VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Programming Intergration Project (CO3101)

Group 4:
"Research and build AI chatbots using Retrieval-Augmented Generation for a music-related website, with Speech-to-Text and Text-to-Speech integration"

Instructor(s): Nguyễn Quốc Minh

Students: Nguyễn Thiện Minh - 2312097
Huỳnh Đức Nhân - 2312420
Phạm Trần Minh Trí - 2313622

HO CHI MINH CITY, DECEMBER 2025

Contents

List of Figures

List of Tables

Member list & Workload

No.	Fullname	Student ID	Problems	% done
1	Nguyễn Thiện Minh	2312097	- Exercise 1: 1.2 - Exercise 2 - Exercise 3: 3.2	100%
2	Huỳnh Đức Nhân	2312420	- Exercise 1: 1.3 - Exercise 2 - Exercise 3: 3.1 - Exercise 4	100%
3	Phạm Trần Minh Trí	2313622	- Exercise 1: 1.1 - Exercise 4 - LaTeX	100%

Table 1: Member list & workload

1 Introduction

2 Prerequisite: ANN, Transformer, LLM

2.1 Artificial Neural Network

Artificial Neural Networks (ANNs) are computational models inspired by the biological neural system. An ANN consists of multiple artificial neurons interconnected through weighted connections, enabling the model to learn complex mappings between input data and output predictions. Neural networks are widely used in classification, regression, and pattern recognition tasks due to their strong representation capability.

2.1.1 Overall Architecture of Neural Networks

A typical feedforward neural network is composed of three main components: an input layer, one or more hidden layers, and an output layer. Information flows from the input layer through hidden layers to produce the final output.

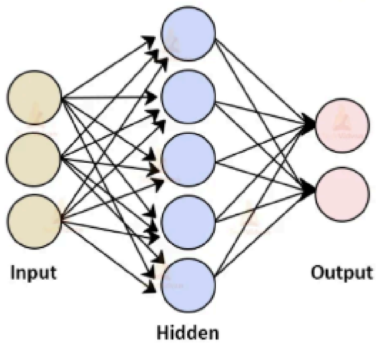

Figure 1: Basic architecture of a feedforward neural network

Each neuron in one layer is usually connected to all neurons in the next layer through weighted edges, allowing the network to model complex relationships.

2.1.2 Mathematical Model of an Artificial Neuron

An artificial neuron performs a weighted summation of its inputs followed by a nonlinear transformation. Given n input features x_i, the neuron computes an intermediate value z as:

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{1}$$

where w_i denotes the weight associated with input x_i, and b is the bias term. The output of the neuron is obtained by applying an activation function $f(\cdot)$:

$$y = f(z) \tag{2}$$

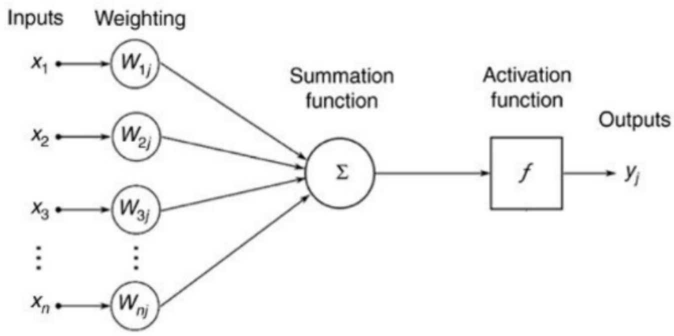

Figure 2: Mathematical structure of an artificial neuron

2.1.3 Activation Functions

Activation functions introduce non-linearity into the neural network. Without them, the network would behave as a linear model and fail to capture complex patterns in data. Common activation functions include:

- **Sigmoid**: maps input values into the range $(0, 1)$, often used in binary classification.
- **Tanh**: outputs values in the range $(-1, 1)$, providing zero-centered activations.
- **ReLU (Rectified Linear Unit)**: defined as $f(z) = \max(0, z)$, widely used due to its simplicity and effectiveness.

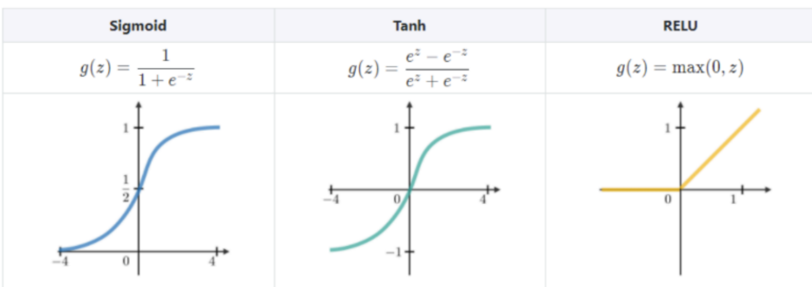

Figure 3: Common activation functions: Sigmoid, Tanh, and ReLU

2.1.4 Single-layer and Multi-layer Perceptrons

A **single-layer perceptron** contains no hidden layers and can only solve linearly separable problems. In contrast, a **multi-layer perceptron (MLP)** consists of one or more hidden layers, enabling the network to learn complex nonlinear relationships.

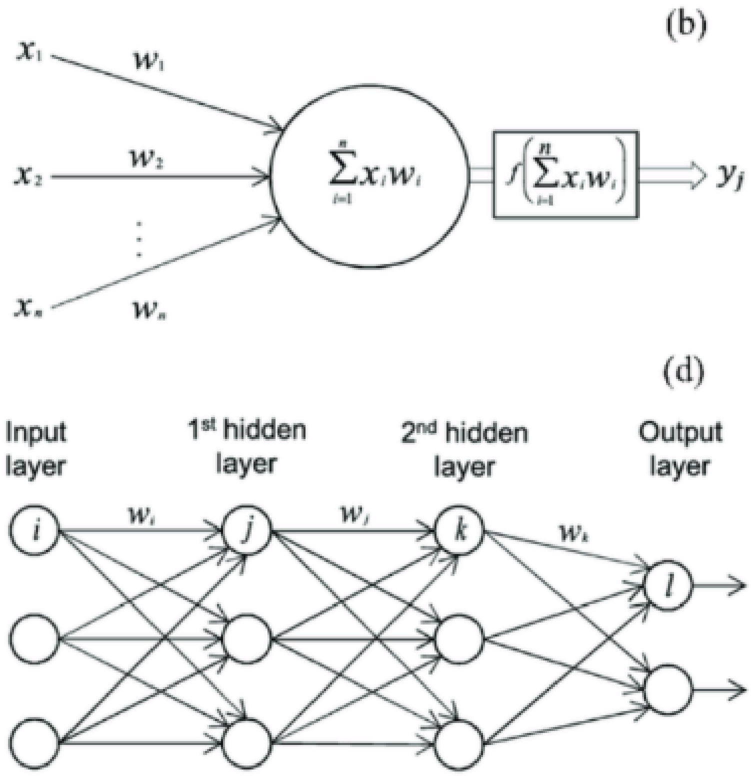

Figure 4: Comparison between single-layer and multi-layer perceptrons

2.1.5 Forward Propagation

Forward propagation is the process in which input data is passed through the network layer by layer to compute the output prediction. At each layer, weighted sums and activation functions are applied to transform the data.

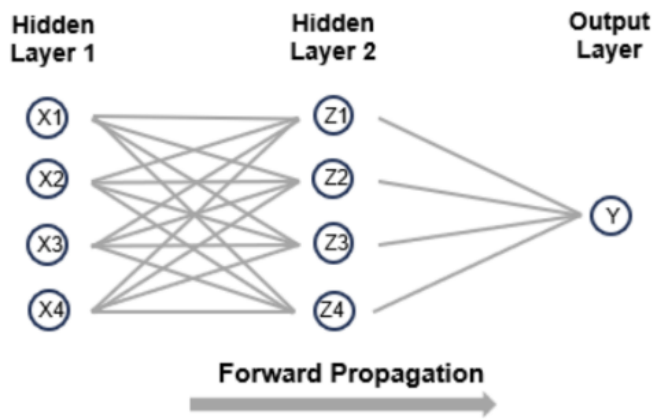

Figure 5: Forward propagation through hidden layers

2.1.6 Loss Function

The loss function measures the discrepancy between the predicted output $\hat{y}$ and the true target value y. It acts as a guidance signal for the learning process, where a smaller loss indicates better model performance. Common loss functions include:

- Mean Squared Error (MSE) for regression tasks:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3}$$

- Cross-Entropy Loss for classification tasks:

$$L = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) \tag{4}$$

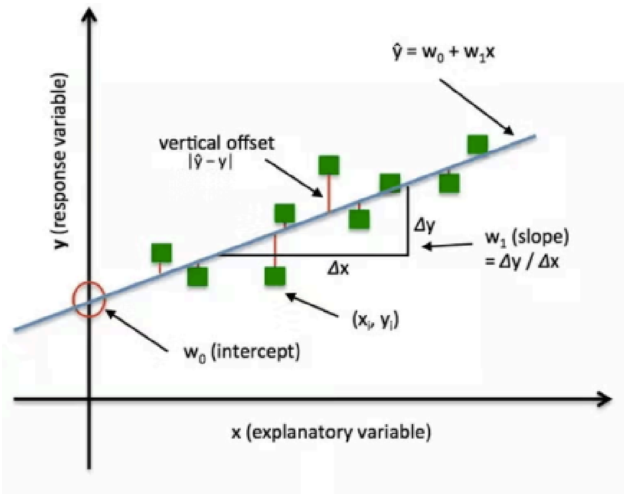

Figure 6: Illustration of prediction error and loss minimization

2.1.7 Backpropagation and Weight Update

Backpropagation is an algorithm used to compute the gradient of the loss function with respect to each weight in the network. Based on these gradients, weights are updated iteratively using optimization methods such as gradient descent.

The training process of a neural network consists of three repeated steps:

1. Forward propagation
2. Backpropagation
3. Weight update

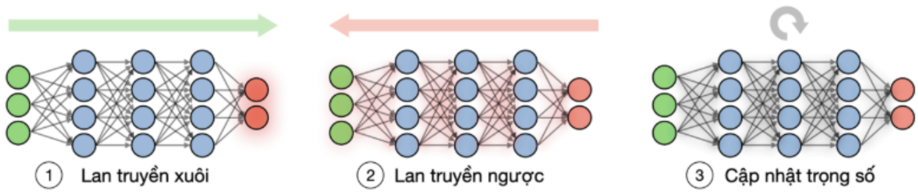

Figure 7: Training process: forward propagation, backpropagation, and weight update

2.2 Transformer architecture

2.3 Large language model

Large Language Models (LLMs) are advanced neural network models designed to understand, generate, and manipulate natural language at scale. They are typically built upon the Transformer architecture and trained on massive text corpora, enabling them to perform a wide range of Natural Language Processing (NLP) tasks such as text generation, question answering, summarization, and dialogue systems.

2.3.1 Evolution and Architecture

The development of LLMs has progressed from traditional statistical language models (e.g., n-gram models) to recurrent architectures such as Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, and ultimately to Transformer-based models. Transformers address the limitations of sequential computation and long-term dependency learning through the self-attention mechanism, making them highly scalable and effective for large datasets.

Based on their architectural design, LLMs can be categorized into three main types:

- **Decoder-only models (causal language models):** These models, such as GPT and LLaMA, predict the next token in a sequence and are primarily used for text generation tasks.
- **Encoder-only models (bidirectional language models):** Models like BERT, RoBERTa, and DistilBERT focus on learning contextual representations of text and are widely used for text understanding tasks such as classification and semantic similarity.
- **Encoder–Decoder models (sequence-to-sequence):** Examples include T5 and BART, which encode the input sequence and then decode it into an output sequence, making them suitable for machine translation, summarization, and question answering.

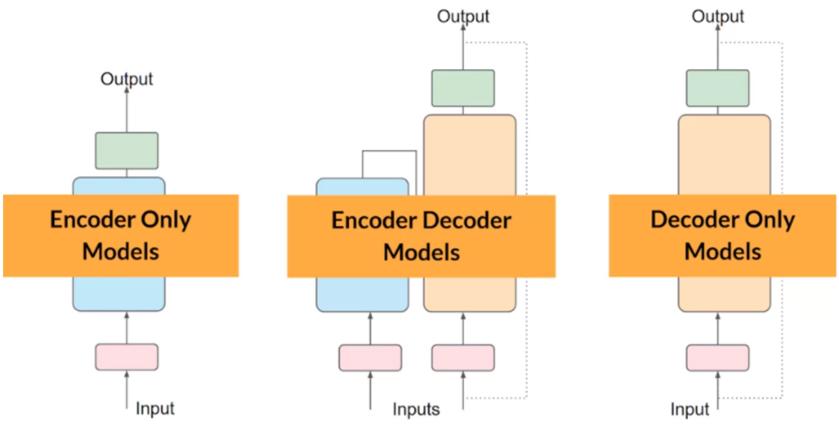

Figure 8: Transformer architectures

2.3.2 Pretraining and Fine-Tuning

LLMs are first trained during a pretraining phase using large-scale datasets such as Common Crawl, Wikipedia, and book corpora. Common pretraining objectives include autoregressive language modeling, masked language modeling, and denoising tasks. These objectives allow the model to learn grammar, semantics, and world knowledge from raw text.

After pretraining, LLMs are adapted to specific tasks through fine-tuning. This process may include Supervised Fine-Tuning (SFT), where human-labeled prompt–response pairs are used, and Reinforcement Learning from Human Feedback (RLHF), where human preferences guide the optimization of model outputs. Parameter-Efficient Fine-Tuning (PEFT) techniques such as LoRA and QLoRA are often employed to reduce computational cost while maintaining performance.

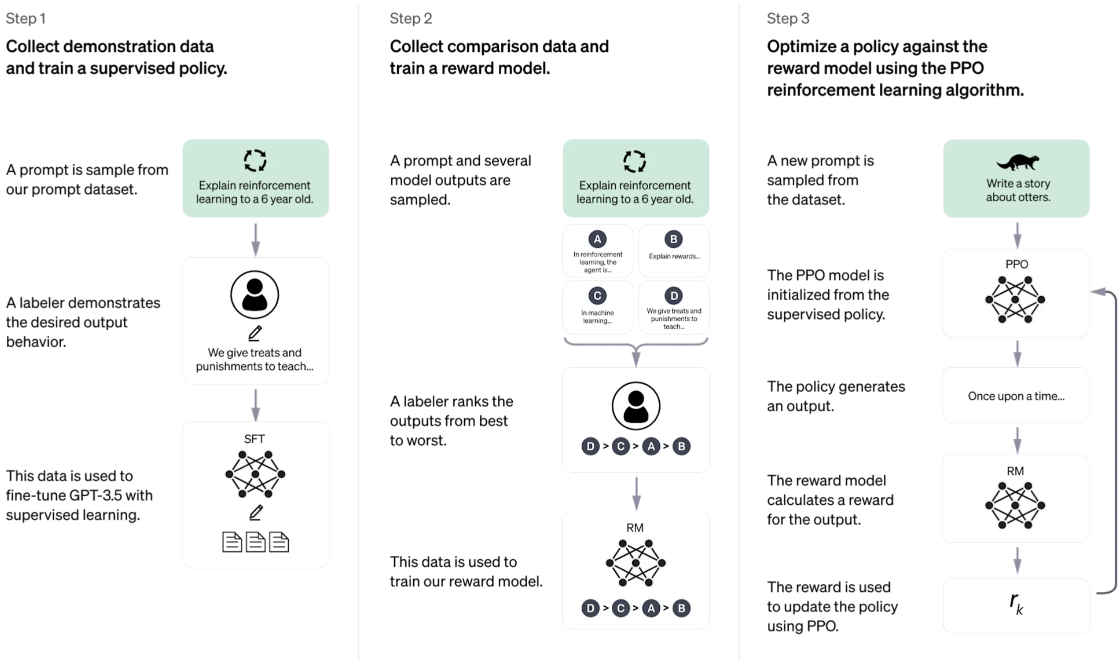

Figure 9: RLHF for ChatGPT

2.3.3 LLMs in Chatbot Systems

LLMs play a central role in modern chatbot systems. To improve factual accuracy and domain specificity, Retrieval-Augmented Generation (RAG) is commonly used. In a RAG-based system, relevant documents are embedded into a vector space and stored in a vector database. When a user submits a query, the system retrieves the most relevant documents and provides them as additional context to the LLM before generating a response. This approach significantly enhances the reliability and explainability of chatbot answers.

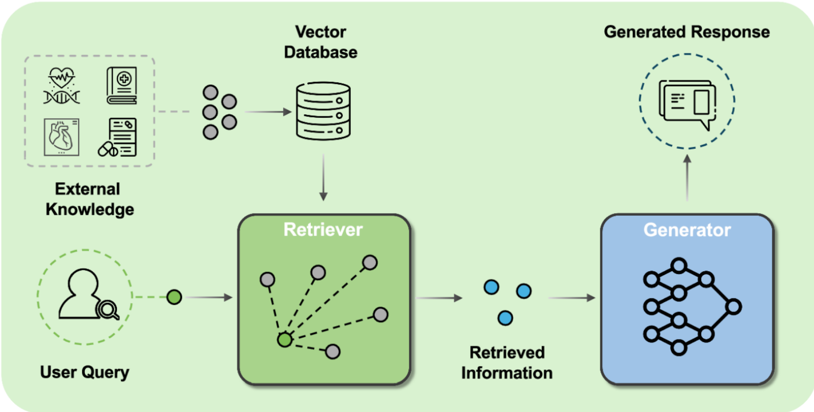

Figure 10: Retrieval augmented generation

In addition, prompt engineering and system instructions are used to control the behavior, style, and scope of the chatbot. For domain-specific applications such as a music information website, these techniques ensure that responses remain relevant, concise, and aligned with predefined constraints.

3 Agent, URAG, LangChain

3.1 AI Agent

An AI agent is a system that leverages an artificial intelligence model to interact with its environment in order to achieve a user-defined objective. Unlike traditional AI systems that produce static outputs, an AI agent integrates reasoning, planning, and action execution to complete tasks in a dynamic and autonomous manner.

3.1.1 Introduction to AI Agents

An AI agent operates by receiving a user request, reasoning about the task, selecting appropriate tools, and executing actions to fulfill the objective. This process allows agents to solve multi-step problems that require interaction with external systems and continuous adaptation based on feedback.

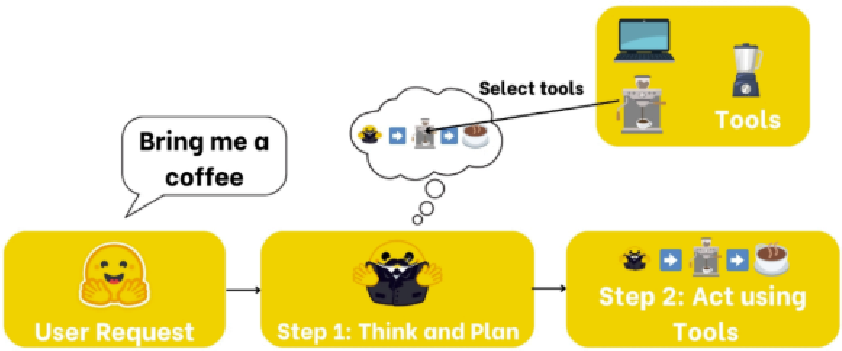

Figure 11: High-level AI agent workflow: user request, reasoning, and tool-based action

3.1.2 Core Components of an AI Agent

AI agents consist of two main components:

- **The Brain (AI Model)**: This component is typically a Large Language Model (LLM) that performs reasoning, understands context, plans future steps, and decides which actions or tools should be used.
- **The Body (Capabilities and Tools)**: This component enables the agent to act within its environment. It includes external tools and APIs such as search engines, code execution environments, and databases, as well as memory and observation mechanisms.

3.1.3 Levels of Agent Agency

AI agents can be categorized according to their level of autonomy, commonly referred to as their *agency level*. As the agency level increases, the agent gains greater control over program execution and decision-making.

3.1.4 Thought–Action–Observation Cycle

The behavior of an AI agent follows a continuous loop known as the *Thought–Action–Observation* cycle. In this cycle, the agent first reasons about the problem, then performs an action, and finally observes the result of that action. The observation is used to update the agent's internal state and guide subsequent reasoning steps.

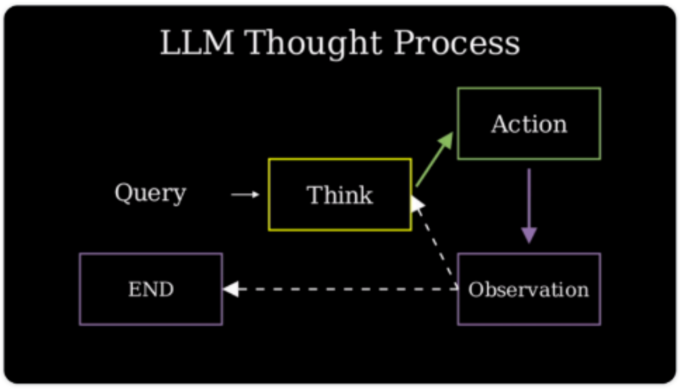

Figure 12: The Thought–Action–Observation (TAO) cycle in an AI agent

This iterative cycle enables agents to adapt their behavior dynamically based on environmental feedback.

3.1.5 Practical Illustration of the TAO Cycle

A common example of the Thought–Action–Observation cycle is a task that requires external information, such as retrieving real-time weather data. The agent reasons about the user's request, calls an appropriate external API, observes the returned data, and then produces a response.

Figure 13: Reasoning and action steps using an external weather API

3.1.6 Observation and Response Generation

After executing an action, the agent observes the outcome and integrates the feedback into its internal context. Based on this updated information, the agent performs additional reasoning to generate the final response for the user.

Figure 14: Observation feedback and final response generation

3.1.7 Internal Reasoning: Chain-of-Thought and ReAct

Reasoning is a critical capability of AI agents. Two widely used prompting approaches are:

- **Chain-of-Thought (CoT)**: A technique that encourages the model to reason step-by-step before generating a final answer.
- **ReAct (Reasoning and Acting)**: A technique that interleaves reasoning steps with actions, allowing the agent to think, act using tools, observe results, and continue reasoning until the task is completed.

3.1.8 Actions: Enabling Interaction with the Environment

Actions allow an AI agent to engage with its environment and accomplish tasks. Common categories of actions include information gathering, tool usage, environment interaction, and communication.

3.1.9 Observation and Adaptation

After executing an action, the agent observes the outcome and integrates the feedback into its internal context. This enables the agent to update its memory, refine its strategy, and improve performance in future interactions.

3.2 Unified Hybrid RAG

3.3 Introduction to LangChain

LangChain is a comprehensive framework designed to support the development, deployment, and monitoring of applications powered by Large Language Models (LLMs). It provides modular components and abstractions that simplify the construction of complex LLM-based systems such as Retrieval-Augmented Generation (RAG) pipelines and agentic workflows.

LangChain supports the full lifecycle of an LLM application. During the development phase, developers can build applications using LangChain's core components, including prompt templates, chains, retrievers, vector stores, and integrations with third-party tools and APIs. For more advanced agentic behaviors, LangChain introduces LangGraph, which enables the definition of multi-step agents with explicit states, nodes, and control flows. This graph-based design allows agents to reason, invoke tools, and iterate over multiple steps in a structured and controllable manner.

In the production phase, LangChain is complemented by LangSmith, a platform that provides observability, debugging, and evaluation capabilities. LangSmith allows developers to inspect prompt execution, track intermediate steps, monitor latency and costs, and systematically evaluate model outputs. These features are particularly important for RAG systems, where both retrieval quality and generation accuracy must be continuously assessed.

For deployment, LangChain offers the LangGraph Platform, which facilitates the deployment and scaling of agentic workflows. This platform-oriented approach makes LangChain suitable not only for experimentation but also for real-world applications.

Within a RAG architecture, LangChain structures the workflow into two main stages. The first stage is indexing, an offline process that involves loading documents, splitting them into chunks, generating embeddings using an embedding model, and storing these embeddings in a vector database. The second stage is retrieval and generation, which occurs at runtime: relevant document chunks are retrieved from the vector store based on the user query, combined with the query into a structured prompt, and passed to an LLM to generate a final response.

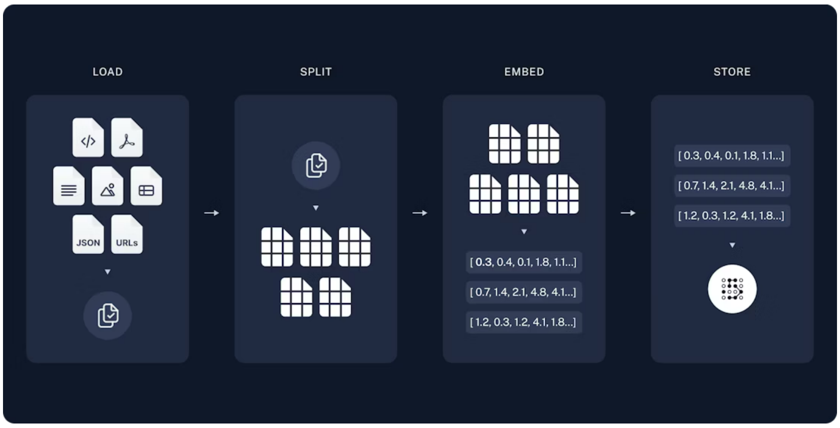

Figure 15: RAG - Indexing

LangChain provides a flexible and extensible foundation for building RAG-based and agent-driven applications, enabling developers to integrate LLMs, retrieval systems, and external tools into a unified and production-ready framework.

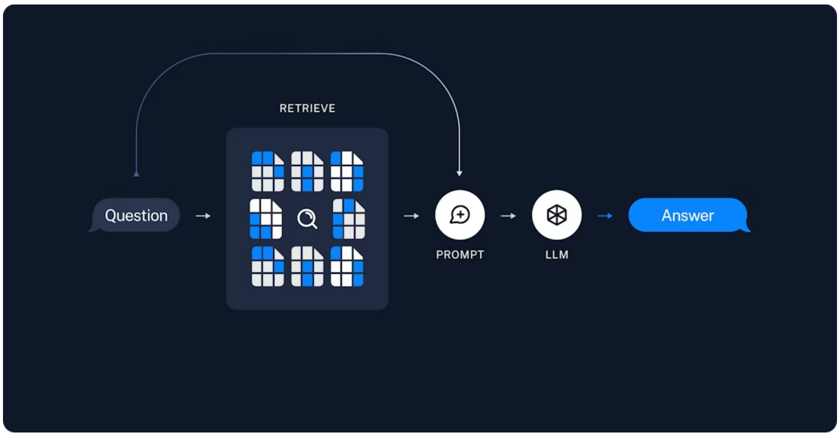

Figure 16: RAG - Retrieval and Generation

4 Retrieval-Augmented Generation for Large Language Models: A Survey

Link to the paper: https://arxiv.org/abs/2312.10997 [1]

4.1 Overview of RAG

Retrieval-Augmented Generation (RAG) is a paradigm that enhances large language models (LLMs) by incorporating external knowledge retrieved at inference time. Instead of relying solely on parametric knowledge stored in model weights, RAG dynamically retrieves relevant documents and uses them as additional context for response generation.

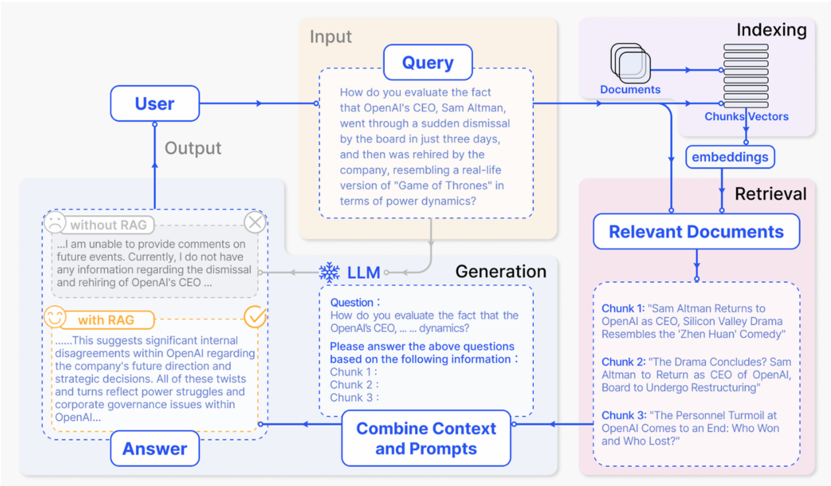

Figure 17: RAG overview

4.1.1 Naive RAG

The Naive RAG framework follows a simple *Retrieve–Read* pipeline consisting of three main stages: indexing, retrieval, and generation. During indexing, raw data sources such as PDFs, HTML pages, or Word documents are converted into plain text, segmented into smaller chunks, and encoded into vector representations stored in a vector database. In the retrieval stage, the user query is embedded and compared with stored vectors using similarity metrics to obtain the top-K most relevant chunks. Finally, in the generation stage, the LLM produces an answer based on the user query and the retrieved context, optionally incorporating conversation history.

Despite its simplicity, Naive RAG suffers from several limitations. Retrieval may lack precision and recall, resulting in irrelevant or missing information. During generation, the model may hallucinate content not supported by the retrieved context or produce outputs with bias or irrelevance. Moreover, effectively integrating retrieved information across different tasks remains challenging, often leading to redundant, incoherent, or overly extractive responses.

4.1.2 Advanced RAG

Advanced RAG aims to improve retrieval quality and context utilization through enhanced pre-retrieval and post-retrieval techniques. Pre-retrieval optimization focuses on improving indexing structures and query formulation, including data granularity control, metadata alignment, mixed retrieval strategies, and query rewriting or expansion. Post-retrieval optimization emphasizes effective context integration, such as re-ranking retrieved documents to prioritize relevance and compressing context to reduce noise and prompt length.

4.1.3 Modular RAG

Modular RAG extends beyond fixed retrieval-generation pipelines by introducing specialized, interchangeable modules. These include search modules for heterogeneous data sources, RAG-Fusion for multi-query expansion and re-ranking, and memory modules that maintain a continuously updated retrieval memory pool. Additional components such as routing, prediction, and task adapters allow RAG systems to dynamically select retrieval pathways and adapt to downstream tasks.

This modular design enables flexible retrieval patterns, including Rewrite–Retrieve–Read, Generate–Read, and hybrid retrieval strategies that combine keyword-based, semantic, and vector-based search. As a result, Modular RAG exhibits strong adaptability and scalability across diverse applications.

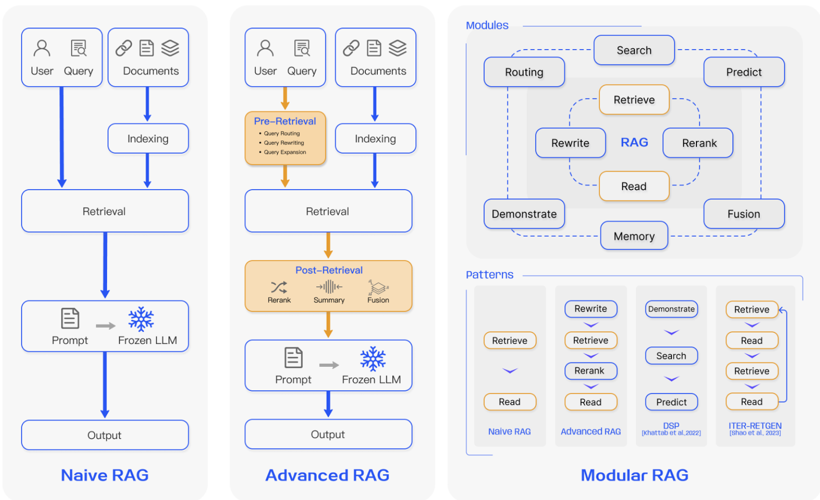

Figure 18: Types of RAG

4.2 Retrieval

Retrieval is a core component of RAG, responsible for identifying and supplying relevant external knowledge to the generation model. The effectiveness of a RAG system heavily depends on retrieval source selection, indexing strategies, query optimization, and embedding quality.

4.2.1 Retrieval Sources and Granularity

Retrieval sources can be categorized into unstructured, semi-structured, and structured data. Unstructured text data, such as Wikipedia articles or domain-specific documents, is the most common source. Semi-structured data, including PDFs with tables, presents challenges due to structural complexity, while structured sources like knowledge graphs offer precise and verified information at the cost of higher construction and maintenance effort.

Retrieval granularity ranges from tokens and sentences to chunks and full documents. Coarse-grained retrieval provides richer context but may introduce redundancy and noise, whereas fine-grained retrieval improves precision but risks losing essential semantic information.

4.2.2 Indexing Optimization

Indexing optimization techniques aim to balance context richness and efficiency. Chunking strategies play a critical role, where large chunks capture broader context but increase noise and computational cost, while small chunks reduce noise but may lack sufficient information. The *Small-to-Big* approach mitigates this trade-off by retrieving smaller units and expanding context hierarchically.

Metadata attachments, such as page numbers or timestamps, enable filtered retrieval and scoped search. Structural indexing methods, including hierarchical document structures and knowledge graph indices, further enhance retrieval speed and relevance. Techniques like Reverse HyDE leverage LLMs to generate potential questions that each chunk can answer, improving retrievability.

4.2.3 Query Optimization

Query optimization improves retrieval effectiveness by refining or expanding user queries. Query expansion and multi-query techniques enrich the query with additional context, while sub-query decomposition breaks complex questions into simpler ones. Query transformation methods include rewriting queries, generating hypothetical answers (HyDE), and step-back prompting to retrieve higher-level contextual information.

Query routing mechanisms further enhance retrieval by directing queries to appropriate data sources or pipelines using metadata-based or semantic routing strategies.

4.2.4 Embeddings and Adapters

Modern RAG systems often employ hybrid retrieval that combines sparse retrievers, such as BM25 for keyword matching, with dense retrievers based on neural embeddings for semantic understanding. Embedding models can be fine-tuned for domain-specific tasks, with LM-supervised retrievers aligning retrieval objectives with generation outcomes using LLM feedback.

When fine-tuning is impractical, adapter-based methods provide lightweight alternatives. These include prompt retrievers, bridging modules that transform retrieved content into LLM-friendly formats, and plug-in knowledge generators that replace or augment traditional retrievers in white-box settings.

4.3 Generation

Generation is the final stage in a Retrieval-Augmented Generation (RAG) pipeline, where the Large Language Model (LLM) produces the final response based on the user query and the augmented context retrieved from external knowledge sources. This stage directly determines the quality, coherence, and usefulness of the system's output.

4.3.1 Overview of the Generation Stage

In the basic RAG setting, also referred to as *Naive RAG*, the generation stage receives the user query together with the retrieved documents as input. These elements are combined into a single prompt, which is then passed to a frozen LLM to generate the final answer.

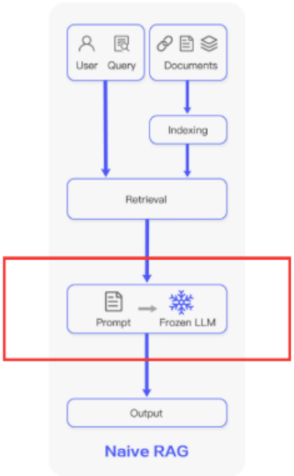

Figure 19: Generation stage in Naive RAG, where retrieved context is directly passed to a frozen LLM via a prompt

The generated response is expected to be fluent, context-grounded, and aligned with the retrieved evidence. However, this approach may suffer from noisy or redundant context, which can degrade answer quality.

4.3.2 Context Curation for Improved Generation

To address the limitations of Naive RAG, advanced RAG systems introduce a *context curation* step before generation. The goal of context curation is to improve relevance and reduce noise in the retrieved context supplied to the LLM.

Key components of context curation include:

- **Reranking**: reorders retrieved documents to prioritize the most relevant ones.
- **Context Selection or Compression**: filters, summarizes, or shortens retrieved content to avoid overly long prompts.

Figure 20: Advanced RAG with post-retrieval context curation before generation

By providing a more concise and focused augmented prompt, context curation enables the LLM to generate higher-quality and more accurate responses.

4.3.3 LLM Fine-tuning for Generation

Beyond improving the quality of the input context, generation performance can be further enhanced by fine-tuning the LLM itself. The objective of LLM fine-tuning is to adapt the model to domain-specific knowledge and desired response styles.

The key benefits of LLM fine-tuning include:

- **Domain Adaptation**: improves understanding and reasoning in specialized domains.
- **Output Alignment**: aligns tone, structure, and formatting with predefined guidelines.
- **Reduced Hallucination**: encourages stronger reliance on retrieved evidence.

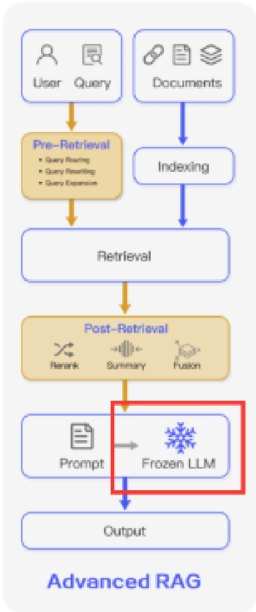

Figure 21: Generation stage using a fine-tuned LLM in an Advanced RAG pipeline

As a result, fine-tuned generation produces responses that are more accurate, coherent, and context-grounded, making it suitable for domain-specific and high-stakes applications.

4.4 Augmentation

Augmentation is a key process in Retrieval-Augmented Generation (RAG) systems that enhances answer quality by iteratively refining retrieval and generation. Instead of performing retrieval only once, the system can retrieve additional information, reformulate queries, or adjust context dynamically during the answering process.

4.4.1 Overview of the Augmentation Process

In the augmentation process, retrieval and generation are tightly coupled in an iterative loop. After each generation step, the system evaluates whether the current information is sufficient or if additional retrieval is required before producing the final response.

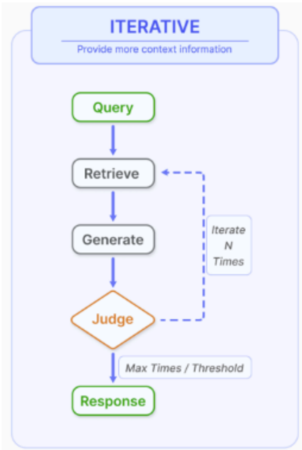

Figure 22: Overview of the iterative augmentation process in RAG

The main purposes of augmentation are:

- Handling complex or multi-step reasoning questions.
- Producing more accurate and context-grounded answers.

4.4.2 Iterative Retrieval

Iterative retrieval repeatedly alternates between retrieving new context and generating partial answers. Each generation step provides new signals that guide the next retrieval, gradually enriching the available context.

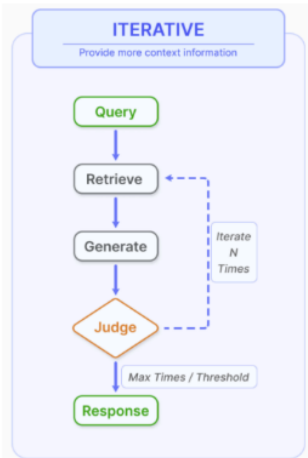

Figure 23: Iterative retrieval with repeated retrieval–generation cycles

The key ideas behind iterative retrieval include:

- Retrieval is performed again based on what has been generated so far.

- Context is progressively enriched to support step-by-step reasoning.

Pros and Cons:

- **Advantages**: more comprehensive and targeted information.
- **Disadvantages**: risk of semantic drift or accumulation of irrelevant context.

4.4.3 Recursive Retrieval

Recursive retrieval refines the query step-by-step using feedback from previous retrieval results. Each iteration clarifies what information is still missing, leading to progressively more relevant context.

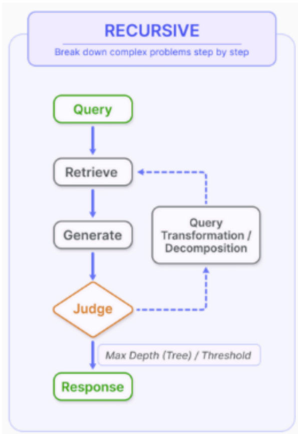

Figure 24: Recursive retrieval with query transformation and decomposition

Key characteristics of recursive retrieval include:

- Construction of intermediate reasoning structures such as chain-of-thought or clarification trees.
- Continuous reformulation of the query to target more specific information.

Pros and Cons:

- **Advantages**: improved accuracy and relevance over multiple iterations.
- **Disadvantages**: increased latency due to multiple refinement steps.

4.4.4 Adaptive Retrieval

Adaptive retrieval allows the LLM to dynamically decide when and what to retrieve during generation. Instead of enforcing retrieval at fixed stages, the model monitors its own confidence and triggers retrieval only when additional context is required.

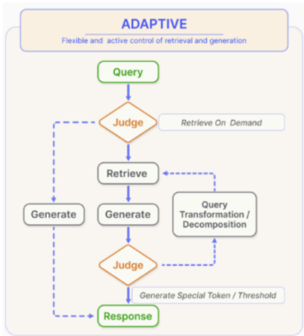

Figure 25: Adaptive retrieval where the LLM actively controls retrieval decisions

The key ideas of adaptive retrieval are:

- Retrieval is performed on demand rather than at predefined stages.
- The LLM behaves like an agent, evaluating intermediate outputs and deciding whether to retrieve more context or proceed to the final answer.

4.4.5 Summary

In summary, augmentation enhances the RAG pipeline by enabling iterative and adaptive interaction between retrieval and generation. Iterative, recursive, and adaptive retrieval strategies provide increasing levels of flexibility and reasoning capability, allowing RAG systems to better handle complex queries and produce more accurate, context-aware responses.

4.5 Task & evaluation

4.6 Discussion

5 Reranking, RAG-Reasoning, RAG-RL

5.1 Reranking

5.1.1 Limitations of Vector-Based Retrieval

In large-scale Retrieval-Augmented Generation (RAG) systems, retrieval is typically performed using dense vector embeddings and approximate nearest neighbor (ANN) search. While this approach is highly efficient and scalable, it introduces fundamental limitations.

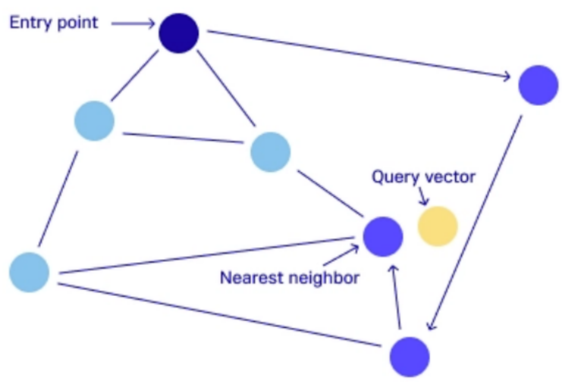

Figure 26: Approximate nearest neighbor search may return vectors that are close in embedding space but not truly semantically relevant

Dense retrieval commonly relies on a bi-encoder architecture, where queries and documents are encoded independently into vector representations. ANN search then finds the nearest vectors in the embedding space. However, vector similarity does not necessarily correspond to true semantic relevance. As a result, the retrieved top-k documents may lack sufficient reasoning, logical structure, or supporting evidence required by the LLM. Noisy or weakly relevant context can increase confusion and hallucination during generation.

5.1.2 Reranking as a Post-Retrieval Refinement Step

To mitigate the limitations of vector-based retrieval, reranking is introduced as a post-retrieval refinement step in the RAG pipeline. Instead of directly passing the retrieved documents to the generator, reranking evaluates and reorders them based on more accurate relevance estimation.

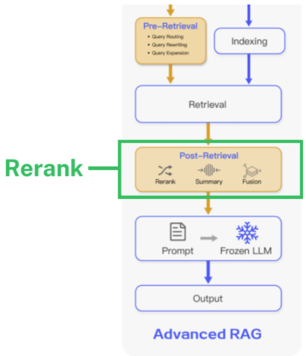

Figure 27: Reranking as an intermediate step between retrieval and generation in Advanced RAG

In this stage, each retrieved document is assessed together with the query, and the documents are reordered to prioritize those that are most relevant. By selecting a smaller set of high-quality passages, reranking reduces noise in the prompt and improves factual accuracy, coherence, and reliability of the final LLM-generated response.

5.1.3 Bi-Encoder versus Cross-Encoder for Reranking

A key architectural distinction in RAG systems lies between bi-encoders used for retrieval and cross-encoders used for reranking.

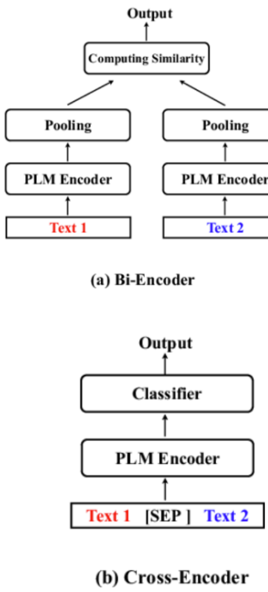

Figure 28: Comparison between bi-encoder retrieval and cross-encoder reranking architectures

Bi-encoders encode the query and documents separately and compute relevance using vector similarity. This design enables fast retrieval over large document collections but captures only coarse semantic similarity. In contrast, cross-encoders jointly encode the query and document within a single model, allowing full cross-attention between them. This results in significantly more accurate relevance estimation.

Due to their higher computational cost, cross-encoders are typically applied only to the top-k candidates returned by the bi-encoder. This two-stage design effectively balances scalability and accuracy in modern RAG systems.

5.1.4 Summary

In summary, reranking plays a critical role in bridging the gap between efficient retrieval and accurate generation. By refining retrieved results using more precise relevance modeling, reranking improves context quality, reduces hallucination, and significantly enhances the overall performance of Retrieval-Augmented Generation systems.

5.2 Towards Agentic RAG with Deep Reasoning: A Survey of RAG Reasoning Systems in LLMs

Link to paper: https://arxiv.org/abs/2507.09477 [5]

Retrieval-Augmented Generation (RAG) has emerged as a powerful paradigm for enhancing large language models (LLMs) by grounding generation in external knowledge sources. However, traditional RAG pipelines primarily focus on surface-level semantic retrieval and often struggle with multi-hop reasoning, noisy contexts, and complex decision-making tasks. The survey by Li et al. proposes a comprehensive framework that systematically analyzes how reasoning capabilities can be deeply integrated into RAG systems, moving toward more agentic and intelligent architectures.

The survey categorizes existing approaches into three major paradigms. The first is Reasoning-Enhanced RAG, where reasoning is explicitly incorporated to improve retrieval, integration, and generation. At the retrieval stage, techniques such as reasoning-aware query reformulation, retrieval planning, and retriever model enhancement aim to obtain evidence that is more relevant to downstream reasoning tasks. During integration, retrieved documents are assessed, filtered, and fused using reasoning-driven relevance assessment and information synthesis mechanisms. At the generation stage, context-aware and grounded generation methods ensure that the model's outputs remain faithful to retrieved evidence and follow coherent reasoning paths.

The second paradigm is RAG-Enhanced Reasoning, which treats retrieval as a tool to directly support the reasoning process of LLMs. In this setting, models retrieve external knowledge such as structured knowledge bases, web content, tools, or prior experiences to assist in complex reasoning tasks including mathematical problem solving, fact verification, and planning. In-context retrieval of examples and historical interactions further enables LLMs to adapt reasoning strategies dynamically based on retrieved demonstrations or memories.

The third paradigm, Synergized RAG-Reasoning, represents the most advanced integration, where retrieval and reasoning are tightly interwoven within an agentic workflow. These systems interleave reasoning steps with retrieval actions using chain-based, tree-based, or graph-based reasoning workflows. Moreover, agent orchestration techniques, including single-agent and multi-agent systems, allow LLMs to autonomously plan, retrieve, reason, and verify information. Such agentic RAG systems exhibit improved robustness, interpretability, and adaptability in complex tasks.

Category	Method summary	Related papers

Reasoning-Aware Query Reformulation (§3.1.1)	Reformulates the original query to better retrieve reasoning-relevant context. This includes query decomposition (breaking complex queries into simpler ones) , reformulation (recasting ambiguous queries) , and expansion (enriching the query via CoT).	e.g., Collab-RAG (Xu et al., 2025b), DynQR (Anonymous, 2025), DeepRetrieval (Jiang et al., 2025)
Retrieval Strategy and Planning (§3.1.2)	Covers global retrieval guidance. This involves advance planning to generate a retrieval blueprint before execution or adaptive retrieval methods that predict whether and how to retrieve based on query complexity.	e.g., PAR-RAG (Zhang et al., 2025d), LPKG (Wang et al., 2024b), FIND (Jia et al., 2025)
Retrieval Model Enhancement (§3.1.3)	Enhances retrievers with reasoning. This is done by leveraging structured knowledge (like KGs with GNNs or symbolic rules) or integrating explicit reasoning (like CoT) with the query.	e.g., GNN-RAG (Mavromatis & Karypis, 2024), RuleRAG (Chen et al., 2024c)
Relevance Assessment & Filtering (§3.2.1)	Uses deeper reasoning to assess the relevance of retrieved fragments. This can involve using "assessor experts" to select faithful evidence or models to filter non-entailing passages.	e.g., SEER (Zhao et al., 2024c), M-RAG-R (Yoran et al., 2024)
Information Synthesis & Fusion (§3.2.2)	Fuses relevant snippets into a coherent evidence set after they are identified. Methods include aggregating sub-question answers or building a reasoning graph to evaluate and aggregate knowledge.	e.g., BeamAggR (Chu et al., 2024), DualRAG (Cheng et al., 2025), CRP-RAG (Xu et al., 2024)
Context-Aware Generation (§3.3.1)	Ensures outputs remain relevant and reduces noise. This includes selective-context utilization (pruning or re-weighting content) and reasoning path generation (building explicit logical chains).	e.g., Open-RAG (Islam et al., 2024), RARE (Wang et al., 2025d), Self-Reasoning (Xia et al., 2025b)
Grounded Generation Control (§3.3.2)	Introduces verification mechanisms to anchor outputs to retrieved evidence. This is done via fact verification , citation generation , and faithful reasoning (ensuring steps adhere to evidence).	e.g., RARR (Gao et al., 2023a), TRACE (Fang et al., 2024), AlignRAG (Wei et al., 2025b)
Knowledge Base (§4.1.1)	Retrieves from KBs storing arithmetic, commonsense, or logical knowledge. This can include formal lemmas for math , legal precedents , or code snippets.	e.g., Premise-Retrieval (Tao et al., 2025), ReaRAG (Lee et al., 2025), CBR-RAG (Wiratunga et al., 2024)

Web Retrieval (§4.1.2)	Accesses dynamic online content like web pages, news, or social media. It is used for fact-checking by verifying claims step-by-step or for QA by iteratively refining reasoning.	e.g., ALR² (Li et al., 2024d), RARE (Tran et al., 2024), Open-RAG (Islam et al., 2024)
Tool Using (§4.1.3)	Leverages external resources like calculators, libraries, or APIs to enhance reasoning interactively. This improves numerical accuracy and computational robustness.	e.g., TATU (Li et al., 2024g), TRICE (Qiao et al., 2024), Re-Invoke (Chen et al., 2024a)
Prior Experience (§4.2.1)	Retrieves past interactions or successful strategies stored in a model's internal memory. This includes leveraging past decisions for planning or recalling conversational histories for adaptive reasoning.	e.g., RAP (Kagaya et al., 2024), JARVIS-1 (Wang et al., 2024f), EM-LLM (Fountas et al., 2024)
Example or Training Data (§4.2.2)	Retrieves external examples from demonstrations or training data. This provides relevant exemplars to guide the model in emulating specific reasoning patterns.	e.g., MoD (Wang et al., 2024c), RE4 (Li et al., 2024c), UPRISE (Cheng et al., 2023)
Chain-based (§5.1.1)	Interleaves retrieval operations between the linear "step-by-step" reasoning of a Chain-of-Thought (CoT) to avoid error propagation. Methods can also add verification or filtering steps.	e.g., IRCOT (Trivedi et al., 2023), Rat (Wang et al., 2024g), CoV-RAG (He et al., 2024a), RAFT (Zhang et al., 2024a)
Tree-based (§5.1.2)	Explores multiple reasoning pathways. Tree-of-Thought (ToT) methods build a deterministic reasoning tree. Monte Carlo Tree Search (MCTS) methods use probabilistic tree search to dynamically prioritize exploration.	ToT: e.g., RATT (Zhang et al., 2025a), Tree of Clarifications (Kim et al., 2023) MCTS: e.g., AirRAG (Feng et al., 2025), MCTS-RAG (Hu et al., 2025)
Graph-based (§5.1.3)	Walk-on-Graph uses graph learning techniques (like GNNs) to retrieve and reason over graph-structured data. Think-on-Graph integrates graph structures into the LLM's reasoning loop, letting the LLM decide which node to explore next.	Walk-on-Graph: e.g., QA-GNN (Yasunaga et al., 2021) Think-on-Graph: e.g., ToG (Sun et al., 2024b), Graph-CoT (Jin et al., 2024)
Single-Agent (§5.2.1)	A single agent interweaves retrieval into its reasoning loop. This is achieved via Prompting (e.g., ReAct) , Supervised Fine-Tuning (SFT) , or Reinforcement Learning (RL).	Prompting: e.g., ReAct (Yao et al., 2023b) SFT: e.g., Toolformer (Schick et al., 2023) RL: e.g., Search-R1 (Jin et al., 2025)

Multi-Agent (§5.2.2)	Uses multiple agents for collaboration. Decentralized systems use specialized agents that work together. Centralized systems use a hierarchical (e.g., manager-worker) pattern for task decomposition.	Decentralized: e.g., M-RAG (Wang et al., 2024) Centralized: e.g., HM-RAG (Liu et al., 2025), Chain of Agents (Zhang et al., 2024c)

The survey highlights a clear evolution from static retrieval pipelines toward dynamic, agent-based RAG systems with deep reasoning capabilities. It identifies key challenges such as efficiency, evaluation, and controllability, while outlining future research directions that aim to unify reasoning, retrieval, and agent learning into a coherent framework for next-generation LLM systems.

5.3 RAG-RL: Advancing Retrieval Augmented Generation via RL and Curriculum Learning

Link to paper: https://arxiv.org/abs/2503.12759 [3]

6 Implementation: LangChain, Text-to-speech, Speech-to-text

6.1 Building an AI Agent Chatbot with LangChain

We implement AI agent chatbot using the LangChain framework, designed to support a music-related website through Retrieval-Augmented Generation (RAG) combined with speech-based interaction. LangChain enables the construction of agentic systems by integrating large language models (LLMs) with external tools, memory, and retrieval mechanisms, allowing the chatbot to reason, decide actions, and iteratively solve user queries.

An **AI agent** in LangChain is defined as a system that combines a language model with a set of tools, enabling it to select and invoke appropriate tools based on the task context. Tools act as functional interfaces that extend the model's capabilities beyond pure text generation, such as searching documents or querying the web.

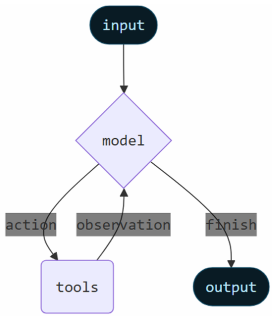

Figure 29: AI Agent

6.1.1 RAG Agent Architecture

The chatbot is built around a RAG agent architecture consisting of three main stages: indexing, retrieval, and response generation. During the indexing phase, nearly 100 Wikipedia articles related to Music are loaded using LangChain's `WikipediaLoader`. These documents are embedded using the `gemini-embedding-001` model and stored persistently in a `Chroma` vector database, enabling efficient semantic similarity search.

6.1.2 Agent Tools

The agent is equipped with multiple tools to enhance its reasoning and information access:

- **Context Retrieval Tool**: Retrieves the top-k (with $k = 10$) most relevant documents from the Chroma vector store based on semantic similarity.
- **Web Search Tool**: Uses DuckDuckGo to fetch real-time search results, including URLs, titles, and snippets, enabling access to up-to-date information.
- **Web Loader Tool**: Employs LangChain's `WebBaseLoader` to extract and process raw HTML content from selected web pages.

These tools are wrapped and exposed to the agent, allowing it to dynamically decide whether to rely on internal knowledge, retrieved documents, or external web sources.

6.1.3 Agent Configuration and Memory

The agent is initialized with a system prompt that defines its role and behavior. The conversational backbone uses the `gemini-2.5-flash` chat model for response generation. To maintain contextual coherence, short-term memory is incorporated, enabling the agent to remember and reference previous turns within a single conversation thread.

6.2 Text-to-Speech

Text-to-Speech (TTS) is a technology that converts written text into synthesized speech, enabling AI systems to produce audible responses for users. In conversational AI and Retrieval-Augmented Generation (RAG) systems, TTS plays an important role in enhancing user interaction by delivering information in a natural and accessible audio format.

6.2.1 Text-to-Speech Pipeline

A typical Text-to-Speech system follows a multi-stage processing pipeline that transforms raw text into an audio signal. The main stages of this pipeline are illustrated in Figure 30.

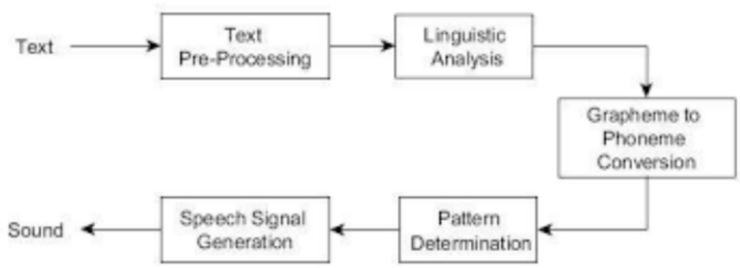

Figure 30: General Text-to-Speech pipeline from text input to synthesized speech output

The process begins with **text pre-processing**, where the input text is normalized by handling numbers, abbreviations, and formatting. Next, **linguistic analysis** examines grammatical structure and syntactic features. The **grapheme-to-phoneme conversion** stage maps characters to phonemes to determine correct pronunciation. Based on this phonetic representation, **pattern determination** defines prosodic features such as rhythm, pitch, and stress. Finally, **speech signal generation** synthesizes the waveform that produces the audible speech output.

6.2.2 gTTS: Google Text-to-Speech

gTTS (Google Text-to-Speech) is a lightweight Python library that provides an easy-to-use interface for converting text into speech. It leverages Google Translate's text-to-speech service to generate spoken audio from textual input and outputs the result as an MP3 file.

The simplicity of gTTS makes it suitable for rapid prototyping and integration into AI applications where basic speech output is required. Figure 31 shows a minimal example of using gTTS to synthesize Vietnamese speech from text.

```python
from gtts import gTTS

text = "Xin chào, đây là gTTS"
tts = gTTS(text, lang='vi')
tts.save("output.mp3")
```

Figure 31: Example of using gTTS in Python to generate an MP3 file from text

In this example, the input text is passed to the gTTS library along with the target language code. The generated speech is then saved as an audio file, which can be played back by the system. While gTTS is easy to use and freely accessible, it provides limited control over voice characteristics and requires an internet connection.

Overall, Text-to-Speech systems such as gTTS complement language models by enabling multimodal interaction, allowing AI-driven applications to communicate information not only through text but also through natural-sounding speech.

6.3 Speech-to-text

7 Evaluation

8 Conclusion

References

[1] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey.* Preprint. 2023. DOI: 10.48550/arXiv.2312.10997. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997.

[2] Google AI for Developers. *Google AI Studio.* Accessed: 2026-01-08. Google. 2025. URL: https://ai.google.dev/aistudio.

[3] Jerry Huang et al. *RAG-RL: Advancing Retrieval-Augmented Generation via RL and Curriculum Learning.* Preprint. 2025. DOI: 10.48550/arXiv.2503.12759. arXiv: 2503.12759 [cs.CL]. URL: https://arxiv.org/abs/2503.12759.

[4] LangChain Documentation. *Build a RAG agent with LangChain.* Accessed: 2026-01-05. LangChain. 2025. URL: https://docs.langchain.com/oss/python/langchain/rag.

[5] Yangning Li et al. *Towards Agentic RAG with Deep Reasoning: A Survey of RAG-Reasoning Systems in LLMs.* Preprint. 2025. DOI: 10.48550/arXiv.2507.09477. arXiv: 2507.09477 [cs.CL]. URL: https://arxiv.org/abs/2507.09477.

[6] OpenAI. *Introducing ChatGPT.* Accessed: 2026-01-08. OpenAI. 2022. URL: https://openai.com/index/chatgpt/.