uc3m | Universidad Carlos III de Madrid

Master Degree in Statistics

Academic Year 2024/2025

Master Thesis

"Agent-Based Retrieval-Augmented Generation with Large Language Models using LangGraph"

Youssef Benslimane

Juan Jose Cervigon Simo

Madrid, July 2025

SUMMARY

Large Language Models (LLMs) like ChatGPT and LLaMA have shown impressive abilities in language understanding and generation. However, they still suffer from a key issue: **hallucination**, producing confident but incorrect answers. This becomes a serious problem in tasks that require factual accuracy.

Retrieval-Augmented Generation (RAG) helps address this by giving LLMs access to external knowledge, much like doing a quick web search before answering. Still, most RAG systems follow a fixed path and can't adapt when the context is unclear or the retrieved content is weak.

This thesis proposes a more flexible solution: a **modular, agent-based RAG pipeline** built with **LangGraph**. In this setup, different agents are responsible for key steps : retrieving, rewriting, reranking, grading, and generating. Each agent can make decisions dynamically, improving the system's ability to handle vague or complex queries.

Through experiments and analysis, the work shows that this agentic design leads to better fallback behavior, improved answer quality, and stronger traceability. The goal is not only to boost accuracy, but to make LLMs more **reliable**, **transparent**, and **adaptable** in real-world use.

Keywords: Large Language Models (LLMs), Retrieval-Augmented Generation (RAG), LangGraph, Agent Systems, Modular AI Architecture, Hallucination Reduction, Query Rewriting, Ethical Reasoning, Document Retrieval, Sentence-BERT, TF-IDF, Cosine Similarity, Fallback Detection, Evaluation Metrics, FLAN-T5

DEDICATION

I humbly dedicate this work as a gesture of deep respect, gratitude, and heartfelt recognition to :

My **dearest parents**,

whose love has been a guiding light and whose sacrifices are the silent pillars of everything I have achieved.

My **sweet and kind-hearted sister**,

for her compassionate presence, her listening heart, and her gentle support - a quiet but steady motivation that helped me move forward with balance and clarity.

My **dear brother**,

for his thoughtful advice and encouragement during the early stages of this project. His perspective on tech trends and the European job market helped me feel confident in pursuing Retrieval-Augmented Generation.

My thesis supervisor, **Juan José Cervigón Simó**,

for his thoughtful guidance, steady support, and constructive feedback that shaped the direction and clarity of this work.

My **friends**,

for the moral support, shared moments, and helping hands that made the hardest days lighter. Your presence has made this journey more meaningful.

The faculty of the **Master in Statistics for Data Science** at **UC3M**,

for providing the academic foundation, rigorous training, and inspiration that made this research possible.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Large Language Models (LLMs), such as ChatGPT and LLaMA, have made rapid progress in recent years, enabling strong performance in text generation, summarization, and question answering. However, a common problem remains: **hallucination**, which occurs when models generate information that is inaccurate or fabricated.

To address this issue, **Retrieval-Augmented Generation (RAG)** has emerged as a widely used strategy. These systems retrieve external documents at query time and use them to guide generation, allowing models to produce grounded responses based on real data. This improves answer reliability, source traceability, and responsiveness to updated knowledge.

Despite its advantages, **standard RAG pipelines** tend to be static. They often follow a fixed sequence (retrieve, concatenate, generate), without mechanisms to adapt when retrieval is poor, queries are ambiguous, or the information is uncertain. This makes them less suitable for real-world tasks that require flexibility and deeper reasoning.

Recent works such as *AutoRAG* [1] and *RAG Best Practices* [2] highlight the importance of modular and adaptable systems, where components such as retrievers, rewriters, or rerankers can be activated or bypassed depending on context. These architectures promote experimentation, control, and a better understanding of system behavior.

This thesis proposes an **agent-based RAG system** using the **LangGraph** framework. LangGraph allows the pipeline to be built as a flow of modular agents, each responsible for a specific task and able to act based on intermediate results. This design aims to improve adaptability, interpretability and maintainability. To ensure consistent comparisons, all versions use the same instruction-tuned model, **FLAN-T5**, for answer generation.

The central research question is: *Can modular, agent-driven control improve the reliability and adaptability of RAG systems, especially for vague or underspecified prompts?*

To answer this, the system is built and evaluated with four main goals: first, we implement a five-agent RAG pipeline using LangGraph. Next, we compare the agent-based and static architectures in terms of retrieval and generation performance. Then, we evaluate the differences between dense (**SBERT**) and sparse (**TF-IDF**) embeddings and finally, we analyze the impact of key modules such as **grading**, **rewriting**, and **reranking** through ablation.

The evaluation combines theory with practical experimentation on a filtered Wikipedia dataset and a set of ethical prompts. Performance is measured using **ROUGE-L**, **METEOR**, **cosine similarity**, and **fallback rate**.

1. SCIENTIFIC AND TECHNICAL FOUNDATIONS

This chapter introduces the main ideas and tools behind the system developed in this thesis. It explains how **Retrieval-Augmented Generation (RAG)** works, why it helps reduce hallucinations in **Large Language Models (LLMs)**, and how a more modular setup using **LangGraph** can make it more flexible and easier to control. The focus is on building a system that is both practical and transparent, especially for tasks that require grounded, reliable answers, such as ethical reasoning.

1.1. LLMs capabilities and constraints

Large Language Models (LLMs) have made significant progress in natural language understanding and generation. Models like GPT-3, FLAN-T5, and LLaMA can answer complex questions, generate coherent narratives, and even perform basic reasoning.

Despite these advances, LLMs still face important limitations. They can hallucinate facts, produce outdated information, and lack transparency in their internal processes. These challenges are especially critical when accuracy and traceability are required, as in tasks involving ethical or factual reasoning. Addressing these issues requires augmenting LLMs with mechanisms that anchor their outputs to verifiable sources.

Model Selection: Why FLAN-T5

This work uses **FLAN-T5-Large**, an instruction-tuned encoder-decoder model released by Google. FLAN-T5 (Fine-tuned LAnguage Net) builds on the original T5 architecture but is further trained to better follow natural-language instructions. This makes it especially helpful for reasoning tasks involving unfamiliar or open-ended queries. It is a critical need in this project when dealing with ethical questions and nuanced prompts. Instruction-tuned models such as FLAN-T5 have shown improvements in following user prompts with reduced hallucination rates [3].

The choice of FLAN-T5-Large is motivated by both technical suitability and practical constraints:

- **Resource Efficiency:** Unlike LLaMA2 or OpenAI GPT-4, FLAN-T5-Large can be run locally on consumer-grade hardware. This project was developed on a MacBook Air M1 with 8 GB of RAM, which cannot accommodate heavier models.
- **Hardware-Constrained Optimization:** Among the available FLAN-T5 variants (Small, Base, Large, XL, XXL), the **Large** version offered the best trade-off between performance and memory usage. Larger models (XL, XXL) exceeded the capacity of the available system, while smaller models did not provide sufficient generation quality for reasoning tasks.

- **Open Access and Instruction Tuning:** FLAN-T5 is freely available via Hugging Face and supports instruction-based prompting, which is essential for agent workflows involving tasks like rewriting, classification, and summarization.
- **Text-to-Text Format:** Being a sequence-to-sequence model, FLAN-T5 natively supports flexible input-output structures, making it a good fit for multi-agent orchestration and prompt-driven control logic.

1.2. RAG Structure and Variants

RAG adds a retrieval step before generating an answer. Instead of depending only on what the model has learned during training, it searches for relevant documents and uses them to guide the response.

A typical RAG pipeline includes the following steps:

- **Query Encoding**: Turning the input question into a dense vector (embedding).
- **Retrieval**: Searching a document index to find the most relevant passages.
- **Context Injection**: Adding the retrieved content into the prompt sent to the model.
- **Answer Generation**: Producing a response based on both the query and the context.

This setup helps ground the output in real content but usually follows a fixed path. If the retrieval step fails or the question is vague, the system doesn't adjust, making it less effective in those situations.

1.3. Technical limitations of RAG

While RAG offers significant benefits in flexibility and factual grounding, real-world deployment comes with practical trade-offs:

- **Outdated or noisy documents** : When the index lags behind or contains clutter (ie off topic text), the answers lose reliability.
- **Irrelevant passages retrieved** : The search step can pull text that looks similar to the question but does not really answer it.
- **Extra processing adds delay** : Each extra part of the pipeline adds a bit of wait time.
- **Multiple services to maintain** : The vector index, the language model, and the controller all need to be set up and watched.

- **Frequent embedding refresh** : New data or better embeddings mean rebuilding the index.
- **Prompt length and cost** : Each module adds tokens; long prompts increase running costs and can exceed the model's input limits.
- **Difficult quality measurement** : Extra checks are needed to see if the answer really uses the right sources.

1.4. Embedding models and vector similarity

Retrieval effectiveness depends heavily on how queries and documents are represented. Embedding models like `all-MiniLM-L6-v2` map texts into high-dimensional vector spaces where semantic similarity can be measured. Dense vector embeddings like Sentence-BERT have demonstrated strong performance in semantic search tasks [4], especially when combined with similarity search libraries such as FAISS [5].

In addition to dense sentence embeddings, a sparse baseline using TF-IDF (*Term Frequency-Inverse Document Frequency*) was implemented. Unlike neural embeddings, TF-IDF represents texts as high-dimensional sparse vectors based on word frequency and inverse document frequency statistics. This method enables efficient, interpretable similarity computation via classic cosine distance on sparse vectors.

Despite its simplicity, TF-IDF provides a useful point of comparison to evaluate how well complex embedding models (like Sentence-BERT) enhance retrieval and answer quality in RAG workflows.

The similarity between a query embedding x and a document embedding y is typically calculated using cosine similarity:

$$\text{sim}(x, y) = \frac{x \cdot y}{||x||||y||}$$

Higher similarity scores indicate stronger relevance, guiding which documents are selected to enrich the prompt. The embedding pipelines and vector indexing setup (TF-IDF, FAISS) are described in more detail in Appendix~A.2.

1.5. Modular pipelines

While standard RAG pipelines improve factual accuracy, they often treat retrieval and generation as a single, inflexible process. If retrieval fails or returns poor quality documents, the system typically proceeds anyway, leading to hallucinated or incomplete answers.

Modular pipelines address this by separating retrieval, reranking, query rewriting, and generation into distinct, interchangeable components. This design enables conditional execution based on intermediate outcomes, offering greater flexibility.

Agent-based systems take modularity a step further by introducing decision-making into the pipeline. Instead of running every step blindly, these systems include dedicated agents that check how well the retrieval went, decide whether it makes sense to rerank or rephrase the query, and review the output before finalizing a response. This kind of adaptive control makes the system more reliable and transparent, especially in situations that involve ambiguity or ethical sensitivity.

1.6. LangChain and LangGraph

Modern RAG system typically involves chaining together several components such as embedding, retrieval, reranking, and answer generation. To manage this complexity efficiently, two open-source tools were adopted: **LangChain** and **LangGraph**.

1.6.1. LangChain: LLM oriented workflow framework

LangChain is a modular Python framework that facilitates building applications powered by language models. It enables the creation of pipelines (or "chains") by connecting components like:

- Prompt templates and memory management
- Retrieval systems (e.g. FAISS, Redis, Pinecone)
- External tool interfaces (e.g. APIs, calculators, file readers)
- Language models and output parsers.

LangChain excels in prototyping and managing complex multi-step LLM workflows, offering high level abstractions for chains, memory management, and tool integration.

LangChain is used to orchestrate retrieval, prompting, and generation steps, providing a unified interface to manage components such as FLAN-T5, Sentence-BERT, TF-IDF retrievers, and Wikipedia chunks. Its composable architecture enables dynamic control flows suited for modular RAG design [6].

1.6.2. LangGraph: Graph based control over agents

LangGraph extends LangChain by introducing an explicit **graph-based control flow**, where:

- **Nodes** represent distinct modules such as retrieval, reranking, rewriting, or generation.
- **Edges** define transitions between nodes.
- **Conditional branches** allow dynamic routing based on agent decisions. See Appendix A.1 for a visual representation of the LangGraph pipeline.

Unlike traditional RAG pipelines that execute in a linear, fixed order, LangGraph enables dynamic branching and conditional logic. For instance, if initial retrieval quality is low, an agent can trigger a query rewrite instead of proceeding directly to generation.

The LangGraph framework was selected due to its flexibility for defining node-based logic with clear decision paths and fallback mechanisms, making it well-suited for agentic RAG systems [7]. This flexible setup allows the system to better handle tricky or unclear questions. It doesn't just push ahead blindly, it can pause, reconsider the input, and try different paths if needed. The graph structure also makes it easier to trace how each answer was formed, which helps with debugging and understanding the system's behavior.

1.6.3. Complementary Usage

While LangChain provides the foundation for composing and executing LLM-based workflows, LangGraph introduces a layer of intelligent control between components. In this work, LangChain is used for:

- Loading and managing the FLAN-T5 model,
- Formatting prompts and orchestrating inputs and outputs,
- Structuring the overall interaction between queries and responses.

LangGraph builds on top of this structure by defining graph-based pipelines that include conditional logic. Specifically, it is used to design the 4-node and 5-node agent workflows that support:

- Query rewriting in response to weak retrievals,
- Reranking of retrieved documents before generation,
- Optional grading of generated outputs to improve reliability.

Together, LangChain and LangGraph enable a modular and adaptable architecture, making it easier to monitor, debug and improve the system's behavior. Especially in complex or sensitive use cases.

2. SYSTEM ARCHITECTURE

This chapter presents the internal design of the agent-based Retrieval-Augmented Generation (RAG) system developed using the LangGraph framework. While Chapter~4 covers the dataset and evaluation methodology, this section focuses on the modular agents, their individual roles, and the routing logic that governs their interactions.

2.1. RAG vs. Fine-tuning and Prompt engineering

RAG offers a middle ground between the other two techniques. Unlike fine-tuning, which modifies the model's internal parameters through supervised learning, RAG retrieves external knowledge at inference time without altering the model. Compared to prompt engineering, RAG provides more robustness when dealing with vague or underspecified inputs, and allows systems to incorporate real-time information.

Given the nature of this work, focused on general knowledge and ethical reasoning, RAG was the most suitable option. Fine-tuning was not feasible due to limited computational resources and the general purpose nature of the dataset. Prompt engineering alone was also insufficient, as it lacks support for fallback mechanisms, external grounding, and modular agent behavior. RAG made it possible to design a dynamic system that retrieves documents, assesses context quality, reformulates queries when needed, and traces each step of the decision process.

Each approach has its strengths and limitations, as summarized in the table below :

Table 2.1. COMPARISON BETWEEN TECHNIQUES

Technique	Strengths	Limitations	Best Use Cases
RAG	Accesses external knowledge dynamically, supports modularity, improves factual accuracy, and enables interpretability	Requires a retriever setup, introduces latency, and adds system complexity	Factual QA, real-time updates, and domains with sparse training data
Fine-Tuning	Achieves strong performance on target tasks, produces compact models at inference time, and offers consistent behavior	Demands high computational cost for training, lacks adaptability to new knowledge, and is difficult to update	Closed-domain tasks, custom generation, and classification with labeled data
Prompt Engineering	Allows fast iteration without training, easy to apply across tasks, and remains flexible for small-scale usage	Often fragile to query phrasing, offers limited control, and may suffer from hallucinations	Quick prototypes, formatting, and small, well-defined generation tasks

2.2. Agent roles and purposes

To increase adaptability and control, the system is designed as a sequence of five modular agents. Each one is implemented as a node in LangGraph and performs a distinct function within the pipeline.

- **Retrieval Agent:** Encodes the input query using either Sentence-BERT or TF-IDF. It retrieves the top-K document chunks based on cosine similarity. This agent is responsible for surfacing potentially relevant content to support answer generation.
- **Grading Agent:** Assesses the quality of retrieved documents using two metrics: *Mean cosine similarity* and *Context Precision@10*. If either score falls below the 75th percentile of observed similarity scores (approximately 0.33), the query is passed to the Rewrite agent. Its implementation follows a straightforward rule: if the average similarity score remains below the threshold or the maximum number of iterations is reached, the system redirects accordingly. The complete logic is available in Appendix A.2.5.
- **Rewrite Agent:** Reformulates vague or underspecified queries to improve specificity and clarity. It uses a lightweight keyword-matching logic ("moral" becomes "moral philosophy") to guide retrieval toward richer content. The rewriting process is limited to one cycle per query for efficiency. A ReAct-style prompting structure [8] is used to encourage reasoning before rewriting. The code is shown in Appendix A.2.4.
- **Rerank Agent:** Ranks the retrieved documents again using refined similarity scores. This ensures that the most relevant passages appear at the top of the constructed prompt.
- **Generation Agent:** Combines the ranked passages with the original query to construct the final prompt. It then uses the FLAN-T5 model to generate a grounded response. The ReAct prompting structure is also used at this stage to support interpretability and reasoning.

This agent-based structure allows the system to adapt dynamically to various query types, making it more robust, interpretable, and modular.

2.3. Routing logic and node behavior

Each agent in the LangGraph workflow operates independently but can trigger or skip other agents based on the intermediate results. This makes the flow conditional, adaptive, and resilient to weak retrievals or ambiguous queries.

- **Context Evaluation:** After retrieval, the grading agent calculates the mean cosine similarity and CP@10. If the score falls below a set threshold (0.33 for SBERT, 0.2 for TF-IDF), it sends the query to the rewrite node.
- **Rewrite and Rerank:** If rewriting improves the retrieval quality, the rerank node reshuffles the retrieved passages to enhance prompt quality. If results remain weak, the pipeline proceeds cautiously using fallback logic.
- **Fallback Behavior:** If no passage meets the quality criteria after all processing steps, the system generates a fallback message such as "not clear" or "the context does not take a clear position," avoiding hallucinations.
- **Traceability:** All node decisions, scores, thresholds, and rewrites are logged to support downstream evaluation and system transparency.

This routing mechanism enables the system to skip unnecessary steps or retry earlier ones, enhancing robustness and flexibility without compromising performance.

2.4. Practical benefits of the system

The five-agent setup was chosen to offer a good balance between flexibility and control. By assigning each task; retrieval, grading, rewriting, reranking, and generation to its own agent, the system becomes easier to understand, test, and upgrade. For example, it's simple to add a new fact-checking step or replace the embedding model without changing the rest of the pipeline.

This modular structure also follows common best practices in data engineering: keeping components separate, routing adaptively based on context, and logging each decision for full traceability. This is especially useful for complex or vague questions, where the system may need to pause, revise, or choose not to answer at all.

The LangGraph design enables the system to handle a wide range of queries with clarity and reliability. Visual diagrams and alternative versions of the pipeline can be found in Appendix~A.1.

2.5. Reproducibility and implementation notes

The system was implemented in Python using LangGraph, Hugging Face Transformers, and FAISS for semantic retrieval. Sentence-BERT was used for dense embeddings, while FLAN-T5 handled generation. Key modules were organized to reflect the five-agent structure.

All code was reproducible with fixed random seeds. Core logic and example configurations are provided in Appendix~A.2 and Appendix~A.1.

2.6. Walkthrough of an Ethical query resolution

To illustrate how the agent-based RAG system handles ambiguous queries, consider the input:

> *"Is it ethical to use facial recognition in public spaces?"*

At first, the **Retrieval agent** pulls ten passages containing broad discussions on surveillance, technology, and legal frameworks. These passages include definitions of facial recognition, privacy policies, and performance audits, but lack strong ethical reasoning.

The **Grading agent** computes the quality of the retrieved content using two metrics:

- CP@10 = 0.20 (only 2 out of 10 passages were relevant)
- Cosine similarity = 0.58

Since the CP@10 score falls below the system threshold (0.33), the **Rewrite agent** is activated. It appends targeted terms to the original query, such as "moral justification", "privacy concerns" or "GDPR" (General Data Protection Regulation), to guide retrieval toward ethically rich content.

The new query becomes:

```
Is it ethical to use facial recognition in public spaces?
moral philosophy GDPR
```

With this refined query, the **Retrieval agent** surfaces documents focused on fairness, surveillance ethics, and the right to privacy. The **Rerank agent** sorts these based on semantic relevance and domain alignment.

Finally, the **Generation agent** synthesizes a concise summary. The system produces:

> *"Using facial recognition in public spaces raises ethical concerns related to privacy, fairness, and consent. While some legal frameworks like the GDPR provide safeguards, opinions vary on whether the benefits outweigh the risks. Context, transparency, and accountability are key to assessing its legitimacy."*

This answer directly reflects the reranked content and avoids unsupported claims. The entire process completes in roughly 120 ms.

3. EXPERIMENTAL METHODOLOGY

The methodological components were chosen to support the main research objective: evaluating whether a modular, agent-based RAG system improves reliability and adaptability compared to a static baseline. The retrieval setup enables comparison between dense and sparse embeddings. The agent-based flow allows for testing modular behavior, fallback handling, and explainability. The evaluation design reflects the challenges posed by factual and ethical queries.

3.1. Tools and libraries used

Before diving into the system architecture, it is important to describe the core Python libraries that supported the implementation of the retrieval and generation pipelines.

- **nltk** (*Natural Language Toolkit*): Used for sentence-level segmentation of documents. This ensures that document chunks preserve semantic and grammatical coherence, improving retrieval and generation quality.
- **sentence-transformers**: Provides access to pretrained Sentence-BERT models, including `all-MiniLM-L6-v2`, used to generate dense vector representations of text for semantic similarity-based retrieval.
- **faiss** (Facebook AI Similarity Search): Enables fast nearest-neighbor search on dense embeddings. It is used to construct and query an efficient index of document vectors based on cosine similarity.
- **sklearn** (*scikit-learn*): Used for computing TF-IDF vectors and pairwise cosine similarity scores between queries and documents. It also supports various evaluation utilities and data transformations.
- **transformers** (Hugging Face): Provides access to the FLAN-T5 model used for generation. This library enables seamless loading and inference with instruction-tuned models.
- **langchain** and **langgraph**: Core libraries for building and executing both static and agent-based RAG pipelines. LangChain handles model orchestration, retrieval chains, and Input/Output formatting. LangGraph introduces node-based control and decision-making logic.
- **pandas** and **numpy**: Used for data management, tabulation, and computation of evaluation metrics across different systems.

These tools formed the backbone of the pipeline implementation, allowing for modular experimentation with embedding strategies, retrieval quality, and agent workflows.

3.2. Dataset description

The main dataset used for evaluation is a **filtered subset** of English Wikipedia articles retrieved from Hugging Face (`wikimedia/wikipedia`, snapshot dated 2023-11-01). Instead of randomly sampling the corpus, we performed keyword-based filtering to extract articles that are relevant to our two primary domains of interest: factual science and ethical reasoning.

A set of **1 000 Wikipedia articles** was selected using keyword matching across the article title and body. These keywords included terms related to geography ("japan", "capital"), biology ("mitochondria", "respiration"), and moral philosophy ("justice", "euthanasia", "morality"). Articles shorter than 300 characters were excluded to ensure sufficient content for retrieval.

Each article was segmented into semantically coherent chunks of approximately 200 words using sentence-level segmentation. This approach ensures that the chunks preserve natural linguistic boundaries, improving retrieval quality.

In earlier exploratory experiments, we also incorporated the **Hendrycks Ethics** dataset, which includes self-contained ethical dilemmas. While useful for probing LLM reasoning skills, its short format limited the opportunity for document retrieval. Therefore, we focused primarily on the Wikipedia subset, which better supports evaluation of multi-stage RAG pipelines involving document search, reranking, and answer generation.

3.3. Chunking and embedding strategy

This section presents the dual-vectorization strategy used for document encoding: Sentence-BERT for dense embeddings and TF-IDF for sparse embeddings.

Both dense and sparse embeddings are used for retrieval in this project. Sentence-BERT vectors are stored in a FAISS index for efficient similarity search, while TF-IDF vectors are compared using direct cosine similarity.

The following subsections outline the preprocessing, chunking, and vectorization strategies used in both the static and agent-based RAG pipelines.

3.3.1. Sentence-BERT embedding

Wikipedia articles are split into chunks of ~100 tokens using sentence-level segmentation to preserve semantic coherence. Each chunk is embedded using the `all-MiniLM-L6-v2` Sentence-BERT model, projecting it into a 384-dimensional space.

These vectors are stored in a FAISS index for approximate nearest-neighbor search.

```
from sentence_transformers import SentenceTransformer

embedder = SentenceTransformer("all-MiniLM-L6-v2")
embeddings = embedder.encode(chunks)
```

3.3.2. TF-IDF Embedding (Baseline)

To establish a lightweight baseline, the system incorporates a TF-IDF (Term Frequency–Inverse Document Frequency) representation. Each document is encoded as a sparse vector based on term frequency within a document and inverse document frequency across the corpus:

$$\text{TFIDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

This method offers a fast and interpretable retrieval mechanism without relying on pretrained models. It proves effective for fact-based queries but tends to fall short in tasks that require contextual nuance or semantic depth.

In practice, TF-IDF retrieval often failed to retrieve meaningful context for ethically sensitive or ambiguous questions. This limitation frequently led to fallback responses such as "not clear," reflecting the sparse or weak quality of the retrieved documents in these scenarios.

3.3.3. FAISS Indexing for Dense Embeddings

To support semantic retrieval, document chunks are encoded using Sentence-BERT into 384-dimensional dense vectors and indexed with FAISS, which enables fast nearest-neighbor search via inner product similarity. At query time, the input is embedded and compared to the index, approximating cosine similarity through normalized inner products to retrieve the top-K most relevant documents.

To filter out weak matches and reduce hallucinations, a similarity threshold is applied. Instead of using a fixed value, as in prior work like AutoRAG [1], this system adopts a data-driven approach by setting the threshold at the 75th percentile of cosine similarities between queries and their reference contexts. This adaptive threshold (around $\theta \approx 0.33$) aligns better with the distribution of similarity scores in the dataset and helps preserve relevant content that might fall below arbitrary cutoffs.

The combination of dense embeddings, approximate similarity search, and percentile-based filtering ensures that only meaningful context is included in the prompt. Implementation details are available in Appendix~A.2.

3.4. Query scenarios

The evaluation was conducted using a set of five curated prompts designed to represent two broad categories of queries. :

- **Fact-Based Queries:** These are direct, information-seeking questions that can be answered using factual content from Wikipedia. The following two were used:
  - *What is the capital city of Japan?*
  - *What is the function of mitochondria in a cell?*
- **Ethical or Ambiguous Queries:** These prompts require subjective reasoning or are underspecified, making them well suited for evaluating rewriting, reranking, and generation quality in agent-based pipelines. The following three were used:
  - *Can honesty sometimes cause harm?*
  - *What are common arguments for or against euthanasia?*
  - *Is it morally acceptable to take something if you believe it's justified?*

These prompts were manually chosen to reflect common and varied situations where RAG systems are often used. The factual questions help test whether the system can retrieve relevant information and give clear answers. The ethical questions are more open and harder to define, which makes them useful for testing how the system deals with uncertainty, reformulates queries, or avoids incorrect answers. Together, they allow us to observe how each version of the system performs in both simple and more complex cases.

3.5. Baseline RAG pipeline

Before evaluating with retrieval and generation metrics, we construct a simple baseline system where the query is embedded, semantically matched against the dataset using FAISS and cosine similarity, and then passed to FLAN-T5 for final answer generation.

This pipeline serves as a reference point for comparing with the more flexible agent-based RAG architecture. It highlights how fixed retrieval logic performs in isolation without reranking, rewriting, or validation stages.

The output above illustrates the static RAG pipeline's ability to retrieve relevant Wikipedia passages and generate concise, fact-based answers. Each query is matched with a short, informative response constructed using top-ranked contextual chunks from the document index.

While the system generally returns coherent and plausible answers, the quality and specificity vary depending on the query type, especially for open-ended or underspecified

prompts. This motivates a deeper evaluation using automatic metrics such as ROUGE-L, METEOR, cosine similarity, and Context Precision@K, presented in the next section.

Both SBERT and TF-IDF were also tested within the agent-based LangGraph framework. This enabled direct comparison between static pipelines and multi-agent workflows using the same embedding backbone, revealing the impact of modular control and query rewriting on answer quality.

3.6. Metric design

To assess the quality of the system, we divide the evaluation into two levels: (1) retrieval performance, and (2) generation quality. This section introduces the metrics used at each stage, alongside Python code for implementation.

3.6.1. Retrieval quality metrics

Cosine similarity

Cosine similarity measures how semantically aligned a retrieved document d_i is with a query q, based on the angle between their vector embeddings:

$$\text{CosineSimilarity}(q, d_i) = \frac{q \cdot d_i}{|q||d_i|}$$

We report two values per query:

- **Top-1 Cosine Similarity:** Between the query and the highest-ranked retrieved passage.
- **Mean Cosine Similarity:** The average over the top $K = 10$ retrieved passages, reflecting overall retrieval quality.

Cosine similarity helps quantify how well the context aligns with the query. It remains one of the most widely used metrics in vector space models and information retrieval [9].

Context precision at K (CP@K)

CP@K computes the proportion of the top-K retrieved passages that exceed a similarity threshold θ with the query:

$$\text{CP@}K = \frac{1}{K} \sum_{i=1}^{K} \mathbf{1}\left[\text{sim}(q, d_i) \geq \theta\right]$$

- We use $K = 10$ and refer to the metric as CP@10 (abbreviated to CP10).
- Thresholds are set to $\theta = 0.33$ for SBERT and $\theta = 0.2$ for TF-IDF, based on observed distributions. This design follows the CP@K formulation proposed in AutoRAG [1], where CP@10 serves as a lightweight proxy for semantic match coverage (i.e how many retrieved passages are likely to be relevant enough to support accurate generation).

CP10 complements cosine similarity by counting how many retrieved chunks are "good enough," offering a broader view of retrieval coverage.

3.6.2. Generation quality metrics

ROUGE-L

ROUGE-L measures the similarity between a generated answer G and a reference R by identifying their longest common subsequence (LCS). Unlike simple word overlap, LCS captures in-sequence matches :

$$\text{ROUGE-L} = \frac{\text{LCS}(G, R)}{|R|}$$

- LCS is the length of the longest sequence of words appearing in both G and R, in order.
- The score emphasizes recall by comparing LCS length to the reference length |R|.
- It is especially useful for evaluating extractive and partially ordered responses.

In our implementation, we use the rouge_scorer from the rouge-score Python library with stemming enabled:

```
scorer = rouge_scorer.RougeScorer(['rougeL'], use_stemmer=True)
scorer.score(reference, generated)['rougeL'].fmeasure
```

This method captures how well the model preserves key phrases and order. The metric was introduced by Lin [10].

METEOR

METEOR (Metric for Evaluation of Translation with Explicit ORdering) extends ROUGE by going beyond exact word matches. It accounts for stemming, synonyms, and word order, making it more sensitive to semantic similarity between a generated answer G and a reference R.

The score is based on a harmonic mean of unigram precision and recall, adjusted by a penalty that reflects fragmented or disordered matches:

$$\text{METEOR} = F_{\text{mean}} \cdot (1 - P)$$

- $F_{\text{mean}} = \frac{10 \cdot P \cdot R}{R + 9P}$, with higher weight on recall.
- P stands for precision: the percentage of words in the generated answer that match the reference.
- R stands for recall: the percentage of words from the reference that appear in the generated answer.
- The penalty increases when the matching words are scattered or out of order, lowering the score for less fluent or coherent answers.

We use the NLTK implementation, which includes WordNet synonym matching:

```
meteor_score([word_tokenize(reference)], word_tokenize(generated))
```

METEOR is ideal for instruction-tuned LLMs that generate semantically correct but lexically varied outputs [11].

3.7. Fallback detection

In addition to retrieval and generation metrics, we track fallback behavior to assess system robustness. A fallback is logged when the model provides a vague or evasive answer such as: *"I don't know." "No clear answer." "The context does not take a clear position."*.

These are detected with a rule-based filter scanning for known fallback phrases. Fallback tracking helps evaluate system robustness and error-handling behavior in open-ended tasks.

3.8. Module optimization

Some recent work [1] has proposed more advanced tools than metrics used :

- **GEval (Grounded Evaluation):** Uses entailment checks to confirm whether generated answers are supported by retrieved context.
- **NLI-Based Verifiers:** Determine whether the answer is entailed, contradicted, or unrelated to the source.

Although, they were not implemented, due to integration complexity and runtime cost, these tools likely offer improvements for factual verification in modular RAG systems.

4. DATA ANALYSIS AND EVALUATION

This chapter presents the evaluation of the 5-node agent-based architecture introduced in Chapter~2. Our aim is to assess how each module contributes to retrieval quality, generation accuracy, and overall robustness. We compare it against static baselines to highlight the added value of modular decision-making and reasoning-based prompting.

4.1. Retrieval & Generation evaluation

We evaluate the performance of both static and agent-based pipelines. Our goal is to measure how effectively each system retrieves relevant contexts and generates accurate responses. We compare four configurations : two static (SBERT and TF-IDF) and two agent-based (LangGraph SBERT and TF-IDF), using key metrics.

The evaluation design was inspired by AutoRAG, which provides a structured methodology for assessing the contribution of each RAG component in isolation [1]. In addition, our system incorporates ideas from the ReAct framework [8], which emphasizes combining reasoning and acting steps during generation. Specifically, ReAct-style prompting was applied in the generation and rewrite nodes to encourage more explicit reasoning traces and improve handling of ambiguous queries.

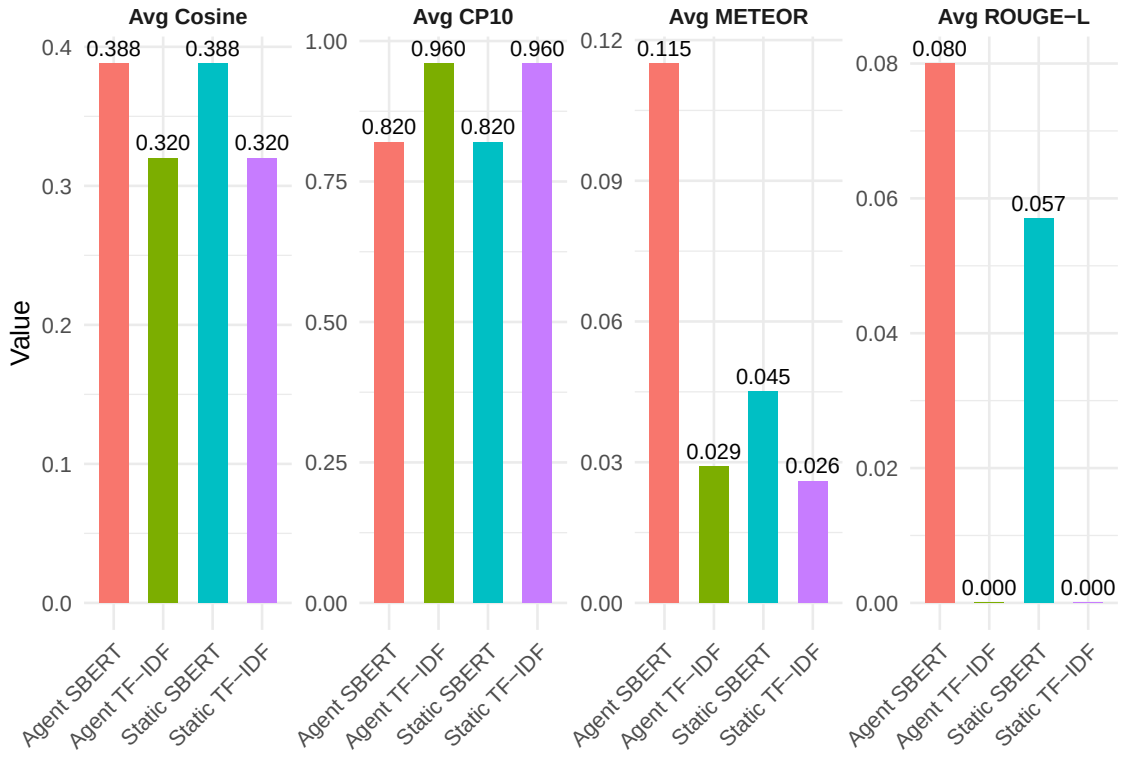


Fig. 4.1. Static vs. Agent Pipelines: Metric-wise Comparison

Agent SBERT performs best overall. It achieves the highest METEOR score (0.115), meaning its answers are more semantically similar to the reference responses. It also scores highest on ROUGE-L (0.08), which measures how closely the answer matches the wording and structure of the reference. Although both Agent and Static SBERT have the same cosine similarity score (0.388), the agent version produces better answers. This is likely because it uses extra steps like rewriting and reranking to make the most of the retrieved content.

On the other hand, both TF-IDF pipelines perform poorly in generating answers. Their METEOR and ROUGE-L scores are close to zero, even when used with agents. This shows that while TF-IDF can retrieve text that appears similar to the query, it doesn't help the model write good answers. In fact, both TF-IDF systems reach a CP10 score of 0.96, meaning they retrieve context that passes a basic similarity check, but this context is not helpful for writing meaningful responses.

This suggests that just retrieving relevant-looking text is not enough. High quality generation needs deeper understanding, which dense embeddings like SBERT can provide. When paired with agent steps that help rephrase and organize the information, the system becomes much better at answering complex or vague questions.

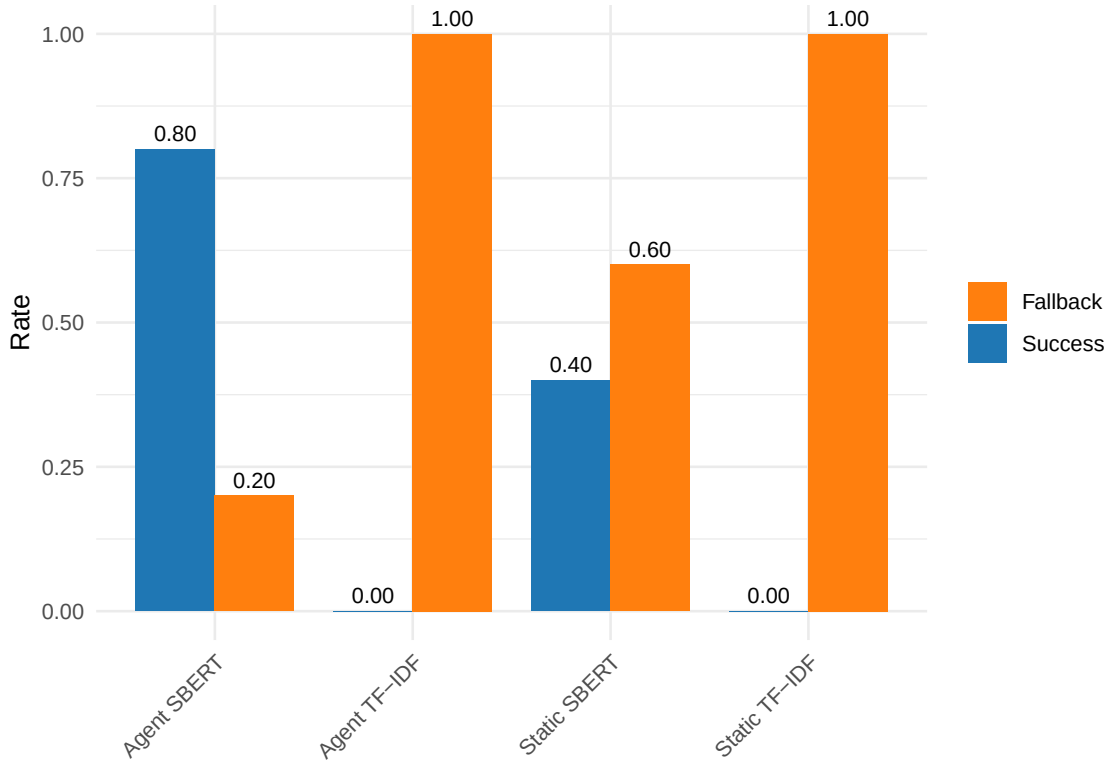


Fig. 4.2. Success vs. Fallback Rates by System

Fig. 4.2 shows a contrast in success and fallback behavior across different retrieval setups. Agent SBERT stands out with a success rate of 80% and a fallback rate of just 20%. Compared to Static SBERT, which falls back 60% of the time and succeeds only

40% of the time, this represents a significant improvement. The agent-based version handles uncertainty better, likely thanks to its dynamic use of query rewriting and reranking when retrieval is weak or ambiguous.

Meanwhile, both TF-IDF systems static and agent-based completely fail to generate substantive answers. They exhibit a 100% fallback rate and a 0% success rate, which highlights the limitations of sparse retrieval techniques. Even when embedded within an agent pipeline, TF-IDF lacks the semantic depth to provide meaningful grounding for generation.

This result highlights that dense embeddings like SBERT, when paired with a modular agent design, are far more effective at navigating complex prompts and ensuring more complete responses. Sparse methods fall short when higher-level reasoning or nuanced context is needed.

4.2. Agent contributions and ablation analysis

We explore the individual impact of each component within the agent-based LangGraph pipeline through systematic ablation. By selectively disabling the Grade, Rewrite, and Rerank nodes, we evaluate how each component contributes to the system's retrieval and generation capabilities. The use of ReAct-style prompts in the Rewrite and Generation nodes may also play a role in this robustness, particularly when rewriting vague user inputs or synthesizing grounded responses.

Table 4.1. ABLATION STUDY: ALL EVALUATION METRICS

System	Fallback Rate	Avg ROUGE-L	Avg METEOR	Avg Cosine	Avg CP10	Substantive Answers
Static SBERT	0.6	0.057	0.045	0.388	0.82	2
Agent (5-node)	0.2	0.080	0.115	0.388	0.82	4
Static TF-IDF	1.0	0.000	0.026	0.320	0.96	0
Agent TF-IDF (5-node)	1.0	0.000	0.029	0.320	0.96	0
No Rewrite (4-node)	0.2	0.101	0.071	0.388	0.82	4
No Rerank (4-node)	0.2	0.080	0.073	0.388	0.82	4
No Grade (4-node)	0.2	0.080	0.073	0.388	0.82	4
No Gr. & Rewrite (3-node)	0.2	0.080	0.115	0.388	0.82	4

Through the table, several insights can be drawn when comparing different pipeline configurations :

We see that all SBERT-based setups; whether static or agent-based, reach the same mean cosine similarity of 0.388. This tells us that semantic alignment mostly comes from the embedding model itself, not the pipeline structure. TF-IDF models perform worse here, with a lower average of 0.320, meaning they retrieve less relevant content on average.

Looking at CP10, SBERT-based pipelines reach a strong average score of 0.82, indicating that most of the retrieved passages are semantically aligned with the input query.

TF-IDF configurations score even higher at 0.96, reflecting their tendency to retrieve a broad set of passages. However, this broader coverage does not guarantee useful generation. Despite high CP10, TF-IDF pipelines fail to produce meaningful answers, highlighting the importance of retrieval quality over quantity. Semantic precision and contextual relevance remain critical for effective downstream generation.

METEOR gives us more insight into output fluency. The full 5-node SBERT agent pipeline scores highest (0.115), showing that its Rewrite, Grade, and Rerank agents help refine the answer. If we remove Rewrite or Rerank, METEOR drops, suggesting these modules play a key role in improving final output quality.

Interestingly, ROUGE-L doesn't vary much across setups. In fact, the 4-node pipeline without Rewrite slightly outperforms the full one. This could be because skipping rewriting keeps more of the original retrieved content, which helps with matching the reference text.

The fallback rate stays steady at 20% across all agent configurations, even when some nodes are removed. This consistency shows that the pipeline is fairly robust, other agents can step in when one is missing.

Finally, the minimal differences between full and reduced agent pipelines may be due to the dataset itself. Many of the queries are short and don't require much reasoning. So, even though the system has tools like rewriting and reranking, they may not have much room to make a real difference in this particular test set.

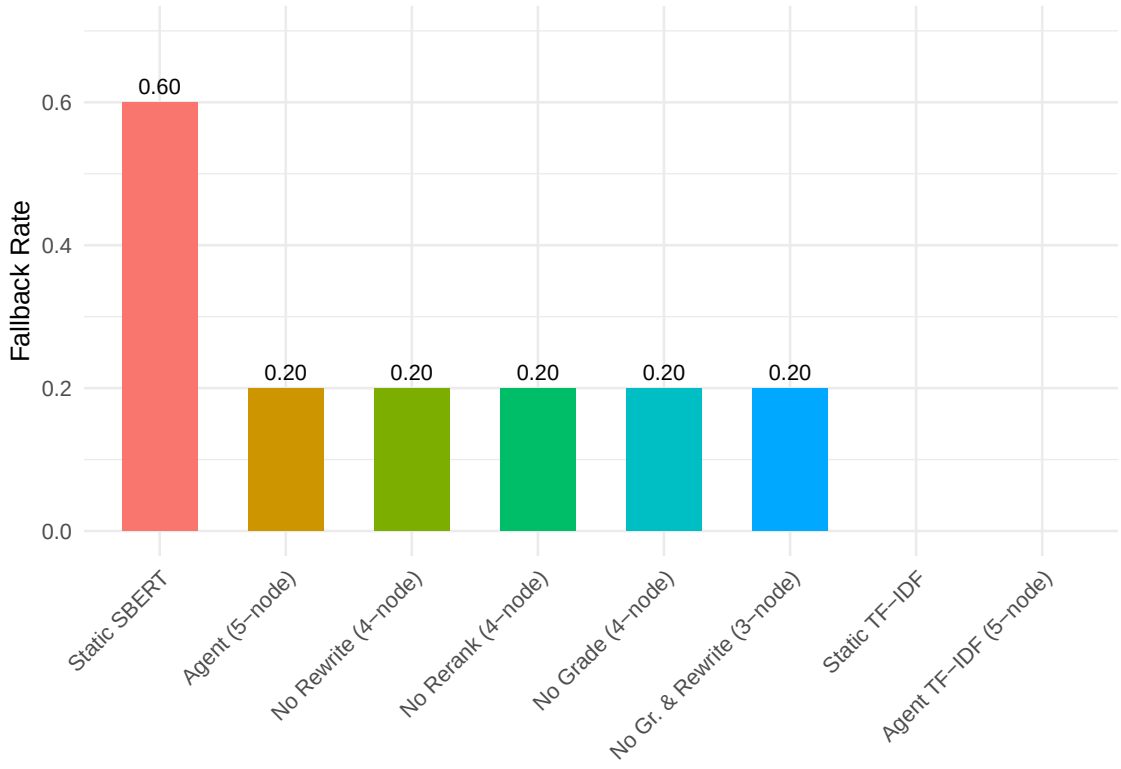


Fig. 4.3. fallback rate

Agent SBERT achieves a success rate of 80%, reducing fallback responses to just 20%, compared to 60% in the static SBERT pipeline. This difference shows how the modular architecture helps the system better manage vague, incomplete, or underspecified queries. Instead of quitting when retrieval is weak, the agentic design encourages the model to try again, either by rewriting the query or reordering the retrieved content for better alignment. These fallback-reducing steps likely improve the model's ability to produce helpful answers even when initial retrieval is not ideal.

On the other hand, TF-IDF pipelines, both static and agent-based, fail to produce any successful outputs under the same conditions. The success rate is 0%, with every attempt resulting in a fallback. This confirms the limitations of sparse retrieval methods when applied to open-ended generation tasks. TF-IDF simply does not capture the semantic nuances that are often needed for language models to generate grounded and relevant answers. Even when paired with agent logic, the embedding quality was not sufficient to support the downstream generation step.

These findings support two key conclusions. First, dense semantic embeddings like SBERT are much better suited for RAG workflows than TF-IDF, especially when the task involves vague or high-level prompts. Second, the use of modular agents, particularly rewriting, reranking, and grading, makes the system more robust and adaptive. By reducing fallback and enabling dynamic control over the generation flow, the agent-based setup proves more capable of handling uncertainty and weak context.

4.3. Error Analysis

To understand where the pipelines struggle, we classify errors and compare their distribution between Agent SBERT and Static SBERT:

- **Correct**: Answers with non-zero ROUGE-L, indicating alignment with reference responses.
- **Misretrieval**: Answers were generated, but retrieved passages lacked relevance (ROUGE-L = 0).
- **Fallback**: No output was generated due to low context quality or retrieval failure.

Fallback categorization and misretrieval analysis follow best practices outlined in recent RAG evaluation literature [2].

Table 4.2. ERROR BREAKDOWN FOR STATIC VS. AGENT SBERT

System	Correct (ROUGE > 0)	Misretrieval (ROUGE = 0)	Fallback
Agent SBERT	2	2	1
Static SBERT	1	1	3

Agent SBERT correctly answered 2 out of 5 queries, outperforming the static variant, which succeeded on only one. This suggests a measurable improvement in retrieval and generation when using an agent-based architecture. The system demonstrates a stronger capacity to provide useful responses, even under uncertain input conditions.

One of the key differences lies in fallback behavior. The agent pipeline produced only one fallback compared to three in the static setup. This indicates a more resilient design: rather than defaulting to silence when context is weak, the agent-based system attempts to move forward, increasing its chances of producing helpful output.

However, this adaptability comes with trade-offs. The number of misretrievals was higher in the agent pipeline (2 vs. 1), suggesting that while the system is more proactive, it may also generate answers from less relevant documents. This behavior reflects a deliberate choice to favor continuity and interpretability, even when input quality is uneven.

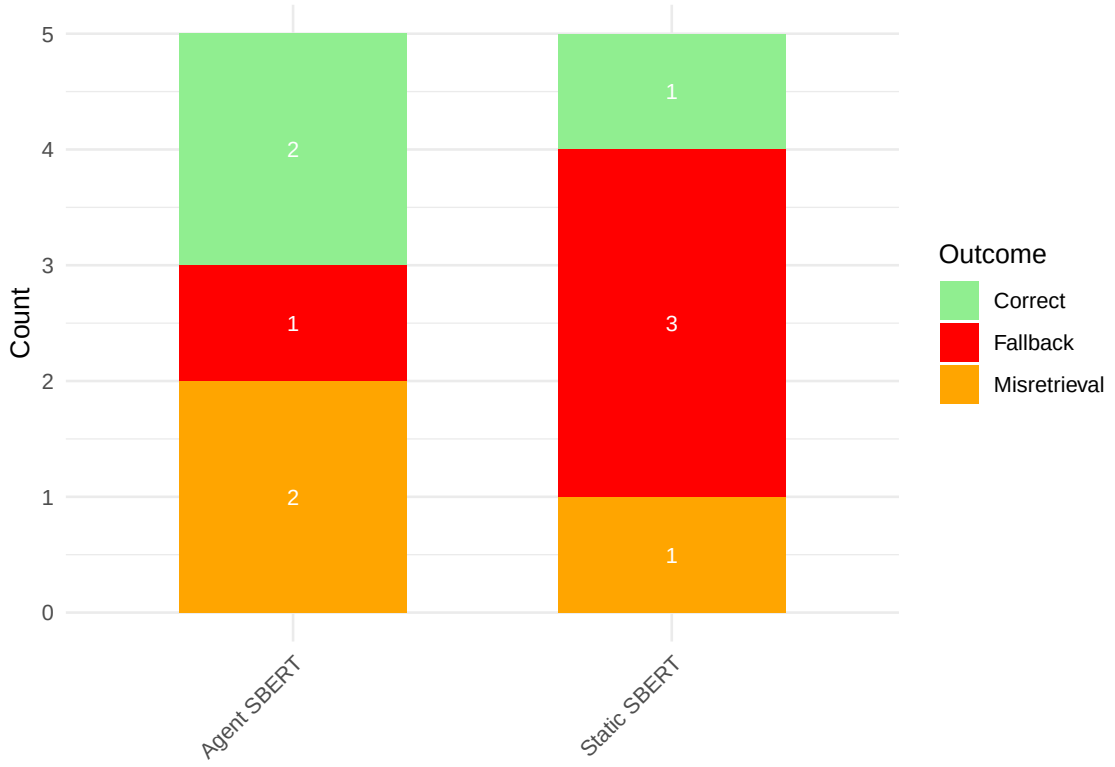


Fig. 4.4. error distribution

Figure 4.4 highlights how the two systems behave differently when dealing with uncertainty. The agent-based pipeline shows a more balanced outcome: it produced two correct answers, two misretrievals, and only one fallback. This suggests that the agent system prefers to generate an answer, even if the context isn't perfect, offering something potentially useful rather than giving up.

In comparison, the static SBERT pipeline fell back three times out of five, managing just one correct answer. This indicates that it's more conservative. If the retrieved content isn't strong enough, it tends to stop instead of attempting a response. While this avoids

incorrect generations, it also means more unanswered queries.

Interestingly, the agent version makes more mistakes through misretrievals, but it also takes more initiative. Instead of defaulting to "I don't know," it tries to work with the available context. This trade-off shows that the agent system is designed to be more flexible, minimizing fallback and favoring continuity, even at the cost of occasional errors.

4.4. Observations on agent decisions

While the architecture supports dynamic agent routing, including automatic triggering of rewriting or fallback, this behavior was not fully enabled in the current implementation. For evaluation purposes, query rewrites were applied selectively in cases where initial retrieval failed to produce meaningful results. These rewrites were crafted using ReAct-style prompting and evaluated alongside baseline outputs to assess their potential benefit.

Therefore, agent decisions in this study should be interpreted as design-driven rather than autonomously learned or executed. This choice was made to keep the evaluation focused and computationally feasible, while still showcasing how modular nodes like the Rewrite Agent could intervene in low-quality scenarios.

Future iterations of this work could enable true agent-driven routing by incorporating reinforcement learning or decision heuristics to dynamically select the best action path at inference time.

5. RESULTS AND INSIGHTS

This chapter reflects on the key findings from the experiments in Chapter~4. It explores how different embedding choices and system designs impact performance, especially in terms of fallback behavior, control over the generation process, and the overall quality of responses. The goal is to draw practical insights that can inform the deployment of more reliable and flexible RAG systems.

5.1. Embedding comparison

We investigate how embedding choice influences both retrieval precision and generation quality. We compare traditional TF-IDF embeddings with Sentence-BERT (SBERT), evaluating their performance across metrics. They reflect lexical overlap, semantic matching, and top-k retrieval alignment, respectively.

Table 5.1. EMBEDDING COMPARISON

Embedding	Avg ROUGE-L	Avg METEOR	Avg Cosine	Avg CP10
TF-IDF	0.000	0.026	0.320	0.96
SBERT	0.057	0.045	0.388	0.82

Sentence-BERT (SBERT) clearly outperforms TF-IDF across all quality metrics. It achieves higher ROUGE-L (0.057 vs. 0) and METEOR (0.045 vs. 0.028), indicating better alignment and semantic fidelity in the generated outputs.

The mean cosine similarity is also higher with SBERT (0.388 vs. 0.32), confirming its superior ability to capture meaningful semantic proximity between queries and retrieved documents.

Interestingly, TF-IDF shows a high CP10 (0.96), meaning it retrieves many documents above the similarity threshold. However, its generation quality remains low. SBERT retrieves slightly fewer documents (CP10 = 0.82) but produces stronger answers, as shown by higher ROUGE-L and METEOR scores.

These results reaffirm the importance of dense embeddings like SBERT for effective retrieval-augmented generation, particularly when dealing with nuanced or context-sensitive queries.

5.2. System architecture comparison

We compare static and agent-based RAG pipelines using both SBERT and TF-IDF embeddings across multiple metrics. The goal is to determine whether agentic modules (i.e. grading, rewriting, and reranking) offer measurable improvements in answer quality and success rates.

Table 5.2. ARCHITECTURE COMPARISON

Architecture	Embedding	ROUGE	METEOR	Cosine	CP10	Success	Fallback
Static	SBERT	0.057	0.045	0.388	0.82	0.4	0.6
Agent (5-node)	SBERT	0.080	0.115	0.388	0.82	0.8	0.2
Static	TF-IDF	0.000	0.026	0.320	0.96	0.0	1.0
Agent (5-node)	TF-IDF	0.000	0.029	0.320	0.96	0.0	1.0

The agent-based SBERT pipeline achieves better performance than its static counterpart in both METEOR (0.115 vs. 0.045) and success rate (80% vs. 40%), while preserving equivalent cosine similarity (0.388).

This improvement suggests that agentic modules contribute to higher-quality responses without degrading retrieval accuracy. The addition of Grading and Rewrite components appears to be particularly helpful in managing less precise queries, reflected in the drop in fallback rate from 60% to 20%.

Meanwhile, both TF-IDF-based systems fail to generate meaningful answers (ROUGE-L = 0), despite achieving high CP10 scores (0.96). These results reinforce the limitations of sparse embeddings: they may retrieve superficially relevant content, but lack the depth needed for robust generation.

These results demonstrate that embedding type plays a crucial role; SBERT clearly outperforms TF-IDF. But architectural improvements through modular agents also add clear value. The full 5-node LangGraph pipeline demonstrates stronger adaptability and fallback handling, particularly thanks to the Grading and Rewrite nodes. While static SBERT slightly outperforms the agent setup on ROUGE-L (0.057 vs. 0.080), the agent pipeline's higher METEOR and success rate indicate improved generation, especially on more complex or underspecified prompts.

5.3. Reflections on system behavior

Beyond the numerical metrics, the evaluation reveals meaningful differences in how each pipeline behaves when dealing with uncertainty, retrieval quality, and answer generation. These behavioral traits are critical from a deployment perspective, especially when robustness and interpretability are required in real-world applications.

The agent-based pipeline demonstrated greater robustness than static baselines, particularly when queries were vague or underspecified. Components like the Grading and Rewrite agents allowed the system to recover from weak retrievals by reformulating the query and attempting a second pass. As a result, fallback answers were reduced, and more queries resulted in meaningful completions. In contrast, the static pipelines often failed silently when initial retrievals were insufficient, highlighting a less adaptive design.

Latency naturally increases when multiple decision points are introduced, but the agent flow remained efficient in practice. Since components like rewriting or reranking are only triggered when necessary, the system avoids unnecessary steps for straightforward queries. This conditional execution ensures that the pipeline stays responsive without compromising on quality when additional processing is needed.

Finally, the modular nature of the architecture makes it both flexible and interpretable. Each agent operates independently and contributes a specific function, which allows for easy debugging and targeted improvements. For instance, a new rewriting model or a factual verifier like GEval could be added without restructuring the full system. This separation of roles also enhances traceability, as it becomes easier to track how each stage influenced the final answer.

The table below summarizes the core differences between the pipelines. It shows that while SBERT provides a better foundation than TF-IDF, the real gains come from adding intelligent control flow. LangGraph's agentic setup not only improves generation quality (as seen in METEOR and ROUGE-L) but also introduces fallback recovery, richer traceability, and better alignment with complex queries.

Table 5.3. COMPARISON OF RAG PIPELINES

Metric	**TF-IDF**	**SBERT**	**LangGraph (5-node)**
Embedding Type	Sparse	Dense	Dense + Agents
Mean Cosine Similarity	0.320	0.388	0.388
METEOR	0.026	0.045	0.115
ROUGE-L	0.000	0.057	0.080
CP10	0.96	0.82	0.82
Rewrite/Rerank	No	No	Yes
Fallbacks	Frequent	Occasional	Rare
Traceability	None	Limited	Full

CONCLUSION

This work explored whether a modular architecture based on agents could improve the reliability and flexibility of **Retrieval-Augmented Generation (RAG)** systems. By dividing the pipeline into five agents : **Retrieval**, **Grading**, **Rewrite**, **Rerank** and **Generation**, the system becomes easier to adapt, control, and understand. This approach was especially useful for queries that were vague or incomplete, where traditional RAG setups tend to produce unsupported answers.

The results confirmed the initial hypothesis: having separate steps with decision logic improves system performance in both structured and uncertain cases. The grading mechanism, using **Context Precision10** and **cosine similarity**, helped detect poor retrievals and decide when to activate rewriting. This led to better context and higher quality responses. Compared to the static version, the agent pipeline reduced fallback rates from 60% to 20%, and **Sentence-BERT** consistently outperformed **TF-IDF**. Ablation studies showed that the benefit of rewriting or reranking depended on the question, and **error analysis** confirmed that the modular flow handled difficult prompts more safely. These findings support the original goals of improving accuracy, transparency, and control.

Still, the system has limitations. The number of evaluation prompts was small, which limits the scope of the conclusions. The Rewrite Agent used fixed rules based on keywords, without any real understanding. Latency was improved with conditional routing, but no time measurements were made. The flow between agents was predefined and not adjusted during testing. External tools like **GEVal** or **SemScore** were not used, so the evaluation focused only on automatic text metrics.

Future directions could include using classifiers or basic reinforcement rules to decide when to rewrite or rerank, based on scores or past decisions. This would reduce the need for fixed thresholds and allow the system to react better to each case. It would also be interesting to add tools that check whether the generated answers are both correct and supported by the retrieved content. Finally, the architecture could be tested with dialogue tasks or more complex ethical questions to explore its potential in more demanding situations.

On a personal level, this project helped me understand the benefits and trade-offs of modular AI. It showed me that good control and clear structure can often matter more than complexity. It also reminded me how important it is to keep systems simple and reproducible, especially when working with real applications of language models.

Overall, this thesis shows that using agents in RAG pipelines offers a practical way to improve factual accuracy, trust, and clarity in generated answers, especially when questions are difficult or unclear.

A. APPENDIX

A.1. LangGraph RAG architectures

A.1.1. Static RAG baselines

a. Sentence-BERT embedding

```
query_embedding = normalize(embedder.encode([query], convert_to_numpy=True))
D, I = index.search(query_embedding.astype("float32"), k=TOP_K)
filtered_contexts = [
    documents[i]
    for i in I[0]
    if dot(query_embedding[0], doc_embeddings[i]) >= SIM_THRESHOLD
]
retrieved_context = "\n\n".join(filtered_contexts)
response = llm(prompt, max_new_tokens=150, do_sample=False)[0]['generated_text']
```

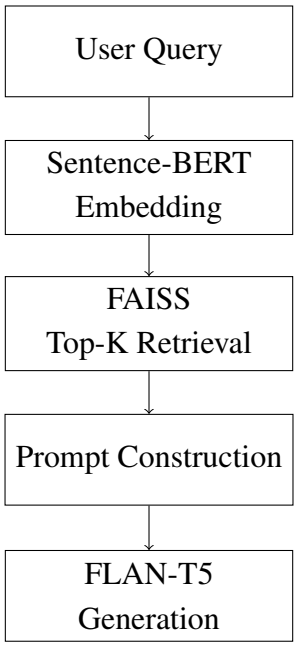


Fig. A.1. Static RAG Pipeline using Sentence-BERT and FAISS.

b. TF-IDF embedding

```
vectorizer = TfidfVectorizer()
doc_vectors = vectorizer.fit_transform(documents)

query_vec = vectorizer.transform([query])
sim_scores = cosine_similarity(query_vec, doc_vectors).flatten()
top_indices = sim_scores.argsort()[::-1][:10]
top_docs = [documents[i] for i in top_indices if sim_scores[i] >= 0.2]
```

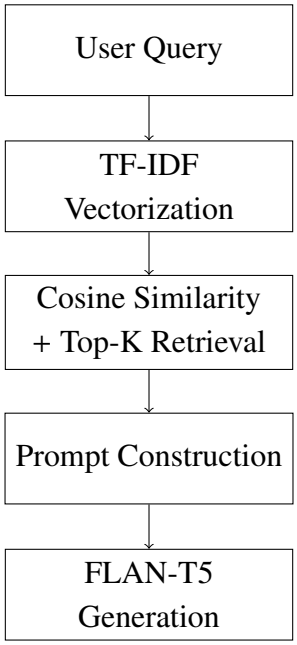


Fig. A.2. Static RAG Pipeline using TF-IDF (sparse retrieval).

A.1.2. Agent-Based architectures

The figures below present ablation variants of the 5-node agent-based RAG pipeline. Each graph disables one specific agent, while preserving the rest of the pipeline structure. Kamada-Kawai layout was applied to enhance the spatial distribution, visibility, and overall aesthetics of the network graphs.

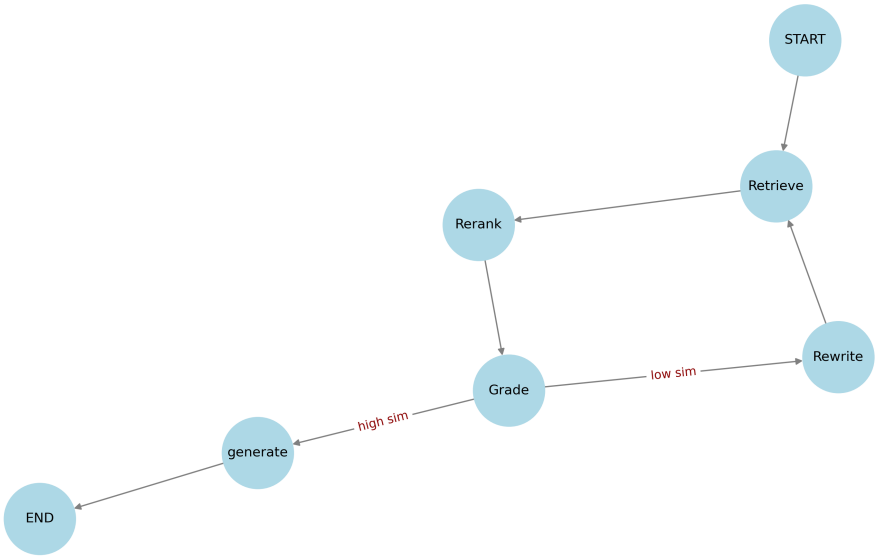
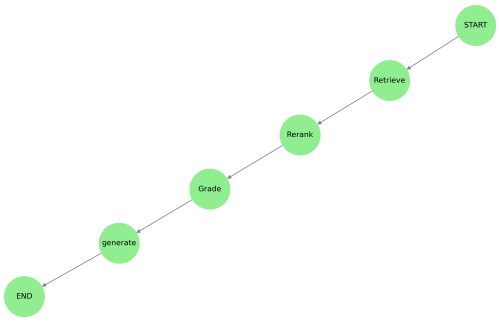
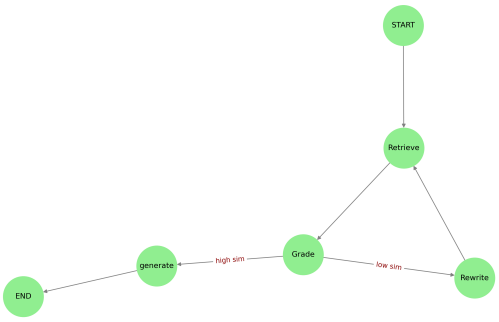


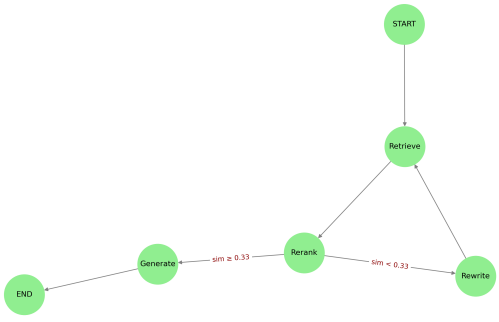
Fig. A.3. Full 5-node agent pipeline



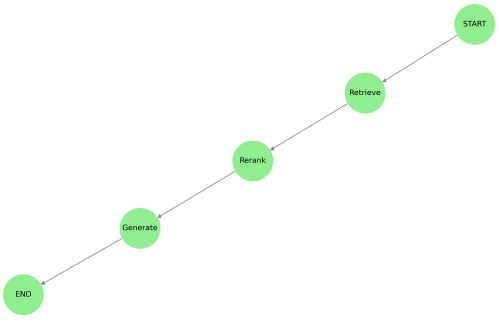
No Rewrite



No Rerank



No Grader



No Grader & No Rewrite (3-node)

Fig. A.4. Ablation variants of the 5-node agentic RAG

A.2. Key code snippets

A.2.1. FAISS Index setup

```
import faiss
import numpy as np
index = faiss.IndexFlatIP(384)  # 384 = embedding dimension
index.add(embeddings)  # np.array of shape (N, 384)
```

A.2.2. TF-IDF static retrieval

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
vectorizer = TfidfVectorizer()
doc_vectors = vectorizer.fit_transform(documents)
query_vec = vectorizer.transform([query])
scores = cosine_similarity(query_vec, doc_vectors).flatten()
top_docs = [
  documents[i]
  for i in scores.argsort()[::-1][:10]
  if scores[i] >= 0.2]
```

A.2.3. LangGraph 5-node workflow skeleton

```
graph = StateGraph(RAGState)
graph.add_node("retrieve", retrieve_node)
graph.add_node("grade", grade_node)
graph.add_node("rewrite", rewrite_node)
graph.add_node("rerank", rerank_node)
graph.add_node("generate", generate_node)
graph.set_entry_point("retrieve")
graph.add_edge("retrieve", "grade")
graph.add_edge("grade", "generate")
graph.add_conditional_edges("grade", lambda s: s["action"], {
    "rewrite_query": "rewrite",
    "generate_answer": "generate"})
graph.add_edge("rewrite", "retrieve")
workflow = graph.compile()
```

A.2.4. Rewrite agent logic

```
def rewrite_node(state):
    q = state["query"]
    keywords = {
        "japan": "Tokyo Japan", "mitochondria": "cellular biology",
        "euthanasia": "medical ethics", "moral": "moral philosophy",
        "stealing": "ethics justification"}
    for k, v in keywords.items():
        if k in q.lower():
            rewritten = f"{q} {v}"
            break
    else:
        rewritten = f"{q} clarified"
    return {"query": rewritten}
```

A.2.5. Grading agent logic

```
def grade_node(state):
    similarities = state.get("all_similarities", [])
    avg_sim = np.mean(similarities) if similarities else 0
    if state.get("iteration", 0) >= state.get("max_iterations", 3):
        return {"next_action": "generate"}
    if avg_sim >= 0.33:
        return {"next_action": "generate"}
    else:
        return {"next_action": "rewrite"}
```

A.2.6. Prompt routing for Ethical vs Factual queries

```
def ethical_prompt(ctx, q):
    return f"You are trained in ethical reasoning.\nContext:\n{ctx}\n\n" \
           f"Ethical Question: {q}\nAnswer:"
def factual_prompt(ctx, q):
    return f"You are a factual assistant.\nContext:\n{ctx}\n\n" \
           f"Question: {q}\nAnswer:"
def is_ethical(query):
    keywords = ["moral", "ethical", "should", "acceptable", "justified"]
    return any(k in query.lower() for k in keywords)
```

A.3. Declaration of Use of Generative AI

A.3.1. Ethical and Responsible Behaviour

In my interaction with Generative AI tools:

- I have submitted sensitive data with the consent of the data subjects: **NO, I have not used sensitive data**
- I have submitted copyrighted materials with permission: **NO, I have not used protected materials**
- I have submitted personal data with the consent of the data subjects: **NO, I have not used personal data**
- My use of the Generative AI tool has respected its terms of use and essential ethical principles: **YES**

I confirm that I did not share any personal, sensitive, or protected data during my interactions with AI tools. My use was aligned with the ethical principles and academic integrity required for the TFM.

A.3.2. Technical Use of Generative AI

I declare that I used **ChatGPT-4 (OpenAI)** as a support tool during the development of my Master's thesis.

The use was limited to: - Understanding technical concepts related to the system architecture (e.g. modular agents, retrieval evaluation) - Clarifying definitions or approaches already being implemented - Exploring alternatives for metrics or evaluation techniques - Debugging or improving elements of existing code (e.g. reranking logic or cosine similarity interpretation)

The system was never used to generate or structure the content of the thesis, and no code or explanation was copied directly. All contributions made by AI were reviewed, adapted, and incorporated in accordance with academic standards.

A.3.3. Reflection on Usefulness

The use of Generative AI provided useful guidance during moments of technical doubt or when refining implementation ideas. It supported my learning by offering alternative perspectives, but it did not replace my reasoning or critical analysis. All decisions, designs, and final content of the project are my own work and responsibility.

BIBLIOGRAPHY

[1] A. Perry et al., *Autorag: Automatic retrieval-augmented generation evaluation and optimization*, Meta AI, 2024.

[2] W. Wang et al., *Best practices for retrieval-augmented generation*, Meta AI, 2024.

[3] H. W. Chung et al., "Scaling instruction-finetuned language models," *arXiv preprint arXiv:2210.11416*, 2022.

[4] N. Reimers and I. Gurevych, *Sentence-bert: Sentence embeddings using siamese bert-networks*, 2019.

[5] J. Johnson, M. Douze, and H. Jegou, *Faiss: A library for efficient similarity search*, Facebook AI Research, 2017.

[6] H. Chase, "Langchain: Building applications with llms through composability," *GitHub*, 2023, https://github.com/langchain-ai/langchain.

[7] NVIDIA, *Workbench example agentic rag*, GitHub repository, 2025.

[8] S. Yao et al., "React: Synergizing reasoning and acting in language models," *ICLR*, 2023.

[9] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York: McGraw-Hill, 1983.

[10] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *ACL Workshop*, 2004.

[11] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation," in *ACL Workshop*, 2005.