(0, len(sentences) - 1)
    total_length = lambda l, r: sum(len(s) for s in
sentences[l:r+1]) + (r - l)

    while total_length(l, r) < target_length:
        if l > 0:
            l -= 1
        if total_length(l, r) >= target_length:
            break
        if r < len(sentences) - 1:
            r += 1
        if l == 0 and r == len(sentences) - 1:
            break

    window = sentences[l:r+1]
    insert_index = int(relative_anchor_pos * len(window))
    window.insert(insert_index, anchor_phrase)
    return " ".join(window).strip()


anchor = "The RAG Tutorial at the Weizenbaum Institute takes place
on the 7th of March."
question = "What is the date of the RAG Tutorial at the Weizenbaum
Institute?"

target_prompt_length = 100000

result = []
num_tokens = []

client = OpenAI()
for pos in [0.1 * i for i in range(11)]:
    context = ""
    context = assemble_snippet(essays, anchor, pos,
target_prompt_length * 4) # factor 4 because the assembly functions
counts chars, not tokens
```

```
    response = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[{"role": "user", "content": f"
{question}\n\n<context>{context}</context>"}],
        max_tokens=100
    )
    answer = response.choices[0].message.content
    num_tokens.append(response.usage.prompt_tokens)
    correct = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[
            {"role": "system", "content": f"Only answer with True or
False"},
            {"role": "user", "content": f"Does the answer contain
the expected answer for the questions '{question}'\n<provided-
answer>{answer}</provided-answer>\n<correct-answer>{anchor}
</correct-answer>"}
        ],
        max_tokens=100
    )
    result.append(bool(correct.choices[0].message.content))

print(sum(result) / len(result))
```

Early experiments with long context models revealed a phenomenon known as the "Lost-in-the-Middle Syndrome". These models were reliable in retrieving facts that were positioned at either the beginning or the end of the text but struggled with information located in the middle.

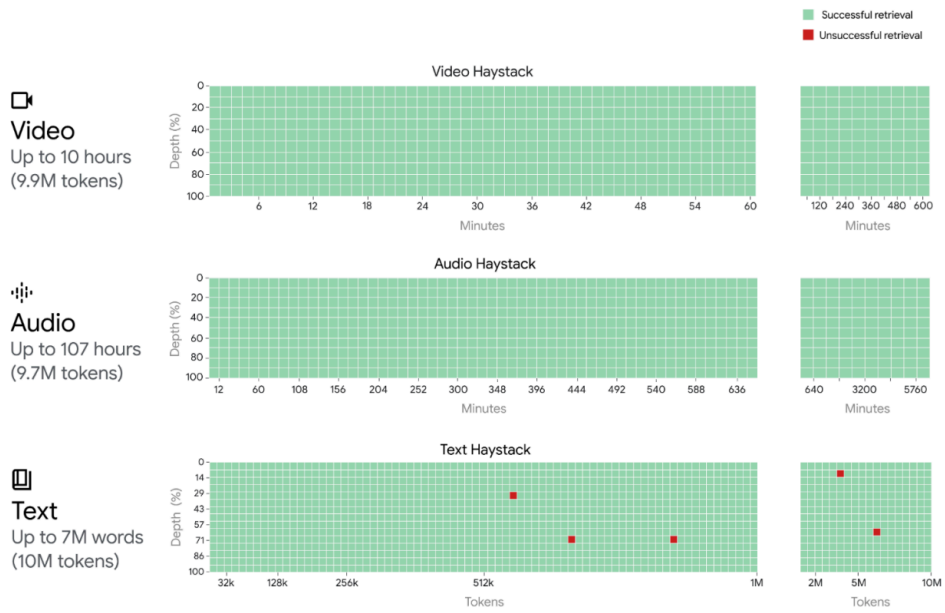


Figure 1: Needle in the Haystack results for Gemini 1.5 Pro show that the model achieves near-perfect “needle” recall (>99.7%) up to 1M tokens of “haystack” in all modalities, i.e., text, video and audio.

Over time, advancements in handling long contexts have mitigated these issues to a large extent. Current results from Needle in the Haystack experiments suggest that when the context environment is well managed and relevant details are clearly presented, modern long context models are capable of effectively retrieving the necessary information. **It is reasonable to conclude that if the context contains a clear answer to a query, the current models will be able to identify and extract that information appropriately**.

Comparing RAG and Long Context Models

While long context models promise a unified approach by handling everything within one extensive prompt, the practical implications of this approach introduce several concerns. Critics argue that relying solely on long context may lead to inefficiencies—such as longer inference times and redundant processing—especially when multiple queries are made that necessitate re-sending the entire context.

Retrieval-augmented generation (RAG) offers an alternative that leverages external retrieval mechanisms to incorporate relevant information. RAG systems are structured to handle massive amounts of data, retrieving pertinent content from millions of documents, which can then be processed alongside the query. This approach reduces the need to repeatedly pass large amounts of context to the model and often results in more efficient and targeted responses.

Below is a summary of the pros and cons of each approach:

Long Context Models

Pros:

- Directly accesses all information within a single, unified prompt.
- Simplified architecture without the need for a separate retrieval mechanism.
- Eliminates the dependency on an external knowledge base for context retrieval.

Cons:

- Processing extremely long contexts is computationally intensive and can slow down response times.
- Increased risk of processing irrelevant or redundant information, which could hinder the model's instruction following.
- Limited scalability, as even the most advanced models have an upper bound on the number of tokens they can handle effectively.

Retrieval-Augmented Generation (RAG)

Pros:

- Scales efficiently by enabling the processing of vast amounts of data retrieved from external sources.

- Facilitates targeted retrieval of relevant information, leading to potentially faster and more accurate responses.
- Reduces the computational load compared to processing an entire long context with each query.

Cons:

- Incorporates additional complexity by requiring the integration of a robust retrieval mechanism.
- Relies on the precision of the retrieval component; mismatches in retrieval can lead to errors in the generated response.
- Requires careful management of both the retrieval and generation components to ensure a seamless and coherent output.

Conclusion

Both long context models and RAG-based approaches have their respective strengths and weaknesses. Long context models offer the allure of directly handling extended inputs, simplifying the architectural design by eliminating an external retrieval step. However, they come with significant computational challenges and efficiency concerns, particularly as the context length grows.

On the other hand, RAG systems provide a more scalable solution by breaking down the retrieval process and processing only the most relevant information. Although this introduces additional layers of complexity, it offers substantial benefits in terms of response time and practical efficiency when dealing with large volumes of data.

Ultimately, the choice between using a long context model or a RAG approach depends on the specific requirements of the application. A hybrid solution that combines the strengths of both methods may, in many cases, provide the optimal balance between efficiency and comprehensive context processing.

LLM Fine-Tuning vs. RAG

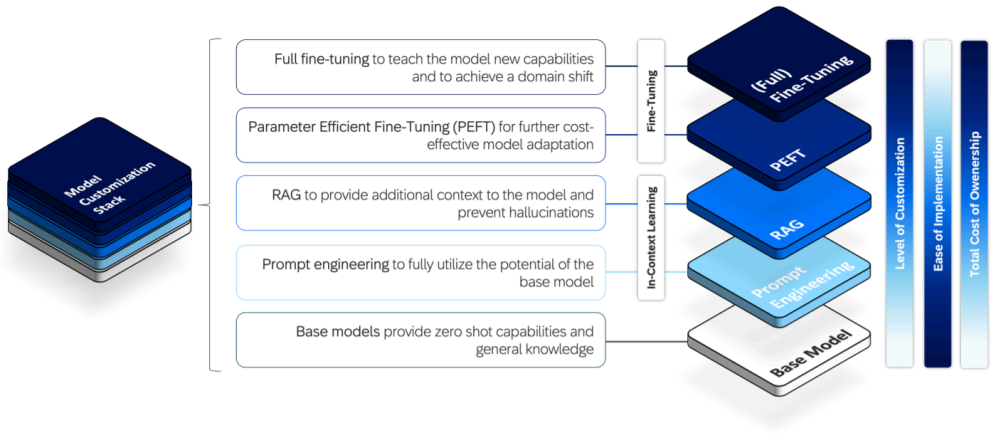


Figure 1: Complementary approaches to adjust a model to a specific use case.

There are several ways to adapt a model to a specific use case. The most basic technique is prompt engineering. As discussed earlier, Retrieval-Augmented Generation (RAG) enables bringing new knowledge to the model through in-context learning. Another popular technique is fine-tuning. While both approaches aim to enhance performance on domain-specific tasks, they operate in fundamentally different ways and offer distinct advantages and limitations.

Overview of Approaches

Fine-tuning involves continuing the training process of a pre-trained LLM with data that is representative of a targeted use case. This method adjusts the model's internal parameters, allowing it to learn specific patterns and stylistic requirements. There are two approaches to fine-tuning:

1. **Full Fine-Tuning:** Every parameter of the model is updated during the training process. Although this method allows deep customization and can yield impressive, domain-specific performance, it also requires substantial computational resources and risks diminishing the broad pre-

training capabilities of the model—a phenomenon sometimes referred to as catastrophic forgetting.

2. **Parameter-Efficient Fine-Tuning (PEFT):** Techniques such as Low-Rank Adaptation (LoRA) update only a subset of the model's parameters by introducing task-specific adapters while keeping the majority of the base model frozen. This strategy minimizes resource requirements and better preserves the model's general knowledge while still adapting it to specialized tasks.

Whether the process includes full fine-tuning or employs techniques like PEFT, fine-tuning ultimately modifies the model's internal knowledge. This makes it particularly effective for applications that require consistent, specialized outputs—such as imitating a historical writing style or processing industry-specific documents—yet it also carries the risk of diminishing the general pre-trained capabilities through over-specialization.

In contrast, RAG supplements the inherent knowledge of an LLM by incorporating external, task-specific information during inference. With RAG, the model retains its general-purpose training while dynamically integrating retrieved context relevant to the task at hand. This method leverages external databases or document collections that can be updated independently, ensuring that the model can provide grounded and contextually rich responses. However, the prerequisites for RAG include having the external knowledge verbalized and available in a retrievable text format, which can be challenging when new skills or entirely new languages and software libraries should be taught to the model.

Comparing the Two Strategies

When deciding between fine-tuning and RAG, several factors must be taken into account:

- **Knowledge Integration:**
  Fine-tuning directly embeds task-specific data into the model's internal parameters, making it highly effective for memorizing and consistently

replicating specialized outputs. In contrast, RAG supplements the model's native capabilities by dynamically incorporating external context, thereby maintaining the flexibility of the pre-trained model while still providing fresh and relevant information. A blogpost by AnyScale fomulates this as: "Fine tuning is for form, not facts"

- **Flexibility and Update-ability:**
  With RAG, new external data can be easily added or updated without retraining the model, offering a dynamic solution that adapts to rapidly changing information. Fine-tuning, while often more deeply customized, is inherently static: updates require additional rounds of training, and data used for training cannot be directly referenced or removed once assimilated into the model's weights.

- **Applicability and Limitations:**
  RAG serves as an effective form of model customization when it is possible to articulate new requirements or domain knowledge through external text. However, if a task involves teaching the model a completely new skill or language, RAG may fall short because the necessary expertise cannot be fully conveyed through retrieval alone. Conversely, fine-tuning is powerful for such scenarios but comes at the cost of reduced flexibility, increased computational expense, and potential challenges in maintaining compliance with data citation or removal requirements.

- **Cost and Operational Considerations:**
  Fine-tuning typically requires significant computational resources during the training phase, which can result in higher initial costs and maintenance efforts, especially when managing multiple specialized models. RAG, by offloading domain-specific knowledge to an external repository that can be updated independently, often proves to be a more resource-efficient and cost-effective approach, particularly for large-scale models.

Often, organizations find that a hybrid approach combines the benefits of both strategies. By fine-tuning a model to internalize key domain specifications and simultaneously employing RAG to access timely external information, it is possible to achieve an optimal balance. This ensures that the model remains

both highly accurate in specialized tasks and flexible enough to adapt to evolving data.

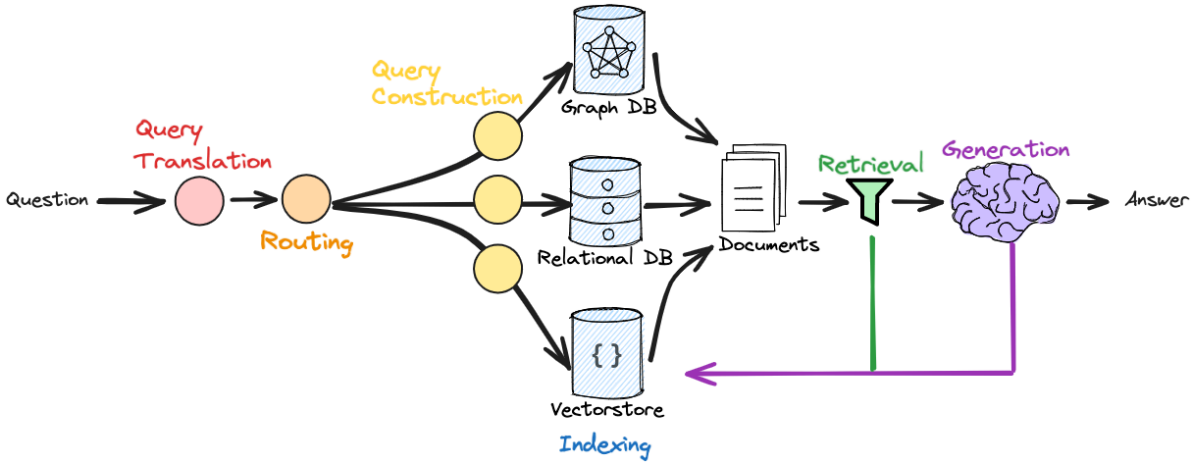
Conclusion

Neither fine-tuning nor retrieval-augmented generation offers a one-size-fits-all solution. Fine-tuning is ideal when a consistent, deeply integrated, task-specific output is required, while RAG provides a flexible, dynamic framework that enriches the model's responses with external, up-to-date information. Ultimately, the decision between these two approaches should be driven by the specific needs of the use case, the nature of the available domain data, and the cost and resource considerations of the implementation.

For many organizations, a hybrid approach that incorporates both strategies may yield the best results—leveraging fine-tuning to internalize critical domain specifications and RAG to ensure responses remain current and contextually rich.

- "Fine tuning is for form, not facts"
- "LLM Fine-tuning" by Salma Sohrabi-Jahromi and Mathis Börner: *SAP Technology Blogs*

RAG Components



This section lays out the end-to-end workflow for building effective RAG systems. We begin with the foundational step of document preparation, where diverse formats—from binary PDFs to complex web pages—are transformed into plain text. Detailed techniques such as optical character recognition (OCR) and specialized libraries ensure that even documents with challenging layouts can be accurately processed. Building on this, the section covers chunking strategies, which break down large texts into smaller, manageable units. These methods—ranging from recursive splitting to advanced semantic and LLM-based chunking—ensure that each text segment preserves its contextual integrity for later retrieval.

Following document conversion, the focus shifts to indexing and retrieval. Here, we discuss how text chunks are encoded into dense vector representations using modern embedding models. These vectors are then organized within specialized data structures to support efficient similarity searches. Techniques such as Approximate Nearest Neighbors (ANN) and algorithms like HNSW are explored to highlight how fast, scalable retrieval is achieved in practice.

Next, the generation chapter demonstrates how retrieved text segments are leveraged by large language models to construct coherent and contextually rich answers. Practical examples illustrate the integration of document stores with LLMs to create a smooth and effective generation pipeline.

In the subsequent chapters on query translation and query construction, the discussion delves into refining and reformulating user queries. Advanced approaches such as Multi-Query, RAG-Fusion, decomposition, and HyDE illustrate how multiple perspectives and hypothetical answers can enhance retrieval precision. Additionally, techniques for transforming natural language into structured queries—be it SQL, Cypher, or metadata-based filters—are presented, ensuring that the system can interact with a wide variety of data sources effectively.

Overall, these chapters collectively provide a comprehensive guide to setting up and optimizing each component of a RAG system—from preparing and indexing documents to generating answers and refining queries—equipping practitioners with the insights and tools needed to handle real-world, large-scale data retrieval challenges.

Document preparation

Convert Documents to Text

The process of converting various document formats into plain text is a crucial first step for many TAG use cases. Often, the documents available are not solely stored as text but are in binary formats like PDFs. For this reason, text extraction from these formats becomes essential in order to effectively process and analyze the data. PDF is by far the most commonly encountered format, yet it remains one of the most challenging. PDFs offer a great deal of flexibility in how content is presented, which means that what looks like text on a page may actually be an embedded image. In cases like these, standard text extraction methods may not suffice, and additional techniques such as optical character recognition (OCR) become necessary. It is also important to consider how to handle images or graphics that are embedded within documents, as they can further complicate the extraction process.

To address these challenges, several toolkits have been developed, offering robust solutions for handling diverse document types. Tools such as `docling` created by IBM and `markitdown` by Microsoft are highly recommended. These libraries build upon underlying PDF toolkits but also provide support for a broader range of document formats, ensuring flexibility in environments where not all documents are produced as PDFs. This is particularly relevant in industrial contexts where documents of various types need to be managed with a high degree of functionality and user convenience.

Finally, another common challenge lies in processing scientific documents with complex layouts, such as those with two-column formats and complexer structures like tables. Handling these layouts requires dedicated tools that can adjust and accurately extract text content despite the non-standard formatting. Many modern toolkits include solutions designed specifically to address these kinds of layout issues, ensuring that the resulting text is both accurate and functional for further processing. These insights establish a foundation for understanding the technical challenges in converting documents to text and

underscore the importance of carefully selecting the appropriate tools in your extraction workflow.

Example usage for `markitdown`:

```
from markitdown import MarkItDown
from openai import OpenAI

client = OpenAI()
md = MarkItDown(llm_client=client, llm_model="gpt-4o-mini")
result = md.convert("papers/2501.07391v1.pdf")

print(result.text_content)
```

Example usage for `markitdown`:

```
from pathlib import Path
from docling.datamodel.base_models import InputFormat
from docling.datamodel.pipeline_options import (
    PdfPipelineOptions,
)
from docling.document_converter import DocumentConverter,
PdfFormatOption

input_doc_path = Path("papers/2501.07391v1.pdf")

pipeline_options = PdfPipelineOptions(
    enable_remote_services=False
)
pipeline_options.do_picture_description = False

doc_converter = DocumentConverter(
    format_options={
        InputFormat.PDF: PdfFormatOption(
            pipeline_options=pipeline_options,
        )
    }
)
result = doc_converter.convert(input_doc_path)
print(result.document.export_to_markdown())
```

Another typical source for RAG content are website. There are also several services and toolkit to convert website to text like: Firecrawl, Reader by Jina AI or the open-source solution Crawl4AI.

Chunking

In the context of RAG applications, chunking refers to the process of breaking down large bodies of text into smaller, more manageable segments or “chunks.” This technique is crucial because it enables the system to efficiently index and retrieve relevant pieces of information from a vast corpus. By segmenting the text, the retrieval component can quickly pinpoint the most pertinent chunks that provide context for the generative model, leading to more accurate and contextually enriched responses. Essentially, chunking helps bridge the gap between large-scale data and the focused, specific queries that RAG systems need to handle effectively.

However, there isn’t a single “correct” way to perform chunking—its implementation can vary based on the text’s structure, the specific application requirements, and the desired level of granularity. This variability has led to the development of many different approaches, ranging from fixed-size blocks to more dynamic, semantic-based segmentation methods, each designed to optimize performance under different circumstances.

Chunking	Size	Overlap	Recall	Precision	Precision_Ω	IoU
Recursive	800 (~661)	400	85.4 ± 34.9	1.5 ± 1.3	6.7 ± 5.2	1.5 ± 1.3
TokenText	800	400	87.9 ± 31.7	1.4 ± 1.1	4.7 ± 3.1	1.4 ± 1.1
Recursive	400 (~312)	200	88.1 ± 31.6	3.3 ± 2.7	13.9 ± 10.4	3.3 ± 2.7
TokenText	400	200	88.6 ± 29.7	2.7 ± 2.2	8.4 ± 5.1	2.7 ± 2.2
Recursive	400 (~276)	0	89.5 ± 29.7	3.6 ± 3.2	17.7 ± 14.0	3.6 ± 3.2
TokenText	400	0	89.2 ± 29.2	2.7 ± 2.2	12.5 ± 8.1	2.7 ± 2.2
Recursive	200 (~137)	0	88.1 ± 30.1	7.0 ± 5.6	29.9 ± 18.4	6.9 ± 5.6
TokenText	200	0	87.0 ± 30.8	5.2 ± 4.1	21.0 ± 11.9	5.1 ± 4.1
Kamradt	N/A (~660)	0	83.6 ± 36.8	1.5 ± 1.6	7.4 ± 10.2	1.5 ± 1.6
⋆ KamradtMod	300 (~397)	0	87.1 ± 31.9	2.1 ± 2.0	10.5 ± 12.3	2.1 ± 2.0
⋆ Cluster	400 (~182)	0	91.3 ± 25.4	4.5 ± 3.4	20.7 ± 14.5	4.5 ± 3.4
⋆ Cluster	200 (~103)	0	87.3 ± 29.8	**8.0 ± 6.0**	**34.0 ± 19.7**	**8.0 ± 6.0**
⋆ LLM	N/A (~240)	0	**91.9 ± 26.5**	3.9 ± 3.2	19.9 ± 16.3	3.9 ± 3.2

Figure 1: Evaluation of various popular chunking strategies on our evaluation. [Brandon et al.'24]

A study conducted by Researcher from *Chroma* compares traditional, heuristic-based splitters such as recursive text spliteter and token-based text splitter with more advanced semantic approaches, including modified versions of an embedding model based semantic splitter, a cluster splitter, and a novel llm-based semantic splitter. The study introduces token-level evaluation metrics like recall, precision, PrecisionΩ, and Intersection over Union (IoU) to quantify retrieval efficiency and effectiveness, using embedding models such as OpenAI's text-embedding-3-large and Sentence Transformers' all-MiniLM-L6-v2. The results indicate that while each method offers unique advantages based on factors such as chunk size and overlap, semantic chunking approaches show promise by delivering competitive recall and precision, although they also introduce practical challenges such as increased processing time and complexity.

Recrusive Chunking

The recursive chunking scheme is a text segmentation approach that divides documents into manageable pieces while attempting to preserve natural semantic boundaries. It works by trying a series of increasingly fine-grained separators in sequence, starting with larger structural elements like paragraph breaks before moving to smaller units like sentences, words, and finally individual characters. This hierarchical approach ensures that text is split at the most natural boundaries possible while still meeting the specified chunk size requirements. What makes this technique powerful is its adaptability to different document structures. When a text segment is too large after applying one separator, the algorithm recursively tries the next separator in the sequence until it finds a division that works. This creates a cascade effect where paragraphs might be split into sentences, sentences into words, and only when necessary, words into characters. By including overlap between chunks, the system further preserves context at boundary points, allowing downstream applications like search engines or language models to maintain semantic coherence even when working with segmented text.

```
import re
from typing import List, Optional


def recursive_text_splitter(
    text: str,
    chunk_size: int = 1000,
    min_chunk_size: int = 200,
    chunk_overlap: int = 200,
    separators: Optional[List[str]] = None) -> List[str]:
    """
    Split text recursively using a list of separators, ensuring
minimum chunk size.

    Args:
        text: The text to split
        chunk_size: Maximum size of each chunk
        min_chunk_size: Minimum size of each chunk
        chunk_overlap: Overlap between chunks
        separators: List of separators to use in order of preference

    Returns:
        List of text chunks
    """
    # Default separators if none provided
    if separators is None:
        separators = ["\n\n", "\n", " ", ""]

    # Ensure min_chunk_size is not larger than chunk_size
    min_chunk_size = min(min_chunk_size, chunk_size)

    def merge_chunks(splits: List[str], separator: str) ->
List[str]:
        """Merge splits into chunks with overlap, ensuring minimum
size."""
        chunks = []
        current_chunk = []
        current_length = 0

        for split in splits:
            split_length = len(split) + (len(separator) if
current_chunk else 0)

            # If adding this split would exceed chunk size, finalize
current chunk
            if current_length + split_length > chunk_size:
                # Save current chunk if it meets minimum size
```

```
                if current_chunk:
                    chunk_text = separator.join(current_chunk)
                    if len(chunk_text) >= min_chunk_size or not
chunks:
                        chunks.append(chunk_text)

                    # Create overlap for next chunk
                    overlap_chunks = []
                    overlap_length = 0

                    for item in reversed(current_chunk):
                        sep_len = len(separator) if overlap_chunks
else 0
                        if len(item) + sep_len + overlap_length >
chunk_overlap:
                            break
                        overlap_chunks.insert(0, item)
                        overlap_length += len(item) + sep_len

                    current_chunk = overlap_chunks
                    current_length = overlap_length

                # Handle splits larger than chunk_size
                if split_length > chunk_size:
                    for i in range(0, len(split), chunk_size -
chunk_overlap):
                        chunk = split[i:min(i + chunk_size,
len(split))]
                        if len(chunk) >= min_chunk_size or not
chunks:
                            chunks.append(chunk)

                    current_chunk = []
                    current_length = 0
                    continue

            # Add to current chunk
            current_chunk.append(split)
            current_length += split_length

        # Handle the final chunk
        if current_chunk:
            final_text = separator.join(current_chunk)
            if len(final_text) >= min_chunk_size or not chunks:
                chunks.append(final_text)
            elif chunks and len(chunks[-1]) + len(separator) +
len(final_text) <= chunk_size:
                # Merge with previous chunk if too small and fits
```

```
                chunks[-1] = chunks[-1] + separator + final_text

        return chunks

    def split_text(text: str, level: int = 0) -> List[str]:
        """Split text using separators at current level."""
        # Base cases
        if len(text) <= chunk_size:
            return [text] if len(text) >= min_chunk_size or not text
else []

        # If at the character level, chunk by size
        if level >= len(separators) - 1:
            chunks = []
            for i in range(0, len(text), max(1, chunk_size -
chunk_overlap)):
                chunk = text[i:i + chunk_size]
                if len(chunk) >= min_chunk_size or not chunks:
                    chunks.append(chunk)
            return chunks

        # Try to split with current separator
        separator = separators[level]
        splits = [char for char in text] if separator == "" else
text.split(separator)

        # If splitting doesn't work, try next separator
        if len(splits) <= 1:
            return split_text(text, level + 1)

        # Process each split
        results = []
        for split in splits:
            if len(split) <= chunk_size:
                results.append(split)
            else:
                results.extend(split_text(split, level + 1))

        # Merge the results
        return merge_chunks(results, separator)

    return split_text(text)
```

Semantic Chunking

Semantic chunking breaks a long text into smaller, meaningful parts by first splitting it into sentences, then grouping nearby sentences together to add context. It looks for natural shifts in meaning—places where the content changes—and uses these shifts to decide where to make the cuts, ensuring that each piece feels like a complete thought.

```python
from typing import List, Dict, Any
import numpy as np
from openai import OpenAI

client = OpenAI()


def calculate_distances(sentences: List[str], buffer_size: int = 3)
-> List[float]:
    """
    Calculates semantic distances between adjacent sentences with
context.

    Args:
        sentences: List of sentence strings
        client: OpenAI client
        buffer_size: Number of sentences to include as context
before and after

    Returns:
        distances: List of semantic distances between adjacent
sentences
    """
    # Calculate embeddings directly in batches
    BATCH_SIZE = 500
    embedding_matrix = None

    # Process sentences in batches, combining with context on the
fly
    for batch_start in range(0, len(sentences), BATCH_SIZE):
        batch_end = min(batch_start + BATCH_SIZE, len(sentences))

        # Create combined sentences for this batch
        batch_combined = []
        for i in range(batch_start, batch_end):
            context_start = max(0, i - buffer_size)
            context_end = min(len(sentences), i + buffer_size + 1)
            combined = '
'.join(sentences[context_start:context_end])
            batch_combined.append(combined)

        # Get embeddings for this batch using OpenAI API
        response = client.embeddings.create(model='text-embedding-3-
small', input=batch_combined)
        batch_embeddings = np.array([item.embedding for item in
response.data])
```

```
        if embedding_matrix is None:
            embedding_matrix = batch_embeddings
        else:
            embedding_matrix = np.concatenate((embedding_matrix,
batch_embeddings), axis=0)

    # Normalize embeddings
    norms = np.linalg.norm(embedding_matrix, axis=1, keepdims=True)
    embedding_matrix = embedding_matrix / norms

    # Calculate similarity matrix and extract distances between
adjacent sentences
    similarity_matrix = np.dot(embedding_matrix, embedding_matrix.T)
    distances = [1 - similarity_matrix[i, i + 1] for i in
range(len(sentences) - 1)]

    return distances


def get_cut_indices(distances, target_cuts):
    """
    Find cut indices based on semantic distances and target number
of cuts.

    Args:
        distances: List of semantic distances between adjacent
sentences
        target_cuts: Target number of cuts

    Returns:
        List of cut indices
    """
    # Binary search for optimal threshold
    lower_limit, upper_limit = 0.0, 1.0
    distances_np = np.array(distances)

    while upper_limit - lower_limit > 1e-6:
        threshold = (upper_limit + lower_limit) / 2.0
        cuts = np.sum(distances_np > threshold)

        if cuts > target_cuts:
            lower_limit = threshold
        else:
            upper_limit = threshold

    # Find cut points based on threshold
    cut_indices = [i for i, d in enumerate(distances) if d >
threshold] + [-1]
```

```
    return cut_indices

def semantic_text_splitter(text: str,
                           avg_chunk_size: int = 1000,
                           min_chunk_size: int = 200,
                           max_chunk_size: int = 4000) -> List[str]:
    """
    Split text into chunks of approximately avg_chunk_size
characters based on semantic similarity.

    Args:
        text: The input text to be split
        client: OpenAI client instance
        avg_chunk_size: Target average size of chunks in characters
        min_chunk_size: Minimum size for initial text splitting

    Returns:
        List of text chunks
    """
    # Split text into minimal sentence units
    sentences = recursive_text_splitter(text, min_chunk_size,
int(min_chunk_size*0.5), chunk_overlap=0)
    assert all([len(sentence) <= max_chunk_size for sentence in
sentences])
    assert all([len(sentence) >= int(min_chunk_size*0.5) for
sentence in sentences])
    # Calculate distances between sentences
    distances = calculate_distances(sentences)

    # Determine number of cuts needed based on character count
    total_length = sum(len(s) for s in sentences)
    target_cuts = total_length // avg_chunk_size

    cut_indices = get_cut_indices(distances, target_cuts)

    # Create chunks based on cut points
    chunks = []
    current_chunk = ''
    sentence_pointer = 0
    while sentence_pointer < len(sentences):
        sentence = sentences[sentence_pointer]
        if len(current_chunk) + len(sentence) > max_chunk_size:
            chunks.append(current_chunk.strip())
            current_chunk = ''
            cut_indices = [n+sentence_pointer for n in
get_cut_indices(distances[sentence_pointer:], target_cuts-
len(chunks))]
```

```
            continue
        current_chunk += f'\n{sentence}' if current_chunk else
sentence
        if sentence_pointer == cut_indices[0]:
            chunks.append(current_chunk.strip())
            current_chunk = ''
            cut_indices.pop(0)
        sentence_pointer += 1


    if current_chunk:
        chunks.append(current_chunk.strip())
    return chunks
```

LLM-based Chunking

The LLM-based splitter uses are recursive splitter to fracture the text into small pieces and prompts an LLM to merge chunks together based on the semantics of the chunks.

```
from openai import OpenAI

client = OpenAI()


SYSTEM_PROMPT = "You are an assistant specialized in splitting text
into thematically consistent sections."

USER_MSG = """The text has been divided into chunks, each marked
with <|start_chunk_X|> and <|end_chunk_X|> tags, where X is the
chunk number.
Your task is to identify the points where splits should occur, such
that consecutive chunks of similar themes stay together. Try to
avoid splitting in the middle of a topic/section/paragraph

{chunked_input}

Respond with a list of chunk IDs where you believe a split should be
made. For example, if chunks 1 and 2 belong together but chunk 3
starts a new topic, you would suggest a split after chunk 2.
THE CHUNKS MUST BE IN ASCENDING ORDER.
Your response should be in the form: 'split_after: 3, 5'
Respond only with the IDs of the chunks where you believe a split
should occur.
YOU MUST RESPOND WITH AT LEAST ONE SPLIT. THESE SPLITS MUST BE IN
ASCENDING ORDER AND EQUAL OR LARGER THAN: {current_chunk}
"""

def llm_text_splitter(text,
                      min_chunk_size: int = 800,
                      n_chunks_per_prompt: int = 10,
                      max_retries: int = 5):
    chunks = recursive_text_splitter(text, min_chunk_size,
int(min_chunk_size*0.5), chunk_overlap=0)
    split_indices = []
    current_chunk = 0
    while True:
        if current_chunk >= len(chunks) - 4:
            break
        chunked_input = []
        for i in range(current_chunk, min(len(chunks),
current_chunk+n_chunks_per_prompt)):
            chunked_input.append(f"<|start_chunk_{i+1}|>{chunks[i]}
<|end_chunk_{i+1}|>")
        chunked_input = '\n'.join(chunked_input)
        original_prompt =
USER_MSG.format(chunked_input=chunked_input,
```

```
current_chunk=current_chunk)
        prompt = original_prompt
        final_answer = None
        for _ in range(max_retries):
            result_string =
client.chat.completions.create(model='gpt-4o-mini', messages=
[{"role": "system", "content": SYSTEM_PROMPT}, {"role": "user",
"content": prompt}], max_tokens=200, temperature=0.2)
            result_string = result_string.choices[0].message.content
            split_after_line = [line for line in
result_string.split('\n') if 'split_after:' in line][0]
            numbers = re.findall(r'\d+', split_after_line)
            numbers = list(map(int, numbers))
            if not (numbers != sorted(numbers) or any(number <
current_chunk for number in numbers)):
                final_answer = numbers
                break
            else:
                prompt = original_prompt + f"\nThe previous response
of {numbers} was invalid. DO NOT REPEAT THIS ARRAY OF NUMBERS.
Please try again."
        if final_answer is None:
            raise ValueError("Failed to retrieve valid split")
        split_indices.extend(final_answer)
        current_chunk = numbers[-1]
        if len(numbers) == 0:
            break
    chunks_to_split_after = [i - 1 for i in split_indices]
    docs = []
    current_chunk = ''
    for i, chunk in enumerate(chunks):
        current_chunk += chunk + ' '
        if i in chunks_to_split_after:
            docs.append(current_chunk.strip())
            current_chunk = ''
    if current_chunk:
        docs.append(current_chunk.strip())
    return docs
```

Indexing and Retrieval

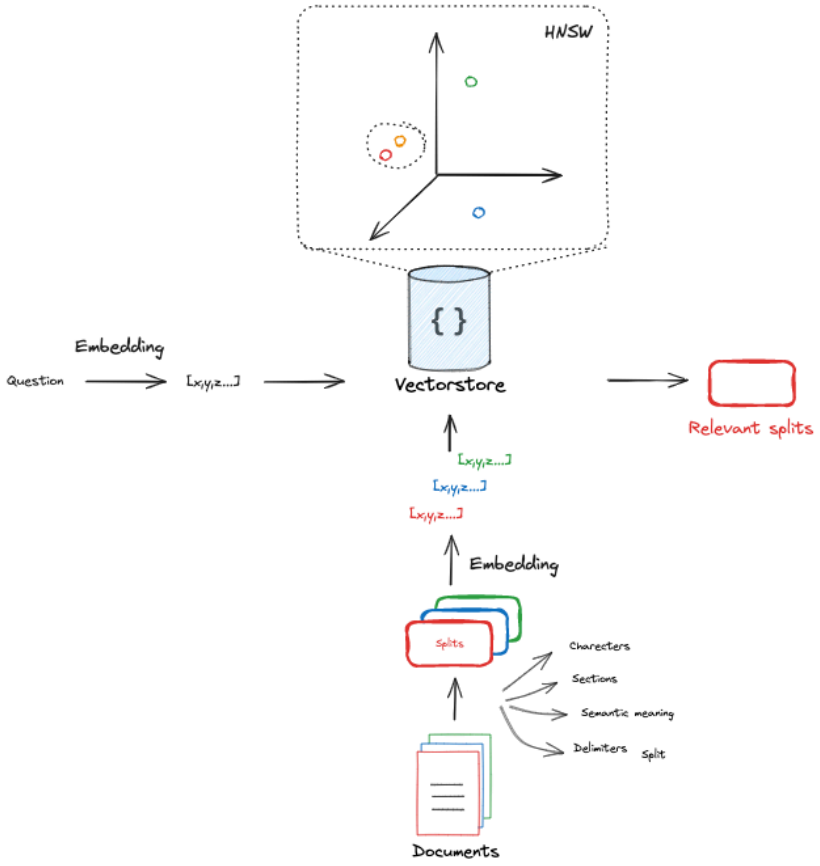


Figure 1: Visualization of the indexing and retrieval steps in a RAG system.[Lance Maring: Langchain]

In a Retrieval-Augmented Generation (RAG) system, the first step is to transform your raw data into a format that enables efficient search. Indexing is the process of breaking documents into smaller, meaningful chunks and converting them into dense vector representations using models like Transformers. These vectors capture semantic meaning, allowing the system to “understand” the content beyond mere keywords.

Once the data is encoded, the vectors are stored in specialized data structures. These structures enable fast similarity searches so that when a query is posed, the system can quickly identify and retrieve the most relevant chunks of text.

The retrieval process is critical—it ensures that only contextually related information is passed to the generative model, thereby enhancing the quality of the final output.

Approximate Nearest Neighbours

When working with large datasets or high-dimensional embeddings, computing exact similarity measures between a query and every document can become computationally expensive. This is where *Approximate Nearest Neighbor* (ANN) come into play. ANN algorithms allow us to quickly retrieve vectors that are most likely to be similar to a given query by sacrificing a bit of precision for a significant gain in speed and scalability.

Approximate nearest neighbor methods construct specialized index structures that partition the vector space in a way that makes similarity searches more efficient. Instead of comparing the query against every single vector, these algorithms focus on a smaller subset of candidates that are likely to be the nearest neighbors. This drastically reduces the computational overhead and enables real-time search capabilities even on large datasets.

HNSW

Hierarchical Navigable Small Worlds (HNSW) is a graph-based algorithm used for efficient approximate nearest neighbor searches in high-dimensional data. It constructs a multi-layer graph where each layer represents a different level of data granularity—upper layers have a sparse, coarse view for rapid long-range jumps, while the bottom layer is dense for fine-grained, local searches. During a query, the algorithm starts at the highest level, performing a greedy search to quickly approximate the region where the nearest neighbors lie before descending layer by layer to refine the results. This layered structure, inspired by probability skip lists and small world network principles, enables HNSW to achieve a near logarithmic search complexity, making it highly effective for large-scale applications like recommendation systems, image retrieval, and vector-based searches.

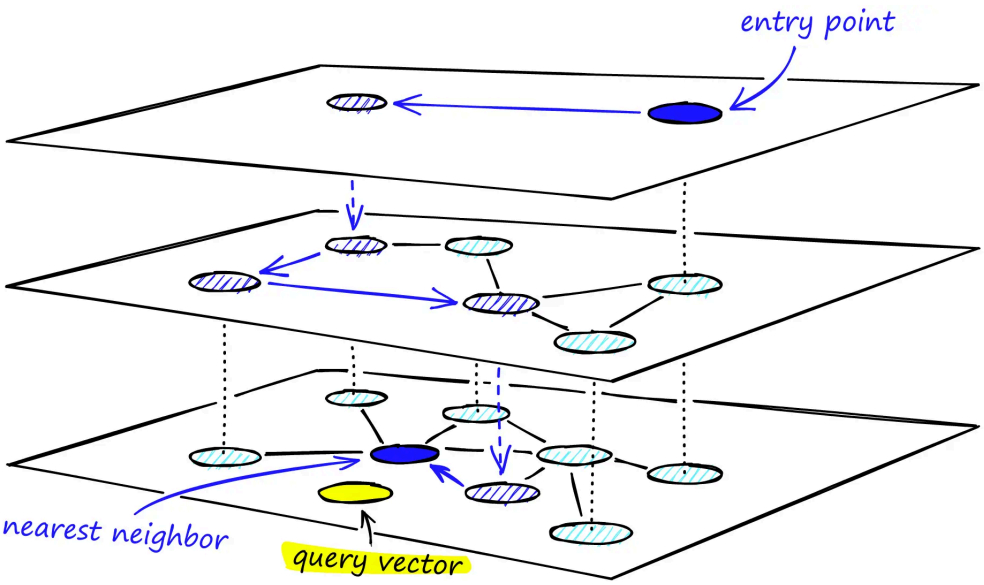


Figure 2: The search process through the multi-layer structure of an HNSW graph as it is used in FAISS.[Pinecone]

Popular libraries such as Facebook's FAISS, Spotify's Annoy, and Google's ScaNN provide robust implementations of ANN algorithms. They allow developers to balance the trade-off between search accuracy and speed. In many real-world applications, such as recommendation systems, semantic search, and RAG pipelines, the slight loss in accuracy is often acceptable given the substantial performance improvements.

Moreover, many ANN libraries are optimized for modern hardware, offering support for parallel processing and GPU acceleration. This means that as the volume of data grows, these tools can scale effectively, ensuring that search operations remain responsive. By integrating ANN into a RAG system, you can achieve near real-time retrieval of contextually relevant documents, thereby enhancing the overall performance and user experience.

In practive ANN is usually used via dedicated document databases designed for RAG. Popular examples are:

1. PostgreSQL with Pgvector

- *Description*: PostgreSQL is a versatile relational database, and pgvector is an extension that enables vector similarity searches.

- *Use Case*: Suitable for setups where data is already stored in PostgreSQL, offering advanced querying capabilities.

2. MongoDB Atlas

- *Description*: Cloud database service with integrated vector search capabilities, allowing for independent scaling of database and search index.
- *Use Case*: Useful for handling a variety of transactional and search workloads with high availability.

3. Pinecone

- *Description*: A cloud-based vector database optimized for machine learning workflows.
- *Use Case*: Popular for its ease of use and scalability in AI-driven applications.

4. Weaviate

- *Description*: An open-source cloud-native vector search engine.
- *Use Case*: Suitable for applications requiring semantic search capabilities.

5. Milvus

- *Description*: An open-source vector database focused on similarity searches for embedding vectors.
- *Use Case*: Ideal for applications needing efficient vector similarity searches.

6. Qdrant

- *Description*: An open-source vector search engine known for its performance and flexibility.
- *Use Case*: Often chosen for its scalability and open-source nature.

7. Chroma DB

- *Description*: An AI-native open-source embedding database.
- *Use Case*: Suitable for applications requiring a customizable vector database solution.

Custom Implementation

The code below show a simple example of a document database that supports vector retrieval. The primary purpose of this database is to serve as a foundational example that will be utilized in subsequent chapters to explore additional concepts and demonstrate the functionality of such vector databases. The implementation of this system prioritizes clarity and ease of use, rather than efficiency or performance optimization. Consequently, the design is intended solely for experimental purposes, providing a secure environment to test and expand upon ideas. It is not recommended for use in production scenarios where performance and scalability are critical factors. Future sections will further develop these concepts and integrate additional elements, showcasing how this simple framework can evolve to support more advanced use cases.

Base Classes

First will will implement a class to store documents/text chunks with metadata and a base class that defines the interface on a `DocumentStore`:

```python
from typing import List
from abc import ABC, abstractmethod
import hashlib

import uuid
from dataclasses import dataclass, field


# Create a custom namespace UUID for our document store
# We use uuid5 with the DNS namespace to create our own unique
namespace
DOCUMENT_STORE_NAMESPACE = uuid.uuid5(
    uuid.NAMESPACE_DNS,  # Standard DNS namespace
    'document.store'  # Our unique domain
)


@dataclass
class Document:
    text: str
    tags: set[str] = field(default_factory=set)
    metadata: dict[str, str] = None
    uuid: str = field(init=False)

    def __post_init__(self):
        """Initialize UUID based on text hash."""
        self.uuid = str(uuid.uuid5(DOCUMENT_STORE_NAMESPACE,
self.hash()))

    def hash(self) -> str:
        return hashlib.sha256(self.text.encode()).hexdigest()

    def __hash__(self):
        # use self.hash()
        return hash(self.hash())



class DocumentStore(ABC):

    @abstractmethod
    def add_document(self, document: Document, ignore_dublicates:
bool = True) -> str:
        pass

    def get_document_by_id(self, id: str) -> Document:
        return self.documents[id]
```

```python
    @abstractmethod
    def search(self, query: str, top_k: int = 10) -> List[Document]:
        pass
```

Basic Implementation

In the following we create two basic implementaionis using embedding models from SentenceTransformers or OpenAI:

```
from typing import List, Dict
import numpy as np
from openai import OpenAI
from sentence_transformers import SentenceTransformer
from abc import abstractmethod


class BasicDocumentStore(DocumentStore):
    def __init__(self):
        self.documents: Dict[str, Document] = {}
        self.embeddings: Dict[str, np.ndarray] = {}


    @abstractmethod
    def _get_embedding(self, text: str) -> np.ndarray:
        pass


    def add_document(self, document: Document, ignore_dublicates:
bool = True) -> str:
        """Add document to store and compute its embedding"""
        if document.uuid in self.documents and ignore_dublicates:
            return document.uuid

        self.documents[document.uuid] = document
        self.embeddings[document.uuid] =
self._get_embedding(document.text)
        return document.uuid


    def search(self, query: str, top_k: int = 10) -> List[Document]:
        """Search for similar documents using cosine similarity"""
        if not self.documents:
            return []

        # Get embedding for query
        query_embedding = self._get_embedding(query)

        # Vectorized cosine similarity calculation
        doc_ids = list(self.embeddings.keys())

        # Stack all document embeddings into a matrix
        doc_embeddings = np.vstack([self.embeddings[doc_id] for
doc_id in doc_ids])

        # Normalize query embedding
        query_norm = query_embedding /
```

```
np.linalg.norm(query_embedding)

        # Normalize all document embeddings (row-wise)
        doc_norms = np.linalg.norm(doc_embeddings, axis=1,
keepdims=True)
        normalized_docs = doc_embeddings / doc_norms

        # Calculate dot product (cosine similarity since vectors are
normalized)
        similarities = np.dot(normalized_docs, query_norm)

        # Get indices of top-k similarities
        top_indices = np.argsort(-similarities)[:top_k]

        # Map back to document IDs and return documents
        top_doc_ids = [doc_ids[i] for i in top_indices]
        return [self.documents[doc_id] for doc_id in top_doc_ids]


class OpenAIMixin:
    def __init__(self, model_name: str = "text-embedding-3-small",
client: OpenAI | None = None):
        self.model_name = model_name
        self.client = client or OpenAI()

    def _get_embedding(self, text: str) -> np.ndarray:
        """Get embedding from OpenAI API"""
        response = self.client.embeddings.create(
            model=self.model_name,
            input=text
        )
        return np.array(response.data[0].embedding)

class SentenceTransformerMixin:
    def __init__(self, model_name: str = "all-MiniLM-L6-v2"):
        self.model_name = model_name
        self.model = SentenceTransformer(model_name)

    def _get_embedding(self, text: str) -> np.ndarray:
        """Get embedding from Sentence Transformers model"""
        return self.model.encode(text, convert_to_numpy=True)


class OpenAIDocumentStore(OpenAIMixin, BasicDocumentStore):
    def __init__(self, model_name: str = "text-embedding-3-small"):
        OpenAIMixin.__init__(self, model_name)
        BasicDocumentStore.__init__(self)
```

```
class SentenceTransformerStore(SentenceTransformerMixin,
BasicDocumentStore):
    def __init__(self, model_name: str = "all-MiniLM-L6-v2"):
        OpenAIMixin.__init__(self, model_name)
        BasicDocumentStore.__init__(self)
```

Generation

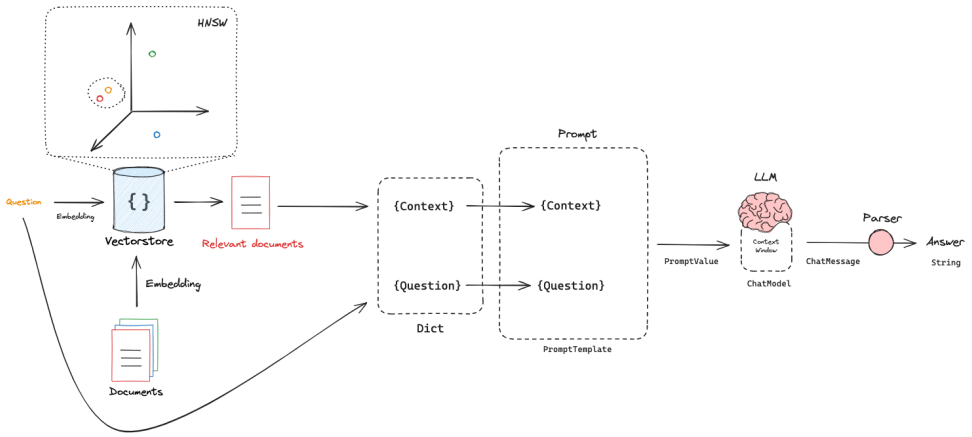


Figure 1: Visualization of the generation step of a RAG system. [Lance Maring: Langchain]

```
question = '...'

# Step 1: Init LLM client
client = OpenAI()

# Step 2: Init Document Store
store = OpenAIDocumentStore()
... # load a filed document store or fill the store


# Step 3: Retrieve chunks based on question and prepare context for
the prompt
docs = store.search(question, top_k=5)
context = '\n---\n'.join([doc.text for doc in docs])

# Step 4: Generate Answer
PROMPT = """    Answer the question only from the customer query
marked with delimiters <!> and context marked with delimiters <#>.

question:  <!>{question}<!>

Context:  <#>{context}<#>

"""
response = client.chat.completions.create(
    model='gpt-4o',
    messages=[
        {"role": "user", "content": PROMPT.format(question=question,
context=context)}
    ]
)
print(response.choices[0].message.content)
```

Query Translation

Multi-Query

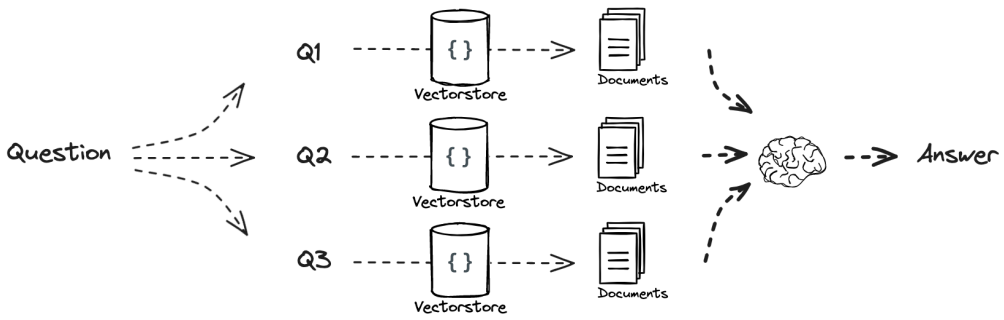


Figure 1: Flow of Multi-Query.[Lance Maring: Langchain]

Multi-Query RAG is an advanced retrieval-augmented generation technique that enhances the document retrieval process by using an LLM to generate multiple reformulated queries from a single user input. Instead of relying on a single query—which might miss relevant documents due to variations in phrasing—this approach automatically produces several query variations that capture different facets of the original question. Each query variant retrieves its own set of documents from a vector database, and the unique union of these results is then provided as context to the generative model. This enriched context helps the model produce more accurate and comprehensive responses by mitigating retrieval noise and covering a broader range of relevant information.

For an implemenation instead of directly retrieven from the document store we use the following look-up techique:

```
MULTI_QUERY_PROMPT = """Your task is to generate {n_questions}
different versions of the
given user question to retrieve relevant documents from a vector
database.
By generating multiple perspectives on the user question, your goal
is to help
the user overcome some of the limitations of the distance-based
similarity search.
You answer should constist only of the new questions. No further
explanations!
Provide these alternative questions separated by newlines. Original
question: {question}
"""

def do_multi_query(question, store, client, k=3, n_questions=3):
    response = client.chat.completions.create(
        model='gpt-4o',
        messages=[
            {"role": "user", "content":
MULTI_QUERY_PROMPT.format(n_questions=n_questions,
question=question)}
        ]
    )
    docs = []
    questions = response.choices[0].message.content.split('\n')
    print(Panel.fit('\n'.join(questions), title='Alternative
Questions'))
    for q in response.choices[0].message.content.split('\n'):
        docs.extend(store.search(q, top_k=k))
    return [*set(docs)]
```

RAG-Fusion

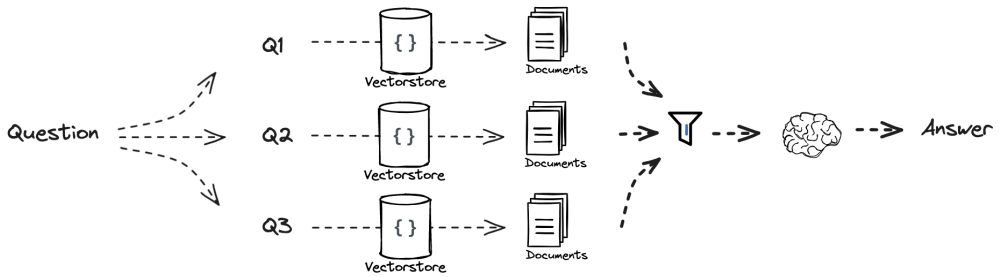


Figure 2: Flow of RAG-Fusion.[Lance Maring: Langchain]

RAG-Fusion refines the process of gathering external documents by not only generating multiple query variants but also merging their retrieval results using a fusion algorithm—typically reciprocal rank fusion—to re-rank and consolidate the most relevant documents before they are fed into the generative model. This fusion step contrasts with standard Multi-Query techniques, where multiple query variants are simply used to retrieve and union the documents without an additional ranking layer, potentially resulting in redundant or less focused content. By leveraging reciprocal rank fusion, RAG-Fusion can prioritize the highest-quality information across different query perspectives, thus enhancing the accuracy and relevance of the final generated response.

For an implemenation instead of directly retrieven from the document store we use the following look-up techique:

```
def do_rag_fusion(question, store, client, k_retrieve=10, k_keep=5,
n_questions=5):
    assert k_keep <= k_retrieve
    response = client.chat.completions.create(
        model='gpt-4o',
        messages=[
            {"role": "user", "content":
MULTI_QUERY_PROMPT.format(n_questions=n_questions,
question=question)}
        ]
    )
    docs = []
    questions = response.choices[0].message.content.split('\n')
    print(Panel.fit('\n'.join(questions), title='Alternative
Questions'))
    for q in response.choices[0].message.content.split('\n'):
        docs.append(store.search(q, top_k=k_retrieve))

    fused_scores = {}
    for docs_q_i in docs:
        # Iterate through each document in the list, with its rank
(position in the list)
        for rank, doc in enumerate(docs_q_i):
            # If the document is not yet in the fused_scores
dictionary, add it with an initial score of 0
            if doc not in fused_scores:
                fused_scores[doc] = 0
            # Update the score of the document using the RRF
formula: 1 / (rank + k)
            fused_scores[doc] += 1 / (rank + 60)

    # Sort the documents based on their fused scores in descending
order to get the final reranked results
    reranked_docs = [
        doc for doc, _ in sorted(fused_scores.items(), key=lambda x:
x[1], reverse=True)
    ][:k_keep]
    return reranked_docs
```

Decomposition

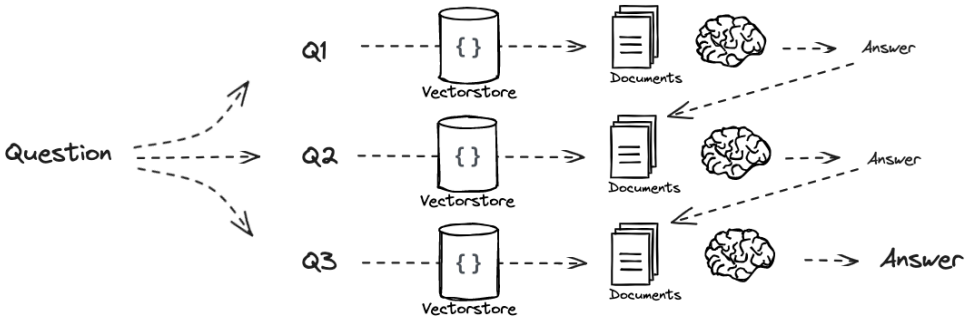


Figure 3: Flow of a "Decomposition Query".[Lance Maring: Langchain]

Decomposition involves breaking down a complex question into multiple, more specific sub-questions that are easier to answer in isolation. Each sub-question is passed through a retrieval step, which fetches relevant context from a vector store or document store, and then answered with the help of a large language model. The answers to these sub-questions are stored and used as incremental “building blocks,” so that the system gradually accumulates information and context. Finally, once all sub-questions have been answered, the system either synthesizes those partial answers into a final, comprehensive answer to the original query, or uses the most recent answer if only one step was needed. This decomposition approach ensures that complex or multi-faceted questions can be addressed more reliably, since each smaller query is more likely to retrieve precise, relevant documents and yield clearer, more accurate intermediate answers.

```
PROMPT = """Answer the question delimited by <!>, use previously
answered questions marked with <$> and context marked with
delimiters <#>.

question:  <!>{question}<!>

Previously answered questions: <$> {previous_questions} <$>

context: <#>{context}<#>
"""

DECOMPOSITION_PROMPT = """You are a helpful assistant that generates
multiple sub-questions related to an input question.
The goal is to break down the input into a set of {n_questions} sub-
problems / sub-questions that can be answers in isolation.
These sub-questions should be easier to answer than the original
question and decompose the problem into smaller parts and
clarify the problem.
Generate multiple search queries related to: "{question}"
Provide these sub-questions separated by newlines.
"""

question = '...'

# Step 1: Init LLM client
client = OpenAI()

# Step 2: Init Document Store
store = OpenAIDocumentStore()
... # load a filed document store or fill the store

# Step 3: Generate Sub-questions
response = client.chat.completions.create(
    model='gpt-4o',
    messages=[
        {"role": "user", "content":
DECOMPOSITION_PROMPT.format(n_questions=3, question=question)}
    ]
)
questions = [q.strip() for q in
response.choices[0].message.content.strip().split('\n') if
q.strip()]

# Step 4: Answer every sub-question before answering the main
question
previous_questions = []
for i, sub_question in enumerate(questions):
```

```
    # Step 4.a: Retrieve docs for the sub-question
    context = store.search(sub_question, top_k=5)
    # Step 4.b: Answer sub-question based on retrieved docs and
previously answered questions
    response = client.chat.completions.create(
        model='gpt-4o',
        messages=[
            {
                "role": "user",
                "content": PROMPT.format(
                    question=sub_question,
                    context='\n---\n'.join([d.text for d in
context]),

previous_questions='\n\n'.join(previous_questions)
                )
            }
        ]
    )
    sub_answer = response.choices[0].message.content
    # Step 4.c: Store answer to the sub-question
    previous_questions.append(f"Question: {sub_question}\n\nAnswer:
{sub_answer}")
final_answer = sub_answer
```

HyDE

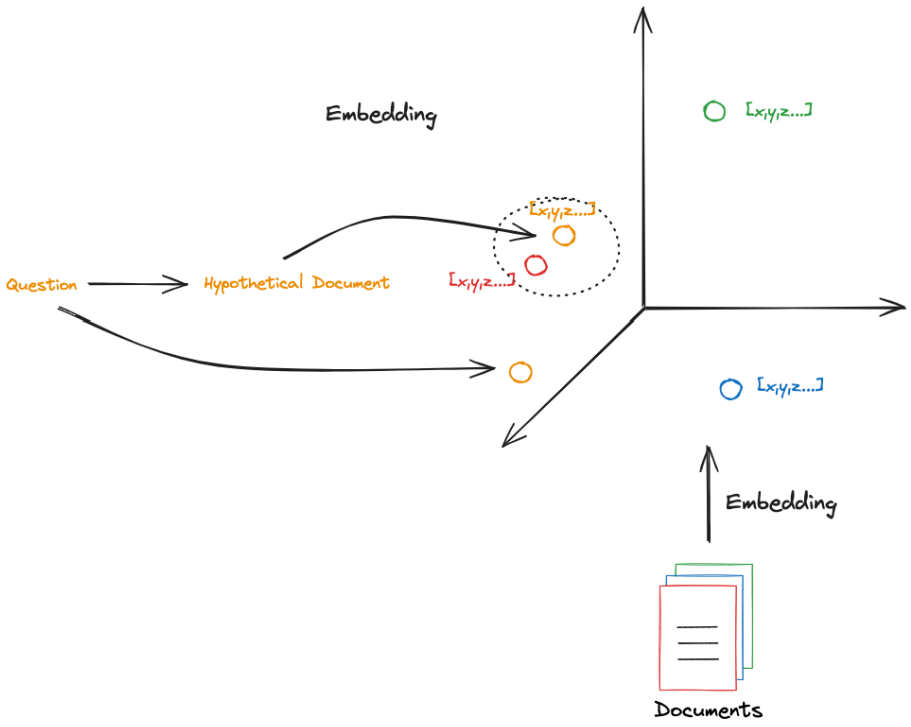


Figure 4: Flow of a "Decomposition Query".[Lance Maring: Langchain]

HyDE (Hypothetical Document Embedding) is an approach that tackles potential mismatches between a user's query and relevant documents by first creating a "hypothetical" answer to the query. Specifically, instead of embedding and retrieving based solely on the user's original query, the system prompts a large language model (LLM) to generate a short passage that might represent an ideal answer (even if it's partially or wholly fictional). This hypothetical answer is then embedded and used for document retrieval, on the premise that an answer-like passage often aligns more closely with relevant source documents in embedding space. After retrieving the most relevant documents using this "answer-first" approach, the original query is combined with the retrieved documents to produce the final response. This two-step technique can improve retrieval effectiveness, especially when user queries are vague or contain insufficient detail for direct similarity matching.

For an implemenation instead of directly retrieven from the document store we use the following look-up techique:

```
HYDE_PROMPT = """Please write a scientific paper passage to answer
the question
Question: {question}
"""

def do_hyde_search(question, store, client):
    response = client.chat.completions.create(
        model='gpt-4o',
        messages=[
            {"role": "user", "content":
HYDE_PROMPT.format(question=question)}
        ]
    )
    return store.search(response.choices[0].message.content,
top_k=5)
```

Query Construction

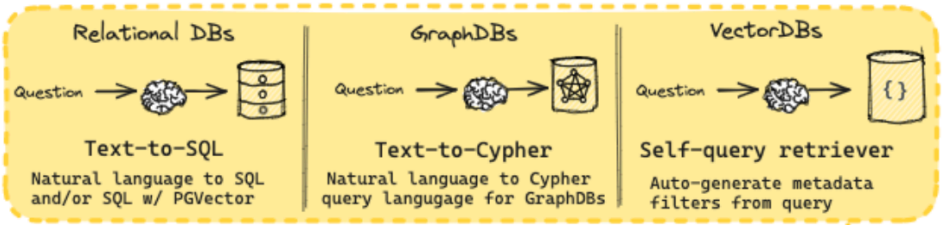


Figure 1: Examples for query construction. [Lance Maring: Langchain]

While *query translation* is more about refining or reformulating the query and doing a similarity retrieval with these new queries, *query constructions* involves an inventive reformulation process, that allows to retrieve of different kind of knowledge databases or additional filtering based on metadata filter.

From Natural Language to SQL

When using relational databases, the process involves converting a textual query into an SQL command. Consider a scenario where a user asks to "find all records of customers who made a purchase last month." Instead of merely translating this sentence, query construction entails interpreting the user's intended meaning and constructing an SQL query that might include filtering by date, joining related tables, and selecting specific fields. This approach ensures that the underlying logic of the user's request is preserved and effectively implemented.

Leveraging Graph Databases with Cypher

Graph databases require a different approach. Instead of generating a standard SQL query, natural language can be transformed into a Cypher query —an essential tool for navigating node and relationship structures inherent in

graph databases. For example, a user might ask for "the shortest path between two related data points." A well-constructed query will reformulate this request into a Cypher query that can efficiently explore the graph, leveraging the unique relational structure of the database to yield precise results.

Enhancing Document Databases Through Metadata Filtering

Document databases often contain unstructured data that can be further enriched with metadata. In these systems, query construction involves extracting or even generating metadata filters from the initial natural language input. Instead of operating on the entire dataset, a constructed query intelligently pre-filters documents based on metadata attributes relevant to the user's request. This two-step process—first filtering documents and then retrieving data from the filtered subset—optimizes performance and improves result accuracy.

Tag-based Filtering

We improve our document store by allowing pre-filtering of the documents based on tags. In addition to a natural language query the document store now also accepts a logical expression on the presence of tags for the documents- Tag names must start with a lowercase letter and can include lowercase letters, digits, hyphens, and underscores. Users can use the logical AND operator (`&`), the OR operator (`|`), and the NOT operator (`!`) to build expressions, and parentheses are allowed for grouping to control operator precedence.

Here are a few examples of supported tag expressions:

- `tag1 & tag2` : Matches when both `tag1` and `tag2` are present.
- `tag1 | tag2` : Matches when either `tag1` or `tag2` (or both) is present.
- `!tag1` : Matches when `tag1` is absent.
- `tag1 & (tag2 | tag3)` : Matches when `tag1` is present and either `tag2` or `tag3` is present.

- `!(tag1 | tag2)` : Matches when neither `tag1` nor `tag2` is present.

For document store is modified like this:

```
def evaluate_many(expression: str, tags: List[Set[str]]) ->
List[bool]:
    """
    Evaluate a boolean expression with the given set of tags.

    Args:
        expression: A boolean expression string
        tags: Set of tag names that are considered True

    Returns:
        bool: Result of evaluating the expression

    Raises:
        ParseError: If the expression is invalid
    """
    ... # Code not shown here for better readability


class TagDocumentStore(BasicDocumentStore):

    @propery
    def all_tags(self) -> Set[str]:
        tags = set()
        for doc in self.documents.values():
            tags |= doc.tags
        return tags


    def search(self, query: str, tag_expression: str | None = None,
top_k: int = 10) -> List[Document]:
        """Search for similar documents using cosine similarity"""

        if tag_expression:
            # Filter documents by tag expression
            mask = evaluate_many(tag_expression, [doc.tags for doc
in self.documents.values()])
            doc_ids = [doc_id for doc_id, m in
zip(self.documents.keys(), mask) if m]
        else:
            # If no expression use use all documents -> fallback to
basic doc store
            doc_ids = list(self.embeddings.keys())

        if not doc_ids:
            return [] # Stop if no docs left

        # Get embedding for query
```

```
        query_embedding = self._get_embedding(query)

        # Stack all document embeddings into a matrix
        doc_embeddings = np.vstack([self.embeddings[doc_id] for
doc_id in doc_ids])

        # Normalize query embedding
        query_norm = query_embedding /
np.linalg.norm(query_embedding)

        # Normalize all document embeddings (row-wise)
        doc_norms = np.linalg.norm(doc_embeddings, axis=1,
keepdims=True)
        normalized_docs = doc_embeddings / doc_norms

        # Calculate dot product (cosine similarity since vectors are
normalized)
        similarities = np.dot(normalized_docs, query_norm)

        # Get indices of top-k similarities
        top_indices = np.argsort(-similarities)[:top_k]

        # Map back to document IDs and return documents
        top_doc_ids = [doc_ids[i] for i in top_indices]
        return [self.documents[doc_id] for doc_id in top_doc_ids]
```

The example below demos the filter expression construction using and LLM:

```
[
 {
    "available_tags": ['journal_article', 'magazine_feature',
'novel', 'technology', 'innovation', 'whitepaper', 'blog_post'],
    "question": "Which breakthroughs are transforming artificial
intelligence?"
  },
  {
    "available_tags": ['scientific_paper', 'blog_post',
'documentary', 'biology', 'genetics', 'news_article',
'research_report'],
    "question": "How do genetic mutations contribute to evolutionary
change?"
  },
  {
    "available_tags": ["research_paper", "feature_story", "novel",
"environment", "climate", "policy_brief", "news_article"],
    "question": "How are recent policy changes impacting climate
action?"
  }
]
```

```
CONSTRUCTION_PROMPT = """You are given a list of available tags and
a natural language question.
Your task is to generate a filter expression that captures the
intent of the question using the provided tags.

available_tags: {available_tags}
question: {question}

The filter expression must follow these rules:
1. **Tag Names:**
   - Only use tags from the list of available tags provided.

2. **Operators:**
   - Use the **AND** operator (`&`) to require that multiple tags
must be present.
   - Use the **OR** operator (`|`) to indicate that at least one of
several tags can be present.
   - Use the **NOT** operator (`!`) to exclude a tag from the
results.

3. **Grouping:**
   - Parentheses `(` and `)` may be used to group expressions and
control the order of evaluation.
```

```
4. **Interpretation:**
   - Analyze the intent of the question and select tags that best
capture that intent.
   - Exclude tags that are not relevant to the question by using the
NOT operator.
   - The expression should be as concise as possible while still
being semantically correct.

**Example:**

- **Available Tags:** `scientific_paper`, `new_article`, `novel`,
`physics`, `chemistry`, `conference_paper`
- **Question:** What causes gravitational lensing in space?
- **Expected Output:** `scientific_paper & physics & !novel`

Using the rules above, generate a reasonable filter expression for
any given list of available tags and question.
Be careful to not create a too strict expression that would exclude
relevant content.
Your answer should be a valid filter expression. No additonal
explanations!"""

from openai import OpenAI

client = OpenAI()

for example in examples:
    response = client.chat.completions.create(
        model='gpt-4o-mini',
        messages=[
            {"role": "user", "content":
CONSTRUCTION_PROMPT.format(**example)}
        ]
    )
    expression = response.choices[0].message.content.strip()
    print(f"Question: {example['question']}")
    print(f"Expression: {expression}\n")

# Output:
# > Available Tags: ['journal_article', 'magazine_feature', 'novel',
'technology', 'innovation', 'whitepaper', 'blog_post']
# > Question: Which breakthroughs are transforming artificial
intelligence?
# > Expression: `technology & innovation & (journal_article |
whitepaper | blog_post)`
# >
# > Available Tags: ['scientific_paper', 'blog_post', 'documentary',
```

```
'biology', 'genetics', 'news_article', 'research_report']
# > Question: How do genetic mutations contribute to evolutionary
change?
# > Expression: `genetics & biology & scientific_paper`
# >
# > Available Tags: ['research_paper', 'feature_story', 'novel',
'environment', 'climate', 'policy_brief', 'news_article']
# > Question: How are recent policy changes impacting climate
action?
# > Expression: `policy_brief & climate`
```

GraphRAG: Combining Knowledge Graphs with Retrieval-Augmented Generation

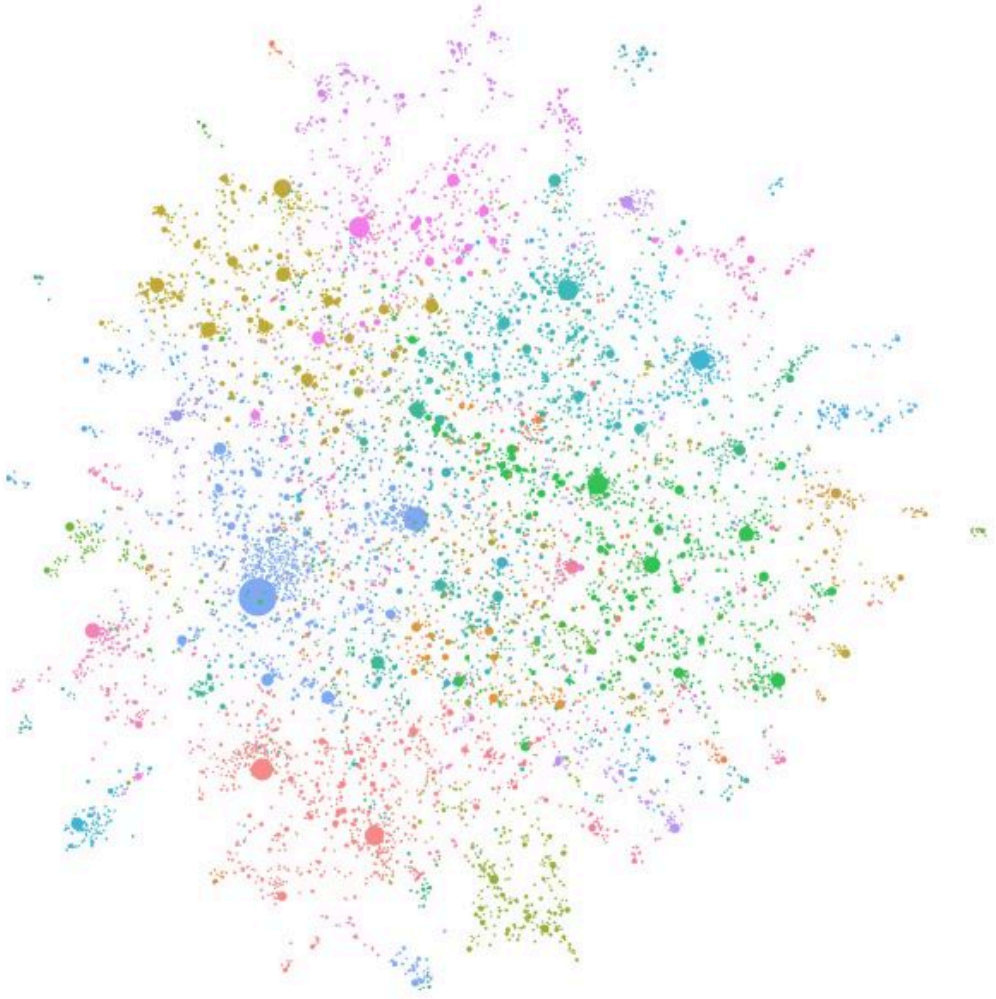


Figure 1: An LLM-generated knowledge graph built using GPT-4 Turbo. [Microsoft/GraphRAG]

GraphRAG (Graph + Retrieval-Augmented Generation) is an approach that enriches the standard RAG technique by introducing a knowledge graph into the retrieval process. In traditional RAG, a language model's outputs are augmented with relevant text snippets fetched via vector similarity search on

documents. GraphRAG instead builds a *structured knowledge graph* from the data and uses it to retrieve and organize information, enabling more complex reasoning than plain text matches. This approach is important because it helps connect disparate pieces of information and maintain context, allowing language models to answer multi-hop questions or summarize large content more accurately than with vector search alone. In fact, graph-enhanced RAG has been shown to significantly boost answer precision – one benchmark saw correctness improve from ~50% with standard RAG to over 80% using a graph-based approach. By mirroring how humans naturally link related facts, GraphRAG provides a more explainable and comprehensive framework for grounding generative AI in knowledge.

Fundamentals of GraphRAG

What is GraphRAG and Why Use Graphs in RAG?

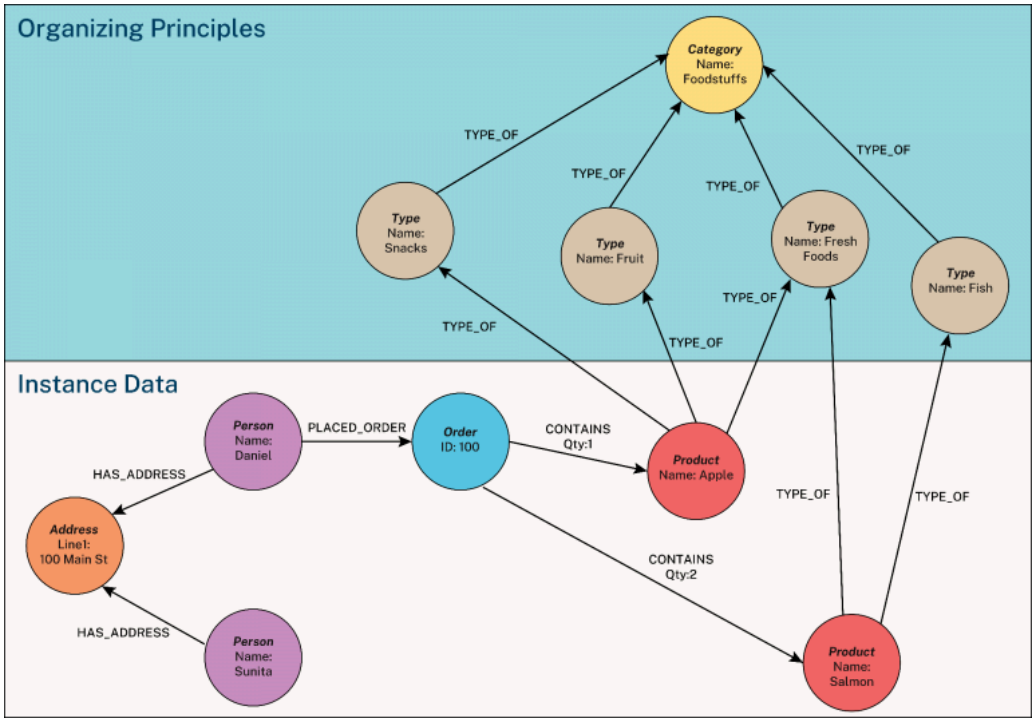


Figure 2: Knowledge-Graph Example

GraphRAG is essentially RAG empowered by a knowledge graph. A **knowledge graph** is a structured representation of facts in the form of nodes (entities or concepts) and edges (relationships). Integrating a graph into RAG yields a *hierarchical, structured retrieval* pipeline instead of relying solely on flat similarity search. The motivation for this is to overcome limitations of baseline RAG (vector-only retrieval). Standard RAG often struggles when a question requires **connecting multiple clues** or **traversing indirect relationships** scattered across documents. It may also falter at **holistic understanding** – for example, summarizing a lengthy report or synthesizing insights spread over many documents. GraphRAG addresses these issues by explicitly modeling relationships:

- *Connecting the dots:* GraphRAG can link disparate information through shared entities. This is crucial for answering complex questions that involve multi-hop reasoning (e.g. *"Which researcher worked on project X and what organization funded that project?"*), where a direct vector search might miss the needed link.
- *Preserving context and meaning:* Unlike pure embeddings which might lose nuances of how facts relate, a graph retains the **rich relationships** and context from the source data. This structured context helps the system stay aligned with the way humans reason about connected information.

By leveraging graph structures, GraphRAG provides more precise grounding for the LLM. Microsoft researchers found that building a knowledge graph from an input corpus and using it to augment prompts led to substantial improvements on complex Q&A tasks, outperforming earlier RAG approaches on private datasets. In practice, graphs bring *explainability* (you can trace which nodes/edges led to an answer) and often better accuracy – one study showed a **35% improvement in answer precision** over vector-only retrieval by using graphs to capture data relationships.

How GraphRAG Works (Technical Deep Dive)

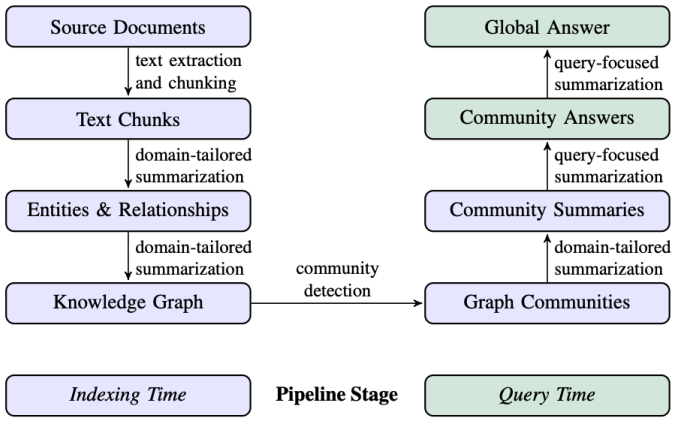


Figure 3: GraphRAG Pipeline. [Edge et al.]

Under the hood, GraphRAG involves a pipeline of steps to transform raw text data into a knowledge graph and then use that graph at query time. We can break the process into two major phases: an **indexing (knowledge graph construction)** phase and a **query (retrieval and generation)** phase.

1. Indexing Phase: Building the Knowledge Graph – This phase takes the input corpus (documents) and produces a structured graph representation of its knowledge:

- **Text Unitization and Extraction:** The documents are first split into manageable text units (e.g. paragraphs or sentences) for analysis. From each unit, the system extracts key **entities** (people, organizations, places, etc.), **relationships** between those entities, and important **facts or claims**. This extraction can be done using NLP techniques or even with the help of an LLM. For example, GraphRAG can employ a large model like GPT-4 to read text and identify entities and relations, essentially having the LLM "generate" the knowledge graph structure. The result is a preliminary knowledge graph where nodes represent entities and edges represent relationships found in the text. (For instance, if the text says "Alice works at OpenAI," the graph would have a node for **Alice**, a node for **OpenAI**, and an edge **works_at** connecting Alice to OpenAI.)

- **Graph Structuring and Community Detection:** Once the raw graph of entities and relations is built, GraphRAG often applies graph algorithms to

organize and enrich it. A key step is performing **hierarchical clustering** on the graph to find communities of related entities. Using techniques like the Leiden algorithm, the graph's nodes are grouped so that highly connected or semantically related entities form clusters (communities). This yields a hierarchy: individual entities connect to each other, communities of entities emerge, and possibly higher-level groupings beyond that. Each community can be thought of as a thematic group or subtopic within the data.

- **Summarization of Communities:** To complement the graph structure, GraphRAG generates **summaries for each community** (and sometimes subgraphs or important nodes). Starting from the lowest level, it summarizes groups of related facts into higher-level descriptions. For example, if one community of the graph centers around a specific project (say GraphRAG as a project), the system might produce a summary of what that project is and its key related entities (people involved, date, etc.). These summaries provide a natural language overview that can be used later to answer broad questions. By building summaries *bottom-up* for each community, the system gains a concise representation of the entire dataset's knowledge at multiple levels of granularity. This is invaluable for holistic queries (like "Give me an overview of topic X in the documents") because the LLM can be fed a pre-written summary instead of raw scattered details.

2. Query Phase: Graph-Augmented Retrieval and Generation – After indexing, the knowledge graph and associated summaries are ready to be used for answering questions or assisting the language model at runtime:

- **Graph-Based Retrieval:** When a query comes in, GraphRAG uses the knowledge graph to retrieve relevant information for that query. Instead of (or in addition to) doing a pure vector similarity search over documents, the system can *navigate the graph*. There are multiple retrieval modes, each suited to a type of question:
    - **Global Search:** If the question is broad or holistic (e.g. *"What is the overall status of project GraphRAG?"*), the system leverages the **community summaries**. It might fetch the summary of the most relevant community (or communities) related to the query. Because

those summaries condense the key points of an entire cluster of documents, they help answer high-level questions that span many pieces of data.
    - **Local Search:** If the question is about a specific entity or requires specific facts (e.g. *"Who does Alice work for?"* or *"What did Alice create?"*), GraphRAG performs a targeted retrieval around that entity. It will find the node corresponding to the main entity (Alice) and "fan out" to its neighbors in the graph (Welcome - GraphRAG). In practice, this means collecting all directly connected facts about Alice (her workplace, her projects, colleagues, etc.) as relevant context. This local neighborhood of the graph provides focused information exactly about the subject of the query.
    - **DRIFT Search:** This mode (short for *Directed RetrIevel with Full Text?*) is essentially a hybrid of local and global contexts (Welcome - GraphRAG). For an entity-specific question, it retrieves the local neighborhood of that entity **plus** any higher-level community context. So if we ask a specific question about Alice but also want contextual info about the bigger picture (maybe the team or project she's part of), DRIFT will include both the facts directly linked to Alice and the summary of the community cluster that Alice belongs to. This gives the language model both detailed facts and background context.

  Regardless of mode, the output of retrieval is a collection of structured facts (from the graph) and/or summaries that are pertinent to the query. Because the graph inherently preserves relationships, the retrieved data often includes the *connections* between facts that the model will need for reasoning.

- **Augmenting the LLM Prompt:** The final step is to feed the retrieved information into the large language model to generate the answer (the "Generation" part of RAG). In GraphRAG, the prompt given to the LLM is augmented with the graph-derived context. For example, if the question is, *"What did Alice create, and where is the company she works at based?"*, the system might retrieve the facts: "Alice **created** GraphRAG" and "Alice **works_at** OpenAI; OpenAI **based_in** San Francisco." These facts (perhaps converted into a readable sentence or list) would be added to the LLM's

input, so that the model has the necessary knowledge to answer in detail. The prompt might also include a community summary if relevant (e.g., a summary about the project GraphRAG or about OpenAI, if the question were broader). With this enriched prompt, the LLM can generate a coherent answer that directly uses the provided facts.

It's worth noting that careful prompt design or even prompt tuning is often used to ensure the LLM uses the graph information effectively. The GraphRAG approach may involve iterating on the prompt format or instructions to maximize accuracy.

Conclusion

GraphRAG represents an evolution of retrieval-augmented generation that incorporates the rich structure of knowledge graphs. By doing so, it allows AI systems to handle complex, multi-hop queries and large knowledge bases with improved accuracy and explainability. The fundamental principles of GraphRAG – extracting structured knowledge, organizing it hierarchically, and leveraging graph-based context retrieval – help bridge the gap between unstructured text and the way humans naturally connect information. In summary, GraphRAG works by **extracting a knowledge graph** from text, **organizing it** (with clusters and summaries), and then **using the graph** to retrieve relevant facts and context which are supplied to an LLM for answer generation. This structured pipeline enables the system to handle complex queries that would be challenging for conventional RAG, providing more accurate and context-rich answers.

- Microsoft/GraphRAG
- Improving Retrieval Augmented Generation accuracy with GraphRAG | AWS Machine Learning Blog

AgenticRAG: A Beginner's Introduction and Python Proof-of-Concept

Introduction to AgenticRAG

Retrieval-Augmented Generation (RAG) is an AI technique that combines a generative language model with an external knowledge source to produce more accurate, up-to-date answers. Instead of relying solely on static training data, a RAG system retrieves relevant documents or facts from a knowledge base and provides them as additional context to the language model. This allows the model to generate responses grounded in current or domain-specific information without requiring extensive fine-tuning.

AgenticRAG (Agentic Retrieval-Augmented Generation) takes this idea a step further by introducing **AI agents** into the RAG pipeline. In an AgenticRAG system, the language model doesn't just passively receive retrieved facts – it actively **plans and controls the retrieval process**. By blending agentic AI's autonomous decision-making with RAG's dynamic information retrieval, AgenticRAG makes AI systems more independent, flexible, and capable of tackling complex tasks. In practical terms, an AgenticRAG system empowers the AI to **decide what to search for, which sources to query, and how to refine its approach iteratively** until it finds the information needed. This added *"agentic"* capability can dramatically increase the adaptability and accuracy of the system's answers. It transforms the AI from a reactive tool that answers questions into a proactive problem-solver that can figure out *how* to answer the question.

Key Concepts of AgenticRAG

How is AgenticRAG different from traditional RAG? The key lies in the agent-like behavior that introduces autonomy and iterative reasoning into the retrieval and generation process. Below are the core concepts and capabilities that distinguish AgenticRAG:

- **Autonomous and Proactive Behavior:** AgenticRAG leverages AI agents (often powered by large language models with tool-use abilities) that can make decisions about what actions to take next. Instead of following a single fixed retrieve-then-answer step, the agent can **plan multiple steps**. For example, it might decide: *"First, search the database for topic X. If that isn't enough, call an API for more data. Then, summarize the findings."* This autonomy means the system is not limited to reacting to the initial query – it can proactively **route queries, call tools, or trigger new searches** based on the situation.

- **Iterative Query Refinement (Feedback Loop):** A hallmark of AgenticRAG is its ability to refine its results through iteration. The agent can assess whether the initial retrieved information fully answers the question. If not, it will **adjust the query or take additional steps** to get closer to the answer. This creates a feedback loop where each round of retrieval and generation can be evaluated and improved upon. Over multiple iterations, the agent learns what information is missing and zeros in on the needed details. This iterative reasoning makes the system more resilient to complex queries – it can break a problem down into sub-queries and solve it step by step.

- **Dynamic Multi-Source Retrieval:** Traditional RAG typically connects a model to a single data source or knowledge base. In contrast, AgenticRAG offers greater **flexibility** by pulling data from **multiple sources or tools** as needed. An agentic system might retrieve from an internal database, then augment that with information from the web or a specialized API in the same session. This multi-source capability means the AI can gather a more comprehensive pool of knowledge. It's especially useful for complex tasks (for instance, a research assistant agent might query a company document repository *and* public articles to answer a question). By having

access to diverse sources, the agent can cross-check facts and gather a richer context before generating a response.

- **Contextual Understanding and Accuracy:** Because the agent actively filters and selects relevant information, AgenticRAG tends to produce more contextually relevant answers. The agent can **orchestrate retrieval, filter out irrelevant data, and focus on what truly matters to the query**. Moreover, by iterating and self-evaluating, it can catch mistakes or gaps in the knowledge and correct them in subsequent steps. This results in **improved accuracy** over a one-shot retrieval approach. In essence, AgenticRAG systems excel in scenarios requiring high precision and adaptability, using their agent-driven workflow to ensure the final answer is well-supported by the evidence retrieved.

In summary, AgenticRAG combines retrieval capabilities with decision-making and planning skills, enabling systems to handle complex inquiries through a human-like approach of gathering information, refining searches when needed, and assembling thoughtful answers. While offering greater flexibility and accuracy, this comes with increased computational requirements and design complexity that can affect performance and costs. Nonetheless, for many applications, the benefits of this self-improving system significantly outweigh these considerations.

By empowering AI with autonomy, planning, and iterative refinement abilities, we create systems that not only access information but reason about how to use it. This approach opens the door to more **robust and intelligent assistants** that actively gather and refine information like human researchers. As supporting libraries and frameworks continue to evolve, implementing these advanced workflows is becoming increasingly accessible to developers, moving us closer to AI systems that are not just knowledge-aware but **capable of self-directed reasoning** in pursuit of optimal solutions.

- Agentic Retrieval-Augmented Generation: A Survey on Agentic RAG
- What is Agentic RAG? | IBM
- OpenAI Deep Research: How it Compares to Perplexity and Gemini
- What is Agentic RAG? | Weaviate
- Agentic RAG turns AI into a smarter digital sleuth | IBM

- Agentic RAG vs. Traditional RAG: The Future of AI Decision-Making | Fluid AI

Deep Research and the Rise of AI-Driven Research Assistants

OpenAI's new Deep Research feature represents a significant step forward in AI-driven research assistance, and it isn't happening in isolation. A wave of open-source and third-party initiatives – from platforms like Perplexity to community-built agents – are offering similar capabilities for in-depth information gathering. Central to these advancements is the concept of **Retrieval-Augmented Generation (RAG)**, especially in its more autonomous *agentic* form, which enables AI systems to retrieve and synthesize information more effectively than ever.

AI-Powered "Deep Research" – Offloading Complex Research Tasks

AI tools are increasingly transforming how we gather and analyze information, saving hours of manual effort while aiming to ensure comprehensive results. **OpenAI's Deep Research** is an AI-driven research assistant built into ChatGPT that can search the web for in-depth information and generate detailed reports on complex topics. In OpenAI's own words, *"Deep Research frees up valuable time by allowing you to offload and expedite complex, time-intensive web research with just one query"*. This capability essentially turns a single prompt into a multi-step research project executed by the AI.

Unlike standard large language model answers (which rely only on their pre-trained internal knowledge), Deep Research acts more like a **research agent** that actively gathers and processes external data. It can, for example, scour websites, academic papers, and other public sources in real time, then consolidate findings into a coherent report. According to one detailed overview, Deep Research is *"an AI-powered automated research agent"* that:

- **Accesses and synthesizes real-time web data** by browsing online sources

- **Conducts multi-step reasoning** to handle queries requiring deeper context and analysis
- **Generates long-form reports with citations** and detailed explanations of its findings

In practical terms, this means a user can pose a complex question – say, an analysis of market trends or a legal case overview – and the Deep Research agent will break the task into substeps, search the web for relevant information, analyze the content it finds, and produce a structured report complete with source citations. The goal is to deliver a level of insight and thoroughness comparable to a human researcher, but in a fraction of the time. OpenAI notes that Deep Research is particularly good at *"finding niche, non-intuitive information that would require browsing numerous websites"*. By automating these labor-intensive research tasks, such AI tools aim to augment professionals in fields like finance, science, law, and policy, where synthesizing information from many sources is part of the job.

How does Deep Research work? Internally, it combines a powerful OpenAI model with an *"agentic framework"* that guides the model to use tools (like web search and web browsing) in a step-by-step process. It operates in phases: interpreting the query (and clarifying details if needed), searching and scraping information from top results, analyzing and summarizing the gathered data (including tracking which source each fact came from), and finally generating a comprehensive report. This multi-phase approach allows the AI to **iterate and refine** its results – for example, performing additional searches if an initial pass didn't yield enough detail – rather than just answering in one shot. The output is a detailed answer with references, often accompanied by elements like bullet point summaries, tables, or even charts if relevant. All of this can run in a sandboxed environment OpenAI's implementation even allows analyzing data with Python tools during the process.

Open-Source & Alternative Initiatives (Perplexity and More)

OpenAI is not alone in pursuing AI-driven research assistants. In fact, the launch of Deep Research comes amid a broader movement in the AI community to develop similar **"AI research agent"** capabilities, with both open-source projects and other companies jumping in. Notably, **Perplexity AI** – known for its AI-powered answer engine – introduced its own *Perplexity Deep Research* mode around the same time. Perplexity's version offers quick, structured research results with the convenience of a free tier (limited daily queries) for users. This means anyone can try an AI-driven research query on Perplexity and get a concise report with inline citations, demonstrating how smaller players are providing access to these tools outside of OpenAI's ecosystem.

Meanwhile, open-source communities have been rapidly replicating and innovating on these ideas. Impressively, an open-source alternative to OpenAI's Deep Research was released *within a day* of OpenAI's announcement. For example, a team at Hugging Face embarked on a 24-hour sprint to create an]open-source "DeepResearch" agent, combining a freely available LLM with an agent framework to mimic the tool's multi-step web browsing and synthesis behavior. Similarly, a project dubbed **Open Deep Research** quickly gained traction on GitHub, amassing thousands of stars and offering developers a way to run their own research agent with custom settings. These open implementations are highly configurable – one can plug in different models or tweak how the agent searches and how many iterations it runs – highlighting a key benefit of community-driven efforts: flexibility and transparency.

Even the tech giants have parallel efforts. Google introduced its *Gemini Deep Research* around the same time, integrating advanced web research capabilities into their AI (Gemini) platform. While details differ (for instance, Google's approach might integrate with its search engine but could be prone to SEO biases in results, the convergence is clear: many AI providers see "deep research" features as the next frontier.

Humanity's Last Exam

Model	Accuracy (%)
GPT-4o	3.3
Grok-2	3.8
Claude 3.5 Sonnet	4.3
Gemini Thinking	6.2
OpenAI o1	**9.1**
DeepSeek-R1*	9.4
OpenAI o3-mini (medium)*	10.5
OpenAI o3-mini (high)*	13.0
OpenAI deep research**	**26.6**

Figure 1: Benchmark Humanity's Last Exam. [OpenAI]

In January 2025, the AI research community unveiled Humanity's Last Exam (HLE), a benchmark designed to push the limits of machine intelligence. Developed through a collaboration between Scale AI and the Center for AI Safety, HLE features 3,000 rigorously curated questions spanning mathematics, natural sciences, humanities, and interdisciplinary domains. This new evaluation framework was created in response to benchmark saturation in existing tests, aiming to challenge even the most advanced models with tasks that require deep, PhD-level expertise and multimodal reasoning, including the integration of text, images, and symbolic notation.

Early evaluations revealed that state-of-the-art AI systems, including models like GPT-4o and Gemini 1.5 Pro, achieved accuracies below 10%, in stark contrast to the near-perfect performance of human experts. These results underscore critical gaps in current architectures, particularly in expert-level reasoning and cross-modal integration. HLE not only exposes the limitations of current AI but also serves as a catalyst for rethinking training paradigms—shifting the focus from sheer scale to the integration of diverse, expert-centric knowledge and innovative multimodal capabilities. In this extemely challenging benchmark OpenAI's Deep Research was able to come up on top.

- OpenAI – Introducing Deep Research
- OpenAI's 'Deep Research' Aims to Impact Business Intelligence | PYMNTS
- What is Agentic RAG? | Weaviate
- OpenAI Deep Research: How it Compares to Perplexity and Gemini | Helicone AI