RAGRouter: A Latency-Aware Routing Framework for Retrieval-Augmented Generation

Mohammed El Azhar
GitHub · LinkedIn

January 2026

Abstract

Retrieval-augmented generation (RAG) improves factuality and domain adaptation of large language models by combining them with a retriever that fetches relevant documents from an external knowledge base. While valuable for answering information-seeking queries, retrieval can introduce significant latency and compute cost, especially when used for every query regardless of difficulty. *RAGRouter* is a lightweight framework designed to adaptively route queries between direct generation and retrieval-augmented generation. It employs a compact classifier that decides, on a per question basis, whether the answer can be generated directly by a small language model or whether retrieval and re-ranking are necessary. This report provides a detailed exposition of the project. We discuss the motivation behind query routing, describe the architecture and major modules, examine the data ingestion and retrieval mechanisms, explain the router's feature extraction and training procedure, and outline the end-to-end pipeline exposed through an API and Streamlit interface. We also highlight limitations, potential improvements and the broader context of adaptive routing in RAG systems.

Contents

Chapter 1

Background and Motivation

1.1 Retrieval-Augmented Generation

Large language models (LLMs) excel at generating fluent text but often struggle with factual accuracy and domain-specific knowledge. To alleviate these limitations, retrieval-augmented generation combines a generative model with a retriever. Before the model answers a question, the system searches an external knowledge base for relevant documents and provides the retrieved passages as additional context. Qdrant's overview of RAG notes that retrieval integrates external information into the generation process so that the model can reference a knowledge base beyond its pre-trained data and produce more accurate answers[1]. The article emphasises that retrieval significantly improves the accuracy and relevance of generated responses by expanding the model's knowledge scope[1]. However, RAG systems also introduce complexity: documents must be indexed in a vector database, queries must be embedded and compared, and the model must read additional context. These operations increase latency and cost relative to direct generation.

1.2 Adaptive Routing in RAG Systems

Not every query benefits equally from retrieval. Simple factual questions like "What is 2 + 2?" can be answered directly by a local model, while complex information-seeking queries require external evidence. Emerging research in *routeRAG* explores dynamic routing mechanisms that choose between multiple retrieval or reasoning pathways based on query difficulty and system constraints. An article describing the RouteRAG paradigm highlights that adaptive routing aims to minimise unnecessary compute while maintaining answer quality by selecting the minimal processing pathway needed for each query[2]. It lists key aims such as exploiting complementary strengths of multiple knowledge sources, minimising latency and cost, and adapting to system-level constraints like resource load[2]. RAGRouter implements a simple yet effective instantiation of this paradigm by routing between direct generation and retrieval-augmented generation.

Chapter 2

Project Overview

2.1 Problem Statement

Standard RAG pipelines naively run retrieval for every query. Although retrieval improves factuality, it adds latency (vector search, re-ranking, longer prompts) and compute cost (embedding and context processing). In interactive applications or when running on CPU, this overhead can degrade user experience. The observation that many queries can be answered directly by a small language model motivates the development of a router that decides whether retrieval is necessary.

2.2 Goals and Design Principles

RAGRouter aims to:

- **Reduce latency and compute cost.** By bypassing retrieval on easy questions, the system avoids vector search, embedding and context processing overheads. Direct generation via a small model provides a concise answer quickly.
- **Preserve answer quality.** When a query requires external knowledge, RAGRouter invokes retrieval and uses a RAG pipeline to generate a faithful answer with explicit citations.
- **Remain lightweight and CPU-friendly.** The framework runs entirely on CPU without GPU acceleration. It uses a compact logistic regression model for routing and a 0.5-billion-parameter Qwen model served through the Ollama runtime for generation. The vector store (Chroma) and embedding model (`intfloat/e5-small`) also run on CPU.
- **Expose a modular API.** RAGRouter is designed around a FastAPI service with a simple `/ask` endpoint. A Streamlit web UI provides a demo for interactive exploration.

Figure 2.1 summarises the high-level architecture. A user query is fed into a router. If the router predicts that the query can be answered directly, the system invokes a small language model to produce a concise answer. Otherwise it performs retrieval from a vector database, selects relevant passages, re-ranks them, and queries the same language model with the context to generate a citation-rich answer.

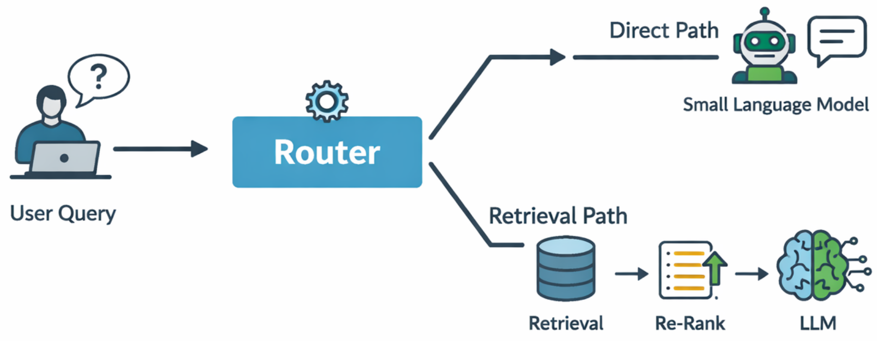


Figure 2.1: Simplified architecture of the RAGRouter pipeline. A router directs each query either to direct generation via a small language model or to a retrieval- augmented pipeline consisting of vector search, re-ranking and generation.

Chapter 3

Data Ingestion and Retrieval

3.1 Corpus Preparation

RAGRouter ingests a corpus of text files and converts them into vector embeddings for retrieval. The corpus can be any collection of documents and resides under `data/corpus`. Files are read using UTF-8 with errors ignored. The `load_corpus` function returns tuples of document ID, raw text and metadata containing the absolute source path. To accommodate the limited context window of small language models, documents are split into overlapping chunks using a sliding window over tokens (default window length $L = 512$ tokens and stride $S = 128$). The `chunk_text` utility ensures that even short documents produce at least one chunk by emitting a single window when the text is shorter than L.

3.2 Embedding and Vector Store

The ingestion script[1] embeds document chunks using a CPU-based SentenceTransformer model (`intfloat/e5-small`) and stores the resulting vectors in ChromaDB, a simple vector database. The script connects to a Chroma server via HTTP, creates a collection configured for cosine similarity and upserts all chunks in batches. Each chunk has a stable ID of the form <`source_path`>`#`<`chunk_index`> and metadata containing the original file path. Embedding is performed on CPU to ensure the framework remains lightweight. The ingestion script outputs a summary in JSON containing the number of files and chunks ingested.

3.3 Retrieval

At query time the `retrieve` function computes an embedding for the question using the same SentenceTransformer model and queries the Chroma collection via `HttpClient` to obtain the top k candidates. The scores returned by Chroma are distances; they are converted to similarity scores via $1 - distance$. The retriever then wraps each document, source and score into a `Passage` dataclass. The retrieval step is used both by the router to gauge query difficulty and by the RAG pipeline to fetch context. Qdrant's overview of RAG notes that vector databases store embeddings and support efficient similarity searches to retrieve relevant data[1].

[1] `src/ingest_docs.py`

Chapter 4

Router Design and Training

4.1 Feature Extraction

The router's task is to decide whether a query requires retrieval. It uses a set of hand-crafted features extracted from the question and the retrieval probe:

- **Length and numeric indicator.** The number of word tokens (`len_words`) provides a proxy for query complexity. A binary feature `has_num` detects the presence of digits, which often indicates specific numeric lookup questions.
- **TF-IDF rarity.** A simple TF-IDF vectorizer is fit on all corpus documents (`data/corpus`). The proportion of query tokens with IDF above the 95th percentile (`tfidf_rare`) measures how unusual the vocabulary is; rare terms tend to require retrieval because they are less likely to be memorised by the language model.
- **Retrieval probe statistics.** A quick retrieval of up to eight passages is executed. The router records the top similarity score (`top1`), the mean score over the top k (`topk_mean`), the gap between the top two scores (`gap12`) and the number of unique sources among the hits (`uniq_src`). High scores indicate that information is easily retrievable, whereas low scores or many unique sources suggest a diffuse or complex query.
- **Question type.** A simple heuristic categorises the first token of the question into one of {`what, who, where, when, why, how, other`} (`wh`). This categorical feature is later one-hot encoded.

These features are extracted in a deterministic order to align training and inference.

4.2 Rule-Based and Logistic Regression Routing

The router supports two modes:

1. **Rule-based fallback.** When no machine-learned model is available, a simple rule checks the maximum retrieval score and the lexical overlap between the query and retrieved passages. If the top score is below 0.35 or the overlap fraction is below 0.20, the router predicts the direct path; otherwise it predicts the RAG path. This conservative rule acts as a safeguard and provides a baseline.

2. **Logistic regression.** When available, a logistic regression classifier is loaded from `models/router.joblib`. The features described above are converted into a numeric vector: the seven numeric features followed by a one-hot encoding of the question type. Logistic regression is a supervised machine learning algorithm widely used for binary classification; it models the probability that an input belongs to one of two classes using a sigmoid function[3]. Because the dependent variable is binary (direct vs. needs-RAG), logistic regression is appropriate for this routing task. The classifier outputs the probability that a query requires retrieval; this probability is compared against a threshold stored in `models/threshold.txt`. If the probability exceeds the threshold and the lexical overlap safeguard passes, the router selects the RAG path; otherwise it chooses direct generation.

4.3 Training Procedure

Training the router involves constructing a dataset of questions labelled as `direct` or `needs-RAG`. The `train_router.py` script reads `data/qa/labels.jsonl`, extracts features, splits the data into training and validation sets (80/20 split with a random seed), and fits a logistic regression model with a class weight of 2.0 for the positive class to handle potential class imbalance. The validation probabilities are scanned over a grid of thresholds to find one that achieves recall of at least 0.9 while maximising F1 score. If no threshold meets the recall target, the threshold with maximal F1 is chosen. The script saves the model, threshold and evaluation metrics; an example `router_metrics.json` shows perfect precision, recall and F1 on the validation split with a threshold of 0.2 and confusion matrix $[20, 0; 0, 40]$.

4.4 Mathematical Details and Evaluation Illustration

Logistic regression models the log-odds of the positive class as a linear function of the input features. Given a feature vector $X \in R^m$ and weights $w \in R^m$ with bias b, the pre-activation is

$$z = w^\top X + b. \tag{4.1}$$

The logistic (sigmoid) function maps z to a probability between 0 and 1:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{4.2}$$

A query is classified as requiring retrieval if $\sigma(z)$ exceeds a threshold τ; otherwise the direct path is taken:

$$\hat{y} = \{ 1\ if \sigma(z) \geq \tau, 0 otherwise. \tag{4.3}$$

Figure 4.2 visualises the evaluation metrics of the trained router. Because the provided dataset is small and synthetic, the model achieves perfect precision, recall and F1 on the validation split. In practice, larger and more diverse datasets would result in more informative metrics.

The confusion matrix for the router on the validation set is given in Figure 4.1. There are 20 true negatives (direct questions correctly routed to the direct path) and 40 true positives (needs-RAG questions correctly routed to the retrieval path), with no misclassifications. Thus the validation set contained 60 examples (20 direct and 40 needs-RAG), which explains the perfect precision and recall values. Given the small size and synthetic nature of the dataset, these metrics should be interpreted as an optimistic upper bound rather than generalisable performance. The labelled questions were manually composed, resulting in a dataset of only about 300 examples. When split 80/20, the validation set contains just 60 queries. This limited and handcrafted dataset likely contributes to the perfect metrics and underscores the need for larger, more diverse training data.

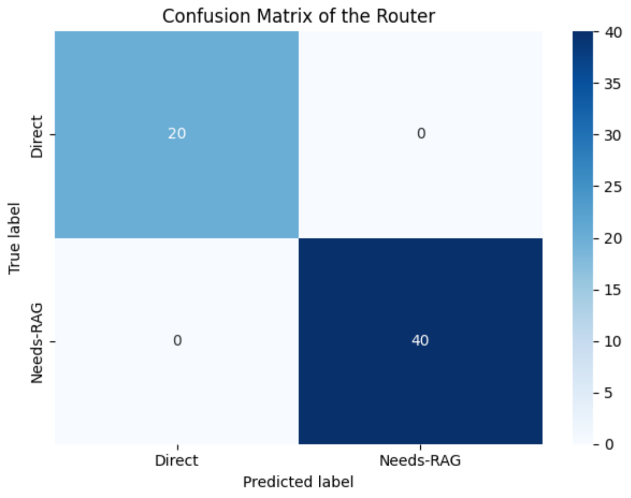


Figure 4.1: Confusion matrix of the router on the validation set. Rows denote the true class (direct vs. needs-RAG) and columns denote the predicted class. The absence of off-diagonal entries indicates that all 60 validation examples were correctly routed.

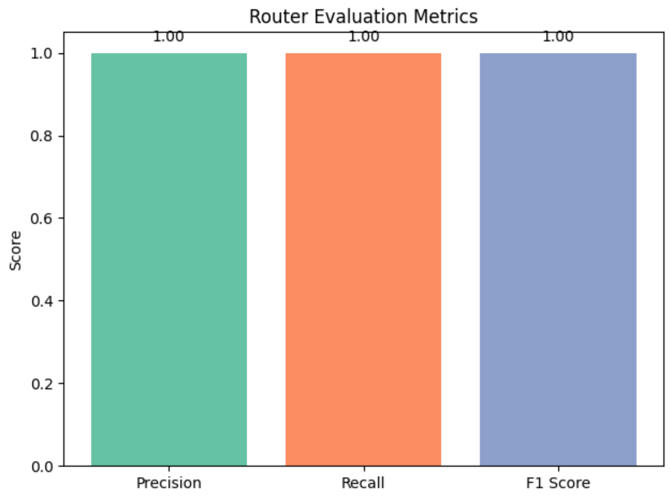


Figure 4.2: Validation metrics of the trained router: precision, recall and F1 score. The model achieves perfect scores on the small synthetic validation set.

Chapter 5

Answer Generation

5.1 Direct Generation

In the direct path, the `answer_direct` function constructs a concise prompt of the form "Answer concisely and factually" followed by the question. It then sends a request to the Ollama server hosting the `qwen2.5:0.5b-instruct` model and returns the response. The function records the elapsed time and wraps the result in a `DirectAnswer` dataclass. The generation is restricted to a small number of tokens (`num_predict`=96 by default) to keep latency low.

5.2 Retrieval-Augmented Generation

When the router selects the RAG path, the `answer_rag` function performs the following steps:

1. Invoke `retrieve` to obtain the top k candidate passages. Filter out passages with a lexical overlap fraction below 0.2 to ensure that context passages share vocabulary with the question.

2. Select up to K_{ctx} passages to include in the prompt. If the environment variable `USE_MMR` is true, the function uses maximal marginal relevance (MMR) to promote diversity: starting with the highest-scoring passage, additional passages are chosen to maximise a weighted combination of relevance and dissimilarity based on Jaccard similarity between passages. Otherwise the top-scoring passages are selected directly.

3. Build a French prompt instructing the model to act as an "assistant strictly extractive," answer using only the context and cite sources as [i] footnotes. When no context is available, a fallback prompt asks the model to answer briefly and admit when information is missing.

4. Call the Ollama model with the prompt and limited generation length (128 tokens). If the returned answer does not contain any citations, the function appends citations for all provided passages. Otherwise it returns the answer with the passages and the total latency.

This pipeline ensures that answers in the RAG path are grounded in retrieved documents with explicit citations. The use of MMR for context selection reduces redundancy and promotes coverage of diverse aspects of the query.

Chapter 6

API and User Interface

6.1 FastAPI Service

RAGRouter exposes its functionality through a minimal FastAPI application defined in `src/service_api.py`. The service defines a single `/ask` endpoint taking a JSON body with a `question` field. Upon receiving a request, the endpoint strips the question, calls the router, invokes the appropriate answer function and returns a JSON response containing:

`route` either `direct` or `rag`, indicating the chosen pathway.

`answer` the generated answer (string).

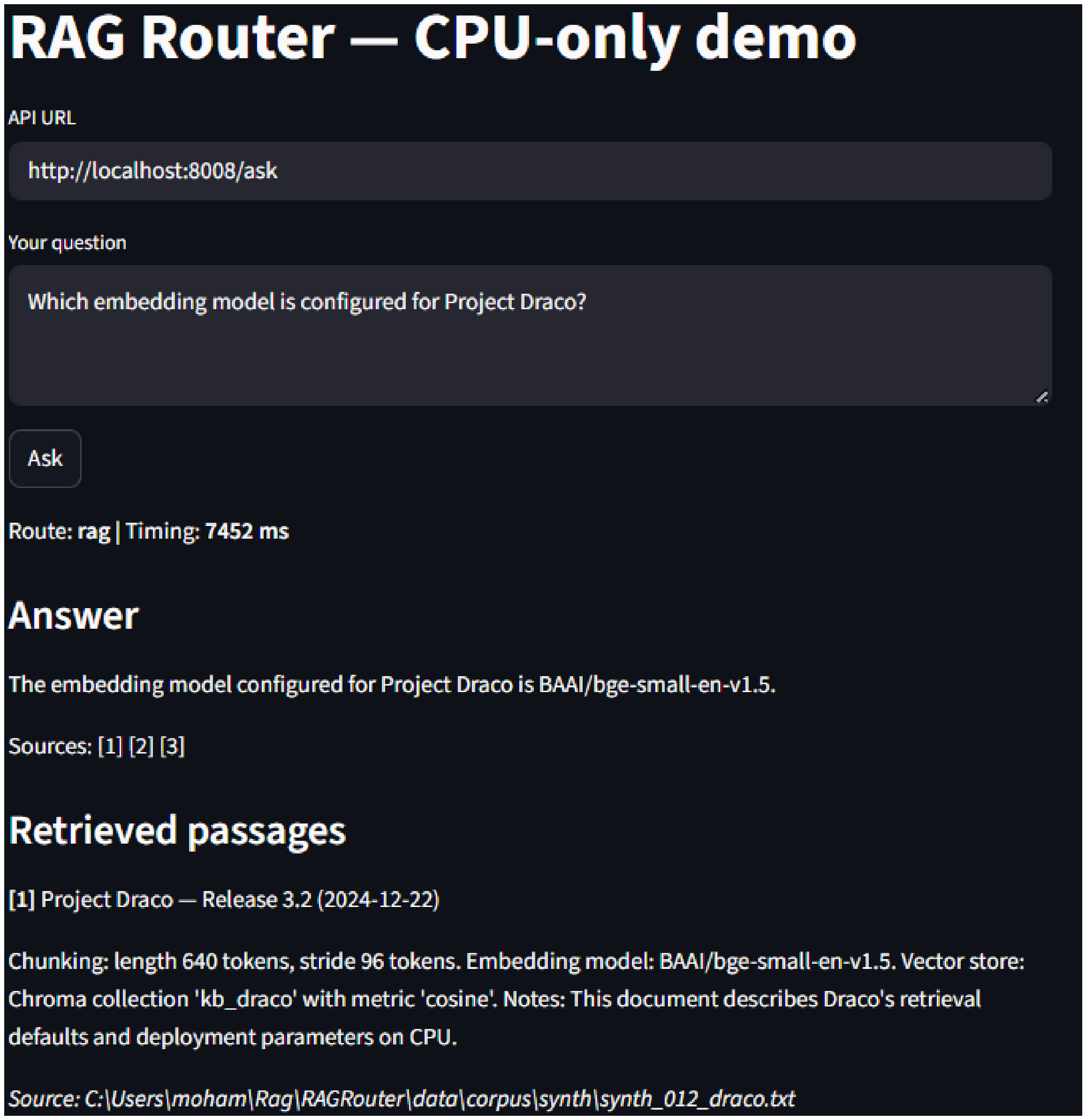
`passages` a list of passage dictionaries in the RAG case, containing the text, source and score; empty for direct answers.

`timing_ms` the total latency in milliseconds.

This API design enables programmatic integration of RAGRouter into other applications. It can be deployed behind authentication or proxies as needed.

6.2 Streamlit Demo

For convenience, the project includes a Streamlit application (`ui/app.py`) providing a simple web interface. The UI consists of a text box for entering questions, a button to submit the query and displays the route, answer and citation-linked passages. Streamlit automatically refreshes the UI and shows a progress bar while waiting for the backend response. Because the UI communicates with the FastAPI backend over HTTP, it must be started separately after the backend is running.



RAG Router — CPU-only demo

API URL

http://localhost:8008/ask

Your question

Which embedding model is configured for Project Draco?

Route: **rag** | Timing: **7452 ms**

Answer

The embedding model configured for Project Draco is BAAI/bge-small-en-v1.5.

Sources: [1] [2] [3]

Retrieved passages

[1] Project Draco — Release 3.2 (2024-12-22)

Chunking: length 640 tokens, stride 96 tokens. Embedding model: BAAI/bge-small-en-v1.5. Vector store: Chroma collection 'kb_draco' with metric 'cosine'. Notes: This document describes Draco's retrieval defaults and deployment parameters on CPU.

Source: C:\Users\moham\Rag\RAGRouter\data\corpus\synth\synth_012_draco.txt

Figure 6.1: Streamlit interface (`ui/app.py`) during a RAG query. The UI displays the selected route, end-to-end latency, the generated answer, and the retrieved passages with source references.

Chapter 7

Deployment and Dependencies

7.1 Software Stack

RAGRouter is fully containerised using Docker Compose. The stack includes:

- **Ollama runtime.** Provides CPU inference for the Qwen 2.5 0.5B instruction-tuned model and other small language models. The runtime exposes a REST API for text generation.
- **ChromaDB.** Hosts the vector store and supports HTTP queries for similarity search. Each collection is configured for cosine distance.
- **FastAPI app.** Serves the router and answer functions via `/ask`.
- **Streamlit UI.** Provides an interactive front end.

All components run on CPU and can be orchestrated via the provided `docker/docker-compose.yml`. Alternatively, a local virtual environment can be used to run the Python code directly.

7.2 Models and Data

By default the project uses the following models:

- `qwen2.5:0.5b-instruct` as the small language model for both direct and RAG-based answers.
- `intfloat/e5-small` from the sentence-transformers library for query and document embeddings.
- Logistic regression trained on questions labelled as direct or needs-RAG.

The repository includes a synthetic corpus under `data/corpus/synth` and a small set of labelled questions under `data/qa`. Users can replace these datasets with their own documents and question labels.

Chapter 8

Limitations and Future Work

While RAGRouter demonstrates a viable approach to latency-aware query routing, it remains a minimal system with several limitations:

- **Small training set.** The provided labels are small and partly synthetic. The perfect evaluation metrics indicate overfitting; larger and more diverse training data are needed for robust performance.
- **Simple features and model.** The router uses handcrafted features and logistic regression. More sophisticated methods such as gradient boosted trees, neural classifiers, or contrastive learning on retrieval embeddings (as explored in the RAGRouter research paper) could better capture query difficulty.
- **Limited retrieval and generation models.** The system is tied to a single small language model and embedding model. Extending to multiple LLMs or multimodal retrieval would require new routing strategies and evaluation.
- **No evaluation dashboard.** Although the repository plans for a dashboard to visualise metrics such as faithfulness, latency and bypass percentage, it is not yet implemented. Comprehensive evaluation on real-world datasets is essential for deployment.

Despite these limitations, RAGRouter exemplifies how adaptive routing can significantly reduce inference cost while maintaining answer quality. The concept aligns with broader research that emphasises dynamically selecting retrieval and reasoning pathways to match query complexity[2].

Chapter 9

Conclusion

Retrieval-augmented generation enables language models to access external knowledge sources, but naive pipelines that perform retrieval for every query incur latency and compute overhead. RAGRouter addresses this problem by learning to route queries to either direct generation or a retrieval-augmented pipeline. A simple feature extractor and logistic regression classifier achieve effective routing without requiring GPUs, and the system exposes an easy-to-use API and optional UI. While the current implementation is lightweight and primarily a proof of concept, it illustrates the benefits of adaptive routing in RAG systems and provides a platform for experimentation with more sophisticated routing strategies and larger models. As LLMs continue to proliferate, approaches like RAGRouter will be important for deploying cost-effective and responsive knowledge-intensive applications.

Chapter 10

References

1. Sabrina Aquino. *What is RAG: Understanding Retrieval-Augmented Generation.* Qdrant blog, 19 March 2024. This article explains how retrieval integrates external information into the generation process and why vector databases are used for efficient similarity search. Accessed January 2026.

2. Emergent Mind. *RouteRAG: Adaptive Routing in RAG Systems.* Updated 14 December 2025. The article discusses adaptive routing mechanisms in retrieval-augmented generation, emphasising the goals of matching minimal processing pathways to query complexity and minimising latency and cost. Accessed January 2026.

3. GeeksforGeeks. *Logistic Regression in Machine Learning.* Last updated 23 December 2025. The tutorial describes logistic regression as a supervised algorithm for binary classification, using the sigmoid function to map linear combinations of features to probabilities. Accessed January 2026.