Prague University of Economics and Business

Faculty of Informatics and Statistics

INNOVATING LARGE LANGUAGE MODELS (LLMS) WITH FINE-TUNED RETRIEVAL-AUGMENTED GENERATION (RAG): DESIGN AND EVALUATION

BACHELOR THESIS

Study programme: Applied Informatics

Author: Šimon Pivoda

Supervisor: MSc. Richard Antonín Novák, Ph.D.

Prague, May, 2025

Acknowledgment

I wish to express my sincere thanks to my thesis supervisor, MSc. Richard Antonín Novák, Ph.D. His tremendous support and strong guidance throughout the whole process of thesis development contributed to its successful completion.

Next, I would like to express my sincere thanks to the AI and open-source communities. In particular, I am deeply grateful to the many researchers, developers, and contributors who continue working to improve Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG) systems. Their openness in sharing their findings and resources with the wider public was incredibly motivating for me and played a major role in helping me build a strong foundation of knowledge for this thesis.

Lastly, I would like to thank Patrick Lewis and his co-authors for their paper, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.”. Their work helped me grasp the RAG concept clearly and became an important reference throughout my thesis.

Abstrakt

Umělá inteligence (AI) zásadně mění způsob, jakým organizace spravují a využívají informace. Její vliv je patrný nejen v každodenním provozu, ale i při strategickém rozhodování na nejvyšší úrovni. V posledních letech se objevila inovativní metoda zvaná Retrieval-Augmented Generation (RAG), která rozšiřuje schopnosti velkých jazykových modelů (LLM) tím, že jim umožňuje přístup k relevantním a aktuálním datovým zdrojům. Díky tomu mohou firmy efektivněji využívat své interní znalosti v reálném čase a aplikovat tak umělou inteligenci v široké škále scénářů.

Přímé nasazení velkých jazykových modelů v podnikových prostředích však naráží na několik zásadních překážek. Hlavními z nich jsou obavy o bezpečnost dat a riziko jejich úniku. Mnohé organizace proto váhají svěřit citlivá interní data těmto modelům. Další výzvou je přesnost odpovědí – jazykové modely často generují nesprávné nebo smyšlené informace, zejména pokud postrádají specifické znalosti z dané domény.

Metoda RAG přináší účinná řešení těchto problémů. Místo spoléhání se na uzavřené tréninkové datasety umožňuje bezpečný a řízený přístup k interním informacím uloženým ve specializovaných úložištích zvaných vektorové databáze. Tímto způsobem dochází k výraznému snížení rizika úniku dat i chyb v odpovědích. Tato práce se zaměřuje na základní principy a klíčové komponenty architektury RAG, včetně efektivních strategií segmentace dokumentů a vyhledávání prostřednictvím vektorových databází a embeddingů.

V praktické části této bakalářské práce je navržen a otestován komplexní systém RAG zaměřený na automobilová data. Pro ukládání a správu vektorových reprezentací je využita databáze PostgreSQL rozšířená o podporu vektorového vyhledávání (PGVector). Hlavním datovým zdrojem je rozsáhlý dataset z platformy Kaggle. Hodnocení systému probíhá pomocí kombinace expertních dotazů a referenční pravdy a zaměřuje se na přesnost výstupů, jejich ověřitelnost a schopnost minimalizovat halucinace.

Výsledky ukazují, že architektura založená na RAG je efektivní při zvyšování přesnosti konverzačních systémů využívajících interní datové zdroje. Výsledkem jsou přesnější odpovědi, vyšší spolehlivost a výrazné omezení výskytu smyšleného obsahu.

Klíčová slova

AI, embedding, Generativní AI, kontextová data, LLM, prompt engineering, RAG, retriever, vyhodnocování modelů

Abstract

Artificial intelligence (AI) is reshaping the way organizations manage and interact with information. Its influence is visible not only in daily operations but also in critical strategic decision-making. In recent years, an innovative approach known as Retrieval-Augmented Generation (RAG) has emerged, extending the capabilities of large language models (LLMs) by providing access to relevant and up-to-date data sources. This enables companies to better leverage their internal knowledge in real time and apply AI across a wide range of use cases.

However, despite the fast advancements in large language models, their direct deployment within corporate environments often faces major challenges. Main among these are concerns related to data security and the risk of data leakage. As a result, many organizations are hesitant to expose sensitive internal data to language models. Additionally, LLMs often suffer from response accuracy issues, including the generation of incorrect or hallucinated responses when lacking domain-specific knowledge.

The RAG method capably furnishes resolutions for these challenges. Safe as well as governed admittance to internal information that is stored within specialized repositories known as vector databases is permitted instead of mainly depending on closed training datasets. This approach greatly curtails the risk regarding data breaches and inaccuracies. This paper digs into the basic tenets and salient constituents of the RAG architecture. Many efficacious document segmentation and retrieval strategies are implemented via vector databases and embeddings.

In the practical part of this thesis, a complex RAG system is proposed and tested, targeting automotive data. A PostgreSQL database enhanced with vector search support (PGVector) is used to store and manage vector representations. The primary data source is the large dataset from the Kaggle platform. System evaluation is conducted using a combination of expert queries and ground truth. The evaluation aims to assess the system's accuracy, output validation, and its ability to minimize hallucinations.

The results indicate that the RAG-based architecture is effective in enhancing conversational systems with internal data sources, resulting in more accurate responses, increased reliability, and a notable reduction in hallucinated or fabricated content.

Keywords

AI, context data, embedding, Generative AI, LLM, model evaluation, prompt engineering, RAG, retriever

Contents

List of figures

Introduction

The introduction of large language models (LLMs) like GPT-3 (Brown et al., 2020) significantly advanced artificial intelligence, particularly in natural language generation and understanding. GPT-3 demonstrated strong performance across a wide range of tasks, including open-ended text generation, translation, and summarization. However, its success marked only the beginning. Since then, LLM development has progressed rapidly, with models expanding exponentially in size—measured by the number of parameters—and exhibiting increasingly complex and emergent capabilities. More recent models, such as Gemini 2.5 (Google DeepMind, 2025), have continued to push the boundaries, establishing LLMs as foundational tools in both real-world applications and AI research.

While LLMs have unlocked remarkable progress in language understanding and generation, they still present several open challenges that limit their broader adoption in critical applications. Key concerns include factual accuracy, privacy and security of sensitive data, limited interpretability, and difficulties with reasoning, mathematical precision, and attribution of sources. Although many modern models incorporate tools like retrieval systems, browsing capabilities, or parameter-efficient fine-tuning (PEFT) to address some of these limitations, such enhancements are not a complete solution. In addition, issues around energy consumption, licensing, and responsible deployment remain active areas of research and debate. Hallucination is another major issue: LLMs can occasionally generate fluent but factually wrong or misleading results, especially when questioned on subjects outside their training data (Ji et al., 2022).

„However, they still exhibit the hallucination problem. Even worse, since LLMs generate highly fluent and convincing responses, their hallucinations become more difficult to identify, and more likely to have harmful consequences.“ (Ji et al., 2022, p. 36)

This restriction is especially troublesome for applications like financial services, healthcare advising tools, automotive knowledge systems and customer support systems where current and accurate information is essential. Businesses looking to implement AI solutions need systems that can access and reason over current, domain-specific knowledge. They cannot rely only on static models that were trained months or years ago.

Retrieval-Augmented Generation (RAG) is a promising strategy for overcoming these obstacles. RAG architectures combine an LLM with an external information retrieval system, as opposed to relying only on a model's internalized knowledge. When a query is presented to the model, it first searches a knowledge base for pertinent documents before producing a response based on the content it has found (Lewis et al., 2020). This hybrid approach guarantees that the outputs are based on current, verifiable sources and lessens hallucinations.

„Crucially, by using pre-trained access mechanisms, the ability to access knowledge is present without additional training. “ (Lewis et al., 2020, p. 3)

Companies wanting to enhance chatbot accuracy or internal database-driven assistants are finding this strategy more and more appealing. Rather than spending a lot of money retraining massive models with fresh data, companies can just keep their knowledge bases and use retrieval methods to dynamically augment the LLM's replies. RAG not only increases accuracy but also greatly increases the model's trustworthiness and adaptability in practical environments.

This paper emphasizes Retrieval-Augmented Generation, investigates its architectures, optimization strategies, and vital function in overcoming static LLMs' constraints in domain specific expert use cases.

Problem Statement and Objectives

Even though Large Language Models (LLMs) have achieved significant success in producing and understanding natural language, they are severely limited because, once trained, their inherent knowledge becomes unchangeable and eventually stale over time. Since LLMs are usually trained on fixed datasets, they cannot incorporate newly emerging information or domain-specific knowledge without extensive and expensive retraining.

Moreover, LLMs produce intelligent but factually incorrect or misleading outcomes and are prone to hallucinations when answering questions outside the scope of their training data (Ji et al., 2022).

Practical applications face a significant difficulty in relation to current and exact information. For example, healthcare, financial advising, or customer support all show this particular knowledge is missing. Companies and institutions want artificial intelligence systems that can logically flow over current data. Instead, one should not rely just on static model parameter values.

The following main research question of this thesis is:

1. How can Retrieval-Augmented Generation (RAG) be used to enhance a general-purpose Large Language Model's performance as well as factual correctness on domain-specific queries?

Rather than retraining the model itself, this paper builds a RAG pipeline augmenting a general-purpose LLM by accessing external, domain-specific information during inference time. The model fetches pertinent informations from a specified database and adds them to its process of producing responses (Lewis et al., 2020).

In this work, we will proceed to answer the research questions according to the following procedure:

- **Design and implement a RAG pipeline**. This conduit allows a pre-trained LLM (e.g., GPT-4o-mini) to dynamically access domain-specific information kept in a SQL database via vector search.

- **Apply the pipeline to an automotive domain dataset** (Auto Market Dataset) to retrieve relevant car market information.
- **Reduce the occurrence of hallucinated or inaccurate responses** by grounding the model's outputs in retrieved, up-to-date evidence.
- **Measure response improvements** if factual accuracy and relevance improve via quantitative measures, such as retrieval exactitude and factual correctness ratio.

Via these outputs, the thesis attempts to show retrieval augmentation can substantially render existing LLMs more factually reliable as well as domain adaptable, absent the need for retraining or else fine-tuning.

Scope and Delimitation

This thesis focuses on applying and assessing Retrieval-Augmented Generation (RAG) method to enhance the factual correctness inside a general-purpose Large Language Model (LLM) at the moment domain-specific queries are addressed. The RAG pipeline's design integrates the extraction from a structured SQL database filled with entries derived from the Car Specification Dataset (Ashrafi, 2023)

The scope of the project is defined as follows:

- **Text Modality Only:** The system exclusively operates on textual data. No processing of images, audio, or multimodal data is considered.
- **Language Focus**: All retrieval and generation are conducted in **English**. The exploration of multilingual RAG is left for future work.
- **Domain-Specific Experimentation**: The retrieval database is constructed from automotive market data. While the RAG methodology is generalizable, experiments and evaluations are confined to the automotive sector.
- **Use of Existing LLMs**: The work leverages **pre-trained, publicly available LLMs** (e.g., OpenAI models) without any further fine-tuning or domain-specific retraining.
- **Dense Retrieval with pgvector**: Vector search is implemented using **pgvector**, an extension for PostgreSQL that enables similarity-based retrieval based on dense embeddings.
- **Prototype Implementation**: The focus is on building a research prototype to demonstrate feasibility and to evaluate factual improvements. Full production deployment, including large-scale optimization, API engineering, and user-facing integration, is beyond the scope.
- **Basic RAG Pipeline**: The pipeline uses standard retrieval-augmented techniques without advanced multi-hop retrieval, reinforcement learning, or multimodal integration.

This thesis emphasizes a case study and demonstrates how retrieval augmentation works to enhance the accuracy, relevance, and dependability of results produced by large language models. Practical experimentation and assessment underline the promise of RAG-based

systems. RAG-based systems can help to address some LLM restrictions and allow for better adaptation of some domain-specific uses.

Methodology Overview

This work employs a mixed-methods approach combining a Systematic Literature Review (SLR), practical system development, and experimental evaluation to investigate techniques for enhancing the factual dependability of Large Language Models (LLMs) via Retrieval-Augmented Generation (RAG). Theoretical foundations were established through a structured literature review based on PRISMA 2020 guidelines, while subsequent system design and validation were methodically aligned with the insights obtained.

Systematic Literature Review

Objectives

The literature review aimed to explore the theoretical and technological background related to:

- Transformer architectures and LLMs
- Fine-tuning and prompting
- Retrieval-Augmented Generation (RAG)
- Knowledge grounding and semantic retrieval

Search Strategy

A broad and iterative search strategy was employed. Structured searches were conducted across leading academic databases, including:

- IEEE Xplore
- arXiv
- Google Scholar

To complement academic sources, general web search engines (e.g. **Google**) were used to locate:

- Technical blogs and engineering posts (e.g. OpenAI, Cohere, LangChain)
- Implementation documentation
- White papers and community best practices

A diverse range of keywords and combinations was applied to capture various dimensions of the research problem. Terms included but were not limited to:

- “Large Language Models”, “LLMs”, “Transformer”
- “fine-tuning”, “prompt engineering”, “zero-shot learning”

- “Retrieval-Augmented Generation”, “RAG pipeline”, “knowledge grounding”
- “embedding models”, “semantic search”, “vector database”

Boolean operators were used to construct queries (e.g., "LLM" AND "retrieval system"). Searches were refined iteratively based on relevance and scope, and synonyms were included where applicable.

While peer-reviewed literature was prioritized, selected non-academic sources were included when they offered original insights, real-world implementations, or were authored by recognized experts in the field.

Inclusion and Exclusion Criteria

To ensure relevance and quality, studies were included if they:

- Were peer-reviewed or widely cited preprints,
- Directly addressed LLMs, RAG, or related topics.

Studies were excluded if:

- They focused on unrelated AI subfields (e.g. image processing),
- Lacked sufficient technical detail,
- Were not accessible in full text.

Screening and Selection Process

A total of 164 records were initially identified through targeted searches in academic databases (arXiv, Google Scholar, IEEE Xplore) and manual collection. After removing 72 duplicates, 92 records were screened by title and abstract. Of these, 43 full-text articles were assessed for eligibility, and 70 studies were finally included in the review. The selection process followed the PRISMA 2020 guidelines and focused on works related to transformer architectures, fine-tuning techniques, and retrieval-augmented generation.

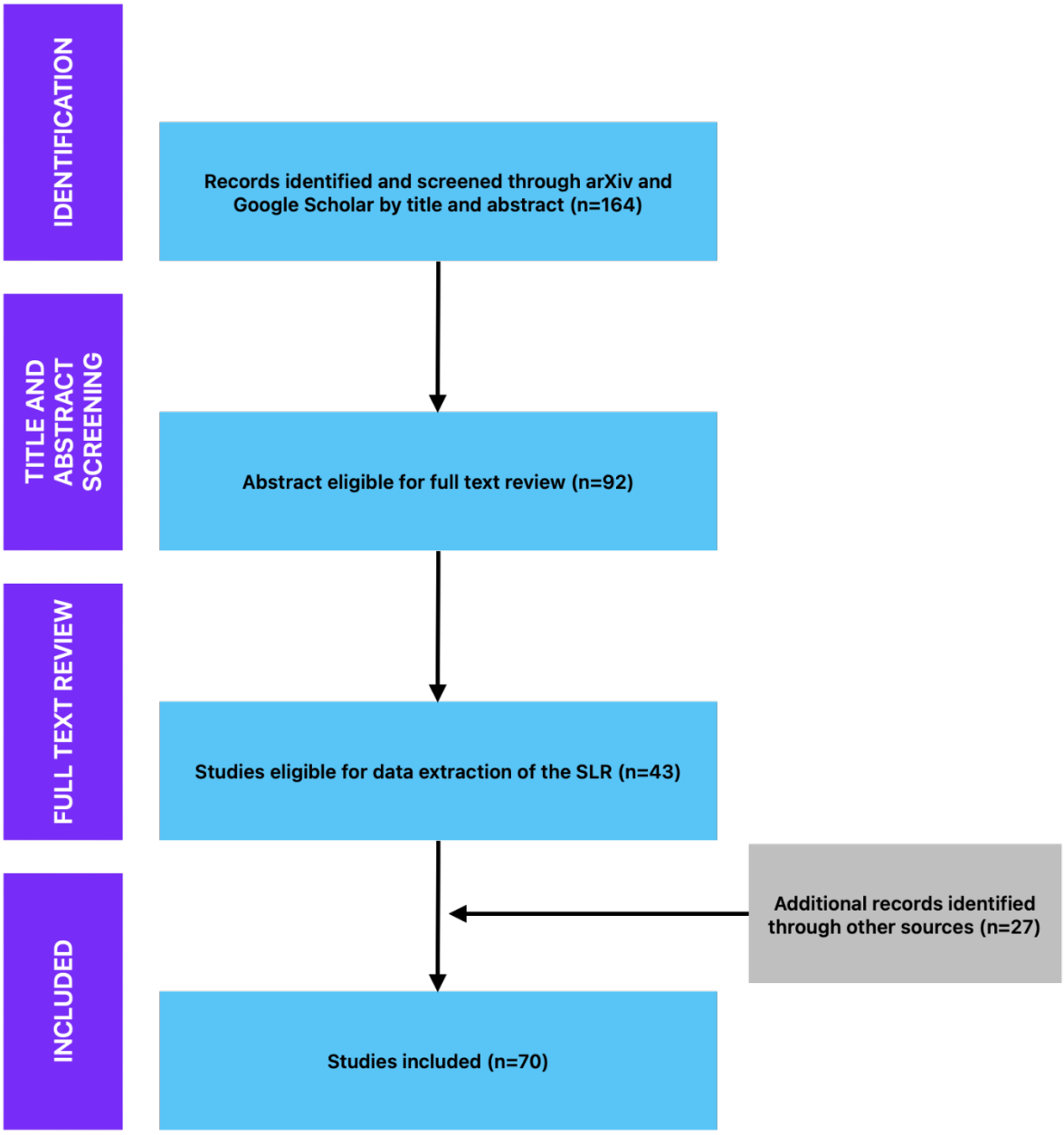


Figure 1 Prism schema (source, author)

The literature study established the conceptual foundation for the design and evaluation phases of this research.

System Design and Experimental Evaluation

Drawing on the insights gathered, a practical system was subsequently designed, implemented, and assessed through the following steps:

1. **Data Preparation**
    - The "Car Specification Dataset" dataset was curated.
    - Entries (make, model, year) were transformed into textual descriptions.
    - Texts were embedded into dense vector representations using OpenAI's embedding API.

2. **Retrieval System Construction**
    - A dense vector retrieval mechanism was built using *pgvector*, an extension for PostgreSQL supporting efficient similarity search.
    - User queries were embedded into the same vector space, and the top-k most semantically similar entries were retrieved.

3. **Integration of the RAG Pipeline**
    - OpenAI's *GPT-4o-mini* model was used for generation tasks.
    - Retrieved documents were dynamically injected into the prompt as additional context.
    - This approach grounded generated responses in domain-specific external information.

4. **Baseline and RAG Output Generation**
    - Two different outputs for each query were generated.:
        - Baseline output: LLM without retrieval augmentation,
        - RAG output: LLM with retrieved context injected.

5. **Evaluation Metrics**

The generated outputs were evaluated using quantitative method:

- Factual accuracy: Measuring correctness against ground truth data,

Eexpert-generated ground truth questions were used as a reference. Each system-generated answer was assessed as either factually correct or incorrect based on comparison with the verified reference answer. This binary evaluation allowed for objective measurement of factual accuracy.

Ethical Considerations

Thesis strictly followed the highest moral guidelines. To a certain extent, the datasets guaranteed adherence to data protection regulations because they were all synthetic or publicly available. No sensitive or private information was used. The evaluation procedure was conducted in a transparent way. It adhered to the principles of fairness, consistency, and impartiality.

Methodological Limitations

Despite careful design, several limitations are acknowledged:

- The domain specificity of the automotive dataset may restrict generalizability to other domains.
- System performance is influenced by the capabilities and limitations of *GPT-4o-mini* at the time of the study.

This approach provides a robust framework for exploring the potential of retrieval-augmented generation to enhance the factual reliability of LLM outputs by methodically integrating theoretical research, real-world system development, and empirical evaluation.

1 Theoretical Background and Literature Review

This chapter investigates foundational ideas and reviews relevant studies that support Retrieval-Augmented Generation (RAG), with a focus on modern large language models (LLMs). It starts from the foundations of LLMs. Then there comes an overview that is of the transformer architecture and also prompting techniques. Fine-tuning methods show the specific tasks models adapt to. The chapter concludes along with an in-depth look at RAG because it improves LLMs through external knowledge retrieval, embedding techniques, and vector search systems like pgvector.

1.1 Foundations of Large Language Models

This section presents foundational definitions and essential concepts underlying contemporary large language models (LLMs). It further explores major technological breakthroughs and critical innovations that have driven significant advancements in the field. The discussion continues by detailing model architectures, including their distinctive characteristics and key functionalities. Finally, the section identifies and critically evaluates the challenges and explores emerging opportunities for further development and application of LLMs.

1.1.1 Definition and Basic Concepts

Large Language Models (LLMs) are deep learning systems capable of processing and generating human-like text by learning statistical patterns from massive text corpora (Brown et al., 2020). Their core function is to estimate the probability of upcoming tokens in a sequence, based on the surrounding context (Bengio et al., 2003). In autoregressive LLMs, this involves predicting the most likely next token, which is then appended to the input, allowing the model to iteratively generate coherent and context-aware text (Radford et al., 2019). For instance, when given a partial phrase such as *"The cat was on the,"* the model calculates the probabilities of all possible continuations and selects or samples the most probable one—like *"mat."* This sequence is repeated until the desired length or semantic unit is achieved.

This step-by-step mechanism is illustrated in Figure 2, where each prediction not only generates a word but also updates the model's contextual understanding. LLMs can continue this process to produce arbitrarily long sequences, maintaining semantic coherence over extended outputs.

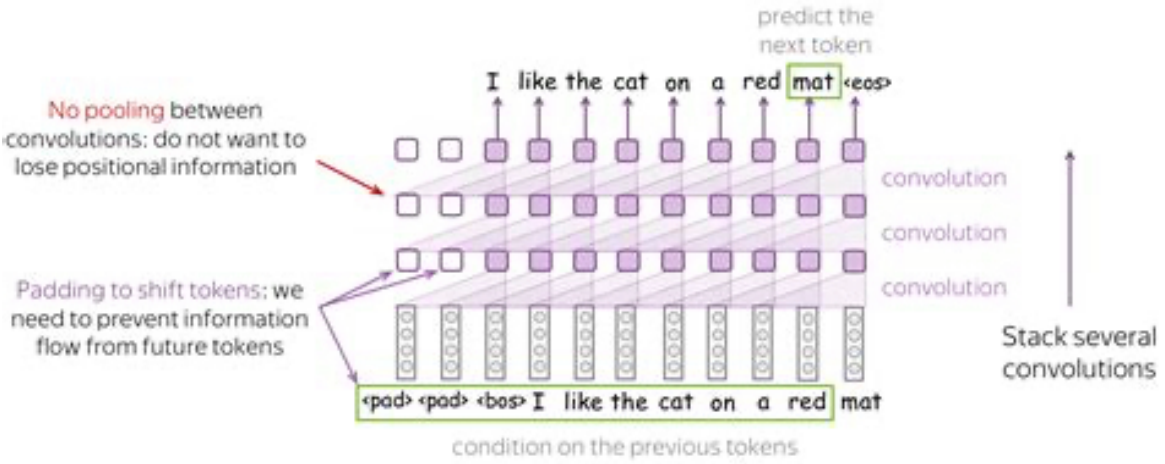


Figure 2 Visualization of language modeling, prediction of the next word based on the previous context (Voita, n.d)

Most modern LLMs are built on Transformer architectures (Vaswani et al., 2017), which utilize attention mechanisms to capture dependencies across all tokens in an input. Unlike earlier architectures, Transformers avoid recurrence and instead rely on self-attention layers, allowing for high parallelization and scalability when training on vast datasets.

The first letter in LLM („large“) refers to the high count of trainable. For example, such as the 175 billion in GPT-3 (Brown et al., 2020) or 540 billion in Google's PaLM (Chowdhery et al., 2022). These parameters, adjusted during training, encode linguistic patterns and factual knowledge.

LLMs are typically trained via self-supervised learning methods where they predict masked or next tokens from unlabelled corpora (Devlin et al., 2019; Radford et al., 2019), eliminating the need for manual annotations. The result is general-purpose models that can later be specialized through either prompting or fine-tuning (Raffel et al., 2020).

A defining capability of LLMs is their generalization power. Once they are pre-trained, they can translate, summarize, answer questions along with generation of dialogue. They are able to perform a range of downstream tasks., often without additional training (Radford et al., 2019; Brown et al., 2020). Techniques like zero-shot, few-shot, and in-context learning enable users to solve new tasks simply by providing instructions or examples in the prompt (Brown et al., 2020; Wei et al., 2022b).

In summary, LLMs capture the statistical structure of language through large-scale self-supervised training. Their performance improves with increased data and model size (Kaplan et al., 2020; Brown et al., 2020), enabling them to produce fluent, relevant, and sometimes even creative outputs.

1.1.2 Key Breakthroughs and Innovations

Large language models (LLMs) have advanced rapidly due to several key innovations, each addressing foundational challenges in language understanding and generate.

- Transformer Architecture (2017): The groundbreaking work of Vaswani et al. (2017) introduced the Transformer architecture, which marked a turning point in neural language modeling. Unlike earlier models that relied on recurrence, Self-attention lets transformers model dependencies throughout entire input sequences efficiently. This enables better handling of long-range context and allows models to be trained in parallel on large datasets. The architecture's encoder-decoder design with multi-head attention and feed-forward layers has since become the foundation of nearly all modern LLMs (Vaswani et al., 2017).
- Unsupervised Pre-training at Scale: Training models on large volumes of unannotated text has proven to be highly effective. This approach, pioneered by models like ELMo (Peters et al., 2018), GPT (Radford et al., 2018), and BERT (Devlin et al., 2019), enables LLMs to acquire generalized linguistic knowledge. The success of GPT-3 (Brown et al., 2020) solidified the strategy of massive-scale pre-training followed by either fine-tuning or prompt-based adaptation. Studies on model scaling (Kaplan et al., 2020) showed that increasing model size and training data leads to predictable improvements in performance across diverse tasks

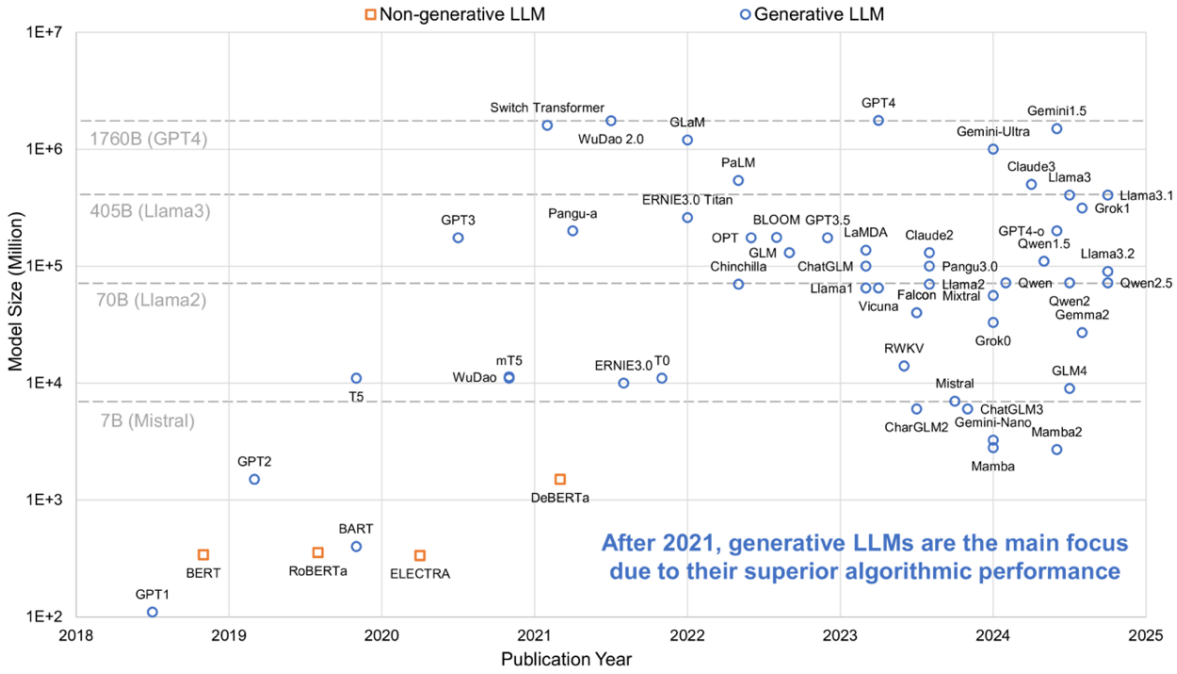


Figure 3 Typical LLM model size in the past seven years (Li et al., 2024)

- Few-Shot and Zero-Shot Learning via Prompting: One of the most surprising capabilities of large-scale LLMs is their ability to perform tasks based solely on textual prompts, without any additional training. GPT-3 demonstrated strong performance using few-shot, zero-shot, and in-context learning, where the model learns the task purely from the prompt itself (Brown et al., 2020). This method significantly reduces the need for task-specific data and training and has led to extensive research on prompt engineering strategies (Liu et al., 2021) to better harness these capabilities.
- Instruction Tuning and Alignment: To make LLMs more responsive to user intent, instruction tuning has been introduced. This method involves training models on

datasets consisting of (instruction, response) pairs to help them better follow natural language commands. Works such as FLAN (Wei et al., 2021), T0 (Sanh et al., 2022), and InstructGPT (Ouyang et al., 2022) have shown that instruction tuning enhances model usability, especially in zero-shot settings. Additionally, reinforcement learning from human feedback has aligned models in order to suit human preferences, resulting in safer and more cooperative outputs (Ouyang et al., 2022).

- Knowledge Retrieval and Augmentation: To overcome limitations related to knowledge cutoffs and factual recall, retrieval-augmented generation has emerged as a promising approach. Instead of relying solely on internal model parameters, RAG enables the model to retrieve relevant documents from external sources (e.g., Wikipedia or vector databases) to inform its responses. This approach, introduced by Lewis et al. (2020) and further developed by Guu et al. (2020), has been shown to improve factual accuracy and task performance, particularly in knowledge-intensive applications.

In summary, the last few years have seen rapid progress driven by these innovations. New architectures (especially Transformers), massively scaled training, novel ways of using models (prompting), and alignment techniques. The confluence of these breakthroughs is what makes today's LLMs both highly capable and increasingly practical.

1.1.3 Transformer-Based Large Language Model Architectures

Transformer architectures, first introduced by Vaswani et al. (2017), have since become the dominant framework in large language model design (Qiu et al., 2020; Bommasani et al., 2021). Their primary innovation—the self-attention mechanism—enables models to evaluate all elements of an input simultaneously, replacing the slower, sequential nature of previous recurrent architectures. This shift has led to greater training efficiency and stronger modeling of long-range dependencies (Vaswani et al., 2017).

Modern Transformer-based models can be categorized broadly into three groups:

- **Encoder-only models** utilize only the encoder portion of the Transformer. They process input sequences bidirectionally, making them suitable for understanding tasks such as text classification, named entity recognition, and information retrieval (Vaswani et al., 2017; Qiu et al., 2020). A prominent example is BERT (Devlin et al., 2019), which is pre-trained using masked language modeling to capture rich contextual information from both left and right context.
- **Decoder-only models** use the Transformer's decoder block and are designed for autoregressive tasks like text generation. These models predict each new token based exclusively on previous tokens (Vaswani et al., 2017; Qiu et al., 2020). GPT-3 (Brown et al., 2020), for instance, follows this architecture and is capable of generating high-quality text from prompts using few-shot and zero-shot learning.
- **Encoder-decoder (sequence-to-sequence) models** combine both components the encoder reads and encodes the full input sequence and the decoder constructs the output sequence incrementally, generating each token in order (Vaswani et al., 2017; Qiu et al., 2020). This design is common in tasks such as machine translation and summarization. Examples include T5 (Raffel et al., 2020)

and BART (Lewis et al., 2020), which are pre-trained on text-to-text tasks and then fine-tuned for specific applications.

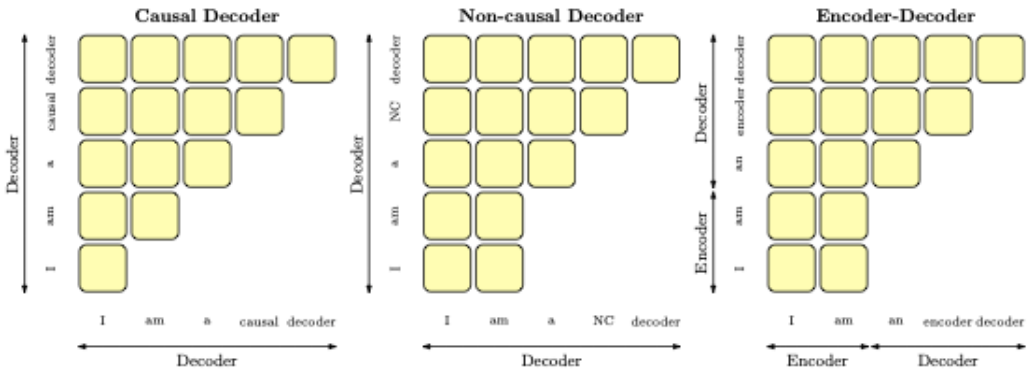


Figure 4 Transformer-based models' graph (attention patterns) (Wang et. al, 2022a)

Each of these architectures uses core elements like feed-forward layers, multi-head attention, position encodings or residual connections (Vaswani et al., 2017). The choice of architecture depends on whether the task emphasizes comprehension, generation, or both (Qiu et al., 2020). A comparison of their attention mechanisms is shown in Figure 4, which illustrates how encoder-only models attend to all tokens bidirectionally, decoder-only models use causal attention, and encoder-decoder models combine both strategies (Vaswani et al., 2017; Qiu et al., 2020).

To meet growing demands for longer context, higher efficiency, and better scalability, several architectural innovations have been introduced:

- **Scaling Efficiency:**
  Efficient attention mechanisms have been developed to reduce the computational cost of traditional self-attention, which scales quadratically with input length. Models like Longformer (Beltagy et al., 2020) and Reformer (Kitaev et al., 2020) use sparse attention or hashing techniques to make processing longer sequences feasible without drastically increasing memory usage.

- **Long Context Handling:**
  Scalability improvements include architectures like Performer (Choromanski et al., 2021), which approximates attention through kernel methods to achieve linear time and space complexity. These advances are critical for training models on very large datasets or for processing long documents.

- **Memory and Retrieval Integration:**
  Extended position encoding methods, such as Rotary Positional Embeddings (RoPE) introduced by Su et al. (2021), improve the model's ability to generalize to longer sequences than those seen during training by encoding positions in a more flexible and extrapolatable way.

- **Sparse Attention and Efficiency Tricks:**
  Retrieval-Augmented Transformers incorporate external sources of knowledge into the model's input. RETRO (Borgeaud et al., 2022), for example, retrieves relevant text chunks from a large database and integrates them during generation, increasing factual accuracy without requiring larger parameter counts.

Together, these architectural choices and enhancements have significantly broadened the capabilities of Transformer-based LLMs. While the original design by Vaswani et al. (2017) remains central, contemporary models continue to evolve by combining efficiency, flexibility, and access to external knowledge.

1.1.4 Current Challenges and Opportunities

Despite major advances, large language models (LLMs) continue to face several important challenges that actively drive current research NLP field.

One persistent issue is factual inaccuracy, often referred to as "hallucination"—where LLMs generate fluent but incorrect or fabricated content (Ji et al., 2022). This appears because models are trained to predict text patterns, not to verify facts. Techniques such as Retrieval-Augmented Generation (Lewis et al., 2020) and reinforcement learning from human feedback (Ouyang et al., 2022) help reduce hallucinations by grounding outputs in external knowledge and aligning them with user expectations. Improving factuality is crucial for building trustworthy AI systems that can be safely used in domains like education, law, and healthcare.

LLMs have a static knowledge base fixed at training time, making them incapable of responding accurately to new information (Karpukhin et al., 2020). Solutions include continual pre-training, retrieval from updated sources, and localized model editing—all aiming to keep models factually current without full retraining. Ensuring up-to-date responses is especially important as LLMs are increasingly used for real-time tasks.

Training and deploying LLMs demand substantial energy and computing resources (Patterson et al., 2022). To improve efficiency, research explores model compression (Sanh et al., 2019), pruning (Han et al., 2015), quantization (Dettmers et al., 2022), and parameter-efficient methods such as LoRA (Hu et al., 2022), which reduce hardware requirements without major performance loss. These methods contribute to the sustainable deployment of LLMs at scale.

LLMs risk amplifying social biases present in their training data (Bender et al., 2021). Approaches like RLHF and curated training data (Gehman et al., 2020) aim to make outputs more fair and socially responsible. However, ethical risks such as stereotyping, misinformation, and toxic content remain active areas of concern, particularly in public-facing AI applications.

Although LLMs can produce convincing text, their reasoning capabilities are limited. Techniques like chain-of-thought prompting (Wei et al., 2022b) and self-consistency decoding (Wang et al., 2022b) improve performance on logic-intensive tasks, but robust

and interpretable reasoning is still a work in progress. Addressing this gap is essential for enabling LLMs to support more complex cognitive tasks.

Efforts to improve factuality, adaptability, efficiency, ethics, and reasoning define current LLM research. Retrieval, fine-tuning strategies, and alignment methods offer promising paths forward, but these systems must evolve further to meet real-world demands and societal expectations.

1.2 Transformer Architecture

The Transformer architecture fundamentally transformed the field of natural language processing by overcoming the major shortcomings of earlier architectures like RNNs and CNNs (Wolf et al., 2019). It introduced a new approach capable of effectively capturing long-range dependencies in sequences without relying on recurrence or convolution layers (Vaswani et al., 2017). Transformers were a major innovation because they rely on attention mechanisms – in particular self-attention – to directly model simultaneously relationships as being between all elements within an input sequence (Uszkoreit, 2017; Vaswani et al., 2017). This design enables highly parallel computation and mitigates the complexity of learning long-range dependencies, making the Transformer especially well-suited for processing long textual inputs and for scaling up to larger model sizes (Wolf et al., 2019; Kaplan et al., 2020). Consequently, Transformers have yielded significant performance gains across a wide range of NLP applications and have become the foundational architecture underpinning modern large language models such as GPT-3, LLaMA, and T5 (Brown et al., 2020; Chowdhery et al., 2022; Kaplan et al., 2020; Raffel et al., 2020; Touvron et al., 2023a), as well as recent systems like GPT-4, LLaMA 2, or Claude (Bai et al., 2022; OpenAI, 2023; Touvron et al., 2023b).

This section provides an in-depth exploration of the Transformer architecture. It begins with a comprehensive overview of its structural elements, detailing the roles of the encoder and decoder modules. Particular attention is devoted to the innovative mechanisms of self-attention and multi-head attention, explaining their central role in enabling effective modeling of contextual dependencies. The concept and implementation of positional encoding are also clarified, highlighting its importance for capturing sequence order. Furthermore, the section contrasts Transformers with traditional sequential models, such as RNNs, illustrating their inherent advantages. Finally, it addresses the scalability of Transformers and examines their profound influence on the development and performance of contemporary large language models.

1.3 Overview of the Transformer Architecture

The Transformer architecture (Vaswani et al., 2017) has a decoder stack along with an encoder stack. A point-wise feed-forward network is then used alongside a multi-head self-attention mechanism for each encoder layer. Residual connections normalize layers around and within both sublayers (Vaswani et al., 2017). Decoder layers are likewise structured yet

also possess a third sublayer for performing multi-head attention over the output from the encoder (i.e., encoder–decoder attention) (Vaswani et al., 2017). First, vector embeddings are created from input tokens then they are augmented by positional encodings, for the incorporation of word-order information. This encoder–decoder architecture is illustrated in Figure 5.

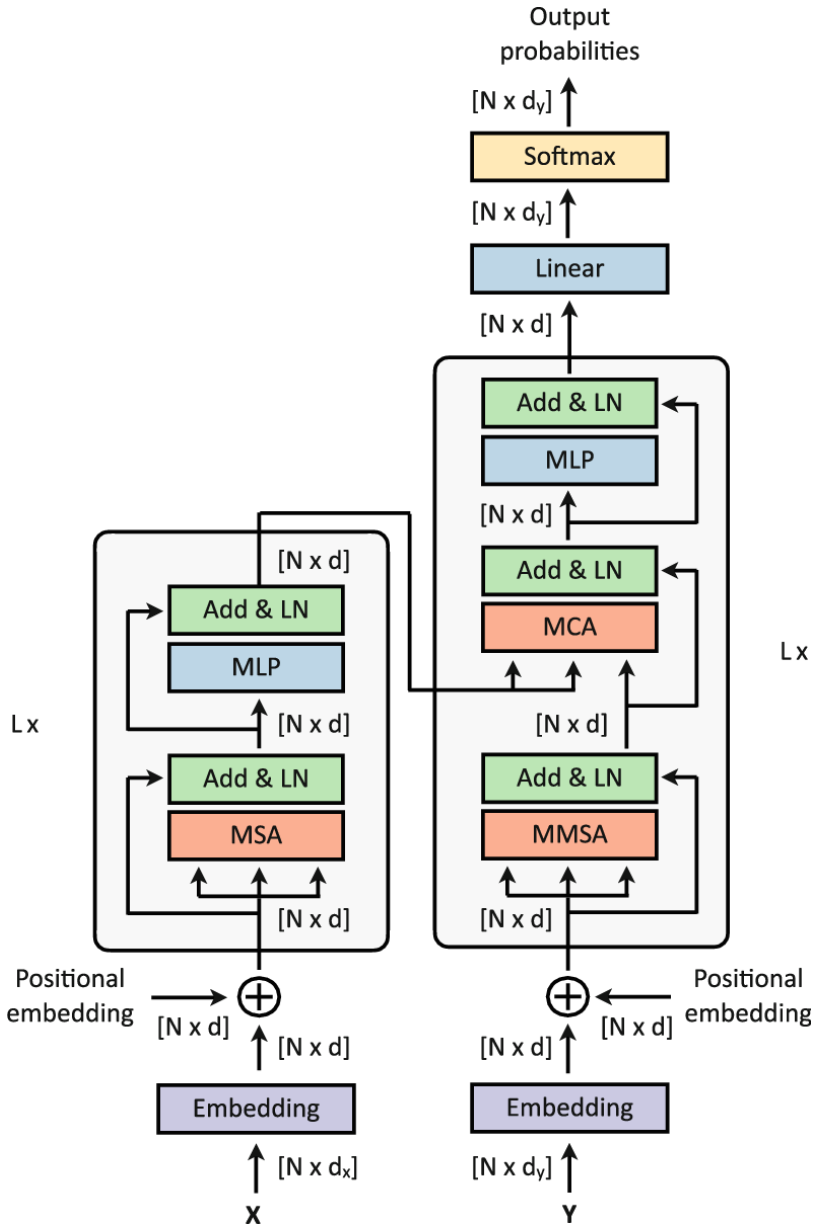


Figure 5 The transformer architecture: It consists of an encoder (left) and a decoder (right) (Courant et. al, 2023)

As a result of the encoding process, the model produces continuous vector representations (contextual embeddings) of the entire input sequence (Vaswani et al., 2017). During decoding, the model uses masked self-attention since each position can attend only to earlier output tokens that are already generated (Vaswani et al., 2017). Encoder–decoder attention then lets the decoder incorporate information from what the encoder outputs throughout each step (Vaswani et al., 2017). Only these attention mechanisms are used by the Transformer. Customary recurrent networks do involve recurrence even though the

Transformer does not rely on recurrence (Vaswani et al., 2017). Consequently, on machine translation benchmarks, the Transformer outperformed prior state-of-the-art RNN and CNN models and also trained significantly faster on modern hardware (Vaswani et al., 2017). In essence, the Transformer's modular encoder–decoder design generalizes the classic sequence-to-sequence model: the encoder transforms the input sequence into an intermediate representation, and the decoder uses that representation to generate an output sequence one element at a time (Cho et al., 2014; Sutskever et al., 2014). We will detail below how this design paved the way for many subsequent models.

1.3.1 Encoder-Decoder Structure

The Transformer is built upon a symmetric encoder–decoder architecture, a framework commonly used in earlier sequence modeling tasks, especially machine translation (Cho et al., 2014). Both the encoder and the decoder consist of N identical layers and leverage attention mechanisms to model relationships within sequences (Vaswani et al., 2017). Each encoder layer comprises a pair of primary components: a position-wise feed-forward neural network and also a multi-head self-attention mechanism (Vaswani et al., 2017). These sublayers are wrapped with residual connections and followed by layer normalization, computed as:

$$LayerNorm\ (x + Sublayer(x)),$$

All encoder layers (including the input embedding layer) share the same dimensionality, ensuring compatibility with residual operations (Vaswani et al., 2017). The encoder processes input sequences by first applying self-attention across all tokens, allowing each to attend to every other, and then passing the result through a feed-forward network applied independently to each token. After N such layers, the encoder outputs a set of contextualized token representations — one per input — that capture global dependencies (Vaswani et al., 2017).

The decoder mirrors this structure but adds an extra sublayer. Each decoder layer consists of three sublayers:

1. **Masked multi-head self-attention**, which restricts each token's visibility to only the previously generated positions — enforcing autoregressive behavior (Vaswani et al., 2017).
2. **Encoder–decoder (cross) attention**, where each decoder token attends to all encoder outputs using the decoder's previous layer as queries and the encoder's output as keys and values (Vaswani et al., 2017).
3. **Feed-forward network**, again applied position-wise (Vaswani et al., 2017).

Like the encoder, decoder sublayers are all wrapped in residual connections then layer normalization follows. This configuration lets the decoder combine the encoded input sequence with the evolving output sequence during its generation of each step (Bahdanau et al., 2015; Vaswani et al., 2017).

As a result, each output token is generated by attending both to the entire input (via encoder–decoder attention) and to the previous output tokens (via masked self-attention).

This dual mechanism is similar in function to the attention-based models used in earlier sequence-to-sequence architectures (Bahdanau et al., 2015), but in the Transformer, attention is implemented entirely within multi-head attention layers in every decoder block (Vaswani et al., 2017).

1.3.2 Self-Attention and Multi-Head Mechanism

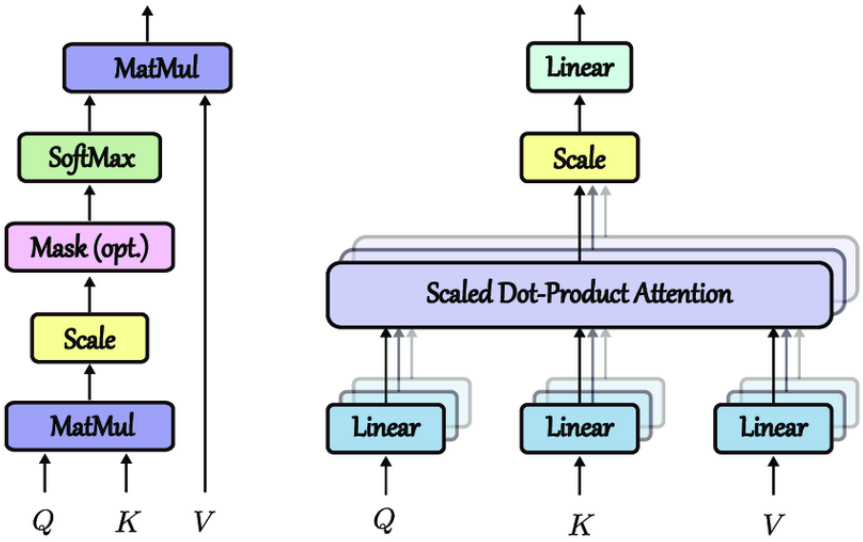


Figure 6 Self-attention mechanism (left) and multi-head attention mechanism (right) based on Vaswani et al. (2017) (Ma et al., 2022)

The Transformer's self-attention mechanism enables each token to attend to all other tokens in the input sequence. This allows the model to capture contextual relationships and long-range dependencies without using recurrence (Vaswani et al., 2017). For each position, the model computes three vectors: a **query** Q, a **key** K, and a **value** V, which are obtained by linear projections of the input embeddings (Vaswani et al., 2017).

The attention mechanism is computed using the following formula (Vaswani et al., 2017, Section 3.2.1):

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

This equation calculates attention since it computes the dot product between queries and keys, scales the result by the square root of the key dimension d_k, upon applying the softmax function to obtain normalized weights. Then these weights are multiplied by the value matrix which is V. This multiplication produces a context-aware representation for each such token (Vaswani et al., 2017).

To allow the model to capture different types of dependencies, the Transformer uses **multi-head attention**, which performs this attention mechanism h times in parallel, using different learned projections (Vaswani et al., 2017). Every head's outputs are joined then mapped linearly to the original model dimension. As stated in the original paper:

"Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions." (Vaswani et al., 2017, p. 5)

The dimensionality of each head is typically set so that:

$$d_k = \frac{d_{model}}{h}$$

which ensures that the total computation remains similar to single-head attention, while providing more expressive power (Vaswani et al., 2017).

1.3.3 Positional Encoding

Since the Transformer processes all tokens in parallel as it lacks any built-in idea about token order, it must incorporate positional information into the input (Vaswani et al., 2017). To compensate for the lack of positional information, Vaswani et al. (2017) added fixed encoding vectors to the input embeddings at the lowest layer of the model. Given that these encodings do have the same dimensionality as token embeddings (d_{model}), they can be summed in an elementwise manner (Vaswani et al., 2017).

Vaswani et al. (2017) proposed a fixed, sinusoidal approach to positional encoding. Each position pos in the sequence is mapped to a vector whose components are defined using sine and cosine functions at varying frequencies. For each even dimension $2i$ and odd dimension $2i + 1$, the encoding is computed as:

$$\mathrm{PE}_{(\mathrm{pos},\,2\mathrm{i})} = \sin\left(\frac{\mathrm{pos}}{10000^{2\mathrm{i}\,/\,\mathrm{d}_{\mathrm{model}}}}\right), \mathrm{PE}_{(\mathrm{pos},\,2\mathrm{i}+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\mathrm{model}}}}\right)$$

These functions assign different frequencies to each dimension, with lower dimensions varying more rapidly (Vaswani et al., 2017). As a result, the encoding vectors form a smooth and continuous function of position, and the difference between any two positions corresponds to a predictable transformation in the encoding space. This structure allows the model to reason not only about absolute positions but also relative distances between tokens (Vaswani et al., 2017).

Although learned positional embeddings were also tested, Vaswani et al. (2017) reported comparable performance with the fixed sinusoidal method. They chose the fixed version because it allows the model to generalize to sequence lengths beyond those seen during training, thanks to its formulaic, unbounded nature (Vaswani et al., 2017).

In summary, positional encoding gives the Transformer access to sequence order while preserving its parallel computation structure. This approach makes it possible for self-attention layers to utilize token position information without the need for recurrence or convolution (Vaswani et al., 2017).

1.3.4 Advantages over Sequential Models

The Transformer architecture (Vaswani et al., 2017) can offer quite a few compelling advantages compared with customary recurrent models (RNNs). By replacing recurrence

with self-attention, Transformers enable more efficient training and capture of long-range dependencies, leading to improved performance in practice (Vaswani et al., 2017). Key advantages include the ability to compute in parallel, shorter dependency paths, flexible context integration, and superior empirical results, as detailed below.

One of the most critical benefits is parallel computation. In RNN-based models, tokens are processed one at a time, which prevents parallel execution and limits the use of modern GPU and TPU hardware. The Transformer, by contrast, computes attention across all tokens simultaneously, allowing all positions in the sequence to be processed in parallel. This greatly speeds up training and makes the model more efficient for large-scale data (Uszkoreit, 2017).

Another major advantage is shorter dependency paths. In RNNs, the information between distant tokens must pass through multiple time steps, which can lead to degradation or loss of information — a common challenge in modeling long sequences. In the Transformer, self-attention enables each token to directly interact with any other token in a single layer. This makes learning long-range dependencies much easier and more reliable (Vaswani et al., 2017).

Transformers also offer flexible contextualization. Through self-attention, each token has for it the ability to attend to different parts in the sequence if there exists context. This gets improved by multi-head attention since the model attends to subspaces simultaneously. Each head can focus on different patterns — such as local syntax, long-range dependencies, or semantic roles — giving the model a richer understanding of the input (Devlin et al., 2019).

As a result of their architecture, Transformers have demonstrated strong practical performance, consistently surpassing RNN-based models in tasks such as machine translation and language modeling, while also requiring less training time (Vaswani et al., 2017). By leveraging the Transformer framework, BERT and GPT-3 have consistently delivered state-of-the-art outcomes across numerous language understanding and generation tasks (Devlin et al., 2019; Brown et al., 2020).

In summary, the Transformer architecture combines computational efficiency, direct global context modeling, and flexible attention mechanisms — making it a superior alternative to traditional recurrent architectures in both theory and practice.

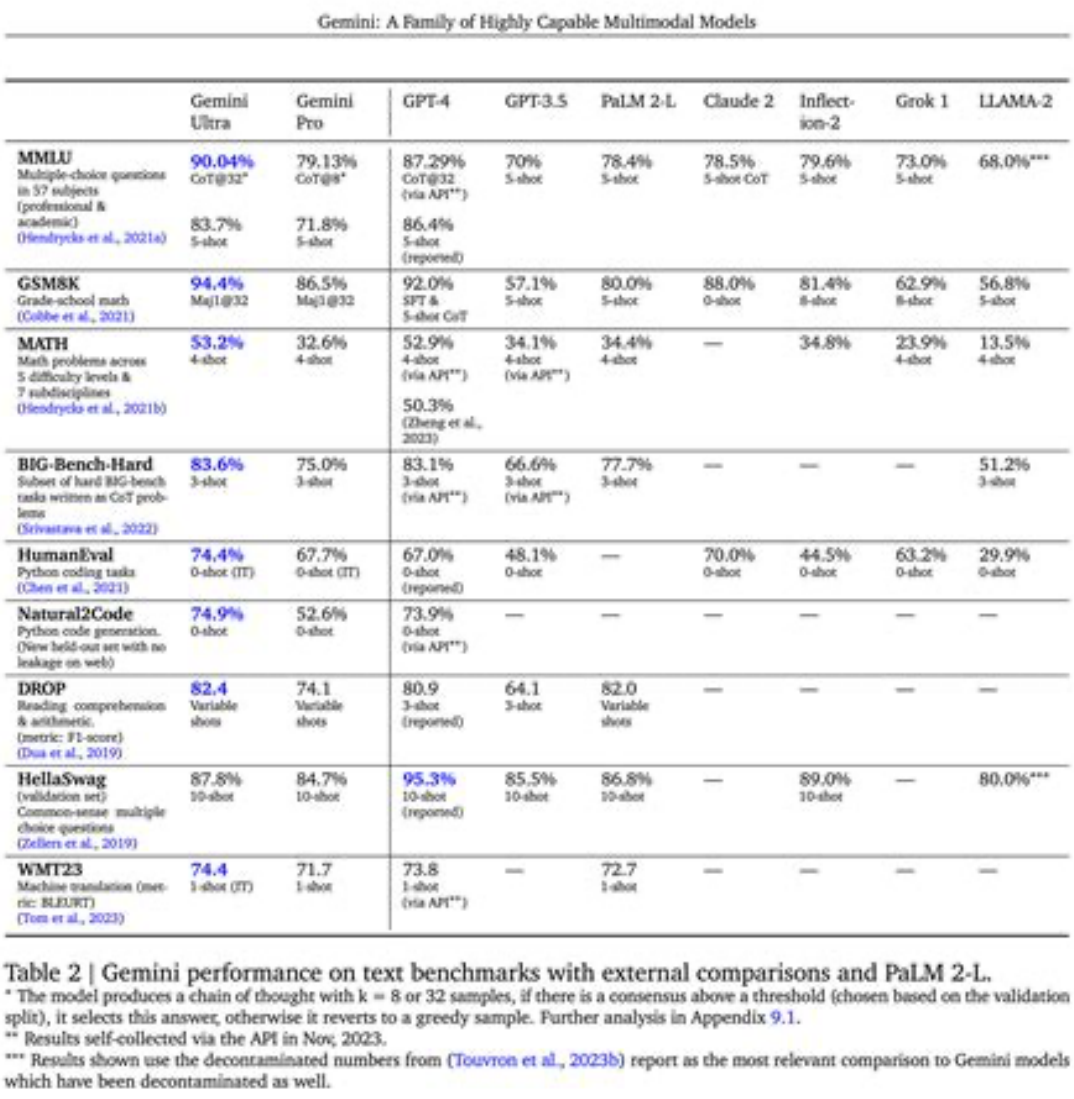
1.3.5 Scalability and Impact on Large Language Models

One of the defining strengths of Transformer-based architectures lies in their scalability. As demonstrated by Kaplan et al. (2020), language models benefit predictably from increases in model size, training data, and computational resources, with loss decreasing along smooth power-law curves. This insight has motivated the development of increasingly larger models, which have shown measurable gains in downstream task performance.

Notably, OpenAI's GPT-3, consisting of 175 billion parameters, achieved impressive few-shot capabilities without task-specific fine-tuning (Brown et al., 2020). Google's PaLM, with 540 billion parameters, further advanced this trend by outperforming earlier models on a

wide range of reasoning benchmarks (Chowdhery et al., 2022). These developments confirmed that larger Transformer models tend to generalize better and acquire more versatile abilities (Brown et al., 2020; Chowdhery et al., 2022).

The trend continued with the release of Gemini Ultra, a large-scale multimodal model developed by Google DeepMind. In a major milestone, Gemini Ultra achieved a score of 90.04 % on the MMLU benchmark, surpassing the average performance of human experts (89.8%) across 57 diverse subjects, including history, law, and medicine (Gemini Team, 2023). This result illustrates the emergence of advanced reasoning capabilities that are not typically present in smaller models. Figure 7 shows Gemini's performance on the BIG-bench evaluation.



Gemini: A Family of Highly Capable Multimodal Models

	Gemini Ultra	Gemini Pro	GPT-4	GPT-3.5	PaLM 2-L	Claude 2	Inflect-ion-2	Grok 1	LLAMA-2
MMLU Multiple-choice questions in 57 subjects (professional & academic) (Hendrycks et al., 2021a)	**90.04%** CoT@32*	79.13% CoT@8*	87.29% CoT@32 (via API**)	70% 5-shot	78.4% 5-shot	78.5% 5-shot CoT	79.6% 5-shot	73.0% 5-shot	68.0%***
	83.7% 5-shot	71.8% 5-shot	86.4% 5-shot (reported)						
GSM8K Grade-school math (Cobbe et al., 2021)	**94.4%** Maj1@32	86.5% Maj1@32	92.0% SFT & 5-shot CoT	57.1% 5-shot	80.0% 5-shot	88.0% 0-shot	81.4% 8-shot	62.9% 8-shot	56.8% 5-shot
MATH Math problems across 5 difficulty levels & 7 subdisciplines (Hendrycks et al., 2021b)	**53.2%** 4-shot	32.6% 4-shot	52.9% 4-shot (via API**)	34.1% 4-shot (via API**)	34.4% 4-shot	—	34.8%	23.9% 4-shot	13.5% 4-shot
			50.3% (Zheng et al., 2023)						
BIG-Bench-Hard Subset of hard BIG-bench tasks written as CoT problems (Srivastava et al., 2022)	**83.6%** 3-shot	75.0% 3-shot	83.1% 3-shot (via API**)	66.6% 3-shot (via API**)	77.7% 3-shot	—	—	—	51.2% 3-shot
HumanEval Python coding tasks (Chen et al., 2021)	**74.4%** 0-shot (IT)	67.7% 0-shot (IT)	67.0% 0-shot (reported)	48.1% 0-shot	—	70.0% 0-shot	44.5% 0-shot	63.2% 0-shot	29.9% 0-shot
Natural2Code Python code generation. (New held-out set with no leakage on web)	**74.9%** 0-shot	52.6% 0-shot	73.9% 0-shot (via API**)	—	—	—	—	—	—
DROP Reading comprehension & arithmetic. (metric: F1-score) (Dua et al., 2019)	**82.4** Variable shots	74.1 Variable shots	80.9 3-shot (reported)	64.1 3-shot	82.0 Variable shots	—	—	—	—
HellaSwag (validation set) Common-sense multiple choice questions (Zellers et al., 2019)	87.8% 10-shot	84.7% 10-shot	**95.3%** 10-shot (reported)	85.5% 10-shot	86.8% 10-shot	—	89.0% 10-shot	—	80.0%***
WMT23 Machine translation (metric: BLEURT) (Tom et al., 2023)	**74.4** 1-shot (IT)	71.7 1-shot	73.8 1-shot (via API**)	—	72.7 1-shot	—	—	—	—

Table 2 | Gemini performance on text benchmarks with external comparisons and PaLM 2-L.
* The model produces a chain of thought with k = 8 or 32 samples, if there is a consensus above a threshold (chosen based on the validation split), it selects this answer, otherwise it reverts to a greedy sample. Further analysis in Appendix 9.1.
** Results self-collected via the API in Nov, 2023.
*** Results shown use the decontaminated numbers from (Touvron et al., 2023b) report as the most relevant comparison to Gemini models which have been decontaminated as well.

Figure 7 BIG-bench evaluation of Gemini Ultra (Gemini Team, 2023)

These gains are made possible in large part due to the Transformer's parallel architecture, which allows model training to scale efficiently across thousands of accelerators. This

capability has enabled the rise of so-called foundation models (Bommasani et al., 2021): massive, general-purpose Transformer systems that can be fine-tuned or prompted for a wide variety of tasks. Today's state-of-the-art models — including GPT-4, Gemini, and Claude — build on this foundation, extending the Transformer's success across natural language, code, vision, and multimodal domains.

1.4 Prompts-LLM instructions

The effectiveness of a language model often depends upon the phrasing of tasks inside its prompt. The model's output can be steered by natural language instructions or example demonstrations or both in practice. Performance in a variety of tasks is largely influenced through prompt design which is careful, as studies do show (Liu et al., 2021; Zhao et al., 2021).

This section reviews the key and main prompting strategies for general LLM use now. The strategies do include zero-shot prompting, few-shot prompting, instruction-style prompts, as well as chain-of-thought prompting.

1.4.1 Zero-Shot and Few-Shot Prompting

Zero-shot prompting refers to the practice for providing a language model with a task instruction. In this method, no examples of just how the input should be transformed into an output are given. For example, a prompt may say: "Translate the following sentence into Czech: 'The sky is blue.'"—and the model must infer the expected format and language without additional cues (Brown et al., 2020). This contrasts with few-shot prompting, where the model is presented with several input–output examples in the prompt, such as previous English–Czech sentence pairs, before being asked to process a new sentence. The few-shot format helps the model generalize from the provided context, improving performance (Brown et al., 2020; OpenAI, 2023).

A visual comparison of these prompting strategies—zero-shot, one-shot, and few-shot—is shown in Figure 8. It illustrates how few-shot learning bridges the gap between raw inference and traditional supervised fine-tuning by inserting context examples directly into the prompt (Brown et al., 2020).

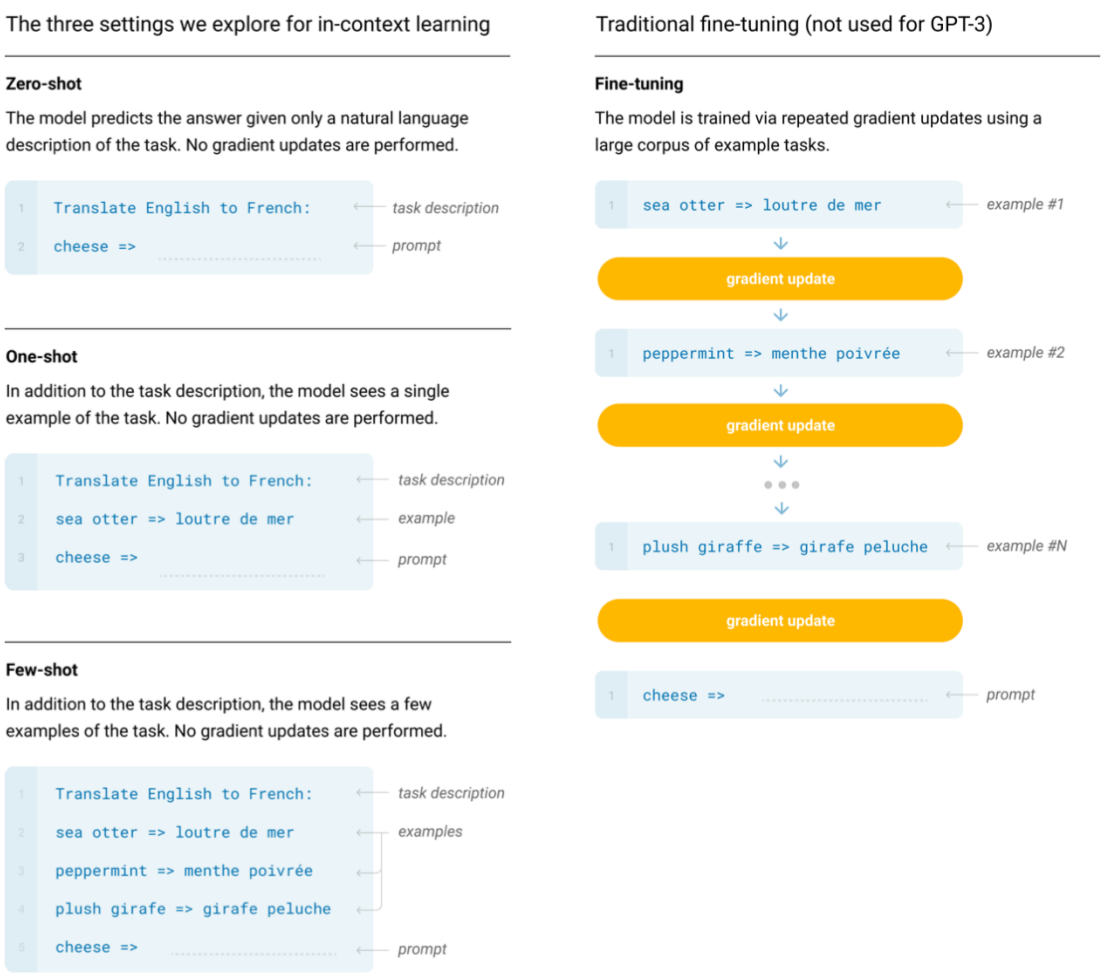


Figure 8 Zero-shot, one-shot and few-shot, contrasted with traditional fine-tuning (Brown et al., 2020)

The difference between these approaches is foundational to modern large language models (LLMs), especially since the release of GPT-3, which demonstrated strong in-context learning without parameter updates (Brown et al., 2020). In traditional NLP pipelines, fine-tuning with task-specific data was necessary. However, GPT-3 and its successors showed that well-crafted prompts—especially few-shot prompts—can unlock capabilities that previously required dedicated training (Chowdhery et al., 2022; OpenAI, 2023).

Recent models further push the boundaries. PaLM 540B showed state-of-the-art few-shot reasoning on benchmarks like GSM8K and DROP (Chowdhery et al., 2022), while GPT-4 scored around 86.4% on MMLU using five-shot prompting, outperforming GPT-3.5 by over 15 percentage points (OpenAI, 2023). On ARC and HellaSwag, GPT-4 achieved 96% and 95% accuracy, respectively—benchmarks testing scientific knowledge and commonsense reasoning (Hendrycks et al., 2020; Zellers et al., 2019).

Furthermore, instruction tuning and chain-of-thought prompting have enhanced zero-shot reasoning. For instance, the phrase *"Let's think step by step"* encourages the model to generate intermediate steps rather than a direct answer. This technique alone improved arithmetic problem-solving accuracy from 17.7% to 78.7% on GSM8K in zero-shot settings

(Kojima et al., 2022). Similarly, Claude and PaLM 2 have demonstrated robust few-shot performance using CoT prompting combined with instruction-following objectives (Bai et al., 2022; Anil et al., 2023).

In summary, zero-shot prompting is quick and requires no labeled data, but is often less precise. Few-shot prompting, by contrast, boosts model performance with minimal effort. As LLMs scale, the performance gap between these two modes narrows, enabling powerful language understanding even in zero-shot settings (Wei et al., 2022a; Touvron et al., 2023a). These prompting strategies now define modern LLM usage, allowing models like GPT-4, Claude, PaLM 2, and Gemini to perform competitive NLP tasks with minimal supervision.

1.4.2 Instruction-Style Prompting

Instruction-style prompting refers to a technique where tasks are conveyed through explicit natural language commands, rather than by providing example-based demonstrations. Instead of few-shot examples, the prompt directly tells the model what to do—for instance, *"Summarize the following text in few sentences"* or *"Translate this sentence into Czech (academic style)."* To further refine outputs, prompts can include role descriptions (*"You are a helpful assistant"*) or output constraints (*"Respond in JSON format"*). This method enables language models to more precisely understand user intent and adhere to specified task formats (Ouyang et al., 2022; OpenAI, 2023).

This approach became widely adopted after the development of InstructGPT, which was fine-tuned using reinforcement learning from human feedback (RLHF). In the foundational work by Ouyang et al. (2022), GPT-3 was trained on thousands of instruction–completion pairs, enabling the model to better follow user directions. Surprisingly, the smaller 1.3B InstructGPT model was preferred over the much larger 175B GPT-3, demonstrating that alignment with user intent can be more important than raw model size. Human evaluators consistently rated InstructGPT's responses as more helpful, relevant, and safe (Ouyang et al., 2022).

Modern instruction-tuned systems like ChatGPT, Gemini, and Claude build on these findings. These models use extensive instruction datasets and alignment techniques to improve the clarity, safety, and helpfulness of their outputs (OpenAI, 2023; Anil et al., 2023; Bai et al., 2022). In addition to RLHF, newer models also integrate techniques like supervised fine-tuning with curated instruction sets, feedback from human reviewers, and reward models that penalize harmful or off-topic responses (Bai et al., 2022; OpenAI, 2023).

Instruction-style prompting has a central role because it aligns large language models (LLMs) with human preferences. Without such guidance, models trained solely to predict the next token can produce responses that are unhelpful, vague, or even harmful (OpenAI, 2023). By providing clear, structured prompts, users enable models to perform complex tasks more reliably, setting the foundation for real-world applications in education, healthcare, software development, and more (Ouyang et al., 2022; Anil et al., 2023).

1.4.3 Chain-of-Thought Prompting

Thought Prompting Chain-of-Thought prompting is a method guiding language models in the production of reasoning steps. This leads toward the final answer, rather than just the answer itself (Wei et al., 2022b). In practice, the prompt provides exemplars with step-by-step solutions instead of only question–answer pairs (Wei et al., 2022b). As Figure 9 illustrates, this approach contrasts with standard prompting, which directly requests an answer (Brown et al., 2020). Studies show that CoT receives a substantial performance increase when prompted on reasoning tasks including arithmetic and common sense (Wei et al., 2022b). For example, providing eight reasoning exemplars to a 540-billion-parameter model (PaLM) achieved state-of-the-art accuracy on the GSM8K math word-problem benchmark (Wei et al., 2022b).

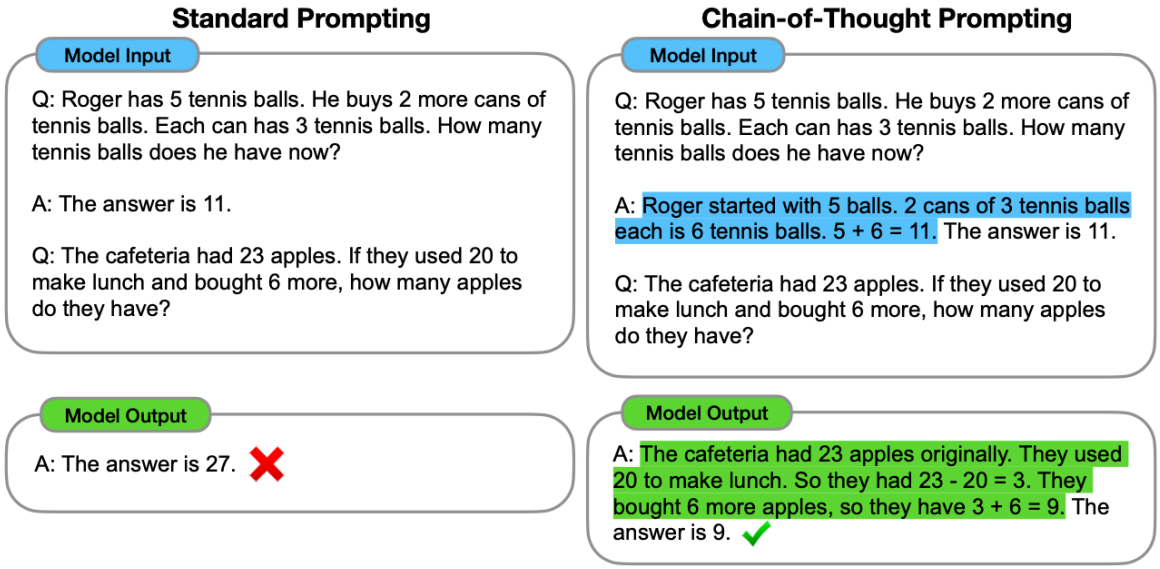


Figure 9 Example of difference bewtween standart prompting (left) and Chain-of-Thought prompting (Brown et al., 2020)

CoT can even be effective with no exemplars (“zero-shot”). Kojima et al. (2022) discovered that simply appending *“Let’s think step by step.”* to a query prompts the model to generate its own reasoning chain, dramatically improving results. This single prompt increased accuracy on the MultiArith dataset from 17.7% to 78.7%, and on GSM8K from 10.4% to 40.7%, using an InstructGPT model (Kojima et al., 2022). These findings suggest that carefully worded prompts can unlock latent reasoning abilities of LLMs (Kojima et al., 2022). Notably, the CoT strategy is now leveraged by cutting-edge models: GPT-4 can be guided with CoT prompts to improve its problem-solving (OpenAI, 2023), Anthropic’s Claude benefits from step-by-step reasoning prompts (Anthropic, 2023), and Google DeepMind’s Gemini is explicitly being designed with an internal “chain-of-thought” process to better tackle complex tasks (Kavukcuoglu, 2025).

1.5 Fine-Tuning pre-trained LLMs

Fine-tuning plays such a major role in the tailoring of specialized tasks and specific domains' demands for meeting large language models (LLMs) (Bommasani et al., 2021; Minaee et al., 2024). Models trained beforehand achieve wide-ranging language understanding from exposure to varied corpora that are wide-ranging (Brown et al., 2020; Devlin et al., 2019). However, they often need more adjustment because their direct use in focused areas fails. Fine-tuning addresses this limitation, for it continues with the training process through the use of targeted datasets in addition to either supervised learning or instruction-based approaches (Bommasani et al., 2021; Minaee et al., 2024). This adaptation makes the model more relevant for the task and aligns it with domain-specific language and conventions. The model's actual effectiveness improves too (Bommasani et al., 2021; Lee et al., 2020). Fine-tuning is a key step for deploying LLMs in real-world contexts.

In this section, we compare pre-training as well as fine-tuning, outline common fine-tuning strategies, along with explore newer parameter-efficient methods because we want to highlight both their advantages and intrinsic limitations (Bergmann, 2024a; Minaee et al., 2024).

1.5.1 Pre-training vs. Fine-tuning

Large language models (LLMs) typically undergo development within two stages. A model is indeed trained upon massive amounts of text within the pre-training stage. Self-supervised objectives like next-word prediction are used for helping it learn broad linguistic patterns. Grammar as well as semantics are better understood from this. Knowledge of the world is also imparted (Devlin et al., 2019; Minaee et al., 2024). Pre-training on huge unlabeled corpora is computationally costly yet yields a base model showing broad language coverage (Brown et al., 2020). This is in contrast. Specific tasks use fine-tuning to adapt this model. Fine-tuning then further trains the pre-trained model on a smaller task-specific often labeled dataset, and this also adjusts the model’s weights in order to specialize its behavior (Bommasani et al., 2021). Bergmann (2024a) notes that fine-tuning “is the process of adapting a pre-trained model for specific tasks or use cases.” Fine-tuning leverages the existing knowledge of the model, and therefore it requires a far less amount of data in addition to compute than training a model that is new from scratch (Minaee et al., 2024). Fine-tuning does effectively apply the targeted optimization such as classification and translation objectives on a broad knowledge that pre-training learned (Bommasani et al., 2021).

One example can show the difference between that of pre-training with fine-tuning. Pre-training offers a base model that generates coherent text upon diverse topics. Yet this might lack optimality for one specific use. Fine-tuning then “sharpens the model’s comprehension of more detailed, specific concepts” because it trains on examples for a given task from before (Bergmann, 2024a). Because the base model has already “learned” general language features, fine-tuning on a small dataset is less likely to overfit than training from scratch (Devlin et al., 2019). Medical question-answering is possible with a model fine-tuned on

medical notes pre-trained on encyclopedia text (Lee et al., 2020; Lewis et al., 2020). Modern foundation models that exist in practice often come along with specialized variants and also a general release. Meta's Llama 2 family, for example, is offered as a code-completion variant, a dialogue-tuned chat model, along with a base model (Touvron et al., 2023b). With these examples, multiple task-specific systems can be created from one fine-tuned pre-trained model. Pre-training builds broad language understanding, also fine-tuning tailors that understanding to particular applications in summary (Bommasani et al., 2021; Minaee et al., 2024).

1.5.2 Supervised and Instruction Tuning

Supervised fine-tuning (SFT) is adapting of an LLM to a task in its simplest form. In SFT, the pre-trained model is trained by use of a dataset of input–output pairs with regard to a given task. Standard supervised objectives work to minimize the prediction error for this training. Through example, this process teaches the model to perform that task, for example, translation or summarization (Raffel et al., 2020; Devlin et al., 2019). Instruction tuning places emphasis on natural-language instructions. It is also known to be a refined variant of supervised fine-tuning. In instruction tuning, each training example includes within it a human-readable instruction that describes the task that needs doing, and it also includes input as well as desired output (Sanh et al., 2021). A prompt may say, as an example, "Answer the following question:" followed by a question with an answer as the output that is the target. Instruction-tuned models can thus learn to interpret and follow diverse natural-language instructions.

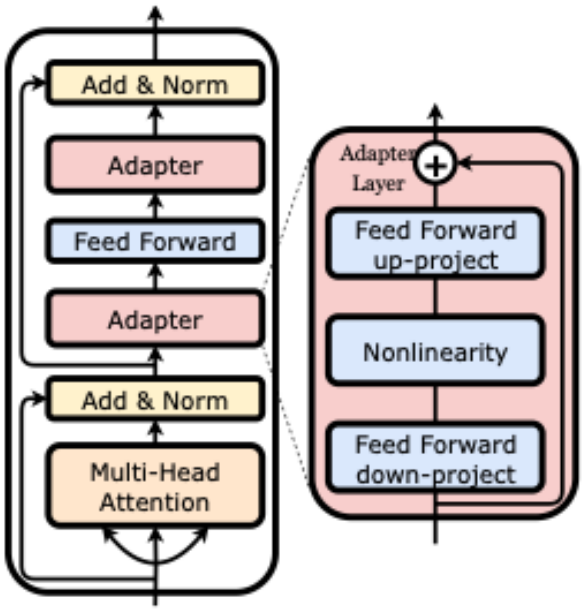
Wei et al. (2021) found instruction tuning improves generalization because instruction-tuning a 137B-parameter model "FLAN" on 60 diverse tasks makes it perform better zero-shot on new tasks. Instruction tuning involves supervised learning upon labeled (input, output) pairs, with a focus on instructions in natural language (Sanh et al., 2021).

OpenAI's InstructGPT is an influential example of instruction tuning along with combined supervision. Ouyang et al. (2022) describe standard supervised learning to fine-tune GPT-3 with collected human-written demonstrations of desired behavior. They also adjusted the model further using reinforcement learning from human feedback (RLHF) afterward to obtain human preference rankings among model outputs. The result came. It was known as an instruct-tuned model. A 1.3B-parameter InstructGPT model had greater preference than the 175B GPT-3 model in their evaluation. InstructGPT showed greater truthfulness and diminished toxicity. It was improved from that base model in some areas. The model reliably learned to follow instructions from users along with aligning to preferences of humans via this two-stage pipeline (RLHF follows supervised tuning) (Ouyang et al., 2022; Christiano et al., 2017). Thus, instruction tuning, often along with RLHF, is central to modern LLM systems, and it aligns a model's outputs with the intent behind natural-language instructions for chat and task following.

1.5.3 Parameter-Efficient Fine-Tuning

Fine-tuning within every parameter of very large LLMs can be prohibitively expensive for a large model. Ding et al. (2023) state that complete fine-tuning demands updating all 175B weights in a 175B-parameter model like GPT-3 which is "almost infeasible" practically. To address this, researchers have developed parameter-efficient fine-tuning methods modifying only a small subset of parameters often by inserting new trainable components into the model. These approaches are in some cases called "delta tuning." They freeze most of the pre-trained model, and they train just the new parameters or just a low-rank update. It does drastically reduce computational cost and reduce storage cost. Generally, it keeps correctness too (Ding et al., 2023; Hu et al., 2021).

One popular method is adapter tuning. Small feed-forward "adapter" modules that have a bottleneck architecture (Figure 10) get inserted in here between transformer layers plus only adapter weights with respect to each task get trained. Houlsby et al. (2019) have demonstrated the fact that adapters perform in a manner near state-of-the-art on GLUE benchmarks. They also add only a few more percent of parameters for each task, at around 3–4%. Only by swapping in different adapters, a single base model can support many tasks, so the model's rest remains fixed (Houlsby et al., 2019; Pfeiffer et al., 2020).



(c) Series Adapter

Figure 10 Series adapter architecture (Hu et al., 2023)

Another approach is LoRA (Low-Rank Adaptation). LoRA freezes the original model and learns additional low-rank matrices A and B (see Figure 11). These matrices can be added to the existing weight matrices, such as in attention layers. As shown by Hu et al. (2021), LoRA can diminish the quantity of trainable parameters by factors of tens of thousands while equaling full fine-tuning performance. For example, we can update only low-rank factors within GPT-3's attention layers, which involves roughly 37.7 million parameters

instead of 175B (Hu et al., 2022). LoRA adds practically no inference latency to the base model since the rank is small (Hu et al., 2021).

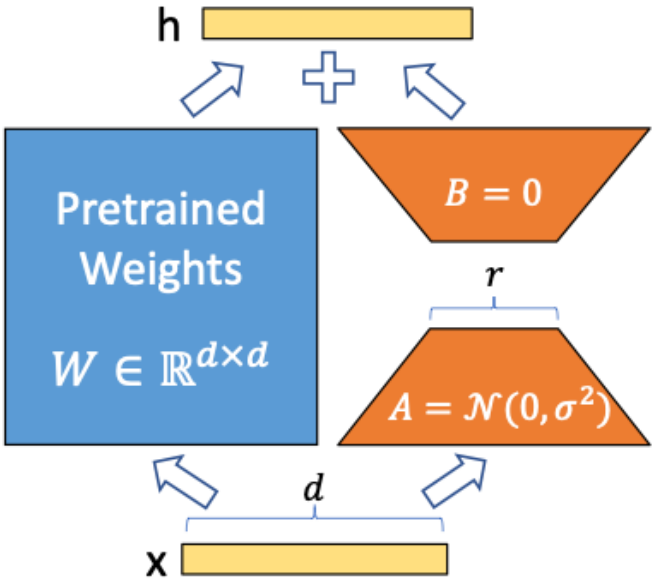


Figure 11 LoRA mechanism (Hu et al., 2021)

Prefix-tuning constitutes a third technique. In prefix tuning, someone prepends a small sequence of continuous "soft prompt" vectors to the input within each layer. These prefix vectors are trained but the entire LLM remains fixed. Li & Liang (2021) achieved performance near to full fine-tuning on generation tasks. They found that one can do this by training on just ~0.1% of parameters (the prefix). Prefix tuning steers the model effectively since it uses contextual signals instead of weight changes. For different tasks, it is often easier practically to store and to swap fixed prefixes (figure 12) than full model weights (Li & Liang, 2021; Lester et al., 2021).

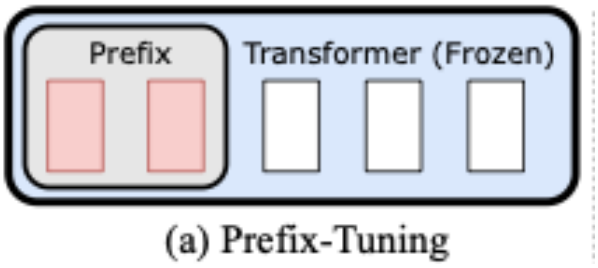


Figure 12 Prefix-Tuning mechanism (Hu et al., 2023)

Adapters, LoRA, and prefix-tuning like delta-tuning methods generally get results similar to full fine-tuning often, considerably lessening memory plus compute. Ding et al. (2023) give a unified analysis of these techniques as well as note that large PLMs can be "effectively stimulated by the optimization of a few parameters." For clarity, see the diagram of the LoRA update in Hu et al. (2021) and the depiction of how prefix vectors attend to the

model's layers in Li & Liang (2021), as they concretely show how these methods operate. Even billion-parameter models can be adapted with ease on modest hardware via these variants while retaining nearly all of the performance from full fine-tuning in practice (Ding et al., 2023).

1.5.4 Limitations and Challenges

Fine-tuning large language models offers powerful customization but also comes with several limitations and challenges. One challenge is overfitting, where a model becomes too narrowly tailored to the fine-tuning dataset and fails to generalize to new inputs (Bergmann, 2024a). Another is catastrophic forgetting – fine-tuning on new data can overwrite or destabilize the model's previously learned knowledge base (Luo et al., 2023). In practice, this means an LLM fine-tuned on a niche domain might lose some of its broader capabilities learned during pre-training.

Fine-tuning large models also incurs heavy resource costs. The process is computationally intensive and often prohibitively costly for models with billions of parameters (Bergmann, 2024a), requiring extensive GPU/TPU time and memory. This leads to a data annotation bottleneck as well: fine-tuning typically requires a substantial amount of high-quality, task-specific training data, which must be curated and annotated by humans. Obtaining such expert-labeled datasets is expensive and time-consuming in many domains (Kumar et al., 2024), limiting the ability to fine-tune LLMs frequently or for highly specialized tasks.

Reinforcement learning from human feedback (RLHF) has been introduced as a strategy to align model outputs with human preferences, but it introduces its own challenges. RLHF fine-tuning requires collecting extensive human feedback and can be complex and unstable to implement in practice (Bergmann, 2024b). Moreover, even after domain fine-tuning (with or without RLHF), alignment issues like hallucinations and biases persist. Fine-tuned LLMs may still generate plausible-sounding but incorrect facts (hallucinations) or reflect undesired biases present in the training data, posing ongoing challenges for safe and reliable deployment (Lin et al., 2024).

1.6 Retrieval-Augmented Generation (RAG)

In recent years, Large Language Models have achieved remarkable succession across a wide range of natural language processing tasks (Brown et al., 2020; Devlin et al., 2019). However, their performance on knowledge-intensive queries is inherently limited by the static nature of their parametric memory. Once trained, these models are unable to access new information or verify facts without costly retraining (Lewis et al., 2020). Retrieval-Augmented Generation (RAG) presents a promising architectural solution to this problem (Lewis et al., 2020). By combining pretrained LLMs with a document retrieval component, RAG systems can dynamically fetch relevant external knowledge at inference time and incorporate it into the generation process (Lewis et al., 2020).

This section introduces the foundational principles of RAG, explores its technical components, and evaluates its strengths and limitations as a hybrid approach for building more factual, flexible, and trustworthy language-based systems.

1.6.1 Concept of RAG

Retrieval-Augmented Generation (RAG) is a hybrid approach with external, retrievable data sources that improves large language models (LLMs) by supplementing their parametric knowledge. This framework is particularly designed in order that it reduces one of the most persistent limitations of LLMs—their inability to dynamically update or verify factual information after training (Lewis et al., 2020).

Lewis et al. (2020) solved this problem when they integrated two components using RAG: a retriever with a generator (Figure 13). The retriever identifies the relevant documents from a large corpus such as Wikipedia for a user query, and the generator, typically a transformer-based sequence-to-sequence model, uses texts that are retrieved to produce a response. Since it improves factual accuracy in knowledge-intensive tasks, this setup permits the system to create answers grounded in external (non-parametric) and internal (parametric) knowledge sources (Lewis et al., 2020).

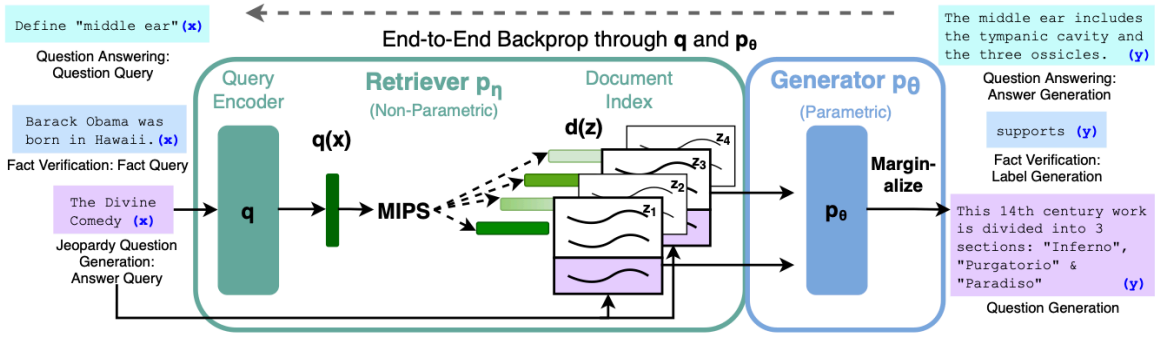


Figure 13 RAG schema: Pre-trained retriever (Query Encoder + Document Index) integrated with Generator (seq2seq), jointly fine-tuned (Lewis et al., 2020)

Prior research had already explored NLP retrieval mechanisms. For example, Field (Guu et al., 2020) and ORQA (Lee et al., 2019) showed that document retrieval has the potential to improve open-domain question answering (Guu et al., 2020; Lee et al., 2019). However, these models did mainly rely on extractive techniques. From documents, they picked spans (Lee et al., 2019; Guu et al., 2020). RAG lets generative output occur instead, giving more flexible complex open responses. This innovation was key in marking a shift from retrieval-assisted extraction. It did augment generation through retrieval in its stead (Lewis et al., 2020).

Izacard and Grave (2020) built up on this foundation and proposed such models like FiD (Fusion-in-Decoder). Because the model could attend to retrieved passages during decoding, these models further improved performance (Izacard & Grave, 2020). Factual consistency improves through evidence integration at the decoding stage rather than during early encoding according to their work (Izacard & Grave, 2020). For the applications in

which factual grounding is quite critical, these particular findings do reinforce RAG-like systems' even broader promise.

Karpukhin et al. (2020) made for us a meaningful contribution using Dense Passage Retrieval (DPR). In the original RAG implementation, DPR is a retrieval model used. DPR maps both queries and passages into one and the same embedding space for the purpose of enabling effective similarity search. Dense vector representations supersede customary keyword-based retrieval such as BM25 so they prove better suited to semantic search (Karpukhin et al., 2020).

Finally, RAG systems can offer yet one more additional benefit. Their outputs are thus more interpretable now. In order to verify the answer's source, users and developers can inspect these documents since retrieved passages are fed into the generation process. It increases transparency and reliability in outputs since it handles typical criticisms of black-box language models (Izacard & Grave, 2020; Lewis et al., 2020).

1.6.2 Parametric vs. Non-Parametric Knowledge

A fundamental concept for Retrieval-Augmented Generation (RAG) is how we distinguish between parametric and non-parametric knowledge within large language models (LLMs). RAG is truly not just only an architectural extension. This dichotomy explains that it is in fact a conceptual solution for a core limitation in language models (Lewis et al., 2020).

Parametric knowledge is what the parameters of a model do encode at a time when the model trains. An LLM such as GPT or BERT, trained on big corpora, internally soaks up factual, linguistic, and common-sense knowledge (Devlin et al., 2019; Petroni et al., 2019). Petroni et al. (2019) showed transformer-based models recall factual statements like “The capital of Czech Republic is Prague?” without any external source in their “*Language Models as Knowledge Bases?*” paper. This suggests that LLMs act as “implicit knowledge bases” since they draw answers from the statistical patterns learned during pre-training.

However, this parametric memory stays intrinsically static. Once trained, an LLM’s knowledge is effectively frozen it cannot natively acquire new facts or adjust to changes without fine-tuning or retraining (Petroni et al., 2019; Emenike, 2025; Lewis et al., 2020). This creates obvious limitations. These limitations exist in scenarios that require up-to-date or niche information. An LLM has a need for new data so that it can retrain, so it will not know of events in 2024 if it was trained in 2022. The knowledge encoded into weights is opaque furthermore there is no way to trace the origin of a given fact nor to cite a source for validation (Guu et al., 2020; Lewis et al., 2020).

Non-parametric knowledge refers to those external explicit sources for information. They include a database, document collection, or knowledge graph able to be queried at inference time, instead. The model is equipped in order to “look up” relevant facts when needed rather than just relying on its stored parameters. In RAG, retrieval accomplishes in fact this kind of process: it is fetching documents that are semantically similar from within an indexed corpus, through a user query. The prompt that was provided to the LLM includes each of these documents (Lewis et al., 2020).

It is important that retrieval-based reasoning is becoming more common. Lewis et al. (2020) along with Karpukhin et al. (2020) showed something. As their research indicated, a retrieval component improves factual accuracy with flexibility. It is possible to update the external knowledge base in an easy way without any model modification. For example, someone may add new data to a vector database and re-embed it since this guarantees the model can access current data. Retrieved content also provides a form of provenance: the system exposes the documents that influenced the answer, which increases trust and transparency (Izacard & Grave, 2020).

Emenike (2025) notes that such a hybrid memory structure of combining static parametric knowledge with dynamic non-parametric memory suits domain-specific applications quite well, where this information may in fact change so frequently or even be too detailed for embedding in these model weights. It is interpretable and adapts in real-time, non-parametric memory. Pure LLMs typically lack those two characteristics (Emenike, 2025; Guu et al., 2020).

Ultimately, parametric memory defines what a model "remembers," while non-parametric memory defines what it can "find out" inside. The RAG architecture combines these two types of knowledge, and it allows the model to generalize language fluently while grounding its responses in retrievable, verifiable evidence (Lewis et al., 2020; Izacard & Grave, 2020). This contrast between parametric and non-parametric knowledge is also illustrated in Figure 14, based on the original architecture described by Petroni et al. (2019).

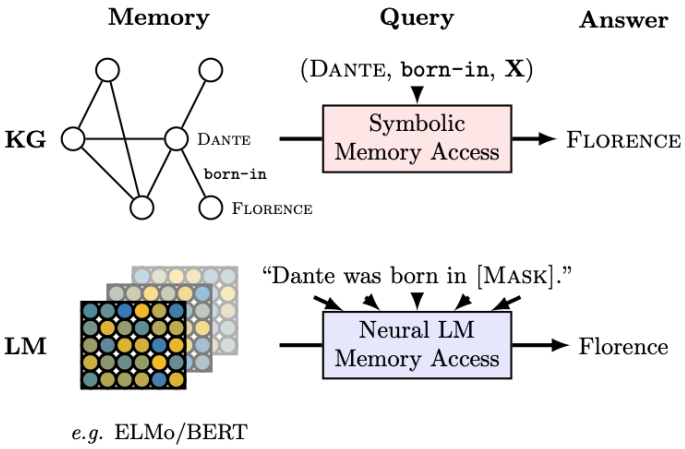


Figure 14 Parametric knowledge retrieval in language models (Petroni et al., 2019)

1.6.3 Embedding: Meaning with Vectors

To enable non-parametric retrieval within RAG systems, a common format suitable for semantic comparison must transform both user queries and knowledge documents. Embeddings accomplish this since they are high-dimensional vector representations of text. These representations capture its meaning in such a mathematically comparable way (Lewis et al., 2020).

Embeddings originate within the field of distributional semantics. The meaning of a word or of a phrase is defined by its contexts there. Mikolov et al. (2013) demonstrated this concept with effectiveness through Word2Vec for words used in similar contexts do tend to have similar vector representations. Their work did introduce the idea that certain relationships could be captured algebraically in a vector space. These relationships, like "king - man + woman ≈ queen," reflect semantic patterns (see Figure 15).

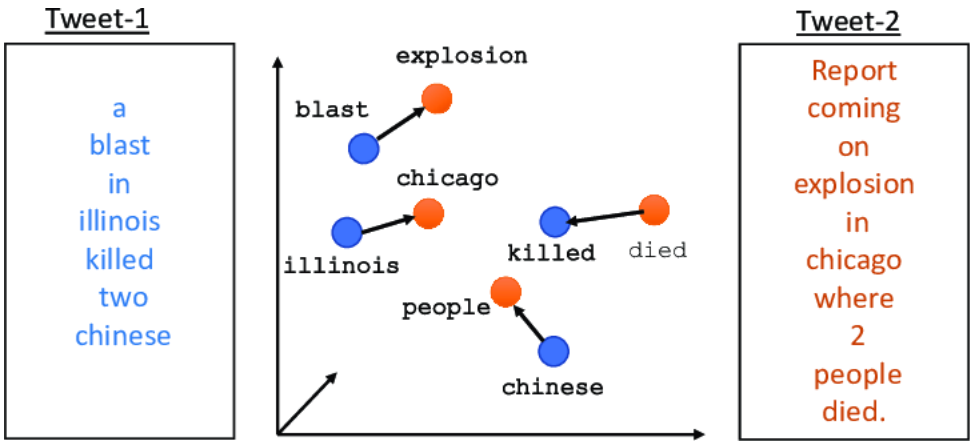


Figure 15 Semantically similar words in a word embedding space. (Iqbal et al., 2021)

Modern language models do build upon this idea and use the deep contextual embeddings. BERT and RoBERTa are models that create vector representations considering a word's context so “bank” embedding changes depending on geographical or financial context (Devlin et al., 2019). Sentence-BERT allows for entire sentences or paragraphs to be represented as just a single vector (Reimers & Gurevych, 2019) as sentence-level embeddings do work. Retrieval tasks do rely on a direct comparison between the queries and the full documents.

An embedding from more of a technical perspective is a fixed-size vector that consists of floating-point numbers generated through use of a neural network encoder having 384, 768 or 1024 dimensions (Devlin et al., 2019). Within the vector space, these embeddings map similar semantic content close together. Karpukhin et al. (2020) leveraged this principle in Dense Passage Retrieval (DPR), in which they encode queries as well as passages into the same space so metrics like the dot product or cosine similarity may directly measure their similarity.

This method is known as dense retrieval, unlike customary sparse retrieval approaches like BM25, because they rely on exact keyword matching. The system can find relevant documents when embeddings retrieve densely even if the query and document use different words to express similar meanings. For example, the query “electric vehicle battery capacity” could still retrieve a document discussing “energy storage in EVs” since their embeddings are close in vector space, even if the terms differ (Johnson et al., 2017).

For indexing, this embedding-based representation is used in RAG systems. Retrieval also uses it. During that offline phase, all of those documents within the knowledge base have been embedded. A vector index stores within it these embedded documents. A query submitted by a user is embedded using the same model along with document embeddings

most similar are retrieved using similarity search (Karpukhin et al., 2020; Lewis et al., 2020). The LLM will then receive these very documents as context for its answering.

The quality of the embedding model matters greatly. Poor quality means the system will generally perform badly. Systems might retrieve passages that are weakly related or passages that are irrelevant because of poor embeddings. That can degrade the level of quality in any answer that the system may generate. A critical design decision exists for any RAG pipeline since anyone selects an embedding model suited to the domain (Reimers & Gurevych, 2019; Izacard & Grave, 2020).

In summary, RAG systems use embeddings that enable semantic similarity search at scale. Regardless of word choice or surface form, the system efficiently identifies the most relevant knowledge to support generation via projecting queries plus documents into the same vector space. Because dense retrieval relies on this functionality, therefore modern retrieval-augmented systems rely on it too.

1.6.4 Similarity Search and Cosine Similarity

Once documents and queries get represented as embeddings within a shared semantic space, identifying relevant documents requires a certain mechanism for a given query. This mechanism seeks similar items a process retrieving vectors nearest (i.e., documents) to one query vector using a similarity metric (Lewis et al., 2020).

Cosine similarity remains a text embedding metric. It is used most of all. Cosine similarity measures the angle between two vectors because it captures just how closely they point in one and the same direction regardless of what their magnitude may be. Two vectors that have a cosine similarity equaling 1 are aligned in a perfect manner with one identical direction. Orthogonal vectors are similar by 0 that indicates there is no correlation (Jurafsky & Martin, 2020). The formal definition is:

$$cosine\_similarity(A, B) = \frac{A * B}{\|A\| * \|B\|}$$

where $A * B$ is the dot product of vectors A and B, and $\|A\|$,$\|B\|$ are their Euclidean norms. This metric is appropriate especially for textual data. In textual data, semantic meaning of two texts has more importance than length or frequency. Cosine similarity turns into something that is mathematically equivalent to what the dot product is because embeddings are often normalized to get to unit length in RAG systems (Reimers & Gurevych, 2019).

For example, consider a query that asks as to “best fuel-efficient SUV 2022”. A vector of high dimension embeds within it this query. The system then calculates cosine similarity between this vector for a query and every vector for a document in the base for knowledge. Documents concerning hybrid or electric SUVs with high MPG ratings will likely have high similarity scores. So, those documents will probably be chosen for retrieval (Karpukhin et al., 2020).

However, in order to calculate cosine similarity for use between a query with every document inside a large corpus may well be computationally expensive. A brute-force

search might work for small datasets, but this becomes impractical when you index many documents. For this balance of retrieval speed and accuracy, modern systems use Approximate Nearest Neighbor (ANN) search algorithms. The Hierarchical Navigable Small World (HNSW) graph is one popular method for allowing efficient navigation through high-dimensional vector spaces by building layered graphs (Malkov & Yashunin, 2016).

ANN algorithms are implemented inside tools like FAISS (Johnson et al., 2017) or pgvector (used in this thesis) in practice to perform fast similarity search using cosine distance. These libraries and extensions allow for embeddings to be indexed and stored and retrieved efficiently in real-time applications. By integrating it with PostgreSQL, pgvector supports cosine similarity directly within SQL queries, thus enabling smooth vector search inside a relational database (Richardson, 2023).

Tuning the number of retrieved results is also important (commonly denoted top-k). Relevant documents may be missed, when k is problematic. If it is too high, irrelevant documents may dilute context passed to the LLM. In most of the RAG setups, typical values for k are from 3 up to 10. Yet this may fluctuate depending on application and model context window restrictions (Lewis et al., 2020).

To put it briefly, there exists semantic matching for embedded queries and documents. Too, cosine similarity allows for this matching. RAG systems combine efficient ANN algorithms together with indexing libraries to scale retrieval to large corpora, maintaining high relevance and speed (Lewis et al., 2020).

1.6.5 Vector Databases and PGvector

As RAG systems do rely on similarity search within large sets of document embeddings, a mechanism for indexing then storing efficiently is necessary. A vector database serves in this role since it is a system that stores these high-dimensional embeddings also it rapidly searches for closest neighbors through the use of similarity metrics like cosine similarity and inner product (Lewis et al., 2020; Johnson et al., 2017).

Vector databases are built for approximate nearest neighbor (ANN) search, with the goal being to find the vectors most similar to a query vector inside a tolerable margin of error (Johnson et al., 2017), unlike customary relational databases, that are optimized for exact matching as well as structured queries. These systems do typically support ANN indexing algorithms that are such as HNSW (Hierarchical Navigable Small World graphs) or IVF (Inverted File Indexes) because they do accelerate search over millions of vectors (Malkov & Yashunin, 2016).

FAISS, Milvus, Weaviate, along with Pinecone are popular standalone vector search engines, offering high performance for scalability. FAISS is widely used by researchers and producers especially to index embedding collections of large scale. FAISS has GPU acceleration to help workloads achieve high throughput according to Johnson et al. (2017).

Yet for projects already using typical relational databases like PostgreSQL, integrating vector search right into SQL workflows is frequently helpful. Here, pgvector, an open-source PostgreSQL extension, gains relevance. pgvector allows for the storage of vector

embeddings within native database columns. It also supports similarity queries through cosine distance, Euclidean distance, and also inner product (Richardson, 2023).

Each row can stand for a document that has its own embedding vector for it. This can occur within a pgvector-enabled database for example a product spec paragraph or article. With appropriate indexing like HNSW or IVF the database can rapidly retrieve the top-k most similar rows to a given query embedding. A query, for example, such as “Which cars have similar performance to the 2020 Audi A4?” can with SQL be embedded like the query that retrieves the nearest document embeddings (Richardson, 2023; Johnson et al., 2017).

This integration lets RAG pipelines stay within the familiar as well as strong SQL environment, while still they leverage modern vector search capabilities. Hybrid queries can also be enabled so they combine structured filters such as "WHERE brand = 'Audi'" with semantic similarity, which is a feature that standalone vector databases do not typically offer (Timescale, 2024).

Pgvector offers a great balance for focused RAG systems especially when using datasets requiring close ties with structured metadata even if it cannot equal FAISS's raw speed for huge datasets (Johnson et al., 2017).

In this thesis, pgvector was used to build a vector index over car specification document embeddings. Relevant entries could be retrieved efficiently by users. The LLM then did receive the results dynamically as some contextual input. This useful design shows vector-native relational databases are feasible. This can support end-to-end RAG systems in real-world scenarios (Richardson, 2023).

1.6.6 RAG schema

Usually, the RAG process occurs throughout two stages: an offline stage when someone prepares and indexes the knowledge base, and an online stage. Queries are processed subsequently, also answers are generated in real time (Lewis et al., 2020; Yeung, 2024).

1. Offline Phase: Preparing the Knowledge Base (see Figure 16)

1. Document Ingestion: The knowledge corpus (e.g., product specs, articles, FAQs) is preprocessed and split into passages or chunks. Each passage is treated as an independent retrievable unit.
2. Embedding Generation: Each passage is converted into a dense vector using an embedding model—commonly a transformer encoder like Sentence-BERT or a bi-encoder from Dense Passage Retrieval (Reimers & Gurevych, 2019; Karpukhin et al., 2020).
3. Indexing: These vectors are stored in a vector database such as FAISS or pgvector. An ANN index (e.g., HNSW) is created to enable fast similarity-based retrieval (Johnson et al., 2017; Richardson, 2023).

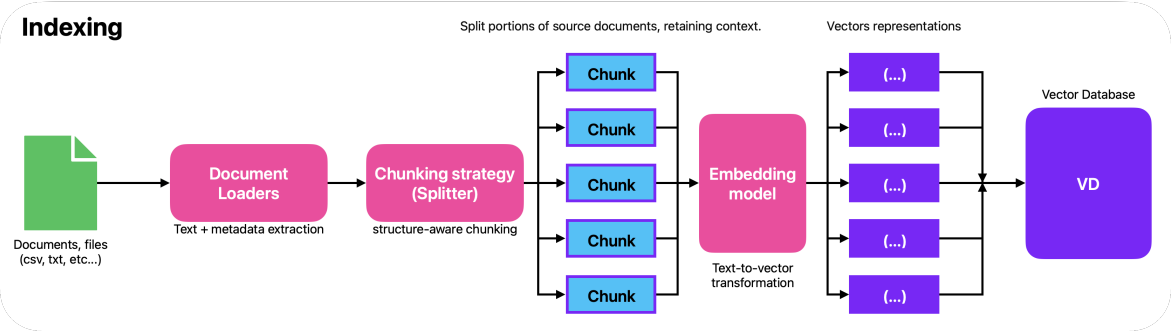


Figure 16 Offline Phase of RAG (source, author)

This phase is computationally intensive but only needs to be run once, or periodically if the corpus is updated.

2. Online Phase: Responding to Queries

1. Query Embedding: When a user submits a question, the system encodes it into a query vector using the same embedding model as used during indexing.
2. Similarity Search: The query vector is passed to the vector database, which retrieves the top-*k* most semantically similar document vectors using cosine similarity or dot product (Malkov & Yashunin, 2016).
3. Context Assembly: The text from the retrieved documents is aggregated into a prompt format. Some systems apply lightweight ranking or filtering (e.g., using metadata or content scoring).
4. Prompt Construction: A new prompt is created by appending the retrieved passages to the original user query, often with separators, headings, or system instructions o guide the model's attention and response format (OpenAI, 2023).
5. Response Generation: This composite prompt is passed to the language model (e.g., BART, T5, or GPT-4), which generates an answer conditioned on the retrieved information.

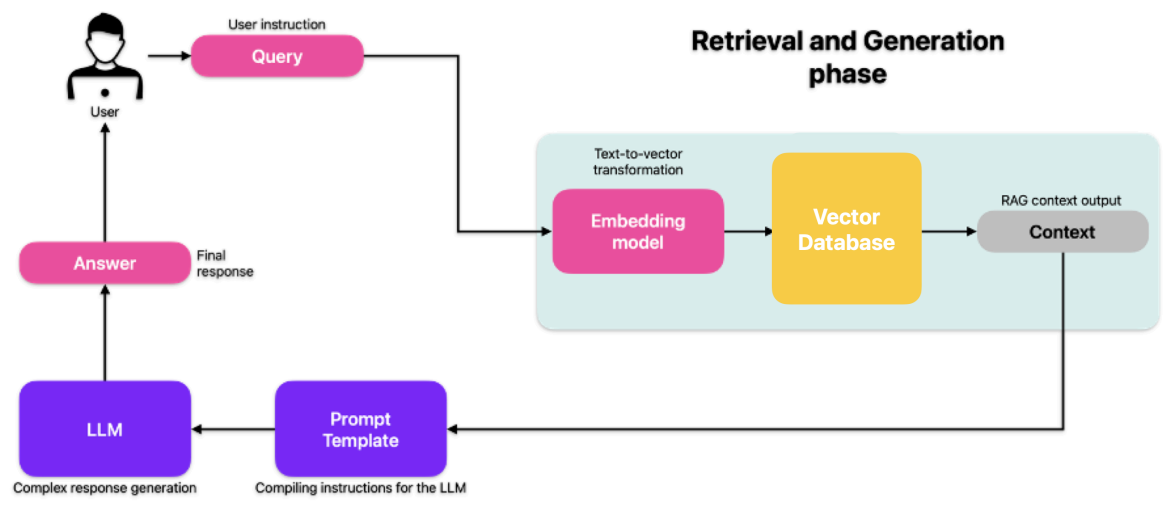


Figure 17 RAG schema (source, author)

Optionally, some implementations will apply post-processing steps such as citations, repetition, or hallucinated content may be filtered.

This architecture is modular along with it meaning that the retriever and generator are able to be trained separately. This simplifies development. Individual components can as a result be tuned for specific domains. For instance, the retriever can be fine-tuned for better capture of automotive terminology (Lewis et al., 2020), while the generator remains unchanged.

One-shot retrieval with a fixed top-k is used in most applications because it is efficient and sufficient (Izacard & Grave, 2020), but some RAG variants like RAG-Token support iterative retrieval where documents can be retrieved dynamically during each step of decoding.

1.6.7 Benefits of Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) gives advantages that address the main limitations of purely parametric large language models (LLMs) (Lewis et al., 2020; Gao et al., 2023). RAG improves transparency, factual accuracy, scalability, coupled with flexibility, via a combination of generative capabilities together with external document retrieval. RAG is in fact a compelling architecture since it has these advantages that are for applications requiring knowledge reliability (Wu et al., 2024).

RAG has an ability for a reduction in hallucinated content that is also one of its most widely cited strengths. LLMs often produce incorrect yet fluent answers, especially when knowledge is of low frequency or specific (Ji et al., 2022). RAG reduces this by supplying the model semantic relevant documents at inference time. Lewis et al. (2020) did show a certain something. Their grounding of generation in the retrieved evidence does lead to a greatly higher factual accuracy in open-domain question answering benchmarks. The passages that were retrieved do provide for an anchor point. The anchor point assists the model in avoiding claims without support (Izacard & Grave, 2020).

Static, parametric models cannot incorporate new information after training unless they are re-trained or fine-tuned, which is resource-intensive and time-consuming. In contrast, RAG systems can be updated simply by modifying the document database (Guu et al., 2020). Access to the latest data in real time is enabled, important for fields like legal tech, health care, or e-commerce. Furthermore, RAG can easily specialize in niche domains since it indexes domain-specific corpora, to avoid the need for costly and risky model fine-tuning (Izacard & Grave, 2020).

Since the architecture for RAG is modular, the retriever and generator can be optimized and developed independently (Lewis et al., 2020). Because of their needs, this allows developers to swap or fine-tune components such as using a domain-adapted retriever with a general-purpose language model. To separate concerns simplifies the experimenting and iterating within applied NLP systems (Gao et al., 2023).

We do not often train quite large LLMs for the purpose of memorizing more facts. RAG enables smaller or mid-sized models to obtain information in a dynamic way. This increases

achievement regarding information-heavy work. Model scaling is not massively required (Karpukhin et al., 2020). Frequent retraining becomes less necessary. This reduces burdens. RAG is in fact a cost-effective alternative for those solely relying on parametric knowledge, as a result of that (Su et al., 2025).

RAG can show particular sources used when answers generate. LLMs of a standard kind cannot do even this. Through this traceability, user trust is improved. Therefore, citations and justification are supported in use cases like customer-facing support systems or scientific summarization. Developers and users can inspect retrieved passages so models' output is better understood (Lewis et al., 2020).

RAG systems possess intrinsic scalability. As new information appears, documents may be added to the vector store, without changes to the model. Furthermore, RAG enables also hybrid reasoning: the model can combine that which it has “memorized” with what it has retrieved, which makes the model stronger in evolving domains or in domains that are unfamiliar (Yeung, 2024). Systems are able to effectively handle common-sense and fact-based queries through the use of this dual approach.

RAG improves generative language systems overall. It also makes each of those systems to be more accurate and adaptable and trustworthy. It offers an elegant architectural compromise regarding static memory with dynamic lookup, which enables LLMs to become more reliable tools for real-world applications (Gao et al., 2023).

1.6.8 Limits of Retrieval-Augmented Generation

Despite its meaningful advantages, limitations exist within Retrieval-Augmented Generation (RAG). Implementing a solid RAG system requires managing some real and abstract challenges like retrieval quality, latency, integration complexity, and explainability gaps. To build systems, these limitations must be understood. Such systems must be reliable and also maintainable (Lewis et al., 2020).

The accuracy for a RAG system depends in a fundamental way on the quality for the retriever. If retrieved top-k documents are irrelevant, outdated, or ambiguous, the language model will probably generate outputs that are incorrect or incoherent (Lewis et al., 2020). Although powerful, dense retrievers are only as good as the underlying corpus along with the embedding model. For semantically similar embeddings, factual relevance is just not guaranteed. Izacard along with Grave (2020) highlighted that retrieval errors can propagate downstream. Because of this the generator may confuse or hallucinate information.

RAG does introduce a further retrieval step that is between the user query and generation. Inference time is thus increased in comparison to pure LLM responses. This includes computing of the query embedding, searching throughout a large vector index, also retrieving documents. A combined prompt will also need to be constructed by you. FAISS and pgvector are fast ANN libraries decreasing this burden yet real-time applications may find added latency concerning (Malkov & Yashunin, 2016; Richardson, 2023).

Current language models are constrained by fixed context lengths (e.g., 4K, 8K, 32K, 128K tokens), limiting how much retrievable information in the prompt they can include. Too

many retrieved documents require summarization or truncation. Because of summarization or truncation, important information may be lost. Also, irrelevant content in the prompt can dilute attention so the model may worsen (Ji et al., 2022). This needs appropriate context and passage choice.

When RAG is being deployed, it involves the integration of multiple different components: the retriever such as DPR, the vector database such as FAISS or pgvector, and also the generator such as GPT. For managing update pipelines and for synchronizing embedding models, careful coordination is what is needed. It is also something that is needed for ensuring encoding between index and retriever (Lewis et al., 2020). Debugging a modular system is also harder: pinpointing the error source between retrieval, prompting, or generation may be unclear.

Sometimes, the language model's parametric knowledge may conflict with the retrieved non-parametric evidence. Facts retrieved can be overridden or conflated by the model (Emenike, 2025). This may happen when the model thinks data is old or wrong. This can be especially problematic in cases where the model saw outdated information throughout pretraining, though someone presented newer data around inference time. It is not possible to guarantee that the model will prioritize the retrieved evidence, and current RAG architectures do not explicitly resolve these conflicts (Wu et al., 2024).

For evaluation, standalone LLMs are simpler than assessing RAG output quality. Standard accuracy metrics may not capture factual consistency, context integration, or retrieval relevance. Whether a correct answer was generated thanks to or despite retrieval is often unclear still. Research has proposed multi-component evaluation via retrieval precision or grounded generation metrics. Such benchmarks are still being researched (Lewis et al., 2020; Ji et al., 2022).

If RAG is applied to private or sensitive data such as internal documents or health records, strict access controls must be enforced. Vector databases can leak data due to embeddings or due to caching. This leakage is inadvertent. Additionally, retrieved context can be exposed within LLM outputs which does raise some privacy risks (Yeung, 2024). Careful design is needed, and it features query logging, sanitization, with retrieval filtering.

RAG ultimately addresses many weaknesses in purely parametric models. However, it is also introducing some new engineering and design challenges. These do include interpretability with latency in system complexity and also managing retrieval fidelity. Careful architectural decisions as well as domain-specific tuning are indeed required when addressing each of these limitations. LLM-based adaptable fact-aware systems can be buildable using RAG even despite these hurdles.

2 Practical Implementation (Design and Development)

This chapter presents the design and implementation of a domain-specific Retrieval-Augmented Generation (RAG) system built for automotive knowledge retrieval. The goal of this implementation is to enhance the factual accuracy and contextual relevance of responses generated by a large language model (LLM), while avoiding the limitations of static pre-trained models (e.g., hallucination or outdated information). The pipeline is structured to allow comparison between a naive, context-free baseline (Basic Pipeline) and a more robust retrieval-enhanced system (RAG Pipeline).

The implementation was done in Python using OpenAI's GPT-4o-mini and text-embedding-3-small APIs, PostgreSQL with pgvector for semantic search, and custom evaluation tools.

2.1 Data Preparation

The raw dataset was sourced from the Kaggle Car Specification Dataset, which provides technical and structural metadata for more than 29,000 cars. While the data is rich, its format is inherently structured (tabular CSV), making it unsuitable for direct ingestion into LLMs or semantic similarity search engines.

2.1.1 Natural Language Transformation

Structured fields such as Model, Year, Fuel Type, and Torque must be transformed into natural-language sentences for LLMs to process effectively. This is necessary for semantic embedding models, which expect paragraph-level inputs to understand and represent meaning in high-dimensional vector space.

A custom Python function was created to convert each row into a descriptive text (see Appendix A):

Code listing 1: Generating a Natural Language Description from Structured Data (source: author)

```
def natural_language_description(row):

    parts = []

# Check if the Model field exists and is not null
if pd.notnull(row['Model']):
    # Start the description with the car model
    parts.append(f"The {row['Model']}")
```

```
# Check if Production years exist
if pd.notnull(row['Production years']):
    # Append production years to the previous part (model)
    parts[-1] += f" produced in ({row['Production years']})"

# Check if Company (manufacturer) exists
if pd.notnull(row['Company']):
    # Add manufacturer information
    parts.append(f"was manufactured by {row['Company']}")

# Check if Body style exists
if pd.notnull(row['Body style']):
    # Add body style information
    parts.append(f"as a {row['Body style']}")

# Check if Segment exists
if pd.notnull(row['Segment']):
    # Add market segment information
    parts.append(f"in the {row['Segment']} segment")

# Check if Cylinders information exists
if pd.notnull(row['Cylinders']):
    # Add engine cylinder configuration
    parts.append(f"with a {row['Cylinders']} engine")

# Check if Displacement exists
if pd.notnull(row['Displacement']):
    # Append engine displacement to the previous part (cylinders)
    parts[-1] += f" of {row['Displacement']}"

# Check if Power(HP) exists
if pd.notnull(row['Power(HP)']):
    # Add horsepower information
    parts.append(f"producing {row['Power(HP)']}")

# … this is replicated to all features. For more see Appendix A

    return " ".join(parts) # Join all parts into a paragraph
```

This generated a new Description column for every record, which was later embedded and used in retrieval. For better understanding Figure 17 was designed.

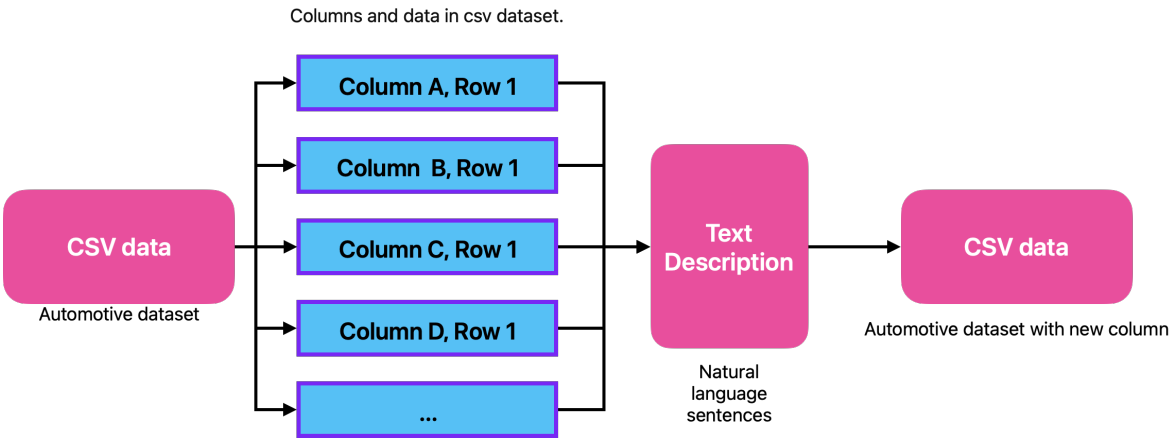


Figure 18 Dataset setup (source, author)

2.1.2 Dataset Filtering for Practicality

Due to testing constraints, the dataset was limited to 10,000 rows:

Code listing 2: Loading and Exporting a 10,000-Row Subset of the Original Data (source: author)

```
# Load the first 10,000 rows of the CSV file into a DataFrame
df = pd.read_csv('processed/cars-for-rag-natural.csv', nrows=10000)

# Save the DataFrame to a new CSV file without the index column
df.to_csv('processed/cars_for_rag_natural_10000.csv', index=False)
```

This subset was selected to preserve a representative cross-section of vehicles from different brands, body types, fuel systems, and performance classes.

2.2 PostgreSQL Integration and Vector Setup

To enable efficient semantic search over the car dataset, the data was imported into a PostgreSQL database and enhanced with vector capabilities using the pgvector extension. This involved creating a structured table, importing all original features along with a newly generated natural-language description, and setting up vector embeddings and indexing for similarity search. These steps laid the foundation for fast and meaningful retrieval based on text representations rather than exact keyword matches.

2.2.1 Table Creation and Data Import

Once the CSV file with the newly generated Description column was ready, the next step was to create a table in PostgreSQL and import the data:

Code listing 3: Initializing the cars Table with Source Data (source: author)

```
/*
Creates a table named cars with various columns like ID, Model, Company, etc.
*/
CREATE TABLE cars (
    "ID" INTEGER,
    "Model" TEXT,
    "Serie" TEXT,
    "Company" TEXT,
    "Body style" TEXT,
    "Segment" TEXT,
    ... -- all original fields
    "Description" TEXT
);

/*
Copy csv data to PostgreSQL database.
*/

\copy cars FROM '/path/to/cars_for_rag_natural_10000.csv' DELIMITER ',' CSV HEADER;
```

All 56 original features were preserved to retain complete domain fidelity. The description field was added as a natural-language textual representation used for retrieval.

2.2.2 Embedding Column and Indexing

After importing the CSV into the database, a new column was added to store vector embeddings, and indexing was configured:

Code listing 4: PostgreSQL Setup for pgvector-Based Retrieval (source: author)

```
-- creates a new column with vector datatype
ALTER TABLE cars ADD COLUMN embedding vector(1536);

-- enables the pgvector extension in PostgreSQL
CREATE EXTENSION IF NOT EXISTS vector;

/*
Creates an index called cars_embedding_idx on the cars table. It uses the IVFFlat indexing
method, which is good for fast similarity search in large datasets. The index is built on
the embedding column, and it uses cosine similarity to compare vectors. The WITH (lists =
100) part means the data is divided into 100 clusters to speed up search.
*/
CREATE INDEX cars_embedding_idx
```

```
ON cars
USING ivfflat (embedding vector_cosine_ops)
WITH (lists = 100);
```

Why lists = 100?

The lists parameter in IVFFLAT indexing determines how the dataset is partitioned into clusters for approximate nearest-neighbor (ANN) search. A commonly used heuristic is to set the number of lists to the square root of the total number of records (i.e., $lists \approx \sqrt{n}$), which balances search accuracy and efficiency (RAPIDS, 2025).

In this project, with approximately 10,000 records, lists = 100 provides a good balance between retrieval **efficiency** (query speed) and **recall** (quality of the nearest neighbors found). Using too few lists slows down queries because more vectors need to be scanned, while too many lists may reduce accuracy by limiting the search to overly small partitions.

2.3 Embedding generation

The system uses OpenAI's text-embedding-3-small model to convert natural language descriptions into vectors. Each vector is 1536-dimensional and captures the semantic content of the vehicle description.

A custom batch-processing script was created to embed descriptions and update the database (see Appendix B):

Code listing 5: Generating and Storing Embeddings in the cars Table (source: author)

```
# Constants for batch processing
BATCH_SIZE = 100 # Number of records processed in one batch

while True:
    # Select records without embeddings from the database
    cur.execute('SELECT "ID", "Description" FROM cars WHERE embedding IS NULL LIMIT %s;',
(BATCH_SIZE,))
    rows = cur.fetchall()

    # Exit loop if no more records to process
    if not rows:
        break

    # Extract IDs and descriptions from records
    ids = [row[0] for row in rows]
    descriptions = [row[1] for row in rows]

    # Generate embeddings using OpenAI model
    response = openai.embeddings.create(input=descriptions, model="text-embedding-3-small")
    embeddings = [item.embedding for item in response.data]

    # Update database records with new embeddings
    for car_id, embedding in zip(ids, embeddings):
        cur.execute("UPDATE cars SET embedding = %s WHERE \"ID\" = %s;", (embedding, car_id))

    # Save changes to database
    conn.commit()
```

For better visualization of the logic behind this function see Figure 18.

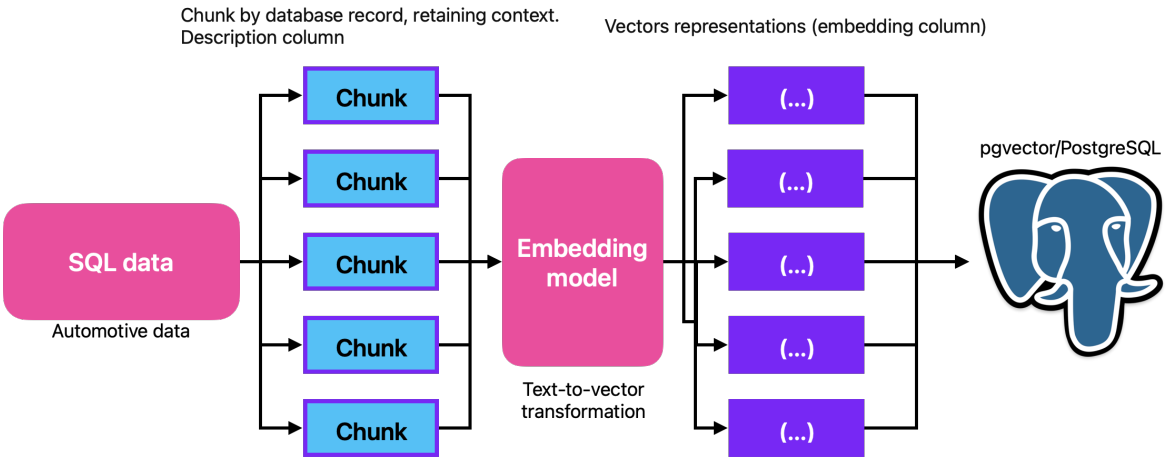


Figure 19 Embedding phase of description data (source, author)

2.4 Pipeline Design

Rather than relying on a single approach, the system was built with two distinct pipelines to test how access to external data influences performance. One uses the language model in isolation, while the other augments its reasoning with retrieved car descriptions from a vector database. This split makes it possible to measure the practical benefits of retrieval-augmented generation (RAG) in a controlled setting.

2.4.1 Two-Pipeline Architecture

The project implements **two separate pipelines** to allow empirical comparison:

- Basic Pipeline: Directly queries GPT-4o-mini without any external knowledge.

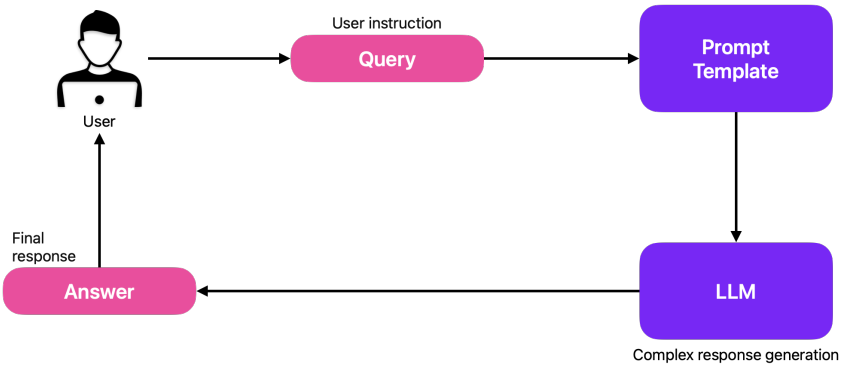


Figure 20 Basic pipeline, (source, author)

- RAG Pipeline: Retrieves relevant descriptions from the database and appends them as context.

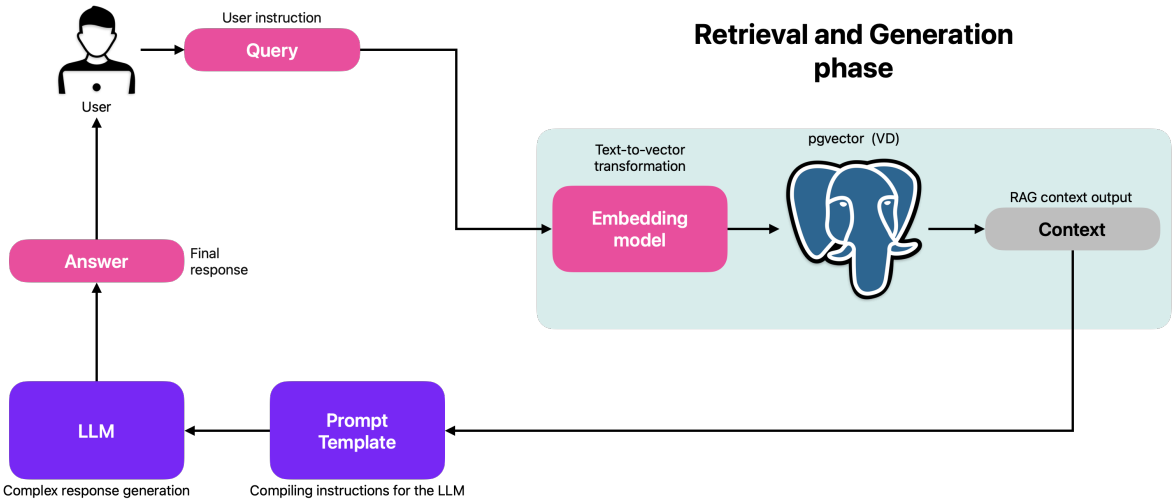


Figure 21 RAG pipeline (source, author)

This setup allows objective comparison of response accuracy and the occurrence of hallucinations.

2.4.2 RAG Retrieval Strategy

In this system, each database row represents a semantic unit (a "chunk"). Choosing LIMIT 5 allows for:

- Sufficient diversity and coverage in expert queries (e.g., "Which Audi models have automatic transmission?")
- Balanced token count (~1000-2500 tokens with context)
- Improved support for **expert-level** queries

This number was chosen based on practical testing and observations during prompt construction.

2.4.3 RAG Pipeline Code Overview

The RAG pipeline is the core of the retrieval-augmented approach. Unlike the basic pipeline that forwards user questions directly to the LLM, this pipeline first retrieves semantically relevant documents (in this case, car descriptions) and uses them as context for generating responses.

The process can be broken down into the following steps:

1. **Query embedding**: The user query is first converted into a high-dimensional vector using the same embedding model (text-embedding-3-small) as was used for the dataset.
2. **Vector similarity search**: This embedding is used to perform an ANN (approximate nearest neighbor) search in the PostgreSQL database using pgvector. The search retrieves the top-k most similar entries based on cosine similarity.
3. **Prompt construction**: The retrieved car descriptions are formatted and concatenated as a system-level prompt.
4. **LLM inference**: The constructed prompt is passed to GPT-4o-mini to generate a final, grounded answer.

Code listing 6: Retriever and Prompt Construction in RAG Pipeline (source: author)

```
class RAGPipeline:
    def retrieve_context(self, query: str, top_k: int = 5):
        # Generate an embedding vector for the input query using the LLM embedding model
        query_embedding = self.get_embedding(query)
        # Convert the embedding vector to a string format suitable for SQL
        embedding_str = "[" + ",".join(str(x) for x in query_embedding) + "]"

        # Query the database to find the top_k most similar records
        # The <=> operator computes the distance between the query embedding and each record's
embedding
        # Results are ordered by similarity (lowest distance first)
        # top_k determines how many most relevant records are retrieved (default is 25)
```

```
        cur.execute("""
            SELECT "ID", "Model", "Description"
            FROM cars
            WHERE embedding IS NOT NULL
            ORDER BY embedding <=> %s
            LIMIT %s;
        """, (embedding_str, top_k))
        # Return the list of most relevant records (as tuples)
        return cur.fetchall()

    def get_response(self, query: str):
        # Retrieve the most relevant context records from the database for the given query
        context_results = self.retrieve_context(query)
        # Format the retrieved context into a readable string for the language model prompt
        # Each record includes the car model and its description
        context = "\n".join([f"Model: {row[1]}\nDescription: {row[2]}" for row in
context_results])

        # Construct the prompt for the language model:
        # - The system message instructs the model to use only the provided context
        # - The user message includes the context and the actual question
        messages = [
            {"role": "system", "content": "You are a helpful assistant. Use only the provided
context to answer the question. Be as precise and concise as possible, directly addressing
the question."},
            {"role": "user", "content": f"Context:\n{context}\n\nQuestion:\n{query}"}
        ]

        # Send the prompt to the language model and get the generated answer
        response = self.client.chat.completions.create(
            model=self.model,
            messages=messages
        )
        # Return only the answer part of the model's response
        return response.choices[0].message.content
```

This design ensures that answers are **based on up-to-date, accurate vehicle specs** retrieved in real time.

2.5 Test cases

Test cases were executed using both pipelines for direct comparison. Below is an example of running both pipelines for a specific question:

Code listing 7: Query Execution Using Basic and RAG Pipelines (source: author)

```
# Import the RAGPipeline and BasicPipeline classes from their respective modules
from rag_pipeline.pipeline import RAGPipeline
from basic_pipeline.pipeline import BasicPipeline

# Initialize the basic pipeline (direct LLM, no context retrieval)
basic = BasicPipeline()

# Initialize the RAG pipeline (LLM with retrieval-augmented context)
rag = RAGPipeline()

# Define the user query to be answered
query = "What is the torque of the Alfa Romeo 166 2.0 Twin Spark?"

# Get and print the answer from the basic pipeline
print("Basic:", basic.get_response(query))

# Get and print the answer from the RAG pipeline
print("RAG:", rag.get_response(query))
```

This comparison demonstrates that the RAG pipeline outperforms the basic approach especially for specific or technical queries where the LLM may hallucinate or lack factual detail.

For complete pipeline implementations, see Appendix C (RAG pipeline) and Appendix D (Basic pipeline).

Chapter 3 will formally evaluate both pipelines using defined ground truth set.

3 Evaluation

Evaluation plays such a major role as the validation for the practical contribution of this thesis. In this work, we assess the implemented Retrieval-Augmented Generation (RAG) system. The evaluation assesses just how effectively the system improves upon factual correctness that is in the automotive domain. Comparing a language model used without access to external knowledge to a model supported by retrieval from a structured knowledge base and ChatGPT acting as a reference benchmark remains the objective.The chapter outlines the evaluation setup, question generation strategy, scoring method, and results that demonstrate the effectiveness of the retrieval-based approach in answering domain-specific, expert-level questions.

In industry scenarios such as customer support, automotive diagnostics, or product comparison tools, factual reliability is crucial. Systems that hallucinate or guess values can cause confusion or errors in decision-making. This evaluation framework is thus not only a research method but also simulates practical deployment conditions.

3.1 Evaluation methodology and setup

The goal of this chapter is to thoroughly evaluate the implemented system's ability to answer factual, domain-specific queries by comparing the performance of three different configurations:

- **Basic Pipeline**, which uses a large language model (LLM) in isolation.
- **RAG Pipeline**, which augments the model with relevant retrieved data.
- **ChatGPT assistant** with direct access to a CSV file containing the dataset, serving as a benchmarking tool but not part of the deployed system.

3.1.1 Motivation and Hypothesis

This evaluation aims to answer the central research question posed in the thesis:

- How can Retrieval-Augmented Generation (RAG) be used to enhance a general-purpose Large Language Model's performance as well as factual correctness on domain-specific queries?

The primary assumption is that the use of retrieval improves factual grounding, especially in expert-level domains such as automotive specification data. RAG should reduce hallucinations, increase factual depth, and enrich the quality of responses by anchoring answers in domain-specific knowledge.

3.1.2 Questions Creation Process

To ensure valid, answerable, and verifiable evaluation, questions were generated by analyzing the dataset itself rather than by inventing them arbitrarily. The key steps were:

1. Feature selection: Ten technical attributes were chosen from the dataset, such as Torque(lb-ft), Electrical motor torque, CO2 Emissions, Turning circle, Cylinders, and Displacement.
2. Random manufacturer-based sampling: To ensure variety, a modified version of the script selected two different manufacturers per feature and chose one valid model each — balancing technical depth with dataset diversity.
3. Ground truth pairing: For each selected vehicle, a natural-language question was formulated (e.g., "What is the torque of the Ford Super Duty Tremor?") and paired with the exact ground truth value.
4. Focus on answerability: Only records with valid data for the selected attribute were used. This ensured that the system could retrieve and respond with the correct value.

3.1.3 Script Example

A key part of the methodology involved using the following script logic (Appendix E):

Code listing 8: Feature-Based Sampling of Vehicle Models (source: author)

```
for feature in features:
    # Filter out already used models and rows without the feature
    available_cars = df[~df['Model'].isin(used_models)]
    random_cars = available_cars.dropna(subset=[feature]).sample(n=10)
    # Keep only one car per manufacturer
    manufacturer_cars = {}
    for _, car in random_cars.iterrows():
        if car['Company'] not in manufacturer_cars:
            manufacturer_cars[car['Company']] = car
    # Randomly select up to two distinct manufacturers
    selected_cars         =         random.sample(list(manufacturer_cars.values()),k=min(2,
len(manufacturer_cars)))
    for car in selected_cars:
        used_models.add(car['Model'])
```

This allowed for a fair, diverse, and valid sample of questions from across the automotive domain.

3.1.4 Prompt Engineering and Test Environment

To keep the evaluation fair and controlled, the following prompt instructions were used:

Code listing 9: Prompt Configuration for Evaluation Pipelines (source: author)

- **RAG system:**

```
System prompt: "You are a helpful assistant. Use only the provided context to answer the question. Be as precise and concise as possible, directly addressing the question."
User prompt: "Context: [retrieved descriptions]\n\nQuestion: [query]"
```

- **Basic pipeline:**

```
System prompt: "You are a helpful assistant. Be as precise and concise as possible, directly addressing the question."
User prompt: "[query]"
```

- **ChatGPT assistant:**

```
You are a specialized data assistant. You receive a CSV file as input, which serves as the source of truth for answering questions.

Rules:
- Answer mainly using information from the CSV file. Do not use general knowledge or make assumptions.
- Be as brief, clear, and direct as possible.
- If the question cannot be answered from the table, reply: “Cannot answer based on the data in the table.”
```

3.1.5 Description Field and Embedding

The embedding was generated using OpenAI’s text-embedding-3-small model. Each car entry was converted into a natural language description containing all available features (torque, emissions, displacement, etc.). If data was missing, that attribute was omitted. This naturally resulted in non-uniform embedding vectors — an important design factor discussed later.

The goal was not to force structure, but to emulate real-world data variability**.** Each row represents a single retrievable unit (“chunk”) and contains the maximum amount of information available for that vehicle.

3.1.6 Scoring Method

Because the goal was to measure factual correctness on expert-level questions (e.g., torque, displacement, CO_2 emissions), a binary scoring approach was chosen. Each answer was rated either as:

- 100 % correct — when the answer exactly or acceptably matched the dataset,
- 0 % incorrect — when the answer was inaccurate, estimated, hallucinated, or missing.

The decision for using only binary scoring also obtained these arguments:

- Only 15 questions were used, and all were factual,
- BLEU, ROUGE or embedding similarity scores were considered excessive,
- Mainly binary scoring reflects practical utility in an expert use case (e.g., “Is the answer right or not?”).

3.1.7 Summary

This evaluation setup offers a controlled, transparent, and domain-grounded way to test factual correctness in the automotive domain. It validates the central hypothesis that RAG pipelines, by incorporating real-world context, produce more accurate and informative answers than base LLMs alone**.**

3.2 Evaluation Results

This section presents the results of the evaluation based on the binary scoring method described earlier. Each of the 15 domain-specific technical questions was answered using three different system configurations:

- Basic Pipeline, without access to any retrieval mechanism,
- RAG Pipeline, with vector-based semantic retrieval of relevant vehicle descriptions,
- ChatGPT Assistant, which had direct access to the complete dataset in CSV format.

Each answer was compared against the exact ground truth derived from the dataset. The results are summarized in Table 3.1.

Table 1 Results of Each Evaluation (source: author)

Question ID	Ground Truth	Basic Correct	RAG Correct	ChatGPT Correct
Q1	1050 lb-ft	100 %	100 %	100 %
Q2	1036 lb-ft @ 4500 RPM	0 %	100 %	100 %
Q3	400 Nm	0 %	100 %	100 %
Q4	740 Nm	100 %	100 %	100 %
Q5	492 g/km	0 %	100 %	100 %
Q6	574 g/km	0 %	100 %	100 %
Q7	270 g/km	0 %	100 %	100 %
Q8	315 g/km	0 %	100 %	100 %
Q9	41 ft	0 %	100 %	100 %
Q10	41 ft	0 %	100 %	100 %
Q11	16	100 %	100 %	100 %
Q12	8	100 %	100 %	100 %
Q13	7994 cm^3	0 %	100 %	100 %
Q14	8128 cm^3	0 %	100 %	100 %
Q15	46.9 ft	0 %	100 %	100 %

Questions:

Table 2 Test Questions (source: author)

Question ID	Question
Q1	What is the torque of the FORD Super Duty Lariat Tremor with the 6.7L V8 (475 HP) engine from the year 2021?
Q2	What is the torque of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?
Q3	What is the electric motor torque of the BENTLEY Flying Spur with the 2.9L V6 (543 HP) hybrid system from the year 2021?
Q4	What is the electric motor torque of the 2023 Ferrari 296 GTS with the 3.0L V6 Turbo 8AT RWD (830 HP) engine from the year 2023?
Q5	What are the CO_2 emissions of the BMW 8 Series (E31) with the 840Ci 6MT (286 HP) engine from the year 1992?
Q6	What are the CO_2 emissions of the BUGATTI Veyron Grand Sport with the 8.0L W16 7AT (1001 HP) engine from the year 2010?
Q7	What is the combined CO_2 emission of the AUDI RS Q8 with the 4.0L TFSIV8 8AT AWD (600 HP) engine from the year 2022?
Q8	What is the combined CO_2 emission of the 2023 Ford Ranger Raptor with the 3.0L V6 EcoBoost 10AT AWD (292 HP) engine from the year 2023?
Q9	What is the turning circle (curb to curb) of the AUDI A8 with the 60 TFSI e V6 quattro 8AT AWD (449 HP) engine from the year 2017?
Q10	What is the turning circle (curb to curb) of the DODGE Durango with the 3.5L V6 8AT (293 HP) engine from the year 2020?
Q11	How many cylinders does the BUGATTI Veyron Grand Sport Vitesse with the 8.0 W16 7AT (1200 HP) engine from the year 2012 have?
Q12	How many cylinders does the 2014 Aston Martin Vantage N430 with the 4.7 V8 6AT (436 HP) engine from the year 2014 have?
Q13	What is the engine displacement of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?
Q14	What is the engine displacement of the CHEVROLET Suburban with the 8.1L V8 4AT RWD (325 HP) engine from the year 2000?
Q15	What is the turning circle of the FORD F-150 Super Cab with the 4.6L (248 HP) engine from the year 2009?

Summary of Results:

- Basic Pipeline: 5/15 correct (33 %)
- RAG Pipeline: 15/15 correct (100 %)
- ChatGPT Assistant: 15/15 correct (100 %)

A detailed breakdown of each question, including full model responses and retrieved contexts, is provided in Appendices F–H.

3.3 Discussion and Interpretation

3.3.1 RAG vs. Basic Pipeline

The results clearly demonstrate the superior factual accuracy of the RAG Pipeline compared to the basic language model setup. The basic pipeline, operating without access to external information, frequently produced hallucinated or approximate answers, particularly when faced with specific, technical queries such as torque values or CO_2 emissions.

Conversely, the RAG pipeline consistently returned accurate answers grounded in retrieved database entries. Its structured context allowed the model to:

- Pinpoint relevant technical data,
- Avoid hallucinating non-existent specs,
- Generate concise and correct answers for every query.

3.3.2 RAG vs. ChatGPT Assistant

Although ChatGPT with CSV access matched the RAG pipeline in correctness, several important distinctions must be highlighted:

- The assistant had access to the entire dataset in raw form, offering complete visibility across all cars and attributes.
- In practice, the assistant occasionally deviated from the specified constraints, leveraging prior knowledge or making implicit assumptions. This behavior, while helpful, introduces non-determinism and undermines strict data integrity.
- ChatGPT's reliance on internet-level knowledge and internal priors does not reflect a deployable, closed-source system, unlike the RAG pipeline, which operates purely on a defined, local dataset.

Therefore, while ChatGPT is a strong benchmark, the RAG pipeline represents a more realistic and controlled solution for production environments requiring reliability, reproducibility, and explainability.

3.4 Evaluation Summary

This chapter has rigorously evaluated the retrieval-augmented generation pipeline developed in this thesis. The experimental setup, question generation, scoring, and comparison were carefully designed to assess the system's factual accuracy in a domain-specific context.

Key findings include:

- RAG improved factual correctness from 33% to 100% compared to a naive LLM approach.
- RAG matched the performance of a specialized assistant with full dataset access, while being more controllable and domain-grounded.
- The approach eliminated hallucinations, addressed expert-level queries, and provided accurate, interpretable outputs.

Ultimately, the evaluation validates the central hypothesis: retrieval-augmented generation significantly enhances the capability of general-purpose language models in domain-specific applications, such as automotive technical support, and presents a viable path toward practical, trustworthy AI assistance systems.

Conclusions

This chapter summarizes all of what the design, the implementation, and the evaluation of a domain-specific Retrieval-Augmented Generation (RAG) system for automotive knowledge retrieval has gained, has contributed, and has resulted in. It reflects on the original objectives as well as assesses the results. It identifies limitations for development and outlines future directions.

Summary of Findings

This thesis investigated the integration of Retrieval-Augmented Generation (RAG) with a large language model (LLM) to enhance factual correctness in the context of expert-level, domain-specific question answering. The chosen domain was the automotive sector, which presents a high density of structured information (e.g., technical specifications such as torque, CO_2 emissions, engine displacement), but also demands precision in language understanding and numerical accuracy.

To address this challenge, a complete RAG pipeline was designed and implemented. The architecture leveraged OpenAI's GPT-4o-mini as the core generative model, used text-embedding-3-small for producing semantically meaningful vector representations, and employed a PostgreSQL database with pgvector to facilitate efficient similarity-based retrieval over a transformed dataset. The source dataset, originally structured as tabular data from Kaggle, was converted into natural language descriptions in order to make it suitable for semantic embedding and retrieval workflows.

Two parallel systems were developed for comparative evaluation: a Basic Pipeline, which relied solely on GPT-4o-mini without external context, and a RAG Pipeline, which dynamically retrieved top-k similar vehicle descriptions from the database and appended them to the input prompt as contextual grounding. Importantly, a third benchmark configuration was introduced — a ChatGPT Assistant with direct access to the full dataset in CSV form — serving as a reference point to measure how well the RAG system could perform relative to an idealized, fully informed assistant.

The evaluation, based on 15 factual and verifiable questions about vehicle specifications, demonstrated a clear performance differential: while the Basic Pipeline answered only 5 out of 15 questions correctly (33%), the RAG Pipeline achieved perfect accuracy (15/15 correct). Notably, the RAG system's performance also matched that of the ChatGPT Assistant, despite the latter having unrestricted access to the full structured dataset. This indicates that semantic retrieval of relevant descriptions is an effective and scalable strategy for domain grounding, even when compared to systems operating with full data visibility.

By integrating retrieval with generation, this thesis effectively answered the central research question — whether RAG can improve factual reliability in domain-specific language model

applications. The results confirm that it can, and that even a relatively lightweight implementation can lead to substantial improvements in precision, control, and answer consistency in technical domains.

Achievement of Objectives

The thesis objectives, as stated in the introduction, are reviewed and evaluated below:

Table 3 Objective Status (source: author)

Objective	Status	Comments
Develop a functional RAG system	Achieved	A two-pipeline system was successfully implemented using OpenAI tools and pgvector-based retrieval.
Transform structured data into retrievable semantic units	Achieved	Tabular vehicle metadata was converted into paragraph-level descriptions via a custom transformation script.
Improve factual accuracy and reduce hallucinations	Achieved	RAG pipeline answered 15/15 technical questions correctly; basic pipeline achieved only 5/15.
Enable comparative evaluation using controlled experiments	Achieved	An empirical evaluation was conducted using a consistent set of test cases and binary scoring.

The system demonstrated improved answer quality, reduced hallucinations, and provided a replicable method for grounding LLMs in structured technical knowledge.

Limitations

While the findings of this thesis affirm the value of Retrieval-Augmented Generation (RAG) in enhancing the factual reliability of language models, it is important to recognize the limitations inherent in the approach, implementation, and evaluation process. These limitations are discussed below to provide context for interpreting the results and to inform future extensions of this work.

One significant limitation stems from the domain specificity of the study. The system was applied exclusively to a structured dataset drawn from the automotive domain. Although RAG as a method is generalizable, the pipeline, query formats, and context structures were all optimized for vehicle specifications. As such, it remains unclear whether the same architecture would yield equivalent results in domains with less structured data or with higher semantic ambiguity (e.g., healthcare, law, or social science). The system's performance may thus not directly translate to these areas without reengineering both the data preprocessing and retrieval logic.

Closely related is the limitation of retrieval scope and sophistication. The system employed a standard top-k cosine similarity search over static dense embeddings, using the text-embedding-3-small model. This approach, while efficient, lacks contextual filtering, ranking refinement, or interpretability layers. More advanced retrieval pipelines often incorporate reranking mechanisms, multi-hop search, or hybrid retrieval combining dense and sparse representations. The absence of these components may result in the inclusion of suboptimal contexts, especially in queries involving ambiguous or multi-faceted vehicle features. A reranker could improve the system's ability to prioritize contextually richer or more targeted entries and filter out semantically adjacent but irrelevant passages.

Another limitation lies in the static nature of the knowledge base. Although RAG pipelines are often viewed as dynamic compared to traditional static LLMs, in this implementation, the system does not automatically handle updates to the underlying dataset. Any new entries would require reprocessing and re-embedding, which—while less expensive than retraining the entire model—is still a barrier to true real-time adaptability. This limits the system's application in domains with rapidly evolving data unless continuous indexing or streaming pipelines are introduced.

In addition, the evaluation relied on a relatively small set of 15 factual queries and used a binary scoring scheme to assess correctness. While this was appropriate for measuring direct factual alignment, it does not account for partial correctness, nuance, fluency, or user satisfaction. Furthermore, no human user study or qualitative rating by domain experts was conducted, which limits insight into perceived answer helpfulness or clarity. While the ChatGPT Assistant was included as a benchmark for comparison, its responses were not blindly rated nor tested for trust calibration. As a result, conclusions about user experience, reliability perception, and comparative satisfaction remain outside the scope of this work.

The work also inherently depends on the capabilities and constraints of GPT-4o-mini, which, although strong in general-purpose language understanding, may still underperform on highly specialized or out-of-distribution inputs. Moreover, the embedding model (text-embedding-3-small) and the retrieval method (pgvector with IVFFlat) impose additional performance ceilings, particularly in terms of vector representation granularity and recall.

Finally, this thesis deliberately focuses on a research prototype. No deployment-level optimization was conducted (e.g., caching, multi-user concurrency, latency handling), and the system was not tested under operational loads. This decision reflects the thesis's academic scope, but it means that considerations such as API security, response time under load, or production reliability remain unexplored.

In summary, while the proposed RAG system achieved its objective of improving factual accuracy on a static domain-specific dataset, several methodological, architectural, and operational limitations must be acknowledged. These constraints inform the boundary conditions of the findings and serve as a foundation for future enhancements.

Future work and direction

While the current implementation of the Retrieval-Augmented Generation (RAG) system has effectively improved factual correctness within the automotive domain, researchers still can develop the system along several promising avenues. These directions build directly from the limitations identified in this thesis. They also suggest ways of improving the architecture along with its practical deployment potential.

A key area for future enhancement lies in the refinement of the retrieval component. The current system uses a simple top-k cosine similarity approach for retrieving semantically similar descriptions from the vector database. While effective to a certain extent, this method may include semantically close but contextually irrelevant entries. Therefore, a logical next step would be to implement a more sophisticated retrieval strategy involving re-ranking mechanisms. For instance, re-ranking retrieved passages using a cross-encoder architecture or attention-based scoring could improve precision by evaluating contextual alignment more deeply. Additionally, hybrid retrieval methods that combine dense vector similarity with traditional keyword filtering may provide better coverage, especially for compound or ambiguous queries.

Another compelling extension is the incorporation of multimodal data. The automotive domain naturally includes rich visual content—images of car models, engine layouts, interior designs, and infographics. Integrating image data into the retrieval process, alongside textual descriptions, could open up new types of queries, such as visual comparisons between models or identification tasks (e.g., “Which model looks like this photo?”). This would require adapting the system for multimodal embeddings and aligning visual and textual modalities during context construction.

Furthermore, dynamic knowledge integration represents an important direction, especially for real-time applications. The current system requires re-embedding the entire dataset if any new data is introduced. A more advanced setup could enable continual learning or streaming updates to the vector index, allowing the system to evolve incrementally as new car models or specifications become available. This could be further enhanced by feedback mechanisms, where user corrections or confirmations are logged and used to refine future responses, creating a self-improving loop over time.

From a systems perspective, scaling the implementation to larger datasets or multilingual domains would also be highly valuable. While the current work focused on a representative subset of 10,000 entries in English, future deployments may involve significantly larger corpora spanning multiple markets and languages. Addressing this would involve optimizing embedding strategies, database performance, and possibly incorporating language detection and translation layers within the retrieval process.

Finally, further experimentation with different language models could be conducted. As open-source and commercial LLMs continue to evolve, it remains an open question whether newer, larger models would continue to benefit from retrieval augmentation, or whether their expanded knowledge capacity may render simple fact-based RAG pipelines redundant in certain cases. Comparative evaluations between RAG-augmented smaller models and

standalone large-scale models could provide valuable insights into the trade-offs between memory, accuracy, and computational cost.

In summary, the work presented in this thesis offers a solid foundation upon which more advanced and production-ready systems can be constructed. Each of the proposed directions represents an opportunity to increase the system's robustness, generalizability, and overall utility, thereby extending the impact of RAG systems in real-world information access scenarios.

Concluding remarks

This thesis has contributed a concrete example of how Retrieval-Augmented Generation can be applied to enhance factual reliability in a structured, expert-driven domain. By bridging neural language modeling with structured database retrieval, the work addresses a fundamental challenge in the field: how to keep language models grounded in verifiable knowledge.

As LLMs become increasingly integrated into everyday systems, ensuring the accuracy, trustworthiness, and interpretability of their outputs becomes paramount. This thesis demonstrates that even a relatively simple RAG pipeline—when paired with a well-structured dataset and clear evaluation—can lead to substantial gains in factual performance.

In conclusion, while the presented system is not yet production-grade, it establishes a strong baseline and a framework upon which more advanced, scalable, and robust retrieval-augmented AI assistants can be built. The insights gained here can support further research into hybrid AI architectures that are not only intelligent, but also reliable.

List of references

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., Chu, E., Clark, J. H., Shafey, L. E., Huang, Y., Meier-Hellstern, K., Mishra, G., Moreira, E., Omernick, M., Robinson, K. et al. (2023). PaLM 2 Technical Report (Version 3). *arXiv*. https://doi.org/10.48550/ARXIV.2305.10403

Anthropic. (2023). *Let Claude think (chain-of-thought prompting) to increase performance*. Anthropic Documentation. Retrieved May 11, 2025, from https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/chain-of-thought

Ashrafi, Y. (2023). *Car Specification Dataset*. Kaggle. Retrieved May 11, 2025, from https://www.kaggle.com/datasets/usefashrfi/car-specification-dataset

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *arXiv*. https://arxiv.org/abs/1409.0473

Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., DasSarma, N., Drain, D., Fort, S., Ganguli, D., Henighan, T., Joseph, N., Kadavath, S., Kernion, J., Conerly, T., El-Showk, S., Elhage, N., Hatfield-Dodds, Z., Hernandez, D., Hume, T. et al. (2022). Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv*. https://arxiv.org/abs/2204.05862

Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., Chen, C., Olsson, C., Olah, C., Hernandez, D., Drain, D., Ganguli, D., Li, D., Tran-Johnson, E., Perez, E., Kerr, J. et al. (2022). Constitutional AI: Harmlessness from AI Feedback. *arXiv*. https://arxiv.org/abs/2212.08073

Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer: The long-document transformer. *arXiv*. https://arxiv.org/abs/2004.05150

Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* (pp. 610–623). ACM. https://doi.org/10.1145/3442188.3445922

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research,* 3, 1137–1155. https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf

Bergmann, D. (2024a, March 15). *What is fine-tuning?* IBM. Retrieved May 11, 2025, from https://www.ibm.com/think/topics/fine-tuning

Bergmann, D. (2024b, November 10). *What is reinforcement learning from human feedback (RLHF)?* IBM. Retrieved May 11, 2025, from https://www.ibm.com/think/topics/rlhf

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S. et al. (2021). On the opportunities and risks of foundation models. *arXiv*. https://arxiv.org/abs/2108.07258

Borgeaud, S., Mensch, A., Hoffmann, J., Cai, T., Rutherford, E., Millican, K. et al. (2022). Improving language models by retrieving from trillions of tokens. *arXiv*. https://arxiv.org/abs/2112.04426

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P. et al. (2020). Language models are few-shot learners. *arXiv*. https://doi.org/10.48550/arXiv.2005.14165

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. *arXiv*. https://arxiv.org/abs/1406.1078

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T. et al. (2021). Rethinking attention with performers. *arXiv*. https://arxiv.org/abs/2009.14794

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A. et al. (2022). *PaLM*: Scaling language modeling with Pathways. *arXiv*. https://arxiv.org/abs/2204.02311

Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences (Version 4). *arXiv*. https://doi.org/10.48550/ARXIV.1706.03741

Courant, R., Edberg, M., Dufour, N., & Kalogeiton, V. (2023). Machine Learning for Brain Disorders: Transformers and Visual Transformers (Version 1). *ArXiv*. https://doi.org/10.48550/arXiv.2303.12068

Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). LLM.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv*. https://arxiv.org/abs/2208.07339

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional Transformers for language understanding. In *Proceedings of NAACL-HLT 2019* (pp. 4171–4186). Association for Computational Linguistics. https://doi.org/10.48550/arXiv.1810.04805

Ding, N., Qin, Y., Yang, G., Wei, F., Yang, Z., Su, Y., Hu, S., Chen, Y., Chan, C.-M., Chen, W., Yi, J., Zhao, W., Wang, X., Liu, Z., Zheng, H.-T., Chen, J., Liu, Y., Tang, J., Li, J., & Sun, M. (2023). Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence, 5*(3), 220–235. https://doi.org/10.1038/s42256-023-00626-4

Emenike, L. (2025). *A straightforward explanation of parametric vs. non-parametric memory in LLMs*. Medium. Retrieved May 11, 2025, from https://lawrence-emenike.medium.com/

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2023). Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv*. https://arxiv.org/abs/2312.10997

Gehman, S., Gururangan, S., Sap, M., Choi, Y., & Smith, N. A. (2020). RealToxicityPrompts: Evaluating neural toxic degeneration in language models. *arXiv*. https://arxiv.org/abs/2009.11462

Gemini Team. (2023). Gemini: A family of highly capable multimodal models. *arXiv*. https://arxiv.org/abs/2312.11805

Google DeepMind. (2025, March 25). *Gemini 2.5: Our most intelligent AI model*. Google Blog. Retrieved May 11, 2025, from https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/

Guu, K., Lee, K., Tung, Z., Pasupat, P., & Chang, M.-W. (2020). REALM: Retrieval-augmented language model pre-training. *arXiv*. https://arxiv.org/abs/2002.08909

Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both Weights and Connections for Efficient Neural Networks. In *Advances in Neural Information Processing Systems*, 28. https://arxiv.org/abs/1506.02626

Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., & Steinhardt, J. (2020). Measuring massive multitask language understanding. *arXiv*. https://arxiv.org/abs/2009.03300

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., & Gelly, S. (2019). Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning (ICML)* (pp. 2790–2799). PMLR. https://arxiv.org/abs/1902.00751

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S. et al. (2022). LoRA: Low-rank adaptation of large language models. In *Proceedings of the 10th International Conference on Learning Representations (ICLR 2022)*. https://openreview.net/forum?id=nZeVKeeFYf9

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv preprint arXiv:2106.09685. https://arxiv.org/abs/2106.09685

Hu, Z., Wang, L., Lan, Y., Xu, W., Lim, E.-P., Bing, L., Xu, X., Poria, S., & Lee, R. K.-W. (2023). LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models (Version 3). *arXiv*. https://doi.org/10.48550/ARXIV.2304.01933

Iqbal, F., Batool, R., Fung, B. C. M., Aleem, S., Abbasi, A., & Javed, A. R. (2021). Toward Tweet-Mining Framework for Extracting Terrorist Attack-Related Information and Reporting. *IEEE Access*, *9*, 115535-115547. https://doi.org/10.1109/ACCESS.2021.3102040

Izacard, G., & Grave, E. (2020). Leveraging passage retrieval with generative models for open-domain question answering. *arXiv*. https://arxiv.org/abs/2007.01282

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., & Fung, P. (2022). Survey of hallucination in natural language generation. *arXiv*. https://arxiv.org/abs/2202.03629

Ji, Z., Yu, T., Xu, Y., Lee, N., Ishii, E., & Fung, P. (2023). Towards mitigating hallucination in large language models via self-reflection. *arXiv*. https://arxiv.org/abs/2310.06271

Johnson, J., Douze, M., & Jégou, H. (2017). Billion-scale similarity search with GPUs. *arXiv*. https://arxiv.org/abs/1702.08734

Jurafsky, D., & Martin, J. H. (2020). *Speech and language processing* (3rd ed., draft). Stanford University. Retrieved May 11, 2025, from https://web.stanford.edu/~jurafsky/slp3/

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R. et al. (2020). Scaling laws for neural language models. *arXiv*. https://arxiv.org/abs/2001.08361

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., & Yih, W.-T. (2020). Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 6769–6781). Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.emnlp-main.550

Kavukcuoglu, K. (2025, March 25). *Gemini 2.5: Our most intelligent AI model*. Google DeepMind Blog. Retrieved May 11, 2025, from https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/

Kitaev, N., Kaiser, Ł., & Levskaya, A. (2020). Reformer: The efficient transformer. *arXiv*. https://arxiv.org/abs/2001.04451

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, 35, 22199–22213. https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf

Kumar, B., Amar, J., Yang, E., Li, N., & Jia, Y. (2024). Selective fine-tuning on LLM-labeled data may reduce reliance on human annotation: A case study using schedule-of-event table detection. *arXiv*. https://arxiv.org/abs/2405.06093

Lee, J., Yoon, W., Kim, S., Kim, D., Kim, S., So, C. H., Kang, J., & Wren, J. (2020). BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, *36*(4), 1234-1240. https://doi.org/10.1093/bioinformatics/btz682

Lee, K., Chang, M.-W., & Toutanova, K. (2019). Latent retrieval for weakly supervised open-domain question answering. *arXiv*. https://arxiv.org/abs/1906.00300

Lester, B., Al-Rfou, R., & Constant, N. (2021). The power of scale for parameter-efficient prompt tuning. *arXiv*. https://arxiv.org/abs/2104.08691

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-T., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *arXiv*. https://doi.org/10.48550/arXiv.2005.11401

Li, J., Xu, J., Huang, S., Chen, Y., Li, W., Liu, J., Lian, Y., Pan, J., Ding, L., Zhou, H., Wang, Y., & Dai, G. (2024). Large Language Model Inference Acceleration: A Comprehensive Hardware Perspective (Version 3). *arXiv*. https://doi.org/10.48550/ARXIV.2410.04466

Li, X. L., & Liang, P. (2021). Prefix-Tuning: Optimizing Continuous Prompts for Generation (Version 1). *arXiv*. https://doi.org/10.48550/ARXIV.2101.00190

Lin, Z., Guan, S., Zhang, W., Zhang, H., Li, Y., & Zhang, H. (2024). Towards trustworthy LLMs: a review on debiasing and dehallucinating in large language models. *Artificial Intelligence Review*, *57*(9). https://doi.org/10.1007/s10462-024-10896-y

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., & Neubig, G. (2021). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv*. https://arxiv.org/abs/2107.13586

Luo, Y., Yang, Z., Meng, F., Li, Y., Zhou, J., & Zhang, Y. (2023). An empirical study of catastrophic forgetting in large language models during continual fine-tuning. *arXiv*. https://arxiv.org/abs/2308.08747

Ma, C., Jiang, J., Li, H., Mei, X., & Bai, C. (2022). Hyperspectral Image Classification via Spectral Pooling and Hybrid Transformer. *Remote Sensing*, *14*(19). https://doi.org/10.3390/rs14194732

Malkov, Y. A., & Yashunin, D. A. (2016). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv*. https://arxiv.org/abs/1603.09320

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient estimation of word representations in vector space*. *arXiv*. https://arxiv.org/abs/1301.3781

Minaee, S., Mikolov, T., Nikzad, N., Chenaghlu, M., Socher, R., Amatriain, X., & Gao, J. (2024). Large language models: A survey. *arXiv* preprint arXiv:2402.06196. https://arxiv.org/abs/2402.06196

OpenAI, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J. et al. (2023). GPT-4 Technical Report (Version 6). *arXiv*. https://doi.org/10.48550/ARXIV.2303.08774

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P. et al. (2022). Training language models to follow instructions with human feedback. *arXiv*. https://arxiv.org/abs/2203.02155

Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L. M., Rothchild, D. et al. (2022). Carbon emissions and large neural network training. *arXiv*. https://arxiv.org/abs/2104.10350

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. *arXiv*. https://arxiv.org/abs/1802.05365

Petroni, F., Rocktäschel, T., Lewis, P., Bakhtin, A., Wu, Y., Miller, A., & Riedel, S. (2019). Language Models as Knowledge Bases? In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)* (pp. 2463–2473). Association for Computational Linguistics. https://doi.org/10.48550/arXiv.1909.01066

Pfeiffer, J., Rücklé, A., Poth, C., Kamath, A., Vulić, I., Ruder, S., Cho, K., & Gurevych, I. (2020). AdapterHub: A Framework for Adapting Transformers. *arXiv*. https://arxiv.org/abs/2007.07779

Qiu, X., Sun, T., Xu, Y., Shao, Y., Dai, N., & Huang, X. (2020). Pre-trained models for natural language processing: A survey. *arXiv*. https://arxiv.org/abs/2003.08271

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). *Improving language understanding by generative pre-training* [Technical report]. OpenAI. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language models are unsupervised multitask learners*. OpenAI. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M. et al. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research, 21*(140), 1–67. http://jmlr.org/papers/v21/20-074.html

RAPIDS. (2025). *IVF-Flat*. In *cuVS: Vector Search and Clustering on the GPU*. NVIDIA. Retrieved May 11, 2025, from https://docs.rapids.ai/api/cuvs/stable/indexes/ivfflat/

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using Siamese BERT-networks. *arXiv*. https://arxiv.org/abs/1908.10084

Richardson, G. (2023, February 6). *Storing OpenAI embeddings in Postgres with pgvector*. Supabase. Retrieved May 11, 2025, from https://supabase.com/blog/openai-embeddings-postgres-vector

Sanh, V., Debut, L., Chaumond, J., & Wolf, T. (2019). *DistilBERT, a distilled version of BERT:* Smaller, faster, cheaper and lighter. *arXiv*. https://arxiv.org/abs/1910.01108

Sanh, V., Webson, A., Raffel, C., Bach, S. H., Sutawika, L., Alyafeai, Z. et al. (2021). Multitask prompted training enables zero-shot task generalization. *arXiv*. https://arxiv.org/abs/2110.08207

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2021). RoFormer: Enhanced transformer with rotary position embedding. *arXiv*. https://arxiv.org/abs/2104.09864

Su, W., Tang, Y., Ai, Q., Yan, J., Wang, C., Wang, H., Ye, Z., Zhou, Y., & Liu, Y. (2025). Parametric Retrieval-Augmented Generation. *arXiv*. https://arxiv.org/abs/2501.15915

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *arXiv*. https://arxiv.org/abs/1409.3215

Timescale. (2024, November 22). *PostgreSQL Hybrid Search Using pgvector and Cohere*. Timescale. Retrieved May 11, 2025, from https://www.timescale.com/blog/postgresql-hybrid-search-using-pgvector-and-cohere

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023a). LLaMA: Open and efficient foundation language models. *arXiv*. https://arxiv.org/abs/2302.13971

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W. et al. (2023b). LLaMA 2: Open foundation and fine-tuned chat models. *arXiv*. https://arxiv.org/abs/2307.09288

Uszkoreit, J. (2017, August 31). *Transformer: A novel neural network architecture for language understanding* [Blog post]. Google AI Blog. Retrieved May 11, 2025, from https://research.googleblog.com/2017/08/transformer-novel-neural-network.html

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *arXiv*. https://arxiv.org/abs/1706.03762

Voita, L. (n.d.). *Convolutional Neural Networks for Text*. Lena Voita. Retrieved May 11, 2025, from https://lena-voita.github.io/nlp_course/models/convolutional.html

Wang, T., Roberts, A., Hesslow, D., Le Scao, T., Chung, H. W., Beltagy, I., Launay, J., & Raffel, C. (2022a). What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization? *arXiv*. https://arxiv.org/abs/2204.05832

Wang, X., Wei, J., Schuurmans, D., Bosma, M., Chi, E. H., Le, Q., & Zhou, D. (2022b). Self-consistency improves chain of thought reasoning in language models. *arXiv*. https://arxiv.org/abs/2203.11171

Wei, J., Bosma, M., Zhao, V., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2021). Finetuned language models are zero-shot learners. *arXiv*. https://arxiv.org/abs/2109.01652

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., & Fedus, W. (2022a). Emergent abilities of large language models. *arXiv*. https://arxiv.org/abs/2206.07682

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022b). Chain-of-thought prompting elicits reasoning in large language models. *arXiv*. https://arxiv.org/abs/2201.11903

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., & Rush, A. M. (2019). Transformers: State-of-the-art natural language processing. *arXiv*. https://arxiv.org/abs/1910.03771

Wu, S., Xiong, Y., Cui, Y., Wu, H., Chen, C., Yuan, Y., Huang, L., Liu, X., Kuo, T.-W., Guan, N., & Xue, C. (2024). Retrieval-Augmented Generation for Natural Language Processing: A Survey. *arXiv*. https://arxiv.org/abs/2407.13193

Yeung, T. (2024, April 5). *Explainer: What Is Retrieval-Augmented Generation?* NVIDIA Developer Blog. Retrieved May 11, 2025, from https://developer.nvidia.com/blog/explainer-what-is-retrieval-augmented-generation

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., & Choi, Y. (2019). HellaSwag: Can a machine really finish your sentence? *arXiv*. https://arxiv.org/abs/1905.078Zhao, T. Z., Wallace, E., Feng, S., Klein, D., & Singh, S. (2021). Calibrate before use: Improving few-shot performance of language models. *arXiv*. https://arxiv.org/abs/2102.09690

Appendices

Appendix A: Dataset setup code

```
import pandas as pd

# Read the CSV file containing car data
# Using low_memory=False to handle mixed data types efficiently
df = pd.read_csv('raw/cars-dataset.csv', low_memory=False)

def natural_language_description(row):
    """
    Transforms structured car data into natural language descriptions.
    Includes all available features from the dataset.

    Args:
        row: A pandas Series containing car attributes

    Returns:
        str: A comprehensive natural language description of the car
    """
    parts = []

    # Basic Information
    if pd.notnull(row['Model']):
        parts.append(f"The {row['Model']}")
    if pd.notnull(row['Serie']):
        parts[-1] += f" {row['Serie']}"
    if pd.notnull(row['Company']):
        parts.append(f"was manufactured by {row['Company']}")
    if pd.notnull(row['Production years']):
        parts.append(f"produced in {row['Production years']}")
    if pd.notnull(row['Body style']):
        parts.append(f"as a {row['Body style']}")
    if pd.notnull(row['Segment']):
        parts.append(f"in the {row['Segment']} segment")

    # Engine and Power Specifications
    engine_specs = []
    if pd.notnull(row['Cylinders']):
        engine_specs.append(f"{row['Cylinders']} cylinders")
```

```
    if pd.notnull(row['Displacement']):
        engine_specs.append(f"{row['Displacement']} displacement")
    if engine_specs:
        parts.append("Engine features: " + ", ".join(engine_specs))

    # Power Outputs
    power_specs = []
    if pd.notnull(row['Power(HP)']):
        power_specs.append(f"{row['Power(HP)']} HP")
    if pd.notnull(row['Power(BHP)']):
        power_specs.append(f"{row['Power(BHP)']} BHP")
    if pd.notnull(row['Power(KW)']):
        power_specs.append(f"{row['Power(KW)']} KW")
    if pd.notnull(row['Total maximum power']):
        power_specs.append(f"{row['Total maximum power']} total maximum power")
    if power_specs:
        parts.append("Power outputs: " + ", ".join(power_specs))

    # Torque Specifications
    torque_specs = []
    if pd.notnull(row['Torque(lb-ft)']):
        torque_specs.append(f"{row['Torque(lb-ft)']} lb-ft")
    if pd.notnull(row['Torque(Nm)']):
        torque_specs.append(f"{row['Torque(Nm)']} Nm")
    if pd.notnull(row['Total maximum torque']):
        torque_specs.append(f"{row['Total maximum torque']} total maximum torque")
    if torque_specs:
        parts.append("Torque specifications: " + ", ".join(torque_specs))

    # Electrical Specifications
    elec_specs = []
    if pd.notnull(row['Electrical motor power']):
        elec_specs.append(f"{row['Electrical motor power']} electrical motor power")
    if pd.notnull(row['Electrical motor torque']):
        elec_specs.append(f"{row['Electrical motor torque']} electrical motor torque")
    if pd.notnull(row['Power pack']):
        elec_specs.append(f"{row['Power pack']} power pack")
    if pd.notnull(row['Nominal Capacity']):
        elec_specs.append(f"{row['Nominal Capacity']} nominal capacity")
    if pd.notnull(row['Maximum Capacity']):
        elec_specs.append(f"{row['Maximum Capacity']} maximum capacity")
    if elec_specs:
        parts.append("Electrical specifications: " + ", ".join(elec_specs))

    # Fuel and Emissions
```

```
    fuel_specs = []
    if pd.notnull(row['Fuel System']):
        fuel_specs.append(f"{row['Fuel System']} fuel system")
    if pd.notnull(row['Fuel']):
        fuel_specs.append(f"{row['Fuel']} fuel type")
    if pd.notnull(row['Fuel capacity']):
        fuel_specs.append(f"{row['Fuel capacity']} fuel capacity")
    if fuel_specs:
        parts.append("Fuel specifications: " + ", ".join(fuel_specs))

    # Performance Metrics
    perf_specs = []
    if pd.notnull(row['Top Speed']):
        perf_specs.append(f"{row['Top Speed']} top speed")
    if pd.notnull(row['Top speed (electrical)']):
        perf_specs.append(f"{row['Top speed (electrical)']} electrical top speed")
    if pd.notnull(row['Acceleration 0-62 Mph (0-100 kph)']):
        perf_specs.append(f"{row['Acceleration  0-62  Mph  (0-100  kph)']}  0-100  km/h
acceleration")
    if perf_specs:
        parts.append("Performance metrics: " + ", ".join(perf_specs))

    # Drivetrain
    drive_specs = []
    if pd.notnull(row['Drive Type']):
        drive_specs.append(f"{row['Drive Type']} drive type")
    if pd.notnull(row['Gearbox']):
        drive_specs.append(f"{row['Gearbox']} gearbox")
    if drive_specs:
        parts.append("Drivetrain: " + ", ".join(drive_specs))

    # Brakes and Tires
    brake_specs = []
    if pd.notnull(row['Front brake']):
        brake_specs.append(f"{row['Front brake']} front brakes")
    if pd.notnull(row['Rear brake']):
        brake_specs.append(f"{row['Rear brake']} rear brakes")
    if pd.notnull(row['Tire Size']):
        brake_specs.append(f"{row['Tire Size']} tire size")
    if brake_specs:
        parts.append("Brakes and tires: " + ", ".join(brake_specs))

    # Dimensions
    dim_specs = []
    if pd.notnull(row['Length']):
```

```
        dim_specs.append(f"{row['Length']} length")
    if pd.notnull(row['Width']):
        dim_specs.append(f"{row['Width']} width")
    if pd.notnull(row['Height']):
        dim_specs.append(f"{row['Height']} height")
    if pd.notnull(row['Front/rear Track']):
        dim_specs.append(f"{row['Front/rear Track']} front/rear track")
    if pd.notnull(row['Wheelbase']):
        dim_specs.append(f"{row['Wheelbase']} wheelbase")
    if pd.notnull(row['Ground Clearance']):
        dim_specs.append(f"{row['Ground Clearance']} ground clearance")
    if dim_specs:
        parts.append("Dimensions: " + ", ".join(dim_specs))

    # Aerodynamics
    aero_specs = []
    if pd.notnull(row['Aerodynamics (Cd)']):
        aero_specs.append(f"{row['Aerodynamics (Cd)']} drag coefficient")
    if pd.notnull(row['Aerodynamics (frontal area)']):
        aero_specs.append(f"{row['Aerodynamics (frontal area)']} frontal area")
    if aero_specs:
        parts.append("Aerodynamics: " + ", ".join(aero_specs))

    # Turning and Cargo
    turn_specs = []
    if pd.notnull(row['Turning circle']):
        turn_specs.append(f"{row['Turning circle']} turning circle")
    if pd.notnull(row['Turning circle (curb to curb)']):
        turn_specs.append(f"{row['Turning circle (curb to curb)']} curb-to-curb turning
circle")
    if pd.notnull(row['Cargo Volume']):
        turn_specs.append(f"{row['Cargo Volume']} cargo volume")
    if turn_specs:
        parts.append("Turning and cargo: " + ", ".join(turn_specs))

    # Weight Specifications
    weight_specs = []
    if pd.notnull(row['Unladen Weight']):
        weight_specs.append(f"{row['Unladen Weight']} unladen weight")
    if pd.notnull(row['Gross Weight Limit']):
        weight_specs.append(f"{row['Gross Weight Limit']} gross weight limit")
    if weight_specs:
        parts.append("Weight specifications: " + ", ".join(weight_specs))

    # Fuel Economy
```

```
    economy_specs = []
    if pd.notnull(row['Combined mpg']):
        economy_specs.append(f"{row['Combined mpg']} combined mpg")
    if pd.notnull(row['City mpg']):
        economy_specs.append(f"{row['City mpg']} city mpg")
    if pd.notnull(row['Highway mpg']):
        economy_specs.append(f"{row['Highway mpg']} highway mpg")
    if pd.notnull(row['High mpg']):
        economy_specs.append(f"{row['High mpg']} high mpg")
    if pd.notnull(row['Extra high mpg']):
        economy_specs.append(f"{row['Extra high mpg']} extra high mpg")
    if pd.notnull(row['Medium mpg']):
        economy_specs.append(f"{row['Medium mpg']} medium mpg")
    if pd.notnull(row['Low mpg']):
        economy_specs.append(f"{row['Low mpg']} low mpg")
    if economy_specs:
        parts.append("Fuel economy: " + ", ".join(economy_specs))

    # Emissions
    emission_specs = []
    if pd.notnull(row['CO2 Emissions']):
        emission_specs.append(f"{row['CO2 Emissions']} CO2 emissions")
    if pd.notnull(row['CO2 Emissions (Combined)']):
        emission_specs.append(f"{row['CO2 Emissions (Combined)']} combined CO2 emissions")
    if emission_specs:
        parts.append("Emissions: " + ", ".join(emission_specs))

    # EV Specifications
    ev_specs = []
    if pd.notnull(row['EV Range']):
        ev_specs.append(f"{row['EV Range']} EV range")
    if ev_specs:
        parts.append("Electric vehicle specifications: " + ", ".join(ev_specs))

    # Additional Specifications
    if pd.notnull(row['Specification summary']):
        parts.append(f"Additional specifications: {row['Specification summary']}")

    # Combine all parts into a single, coherent description
    return " ".join(parts)

# Apply the transformation to create descriptions for all cars
df['description'] = df.apply(natural_language_description, axis=1)

# Save the transformed data to a new CSV file
```

```
# This file will be used for generating embeddings and RAG pipeline
df.to_csv('processed/cars-for-rag-natural.csv', index=False)
```

Appendix B: Embedding Generation Implementation

```
import openai
import psycopg2
import os
from dotenv import load_dotenv
import time

# Configuration for batch processing
# Larger batch size = more efficient but higher memory usage
BATCH_SIZE = 100

# Load environment variables and set up OpenAI API key
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

# Establish database connection
# Using psycopg2 for PostgreSQL connection
conn = psycopg2.connect("dbname=carsdbtest user=simonpivoda")
cur = conn.cursor()

# Set up error logging file
# 'a' mode appends to existing file, preserving previous logs
error_log = open("embedding_errors.log", "a", encoding="utf-8")

while True:
    # Query database for unprocessed records
    # Filters out null and empty descriptions
    # Orders by ID for consistent processing
    cur.execute('SELECT "ID", "Description" FROM cars WHERE embedding IS NULL AND
"Description" IS NOT NULL AND "Description" <> \'\' ORDER BY "ID" LIMIT %s;', (BATCH_SIZE,))
    rows = cur.fetchall()
    if not rows:
        break

    # Extract data from query results
    # ids: list of car IDs
    # descriptions: list of car descriptions to be embedded
    ids = [row[0] for row in rows]
    descriptions = [row[1] for row in rows]

    try:
        # Call OpenAI API to generate embeddings
        # text-embedding-3-small is the latest efficient model
```

```
        response = openai.embeddings.create(
            input=descriptions,
            model="text-embedding-3-small"
        )
        # Extract embeddings from API response
        embeddings = [item.embedding for item in response.data]

        # Update database with generated embeddings
        # Each car record gets its corresponding embedding
        for car_id, embedding in zip(ids, embeddings):
            try:
                cur.execute('UPDATE cars SET embedding = %s WHERE "ID" = %s;', (embedding,
car_id))
            except Exception as e:
                # Log database update errors
                msg = f"DB update error for ID {car_id}: {e}\n"
                print(msg)
                error_log.write(msg)
        # Commit changes to database
        conn.commit()
        print(f"Processed: {len(ids)} records in this batch.")
    except Exception as e:
        # Handle OpenAI API errors
        # Log errors for each affected record
        for car_id, desc in zip(ids, descriptions):
            msg = f"OpenAI error for ID {car_id}: {e}\n"
            print(msg)
            error_log.write(msg)
        # Wait before retrying to avoid rate limits
        time.sleep(10)  # wait before next batch

# Clean up resources
# Close all open connections and files
error_log.close()
cur.close()
conn.close()
print("Done!")
```

Appendix C: RAG pipeline

```
import os
from typing import Dict, Any, List
from openai import OpenAI
import psycopg2
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

class RAGPipeline:
    """
    A Retrieval-Augmented Generation (RAG) pipeline that combines vector search with LLM responses.
    This class handles the entire RAG process from embedding generation to final response generation.
    """
    def __init__(self):
        """
        Initialize the RAG pipeline with OpenAI client, model selection, and database connection.
        """
        # Initialize OpenAI client with API key from environment variables
        self.client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
        # Set the LLM model to use for response generation
        self.model = "gpt-4o-mini"
        # Establish connection to PostgreSQL database
        self.conn = psycopg2.connect("dbname=carsdbtest user=simonpivoda")

    def get_embedding(self, text: str) -> List[float]:
        """
        Generate an embedding vector for the input text using OpenAI's embedding model.

        Args:
            text (str): The input text to generate embedding for

        Returns:
            List[float]: A list of floating-point numbers representing the text embedding
        """
        response = self.client.embeddings.create(
            model="text-embedding-3-small",
            input=text
        )
```

```
        return response.data[0].embedding

    def retrieve_context(self, query: str, top_k: int = 5) -> List[Dict[str, Any]]:
        """
        Retrieve relevant context from the vector database using similarity search.

        Args:
            query (str): The search query to find relevant context
            top_k (int, optional): Number of most similar results to return. Defaults to 25.

        Returns:
            List[Dict[str, Any]]: List of dictionaries containing relevant context
information
        """
        # Generate embedding for the query
        query_embedding = self.get_embedding(query)
        # Convert embedding list to string format for PostgreSQL
        embedding_str = "[" + ",".join(str(x) for x in query_embedding) + "]"

        # Perform vector similarity search in the database
        with self.conn.cursor() as cur:
            cur.execute("""
                SELECT "ID", "Model", "Description"
                FROM cars
                WHERE embedding IS NOT NULL
                ORDER BY embedding <=> %s
                LIMIT %s;
            """, (embedding_str, top_k))
            results = cur.fetchall()

        # Format results into a list of dictionaries
        return [{"id": row[0], "model": row[1], "description": row[2]} for row in results]

    def get_response(self, query: str) -> Dict[str, Any]:
        """
        Generate a response using the RAG approach by combining retrieved context with LLM.

        Args:
            query (str): The user's question or query

        Returns:
            Dict[str, Any]: A dictionary containing:
                - answer: The generated response
                - model: The model used
                - tokens_used: Number of tokens consumed
```

```
                - context_used: The context used for generation
                - error: Error message if something went wrong
                - status: Status of the operation
        """
        try:
            # Retrieve relevant context from the database
            context_results = self.retrieve_context(query)
            # Format context into a readable string
            context       =       "\n".join([f"Model:       {result['model']}\nDescription:
{result['description']}"
                                for result in context_results])

            # Construct the prompt with context and query
            messages = [
                {"role": "system", "content": "You are a helpful assistant. Use only the
provided context to answer the question. Be as precise and concise as possible, directly
addressing the question."},
                {"role": "user", "content": f"Context: {context}\n\nQuestion: {query}"}
            ]

            # Generate response using the LLM
            response = self.client.chat.completions.create(
                model=self.model,
                messages=messages
            )

            # Return the complete response information
            return {
                "answer": response.choices[0].message.content,
                "model": self.model,
                "tokens_used": response.usage.total_tokens,
                "context_used": context_results
            }

        except Exception as e:
            # Handle any errors that occur during the process
            return {
                "error": str(e),
                "status": "failed"
            }
```

Appendix D: Basic pipeline

```
import os
from typing import Dict, Any
from openai import OpenAI
from dotenv import load_dotenv

# Load environment variables from .env file
# This is necessary to access the OpenAI API key securely
load_dotenv()

class BasicPipeline:
    """
    A basic pipeline for interacting with OpenAI's GPT models.
    This class provides a simple interface for sending queries to the model
    and receiving responses without any additional context or retrieval.
    """
    def __init__(self):
        """
        Initialize the basic pipeline with OpenAI client and model configuration.
        Sets up the connection to OpenAI's API using the API key from environment variables.
        """
        # Initialize OpenAI client with API key from environment variables
        self.client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
        # Set the default model to use for completions
        self.model = "gpt-4o-mini"  # or any other model you prefer

    def get_response(self, query: str) -> Dict[str, Any]:
        """
        Get response from OpenAI API for a given query.

        Args:
            query (str): The input query/question to send to the model

        Returns:
            Dict[str, Any]: A dictionary containing:
                - answer: The model's response text
                - model: The model used for generation
                - tokens_used: Number of tokens consumed in the interaction
                - error: Error message if something went wrong (only present on failure)
                - status: Status of the operation (only present on failure)
        """
        try:
            # Create a chat completion request to the OpenAI API
```

```
            response = self.client.chat.completions.create(
                model=self.model,
                messages=[
                    {"role": "system", "content": "You are a helpful assistant. Be as precise
and concise as possible, directly addressing the question."
},
                    {"role": "user", "content": query}
                ]
            )

            # Return successful response with relevant information
            return {
                "answer": response.choices[0].message.content,
                "model": self.model,
                "tokens_used": response.usage.total_tokens
            }

        except Exception as e:
            # Handle any errors that occur during the API call
            return {
                "error": str(e),
                "status": "failed"
            }
```

Appendix E: Data filtering for Test Questions

```
import pandas as pd
import numpy as np
import random

# Load the dataset
df = pd.read_csv('data/processed/cars_for_rag_natural_10000.csv')

# List of features to analyze
features = [
    'Torque(lb-ft)',
    'Electrical motor torque',
    'CO2 Emissions',
    'CO2 Emissions (Combined)',
    'Turning circle (curb to curb)',
    'Cylinders',
    'Displacement',
    'Turning circle',
]

# Function to convert string values to float
# This handles cases where values include units (e.g., "39 ft" -> 39.0)
def convert_to_float(value):
    if pd.isna(value):
        return np.nan
    try:
        # Remove units and convert to float
        return float(str(value).split()[0])
    except:
        return np.nan

# Convert all feature values to float for comparison
for feature in features:
    df[feature] = df[feature].apply(convert_to_float)

# Set to store used models to ensure uniqueness
used_models = set()

# Find two unique cars for each feature
for feature in features:
    print(f"\n{'='*80}")
    print(f"FEATURE: {feature}")
    print(f"{'='*80}")
```

```
# Filter out cars that have already been used
available_cars = df[~df['Model'].isin(used_models)]

# Get random 10 cars that have the feature
random_cars = available_cars.dropna(subset=[feature]).sample(n=10)

# Group by manufacturer and get one car from each manufacturer
manufacturer_cars = {}
for idx, car in random_cars.iterrows():
    manufacturer = car['Company']
    if manufacturer not in manufacturer_cars:
        manufacturer_cars[manufacturer] = car

# Convert to list and randomly select 2 different manufacturers
manufacturer_list = list(manufacturer_cars.values())
if len(manufacturer_list) >= 2:
    selected_cars = random.sample(manufacturer_list, 2)
else:
    selected_cars = manufacturer_list

for car in selected_cars:
    # Add model to used models set
    used_models.add(car['Model'])

    # Print car information
    print(f"\nModel: {car['Model']}")
    print(f"Manufacturer: {car['Company']}")
    print(f"Production Years: {car['Production years']}")

    # Print feature value with context and units
    if feature == 'Torque(lb-ft)':
        print(f"Torque: {car[feature]} lb-ft")
    elif feature == 'Electrical motor torque':
        print(f"Electric Motor Torque: {car[feature]} Nm")
    elif feature == 'Aerodynamics (Cd)':
        print(f"Drag Coefficient: {car[feature]}")
    elif feature in ['CO2 Emissions', 'CO2 Emissions (Combined)']:
        print(f"CO2 Emissions: {car[feature]} g/km")
    elif feature in ['Turning circle', 'Turning circle (curb to curb)']:
        print(f"Turning Circle: {car[feature]} m")
    elif feature == 'Wheelbase':
        print(f"Wheelbase: {car[feature]} mm")
    elif feature == 'Cylinders':
        print(f"Number of Cylinders: {int(car[feature])}")
```

```
        elif feature == 'Displacement':
            print(f"Engine Displacement: {car[feature]} cm³")

        # Print car specifications
        print("\nSpecifications:")
        print(car['Specification summary'])
        print("-" * 50)
```

Appendix F: Questions and Answers of Basic Pipeline

This appendix contains all answers generated by the Basic Pipeline. The responses were produced without access to any external knowledge or retrieval mechanism. Errors such as hallucinations, estimations, or vague approximations are highlighted by binary scoring in Chapter 3.

Torque	What is the torque of the FORD Super Duty Lariat Tremor with the 6.7L V8 (475 HP) engine from the year 2021?	The 2021 Ford Super Duty Lariat Tremor with the 6.7L V8 engine produces 1,050 lb-ft of torque.
Torque	What is the torque of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?	The Bristol Fighter T, equipped with the 8.0L V10 engine, produces a torque of approximately 1050 lb-ft (1,420 Nm).
Electric Motor Torque	What is the electric motor torque of the BENTLEY Flying Spur with the 2.9L V6 (543 HP) hybrid system from the year 2021?	The Bentley Flying Spur with the 2.9L V6 hybrid system in the 2021 model year produces an electric motor torque of 404 Nm (298 lb-ft).
Electric Motor Torque	What is the electric motor torque of the 2023 Ferrari 296 GTS with the 3.0L V6 Turbo 8AT RWD (830 HP) engine from the year 2023?	The 2023 Ferrari 296 GTS with the 3.0L V6 Turbo engine delivers approximately 546 lb-ft (740 Nm) of torque.
CO2 Emissions	What are the CO_2 emissions of the BMW 8 Series (E31) with the 840Ci 6MT (286 HP) engine from the year 1992?	The BMW 8 Series (E31) 840Ci with a 6MT engine (286 HP) from 1992 has CO_2 emissions of approximately 260 g/km.
CO2 Emissions	What are the CO_2 emissions of the BUGATTI Veyron Grand Sport with the 8.0L W16 7AT (1001 HP) engine from the year 2010?	The CO_2 emissions of the 2010 BUGATTI Veyron Grand Sport with the 8.0L W16 engine are approximately 560 g/km.
CO2 Emissions (Combined)	What is the combined CO_2 emission of the AUDI RS Q8 with the 4.0L TFSIV8 8AT AWD (600 HP) engine from the year 2022?	The combined CO_2 emissions of the 2022 AUDI RS Q8 with the 4.0L TFSI V8 engine are approximately 320 g/km.

CO2 Emissions (Combined)	What is the combined CO_2 emission of the 2023 Ford Ranger Raptor with the 3.0L V6 EcoBoost 10AT AWD (292 HP) engine from the year 2023?	The 2023 Ford Ranger Raptor with the 3.0L V6 EcoBoost engine has an estimated CO_2 emission of approximately 269 grams per kilometer (g/km). However, actual emissions can vary based on driving conditions and other factors. For the most accurate and specific data, refer to the manufacturer's specifications or official emissions testing results.
Turning Circle (curb to curb)	What is the turning circle (curb to curb) of the AUDI A8 with the 60 TFSI e V6 quattro 8AT AWD (449 HP) engine from the year 2017?	The turning circle (curb to curb) of the 2017 Audi A8 with the 60 TFSI e V6 quattro is approximately 12.1 meters (39.7 feet).
Turning Circle (curb to curb)	What is the turning circle (curb to curb) of the DODGE Durango with the 3.5L V6 8AT (293 HP) engine from the year 2020?	The turning circle (curb to curb) of the 2020 Dodge Durango with the 3.5L V6 engine is approximately 37.1 feet (11.3 meters).
Cylinders	How many cylinders does the BUGATTI Veyron Grand Sport Vitesse with the 8.0 W16 7AT (1200 HP) engine from the year 2012 have?	The BUGATTI Veyron Grand Sport Vitesse with the 8.0 W16 engine has 16 cylinders.
Cylinders	How many cylinders does the 2014 Aston Martin Vantage N430 with the 4.7 V8 6AT (436 HP) engine from the year 2014 have?	The 2014 Aston Martin Vantage N430 with the 4.7 V8 engine has 8 cylinders.
Engine Displacement	What is the engine displacement of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?	The engine displacement of the BRISTOL Fighter T with the 8.0L V10 engine is 8.0 liters.
Engine Displacement	What is the engine displacement of the CHEVROLET Suburban with the 8.1L V8 4AT RWD (325 HP) engine from the year 2000?	The engine displacement of the Chevrolet Suburban with the 8.1L V8 engine is 8.1 liters, which is equivalent to 496 cubic inches.

Turning Circle (general)	What is the turning circle of the FORD F-150 Super Cab with the 4.6L (248 HP) engine from the year 2009?	The turning circle of the 2009 Ford F-150 Super Cab with the 4.6L engine is approximately 44.2 feet.

Appendix G: Questions and Answers of RAG Pipeline

This appendix presents all answers generated by the Retrieval-Augmented Generation (RAG) Pipeline. The system retrieved relevant vehicle descriptions from a vector database and used them to construct a grounded answer. All 15 answers were rated as correct.

Feature	Question	RAG Pipeline Response
Torque	What is the torque of the FORD Super Duty Lariat Tremor with the 6.7L V8 (475 HP) engine from the year 2021?	The torque of the FORD Super Duty Lariat Tremor with the 6.7L V8 (475 HP) engine from the year 2021 is 1050 lb-ft.
Torque	What is the torque of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?	The torque of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006 is 1036 lb-ft @ 4500 RPM.
Electric Motor Torque	What is the electric motor torque of the BENTLEY Flying Spur with the 2.9L V6 (543 HP) hybrid system from the year 2021?	The electric motor torque of the BENTLEY Flying Spur with the 2.9L V6 (543 HP) hybrid system is 295 lb-ft (400 Nm).
Electric Motor Torque	What is the electric motor torque of the 2023 Ferrari 296 GTS with the 3.0L V6 Turbo 8AT RWD (830 HP) engine from the year 2023?	The electric motor torque of the 2023 Ferrari 296 GTS is 545.8 lb-ft (740 Nm).
CO2 Emissions	What are the CO_2 emissions of the BMW 8 Series (E31) with the 840Ci 6MT (286 HP) engine from the year 1992?	The CO_2 emissions of the BMW 8 Series (E31) with the 840Ci 6MT (286 HP) engine are 492 g/km.
CO2 Emissions	What are the CO_2 emissions of the BUGATTI Veyron Grand Sport with the 8.0L W16 7AT (1001 HP) engine from the year 2010?	The CO_2 emissions of the BUGATTI Veyron Grand Sport (1001 HP) from the year 2010 are 574 g/km.
CO2 Emissions (Combined)	What is the combined CO_2 emission of the AUDI RS Q8 with the 4.0L TFSIV8 8AT AWD (600 HP) engine from the year 2022?	The combined CO_2 emission of the AUDI RS Q8 with the 4.0L TFSIV8 8AT AWD (600 HP) engine is 270 g/km.
CO2 Emissions (Combined)	What is the combined CO_2 emission of the 2023 Ford Ranger Raptor with the 3.0L V6 EcoBoost 10AT AWD (292 HP) engine from the year 2023?	The combined CO_2 emission of the 2023 Ford Ranger Raptor with the 3.0L V6 EcoBoost 10AT AWD (292 HP) engine is 315 g/km.

Feature	Question	RAG Pipeline Response
Turning Circle (curb to curb)	What is the turning circle (curb to curb) of the AUDI A8 with the 60 TFSI e V6 quattro 8AT AWD (449 HP) engine from the year 2017?	The turning circle (curb to curb) of the AUDI A8 with the 60 TFSI e V6 quattro 8AT AWD (449 HP) engine from the year 2017 is 41 ft (12.5 m).
Turning Circle (curb to curb)	What is the turning circle (curb to curb) of the DODGE Durango with the 3.5L V6 8AT (293 HP) engine from the year 2020?	The turning circle (curb to curb) of the DODGE Durango with the 3.5L V6 8AT (293 HP) engine from the year 2020 is 41 ft (12.5 m).
Cylinders	How many cylinders does the BUGATTI Veyron Grand Sport Vitesse with the 8.0 W16 7AT (1200 HP) engine from the year 2012 have?	The BUGATTI Veyron Grand Sport Vitesse from the year 2012 has 16 cylinders.
Cylinders	How many cylinders does the 2014 Aston Martin Vantage N430 with the 4.7 V8 6AT (436 HP) engine from the year 2014 have?	The 2014 Aston Martin Vantage N430 with the 4.7 V8 6AT (436 HP) engine has 8 cylinders.
Engine Displacement	What is the engine displacement of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?	The engine displacement of the BRISTOL Fighter T (2006) with the 8.0L V10 engine is 7994 cm^3.
Engine Displacement	What is the engine displacement of the CHEVROLET Suburban with the 8.1L V8 4AT RWD (325 HP) engine from the year 2000?	The engine displacement of the CHEVROLET Suburban with the 8.1L V8 4AT RWD (325 HP) engine is 8128 cm^3.
Turning Circle (general)	What is the turning circle of the FORD F-150 Super Cab with the 4.6L (248 HP) engine from the year 2009?	The turning circle of the FORD F-150 Super Cab with the 4.6L (248 HP) engine from the year 2009 is 46.9 ft (14.3 m).

Appendix H: Questions and Answers of ChatGPT Assistant

This appendix provides the full responses generated by the ChatGPT assistant when it was given unrestricted access to the structured dataset in CSV format.

Feature	Question	ChatGPT Assistant Response
Torque	What is the torque of the FORD Super Duty Lariat Tremor with the 6.7L V8 (475 HP) engine from the year 2021?	1,050 lb-ft / 1,424 Nm
Torque	What is the torque of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?	1,036 lb-ft @ 4500 RPM / 1,405 Nm
Electric Motor Torque	What is the electric motor torque of the BENTLEY Flying Spur with the 2.9L V6 (543 HP) hybrid system from the year 2021?	400 Nm
Electric Motor Torque	What is the electric motor torque of the 2023 Ferrari 296 GTS with the 3.0L V6 Turbo 8AT RWD (830 HP) engine from the year 2023?	545.8 lb-ft / 740 Nm
CO2 Emissions	What are the CO_2 emissions of the BMW 8 Series (E31) with the 840Ci 6MT (286 HP) engine from the year 1992?	492 g/km
CO2 Emissions	What are the CO_2 emissions of the BUGATTI Veyron Grand Sport with the 8.0L W16 7AT (1001 HP) engine from the year 2010?	574 g/km
CO2 Emissions (Combined)	What is the combined CO_2 emission of the AUDI RS Q8 with the 4.0L TFSIV8 8AT AWD (600 HP) engine from the year 2022?	270 g/km
CO2 Emissions (Combined)	What is the combined CO_2 emission of the 2023 Ford Ranger Raptor with the 3.0L V6 EcoBoost 10AT AWD (292 HP) engine from the year 2023?	315 g/km
Turning Circle (curb to curb)	What is the turning circle (curb to curb) of the AUDI A8 with the 60 TFSI e V6 quattro 8AT AWD (449 HP) engine from the year 2017?	12.9 meters (42.3 feet)
Turning Circle (curb to curb)	What is the turning circle (curb to curb) of the DODGE Durango with the 3.5L V6 8AT (293 HP) engine from the year 2020?	11.8 meters (38.7 feet)

Feature	Question	ChatGPT Assistant Response
Cylinders	How many cylinders does the BUGATTI Veyron Grand Sport Vitesse with the 8.0 W16 7AT (1200 HP) engine from the year 2012 have?	16 cylinders
Cylinders	How many cylinders does the 2014 Aston Martin Vantage N430 with the 4.7 V8 6AT (436 HP) engine from the year 2014 have?	8 cylinders
Engine Displacement	What is the engine displacement of the BRISTOL Fighter T with the 8.0L V10 6MT RWD (1026 HP) engine from the year 2006?	7994 cc
Engine Displacement	What is the engine displacement of the CHEVROLET Suburban with the 8.1L V8 4AT RWD (325 HP) engine from the year 2000?	8128 cc
Turning Circle (general)	What is the turning circle of the FORD F-150 Super Cab with the 4.6L (248 HP) engine from the year 2009?	41.7 feet (12.7 meters)