

Athena: A Plug-and-Play Advisor for Retrieval-Augmented Generation using VectorDB

Ning Liang*, Fabian Wenz†, Jana Giceva†, and Lisa Wu Wills*

*Department of Computer Science, Duke University, Durham, USA

Email: ning.liang@duke.edu, lisa@cs.duke.edu

†Department of Computer Science, TU Munich, Munich, Germany

Email: fabian@wenz.org, jana.giceva@in.tum.de

Abstract—Retrieval-Augmented Generation (RAG) has emerged as a popular technique for addressing several challenges of Large Language Model (LLM) systems, including static model knowledge, hallucination, and limited input sequence lengths. Although RAG mitigates common pitfalls of current LLM systems, its inherent heterogeneity and configurability introduce new challenges. The performance of RAG is crucial for meeting the high-throughput and low-latency demands of LLM services. Different components of RAG operate on different hardware platforms, and their complexity scales with the configurability and complexity of the rest of the system. For example, larger embeddings may enhance retrieval accuracy, but also increase the latency of embedding creation and indexing, thereby compromising the RAG system’s performance and energy consumption.

Thus, a comprehensive characterization of an end-to-end RAG system becomes necessary. In this work, we build an end-to-end RAG benchmarking framework, Athena, that supports various embedding models, vector databases, index/search algorithms, and LLMs. By characterizing the system under various RAG settings built using Athena, we demystify RAG by identifying performance bottlenecks and quantifying the impact of each sub-component on overall system performance. In addition, the plug-and-play, open-sourced Athena framework is designed to assist future RAG research.

Index Terms—Retrieval Augmented Generation (RAG), Vector Database, Large Language Model (LLM)

I. INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable success in a wide range of natural language processing tasks, including question answering, conversational agents, machine translation, summarization, and code generation [4], [7], [9], [12], [30], [33], [36], [50], [58]. Their ability to generate coherent and contextually relevant texts has revolutionized the way users interact with information and language systems. However, despite these capabilities, LLMs still suffer from several critical limitations. These include static model knowledge [14], where the model’s information is fixed at training time and cannot incorporate newly available facts, and hallucination [32], where the model generates plausible-sounding but factually incorrect content. These issues significantly hinder the deployment of LLMs in knowledge-intensive domains [31], [61].

Retrieval Augmented Generation (RAG) has emerged as a promising solution to mitigate these challenges by augmenting

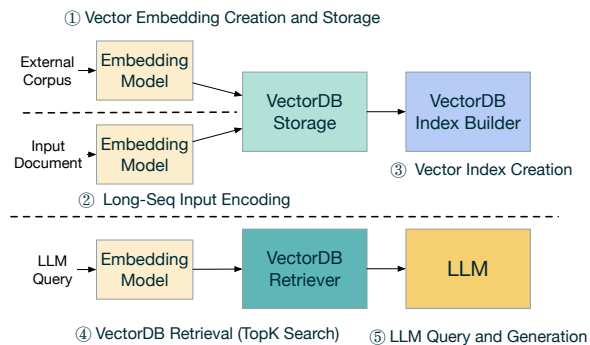


Fig. 1: RAG execution flow that covers common use cases.

LLMs with external knowledge [5], [20], [53]. In a typical RAG pipeline, an external retriever fetches relevant documents or facts from a large corpus, often using vector similarity search, and these retrieved results are appended to the prompt fed into the LLM. Leveraging the information retrieval power of RAG, LLMs can generate high-quality responses, process longer prompts, and require less frequent model training. Due to the effectiveness of RAG, RAG-based LLM systems have been widely adopted in production-scale applications by leading companies such as Google REALM [16], Microsoft GraphRAG [13], and Nvidia InstructRetro [62].

Despite the effectiveness in solving the LLM pitfalls, RAG introduces several new challenges: (1) RAG is a large system that is inherently heterogeneous. According to Figure 1, RAG consists of multiple stages, including embedding models, vector databases, generative LLMs, and optional models such as query re-writers and re-rankers [15]. The vector database usually runs on the CPU server while the models execute on GPUs or other machine learning accelerators. (2) RAG systems are highly configurable. Their performance and accuracy are highly dependent on key factors such as the dimensions of vector embeddings, choice of embedding models and LLMs, vector retrieval algorithms, and the batching strategy throughout the pipeline.

Recent works have focused on optimizing many individual components of RAG, such as vector databases and vector search, from both software [20], [22], [37] and hardware [21], [51], [64] perspectives. However, there is still a lack of a systematic study on the complete RAG pipeline to understand

the interplay of the various components and their configuration settings. This gap makes conducting research in RAG systems particularly challenging. We present Athena — a plug-and-play advisor for Retrieval-Augmented Generation using vector databases. Athena consists of commonly used RAG components such as embedding models, vector databases, vector retrieval algorithms, and LLMs. It is integrated with various open-source frameworks such as vLLM, Ollama, Milvus, and Postgres [28], [45], [60], [63]. The combinations and flexibility of these modules make it possible to build typical RAG systems for various purposes. Athena reports key metrics and assists designers in identifying relationships between sub-components, significantly reducing the effort of RAG design space exploration. Using Athena, we construct several different RAG pipelines by swapping modules and share insights into their performance characteristics and potential optimization opportunities.

II. BACKGROUND

A. Embedding Models

In RAG, unstructured information, such as text and images, is represented as vector embeddings. Typically, these vector representations are derived from pre-trained embedding models. For example, image embeddings are produced using Vision Transformers [11], [68] and Convolutional Neural Networks [68], and text embeddings are generated using bidirectional embedding models (BERT models) [10] and decoder-only LLM-based models [29]. The Massive Text Embedding Benchmark (MTEB) [44] is a state-of-the-art leaderboard that ranks various text embedding models in tasks such as information retrieval, clustering, and translation based on a composite score including nDCG@k, and recall@k evaluated on BEIR [43], [59]. In addition to embedding quality, embedding models also differ in features such as model size, embedding size, and max input tokens, which can affect both RAG system performance and generation accuracy. For example, a larger model may have longer embedding latency and produce a large vector, making retrieval slower. However, the retrieved documents may be more relevant to the question and increase the generation accuracy.

B. Nearest Neighbor Search

Vector representations are retrieved using Nearest Neighbor Search (NNS) algorithms. Given a query vector and a set of stored vectors in a high-dimensional space, NNS identifies K vectors in the collection that are the most similar to the query under distance metrics such as Euclidean distance, inner product, and cosine similarity. This operation is typically referred to as a *TopK search*. The accuracy of a TopK search is measured by its *recall rate*. The recall rate is defined as the percentage of the returned results that are objectively closest to the query vectors. NNS algorithms are divided into two categories based on the tradeoff between speed and accuracy: Exact NNS (ENNS) and Approximate NNS (ANNS).

Approximate Nearest Neighbor Search [39], [57] can significantly reduce TopK search latency and thus improve

scalability on larger datasets by sacrificing the recall rate. A *vector index* is a data structure that organizes the vector embeddings to speed up TopK searches. To use an ANNS algorithm for a TopK search, a vector index structure must be built, and the algorithm efficiently traverses the index structure to perform an approximate search. Multiple popular indexing schemes have been proposed and deployed; choosing which index to build and to perform the TopK search has direct implications for RAG system performance.

In contrast, Exact Nearest Neighbor Search guarantees a perfect recall rate and does not need a vector index. The algorithm is much simpler because it does not search while accessing the index. Instead, ENNS finds the most similar vectors by exhaustively scanning all stored vectors, which usually incurs higher TopK search latency. However, ENNS remains useful in scenarios where higher recall or zero indexing overhead is required [20], [51].

C. Vector Database

Vector database [37], [67], [70] serves as the core component that performs retrieval. It specializes in efficiently storing, indexing, and retrieving vector representations. These vector representations, derived from machine learning models, encode information about unstructured data such as text, image, and audio. In the context of vector databases, a hybrid query is often executed. In addition to the traditional scan, join, and aggregation operators, a hybrid query supports vector operators such as TopK vector similarity search. Different vector databases employ different vector search engines and interact with data in distinct ways. For example, Milvus [63] is a modern in-memory database built upon Faiss [23]. It loads all relevant data into system memory and processes them segment by segment. Pgvector [49], a PostgreSQL [60] extension for vector search, follows traditional relational databases to directly read data from the disk, which makes Pgvector much lighter-weight. We continue to discuss Milvus and Pgvector in Section 3.

D. RAG Pipeline

A RAG execution pipeline typically consists of the following steps/components as depicted in Figure 1: ① Vector Embedding Creation and Storage, ② Long-sequence Input Encoding, ③ Vector Index Generation, ④ VectorDB Retrieval or TopK Search, and ⑤ LLM Query using Retrieval-Augmented Generation and LLM Response. *Vector Embedding Creation and Storage* converts an external plain-text corpus into embeddings using an embedding model. Both the plain texts and their corresponding vector embeddings are stored in a vector database (VectorDB). *Long-sequence Input Encoding* is necessary when an input document is large (e.g., long input contexts for LLM queries) and needs to be converted into vectors in real time using a document encoder as part of the LLM query processing. *Vector Index Generation* is necessary when an approximate nearest neighbor search algorithm is used for retrieval (either for speed or for scale), but it is omitted if an exact nearest neighbor search algorithm is

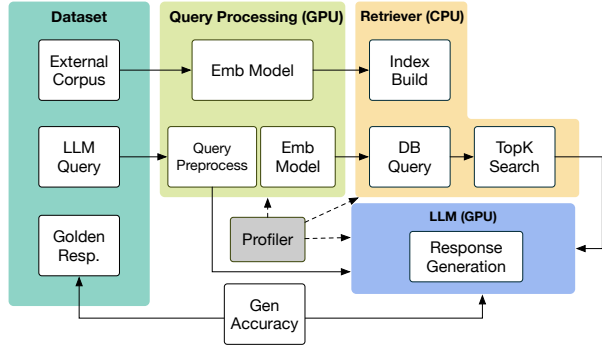


Fig. 2: Athena System Diagram

employed. *VectorDB Retrieval* performs a TopK Search using ANNS or ENNS algorithms and returns the TopK vectors that are most similar to the query vector. *LLM Query using RAG and LLM Response* is the last step where a query is augmented with the TopK search results from the previous step to provide a more accurate response. Note that when a user submits a LLM query (or a prompt), the query needs to be converted using the same embedding model from step ① before it can perform a search in step ④. The augmented query with the retrieved information is then used in step ⑤ to prompt the LLM for a response.

RAG systems also contain optional components such as query rewriters [38] and result rerankers [15]. These peripheral stages assist RAG in case users give vague or complex queries [15], [38], [55]. In query pre-processing, a rewriter can either rephrase the query or decompose a complicated query into multiple simpler ones. After the TopK vectors are retrieved, a reranker model can provide an additional filter to improve retrieval quality.

E. Common Use Cases of RAG

The most common use case of a RAG system is Hyperscale retrieval [6], [20]. RAG execution in this use case closely follows the pipeline introduced in the previous section. The VectorDB stores an enormous amount of external corpora and dynamically retrieves information relevant to users' prompts to facilitate LLMs with external knowledge at inference time. In addition, prior work has shown that Hyperscale retrieval with smaller LLMs generates matching or even better quality responses compared to larger LLMs. For example, with a two trillion token database, RETRO with 7B parameters from Deepmind [6] obtains comparable performance to GPT-3 [8], which has 175B parameters. The embedding and indexing steps ① and ③ are typically performed offline using powerful distributed infrastructure, while the retrieval and generation steps ④ and ⑤ are executed in real time.

RAG is also useful in long-context sequence processing [13], [20], [69]. One notable example is question answering based on a long document that users uploaded in real time. Instead of brute-force processing the entire document, a more effective approach is to treat the document as a knowledge base and retrieve only the relevant information. In this case, the five-step RAG pipeline has a key variation: steps ①, ②, and ③

TABLE I: Vector Indexing Algorithm Parameters

Index Type	Building Params	Search Params
HNSW	$M = 16, ef_construction = 64$	$ef_search = 64$
	$M = 32, ef_construction = 128$	$ef_search = 64$
	$M = 64, ef_construction = 256$	$ef_search = 64$
IVF-PQ	$m = 16, nlist = 1024$	$nprobe = 128$
	$m = 32, nlist = 2048$	$nprobe = 128$
	$m = 64, nlist = 4096$	$nprobe = 128$

must be performed at inference time. The size of the database is also much smaller compared to Hyperscale retrieval.

III. ATHENA FRAMEWORK

As previously discussed, prior work mainly focuses on optimizing individual components of RAG, such as vector databases and nearest neighbor search. The lack of systematic characterization and benchmarking poses challenges to RAG research because RAG execution consists of many heterogeneous components that make it difficult to predict RAG system performance. To support the understanding of RAG and assist RAG system design, we develop the first RAG benchmark framework that evaluates the end-to-end performance of RAG systems.

A. Athena Features

Athena provides the following features:

(1) An End-to-End Framework: Figure 2 illustrates the high-level system overview of our framework. Athena covers many building blocks of RAG. To our knowledge, this is the first end-to-end RAG benchmarking framework. We start with data preparation, which involves creating the vector embeddings from plain-text documents, inserting the vectors into databases, and optionally building an index. This can be done either offline or in real time, depending on the use case. Using the embedding model, we then map plain-text user query questions to the same embedding space. A search query is executed in the selected vector database using the pre-determined nearest neighbor search algorithm. Finally, we augment the search query with the retrieved documents and prompt the large language model using a zero-shot format, meaning the model receives no prior examples and must generate an answer based solely on the provided documents.

(2) A Modular Framework: Given the rapidly evolving and dynamic nature of RAG system development, Athena does not use any performance simulation and examines only real application behavior on CPUs and GPUs. We provide a simple and standardized abstraction interface between each RAG component and support plug-and-play capabilities. It directly receives a system configuration file to customize the RAG system. If the user wants to substitute one component, only minimal changes to the interfaces would be required. For example, our framework is integrated with Python libraries like Huggingface Transformer [17] and frameworks like Ollama [45] and Vllm [28]. Considering the high cost of embedding generation on large datasets, we have also provided an embedding vector dataset on Wikipedia articles with three different vector dimensions (384, 1536, 4096).

However, we have limited freedom in plugging in a new nearest neighbor search algorithm as they require substantial modifications to the vectorDB that supports these indexing schemes. Adding currently unsupported indexing schemes or supporting other vectorDBs are out of scope for this version of Athena and left for future work.

(3) An Advising framework: The number of choices available in each RAG system building block or component results in a very large design space. Our framework works seamlessly with profiling tools and generates insightful data that guides the system design process. The framework integrates AMD μ prof and Nvidia Nsight for CPU and GPU hardware performance metrics. It can generate performance and accuracy metrics such as latency, throughput, recall rate, and generation accuracy under any system configuration. With the guidance of these data, the framework helps its users easily identify bottlenecks in their RAG systems.

B. Athena Components

Using Athena, we can compose different RAG systems by curating the following components. We evaluate system performance with experiments and results presented in Section IV.

Dataset: We use Google’s Natural Questions (NQ) dataset [27] to evaluate RAG systems. These questions are designed for knowledge-intensive tasks. The Meta KILT benchmark [48] collects the NQ dataset and splits it into nq-dev and nq-train. We use nq-dev, which contains approximately 2,800 questions with golden answers in our experiments. The knowledge related to the questions is included in Wikipedia articles. Following an existing method [26], we pre-process all Wikipedia articles by splitting them into chunks that contain 100 tokens. We acknowledge that a different chunking mechanism may lead to different results.

Embedding Model: To study the effect of embedding size quality, we select two embedding models from the MTEB benchmark based on their retrieval task scores and number of output dimensions. BGE-small-en-v1.5 [3] is a BERT-based embedding model trained for vector retrieval tasks. It outputs 384-dimensional vectors and achieves a high retrieval score of 51.68 (retrieval scores range from 0 to 100, higher better, with 100 being the perfect retrieval score). INF-retriever-v1-1.5b [66] is a slightly larger model based on INF-retriever-v1, with an output vector size of 1536 dimensions. Unlike the previous model, it adopts a decoder-only architecture similar to LLMs. It achieves a slightly higher retrieval score of 60.83 on the MTEB benchmark.

Vector Database: We choose two vector databases: Milvus [63] and Postgres [60] with Pgvector [49]. Milvus is an open-source in-memory vector database built for billion-scale vector search and storage. Its vector search engine is built on top of Faiss [23], a high-performance vector search library developed by Meta AI. Milvus supports brute-force, graph-based, quantization-based, and even GPU-accelerated

search algorithms. Pgvector is a vector search extension for PostgreSQL (PG or Postgres). PG has attracted numerous users and has been applied to a broad range of use cases. Unlike modern vector databases specifically designed for vector search, PG uses a standard SQL interface, which supports complex SQL queries with operations such as joins and window functions. It also runs seamlessly with many extensions. In addition, since it is a traditional relational database, it manages data on disk rather than being fully in-memory, which makes it lightweight and cost-effective. Pgvector is one of the most popular vector extensions and has attracted significant attention from academia. Many papers [37], [67] aim to optimize vector search performance based on Pgvector. These two candidates provide a clear contrast in the system architecture of vector databases.

Large Language Model: We use LLaMA-3.1 8B and 70B [46] via the Ollama framework [45] as the backbone language models for response generation in our RAG experiments. Both models are decoder-only transformer architectures released by Meta [1]. With LLaMA-3.1 70B and LLaMA-3.1 8B [46], we are utilizing the latest iteration of each model size still capable of handling input contexts of 11,000 tokens, which is essential for processing the full set of retrieved documents in our setup. The 8B model offers a strong baseline performance with relatively low computational requirements. In comparison, the 70B model is closer in quality to proprietary frontier models such as GPT-4 [47], enabling us to explore the tradeoff between speed, memory footprint, and response quality across LLM scales.

Ollama [45] provides a low-friction interface for local model execution and supports quantized model variants (e.g., q4_K_M) that significantly reduce memory usage by storing model weights in 4-bit precision instead of the default 16- or 32-bit floating point formats. This allows large models like the LLaMA 8B to run efficiently on consumer-grade GPUs, while still making it possible to evaluate the much larger 70B model on machines with larger memory or multi-GPU setups. Quantization not only enables deployment on resource-constrained hardware but also improves inference speed with minimal degradation in accuracy — a crucial factor for practical experimentation at scale.

In our RAG pipeline, the LLM receives a prompt consisting of a zero-shot template that incorporates both the user question and the retrieved documents. The document injection format is based on widely adopted prompt engineering practices in prior RAG literature. The retrieved chunks are output from the vectorDB TopK search, using the selected embedding model and NNS. These chunks are then injected into a prompt template alongside the user query to guide generation. We use zero-shot prompting without fine-tuning to assess the LLMs’ out-of-the-box performance, isolating the effects of retrieval size and vector database behavior on overall system quality.

To evaluate generation quality, we use ROUGE-1 [34] as our primary performance metric, which measures unigram

TABLE II: Representative KPIs reported by Athena for RAG system evaluation.

Dimension	Representative KPIs
Quality	p99 latency, average latency, recall rate, LLM generation accuracy, time-to-first-token (TTFT)
Capacity	VectorDB throughput (QPS), LLM throughput (QPS, tokens/sec), embedding model throughput
Cost	Power consumption breakdown, memory utilization, hardware resource usage

overlap between the generated answers and ground-truth references. ROUGE-1 is widely used in recent RAG and open-domain QA evaluations due to its simplicity and robustness to minor phrasing differences [18], [19], [54]. Given that many benchmarks, such as those derived from Wikipedia-based QA datasets, rely on corpora that have evolved over time, ROUGE-1 provides a reasonable proxy for factual consistency even when exact document matches are no longer guaranteed. This is especially relevant in our setup, where the document corpus may diverge from the one used in earlier benchmark constructions.

Nearest Neighbor Search Algorithms: We evaluate two ANNS algorithms: HNSW and IVF-PQ.¹ HNSW [39] is a commonly used graph-based approximate nearest neighbor algorithm with low latency and high recall rate. It builds a multi-layered proximity graph which usually contains 3 levels. Each layer contains a subset of points from the layer below, with layer 0 containing all data points. Higher layers (i.e., layers 1 and 2) serve as navigation shortcuts, containing exponentially fewer points. During index construction, each new point is assigned a random layer level. The algorithm searches for nearest neighbors starting from the top layer and progressively moves down, maintaining a dynamic candidate list of size $ef_Construction$ at each layer. From these $ef_Construction$ candidates, it selects the M best neighbors to create an edge between them and the new data point. Larger $ef_Construction$ and M values improve search accuracy at increased computational cost. HNSW search follows a greedy search strategy beginning at the top layer. The algorithm maintains a similar dynamic candidate list of size ef_Search , exploring the graph by examining neighbors of the current best candidates. The algorithm returns the approximate TopK when the dynamic list meets a certain termination condition.

IVF-PQ [25], on the other hand, is known for its low memory requirement. Its index construction consists of two phases: Inverted File Indexing (IVF) and Product Quantization (PQ). In the IVF phase, a k -means algorithm partitions the dataset into $nlist$ clusters. Each vector is assigned to its nearest centroid (referred to as IVF centroid). In the PQ phase, residual vectors (original vector minus centroid) are computed within each cluster. Each residual is divided into m equal-length sub-vectors. Each sub-vector is quantized using a separate k -means with 2^{nbit} centroids (referred to as segment centroid), each associated with an id . The original residual vector within each cluster is represented by a more compact m -dimensional vector of ids . A codebook is maintained to map segment centroid ids to the actual segment centroid vectors. At query

¹Unfortunately, Pgvectors does not currently support IVF with product quantization, so we only evaluate HNSW on PG.

TABLE III: Postgres Configurations

Parameter	Value
work_mem, shared_mem, maintenance_work_mem	128GB
effective_cache_size	256GB
max_worker_procs	128
max_parallel_workers, max_parallel_maintenance_workers	64

time, the algorithm identifies the $nprobe$ closest IVF clusters. For each, it computes the residual against the IVF centroid, splits the residual into m segments, and constructs lookup tables containing distances from each query segment to all 2^{nbit} centroids. Approximate distances to all database vectors are computed by summing values from the lookup tables, using their PQ codes.

In our experiment, we built both HNSW and IVF-PQ using three different setups. Table I lists the combinations of parameters. A combination of larger parameters means a larger index building effort and implies better search recall but worse latency. To refer to them effectively, we name them by their index name plus their building effort. For example, an HNSW indexing using low effort is called HNSW_0.

Batch Size: Many components in the pipeline – including the vector database, embedding model, and large language model – support batched queries to improve utilization and throughput. While larger batch sizes can boost throughput by increasing system resource utilization, they may be impractical in latency-critical applications where queries must be served within a tight response window. In such scenarios, waiting to accumulate a large batch can introduce unacceptable delays. In addition, high throughput inevitably sacrifices latency because of resource constraints. To capture this tradeoff, we sweep the batch size in powers of 2 from 1 to 32 and evaluate its impact on system performance.²

Key Performance Indicators (KPIs): Our tool provides a comprehensive set of KPIs to evaluate retrieval-augmented generation (RAG) systems. Since effective RAG design requires balancing quality, capacity, and cost-efficiency, Athena reports metrics across all three dimensions (Table II). Athena collects fine-grained statistics across the query pipeline to capture trade-offs among latency, throughput, and accuracy, while also profiling hardware utilization and power consumption to assess overall cost-effectiveness.

IV. CHARACTERIZATION OF RAG SYSTEMS

A. CPU and GPU System Configurations

RAG systems utilize both CPUs and GPUs. Our experiments are conducted on a Standard_NC40ads_H100_v5 instance on Microsoft Azure [42], which has a 40-core AMD EPYC 9V84

²Postgres uses a modified SQL interface to perform vector search and does not support any batch size greater than 1.

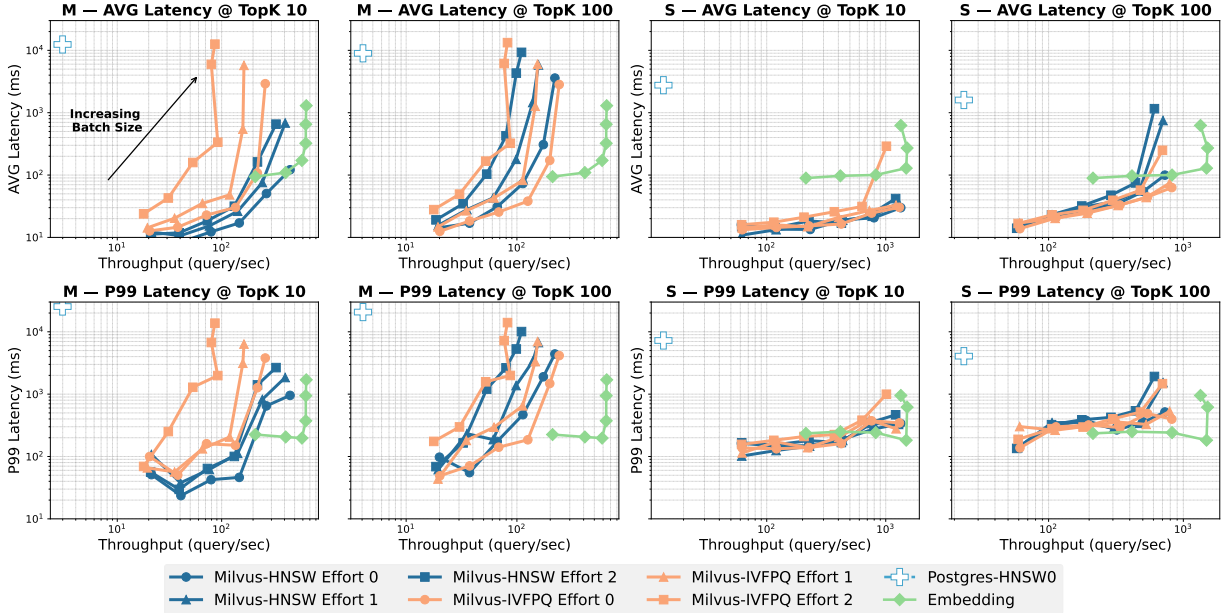


Fig. 3: Embedding generation and vector search throughput and their corresponding average and P99 latencies.

processor [2] running behind a virtual machine hypervisor and 320 GB of DRAM. The system is equipped with an NVIDIA H100 GPU with 94 GB of HBM, connected via a PCIe x16 interface. As mentioned in the previous section, Athena supports two VectorDBs, Milvus and Postgres. Milvus is an in-memory database, so unless memory is limited, it does not interact with disk. In contrast, Postgres stores its data in an Azure P50 premium SSD instance that features a maximum bandwidth of 1,000 MB/s and 30,000 IOPS. Table III shows the Postgres configurations. For experiments requiring access to hardware performance counters, we use a server with a 48-core AMD EPYC 7643 CPU and eight memory channels, each populated with 32GB DDR4-3200 DIMMs. In all experiments, we host the embedding model and the LLM on the same H100 GPU. We host the vector databases on the CPU and do not run any retrieval on the GPU (note: this setup is typically used in practice) because running retrieval on the GPU requires significantly more GPU memory, which dramatically increases the cost of a system.

B. Experimental Setup and Results Analysis

In this section, we examine research questions specifically designed to understand the performance characteristics of the RAG system using various RAG system components, including latency, throughput, recall, memory usage, and power consumption. We design experiments and analyze results to derive insights that assist RAG system developers in making informed tradeoffs.

Q1. How does the choice of each RAG component affect the query latency and throughput?

Experimental Setup To realistically evaluate the performance of an RAG system, it is important to consider that

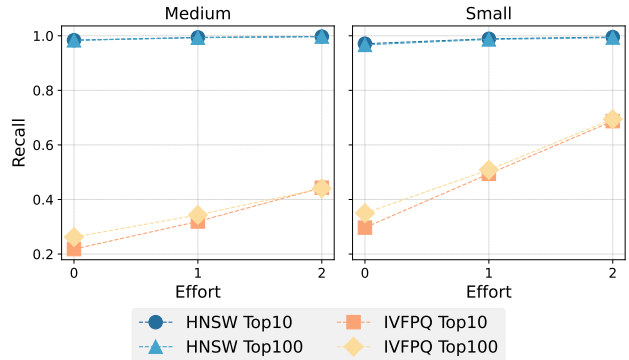


Fig. 4: Recall Rate on different ANNS indices with different build efforts

such systems are designed to serve multiple concurrent users rather than a single isolated request. Since the VM we use contains 40 physical cores, to find the maximum achievable throughput, we launch 40 threads in parallel, with each sending a request of batch size b to both the embedding model and the vector database. We leave the discussion of LLM to a later section. To reflect realistic serving performance, we deploy the embedding model using Ollama [45], an open-source framework. To provide a comprehensive view of system responsiveness, we report both the average latency, which captures typical performance, and the 99th percentile (p99) latency, which reflects tail behavior and worst-case delays. We examine the scalability of the system by varying batch sizes in powers of 2 from 1 to 32. System throughput is computed as the total number of queries (number of requests $\times b$) divided by the total elapsed time to complete all requests. Latency

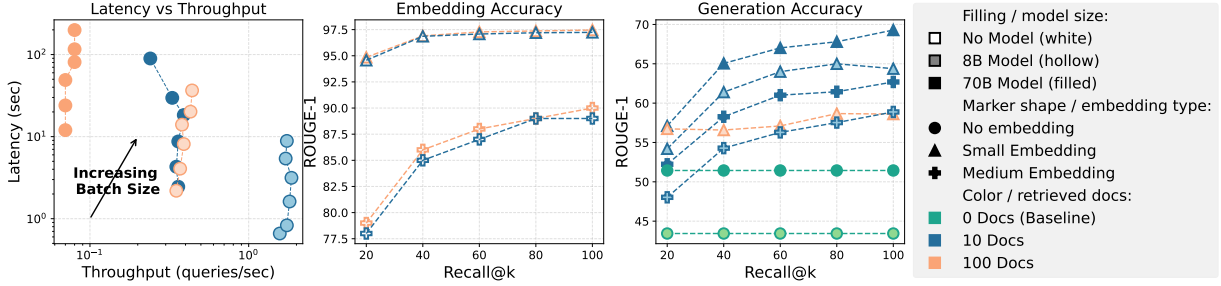


Fig. 5: Latency, embedding accuracy, and generation accuracy under different RAG configurations.

is measured on a per-query basis. Table IV below lists RAG system component configurations for this set of experiments.

TABLE IV: Q1 Experimental Configs

Component	Configurations
Batch Size	1, 2, 4, 8, 16, 32
Vector Embedding Dimension	S (d = 384), M (d = 1586)
VectorDB	Milvus, Postgres
Indexing Algorithm	HNSW, IVF-PQ

Observations Figure 3 plots the average and p99 latencies (top row and bottom row respectively) vs. throughput of RAG queries using the *nq-dev* dataset retrieving Top 10 and Top 100 vectors varying batch sizes (each point in the figure represents a batch size, 1-32 going from left to right) and vector embedding dimensions (small and medium). Both VectorDBs are plotted, as well as the performance of embedding vectors. Data points (i.e., RAG system configurations) closer to the bottom right of the figures are better (i.e., higher throughput and lower latency). Note that in rare cases, embedding vectors become system bottlenecks.

Average latency At a small batch size, average embedding generation latency dominates both medium and small size vectors. Despite the slightly longer latency of IVF-PQ under medium vectors, Milvus HNSW and IVF-PQ have very similar latency. Since Postgres is not an in-memory database, its HNSW latency is approximately three orders of magnitude larger than Milvus.

P99 latency Embedding generation on GPU shows less difference between P99 latency and average latency. This means GPU embedding generation has a more stable and better worst-case performance.

Scalability Generally, increasing the batch size per request of embedding generation and Milvus vector search results in a better throughput until reaching the maximum achievable throughput. At this threshold, the latency continues to increase without any gain in throughput because the requests are stalling for resources. For small vectors, we do not observe this threshold in vector search, even with a batch size of 32. Consequently, GPU embedding generation throughput does not increase after a batch size of eight. For medium vectors, both phases reach maximum throughput at batch size of 8.

HNSW scales slightly better than IVF-PQ, especially when the number of dimensions is higher.

TopK Search Performance IVF-PQ search performance is not affected by the choice of k because the algorithm computes the approximate distance of all pairs and sorts them to obtain the first k results. However, k has a significant impact on HNSW search performance, even on small vectors. This is because in HNSW, the graph traversal needs to run through more nodes until finding k valid candidates. This means the number of distance computations and pointer accesses scales with the value of k .

Q2. How does the choice of each RAG component affect the retrieval recall rate?

Experimental setup While ANNS gives better search performance, its imperfect retrieval recall may cause generation with bad quality. For all six different indices we build, we run all 2800+ questions in the *nq-dev* and compare the retrieved documents with ENNS ground truth. Table V below lists RAG system component configurations for this set of experiments. The results are presented in Figure 4.

TABLE V: Q2 Experimental Configs

Component	Configurations
Indexing Algorithm Effort	0, 1, 2 (low, med, high) effort
Vector Embedding Dimension	S (d = 384), M (d = 1586)
VectorDB	Milvus
Indexing Algorithm	HNSW, IVF-PQ

Observations Although HNSW0 is usually believed to generate a relatively simple graph, it already achieves more than 90% recall for both medium and small vectors. Building a more complex graph under this scenario is not rewarding. However, IVF-PQ_2 achieves a sufficient recall rate at around 70% to support the LLM generation according to our previous findings. IVF-PQ performs slightly better on small vectors. Because we keep the same parameters for vectors of both sizes, the medium vectors are quantized more aggressively compared to the small vectors.

Q3. What retrieval configuration yields the best tradeoff between efficiency and generation accuracy in RAG systems?

Experimental Setup To identify the most practical retrieval configuration for RAG systems, we evaluate the impact of

TABLE VI: Q3 Experimental Configs

Component	Configurations
# of Retrieved Chunks	Top 10, Top 100
Vector Embedding Dimension	S (d = 384), M (d = 1586)
VectorDB	Agnostic because recall rates are set
Retrieval Recall(@K)	20%, 40%, 60%, 80%, 100% (ENNS)
LLM Model	No model, 8B, 70B

three variables on downstream LLM answer quality and system performance. To evaluate the runtime tradeoffs of different LLM configurations, we measure the average latency and throughput across batch sizes for LLaMA 3.1 8B and LLaMA 3.2 70B. We define batch size as the number of LLM queries issued concurrently. This setting reflects the real-world behavior of RAG systems deployed in production environments with multiple simultaneous user requests. By systematically varying the batch size, we assess how well each model configuration scales with parallel demand. Table VI below lists the variations we examined.

Observations Figure 5 plots the average latencies vs. throughput of LLM queries with Top 10 and Top 100 retrieved chunks of both the 8B and 70B models (left). We also plot the generation accuracy of both models with different numbers of chunks at different recall rates (right). The accuracy baseline is obtained by prompting the LLMs without providing any context. To quantify and visualize the embedding accuracy, we measure and plot the ROUGE-1 score between the retrieved chunks at different recalls and the golden answers (middle).

Query Latency and Throughput The 8B model consistently outperforms the 70B model in both latency and throughput. The 8B configuration peaks at batch size 8 in throughput, matching the system’s concurrency limit, while the 70B model fails to scale with batch size due to GPU memory constraints.

As expected, Top100 document inputs result in higher latency than Top10. The only configuration pair with comparable latency and throughput is 70B with 10 documents versus 8B with 100 documents. In all other cases, the 8B model is both faster and more scalable, making it a more efficient choice for real-time or resource-constrained settings.

The 8B model peaks at batch size 8, which aligns with the system’s concurrency setting. In contrast, the 70B model shows degraded throughput beyond batch size 1, likely due to GPU memory limitations that prevent simultaneous inference. Predictably, Top100 documents lead to higher latency than Top10 across all models.

Accuracy Surprisingly, the small embedding model consistently outperforms the medium one across recall levels. This contradicts prior expectations that larger embeddings should capture semantics more effectively [66]. Upon inspecting the retrieved documents (middle plot), we found that the medium embedding model retrieved less relevant content. We suspect that our use of a chunk size of 100 tokens, which is significantly shorter than the medium model’s longer token length, prevents us from exploiting its advantage. However, the significantly larger number of dimensions exaggerates the curse of dimensionality [65].



Fig. 6: HNSW Search Latency vs. Memory Availability (Batch Size = 1).

In line with findings from prior work [52], we observe that using only the Top10 retrieved chunks consistently outperforms Top100 retrieval in generation quality (right plot). Similarly, ANN with approximately 80% recall achieves near-parity with ENN while being significantly faster. These findings indicate that RAG pipelines benefit more from targeted retrieval and prompt efficiency than from exhaustive context aggregation.

Based on these findings, we did not evaluate the configuration combining either the medium or the small embedding model, 70B LLaMA, and Top100 retrieval. These combinations would have resulted in significantly higher computational cost, while both components—medium embedding and Top100—individually underperformed their smaller or shorter counterparts. Since the small embedding model outperformed the medium one, and Top10 documents consistently yielded better results than Top100 in both latency and accuracy, this configuration was unlikely to provide additional insights and would have been dominated by other evaluated settings.

Efficiency vs. Accuracy From the results obtained, we conclude that the best tradeoff between system efficiency and generation accuracy is achieved using Top-10 documents, a small embedding model, and ANN retrieval with ~80% recall rate running on a LLaMA-3.1 8B model. This configuration offers fast, cost-effective inference without significantly sacrificing answer quality. While the 70B model achieves the highest generation accuracy across all settings, its improvement over the 8B model is modest (approximately 5%), and must be weighed against its substantially higher latency and resource requirements. These results demonstrate that scaling up individual components—such as model or embedding size—does not necessarily yield proportionate improvements in end-to-end RAG performance.

Q4. In what scenario would ENNS be acceptable in terms of system performance?

Experimental Setup and Observations Despite the high performance of ANNS, ENNS still needs to be considered when designing a RAG system. Since ENNS is running a brute-force k -nearest-neighbor algorithm in the backend, it offers two key advantages: 1) It always has a perfect recall

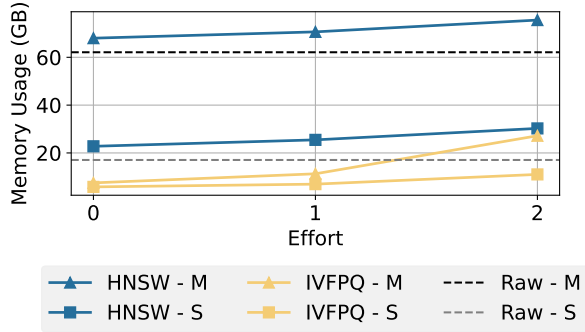


Fig. 7: Memory Requirement vs. Index Build Effort.

rate of 100%, and 2) it does not require an index build. In our experimental setup, we assume data insertion and index building are offline and do not discuss their performance, as all indices we include need multiple hours to build. However, from our experimental results, a perfect recall is not required to saturate the generation accuracy of the LLM. ENNS exhibits long latency and poor scalability, particularly when the system load is high, even in the absence of an index build. At a low system load, the best-case average latency is 1.56 seconds and 4.6 seconds for small and medium vectors, respectively. When Milvus is serving 40 clients, the average latency is approximately 40.2 and 156.4 seconds, and the throughput is 0.92 and 0.25 queries per second. This is over three orders of magnitude larger than the latency and two orders of magnitude smaller than the throughput if an ANN algorithm such as HNSW is used.

Q5. What is the RAG system performance bottleneck given a reasonable generation accuracy?

Experimental Setup Based on the experiment results we obtained, we find that under our RAG setup, a combination of small embedding, HNSW indexing using low effort or IVF-PQ indexing using high effort, and a Llama 3.1 8B model achieved the best balance between generation accuracy and system performance. To understand the system latency bottleneck of RAG systems containing the above components, we present the latency breakdown in Figure 8. We profile the time-to-interaction, shown by time-to-first token (TTFT) [56] for the LLMs, retrieval latency, and embedding generation latency, and report the ratios for three phases.

Observations Our results show that using an in-memory database like Milvus with ANNS, the LLM generation latency always dominates the end-to-end response time. Also, LLM clearly scales worse than vector search when the batch size is larger. This suggests that more resources need to be allocated to host LLMs, and one powerful CPU might be able to serve vector retrieval for many GPUs.

Q6. What is the memory footprint of different search algorithms, and how sensitive is their performance to available memory?

Problem Statement Although HNSW offers superior latency, throughput scalability, and near-perfect recall, it requires

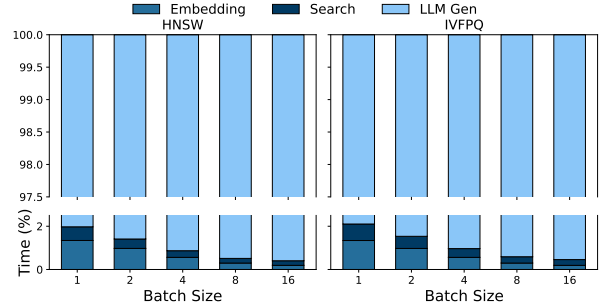


Fig. 8: Latency breakdown of real-time embedding generation, vector search, and LLM generation

access to all original vectors and the entire index graph due to its graph-based design without quantization. This leads to high memory usage, especially in in-memory databases like Milvus for high-performance applications.

Scaling system memory is both costly and limited. Memory capacity is constrained by the number of DIMMs, which in turn is bounded by the number of memory channels and CPU socket pins. The cost of higher-capacity DIMMs increases non-linearly with memory size. For example, a 256GB DDR4 DIMM is much more expensive per GB than a 64GB DIMM. This non-linear cost scaling makes memory one of the most expensive resources in modern systems.

Experimental Setup To evaluate its sensitivity to available system memory, we profile the search latency of HNSW_0 (i.e., low effort HNSW indexing) on medium-sized vectors using a batch size of 1, while systematically restricting the memory accessible to the Milvus runtime. We select the medium embedding vectors because it has higher memory requirements, providing a broader range for system memory limitations. We also include IVF-PQ_2 (i.e., high effort IVF-PQ indexing) running in Milvus and HNSW_0 running in Postgres for comparison.

Observations Figure 7 shows the memory footprint of all 12 ANNS algorithms alongside the corresponding raw data size. Intuitively, IVF-PQ uses less memory than the raw data due to product quantization, while HNSW consumes more due to the additional graph structure. Memory usage grows approximately linearly with the index-building parameter values. Although HNSW scales better and achieves higher recall, its search performance degrades by more than two orders of magnitude when system memory is limited, as shown in figure 6. Postgres requires less memory than Milvus to maintain similar performance, thanks to its tuple-at-a-time execution model, which is more memory-efficient. As discussed earlier, IVF-PQ_2 achieves sufficient recall for LLMs to generate accurate responses. While its throughput scalability and worst-case latency are slightly worse than HNSW, it remains competitive and cost-effective, especially under memory-constrained scenarios.

Q7. Roofline Analysis

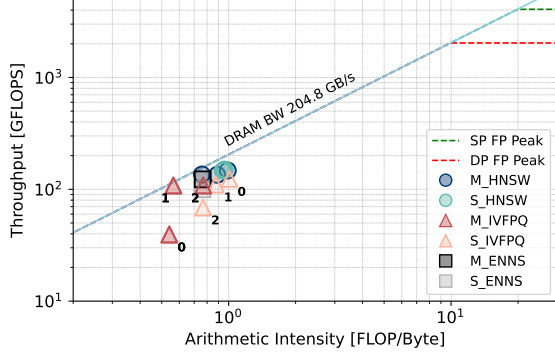


Fig. 9: Roofline model for all vector retrieval algorithms: ANNS with all three efforts are plotted with IVFPQ’s efforts marked

Experimental Setup The roofline model is used to provide hardware limitations on application performance and usually gives insight into future hardware design to accelerate the application kernels. We perform roofline analysis on all search algorithms we run on Milvus by sending as many search queries as possible (limited to 16384 per API call in Milvus) and profiling the floating-point operation throughput and the memory bandwidth of the system.

Observations Figure 9 shows the roofline model of 12 ANNS and 2 ENNS algorithms on both small and medium vectors. All data points stay below the slope instead of the flat line, indicating retrieval algorithms are memory bandwidth-bound. We observe that ENNS and HNSW exhibit more stable roofline characteristics compared to IVFPQ, whose arithmetic intensity and achievable bandwidth do not correlate with search effort. Overall, the arithmetic intensity corresponds to the computational pattern of distance computations, in which the distance between each pair of dimensions is computed once. However, because we search multiple query vectors simultaneously, one reference vector fetched from the database memory should ideally be used more than once, as long as the cache size permits. The result implies that reference vector reuse is not fully exploited. This observation aligns with recent works [35], [51], [64] that accelerate NNS algorithms using processing-in/near-memory, as it directly addresses the bandwidth bottleneck.

Q8. What are the CPU and GPU power consumption tradeoffs in RAG systems?

Problem Statement Total Cost of Ownership (TCO) is an important aspect of modern LLM system design. We do not provide a RAG-TCO model directly because it depends on the actual hardware devices used to host the RAG system. However, power consumption directly affects TCO. Besides TCO, power is essential to an edge RAG system device. Understanding how different RAG components consume power allows developers to identify energy bottlenecks and optimize for cost-efficiency.

Experimental Setup Obtaining exact RAG system power consumption and its breakdown is challenging: the non-

Workload	CPU Package	per DIMM($\times 8$)	GPU
HNSW_0 (Milvus)	207.67	10.64-19.10	-
HNSW_0 (pgvector)	94.84	4.57-5.17	-
IVFPQ_2 (Milvus)	218.70	8.02-13.04	-
Embedding (Small)	-	-	299.21
LLM (8B)	-	-	354.89

TABLE VII: Measured CPU and GPU power (in Watts) at saturated throughput.

trivial off-chip communication, such as CPU-GPU through PCIe, incurs significant power, which is hard to model or profile. However, we can approximately decompose total system power into: CPU package power, DRAM power, and GPU power. We measure CPU and GPU power consumption using AMD μ Prof for CPU package power and nvidia-smi for GPU power metrics. We report DRAM power using Micron’s DDR4 power calculator [40], which estimates power analytically from DIMM datasheet parameters [41] and runtime statistics such as bus utilization and row buffer hit rate. Bus utilization is derived as the ratio of profiled bandwidth to the theoretical maximum bandwidth, under the assumption of evenly distributed accesses across all DRAM channels. Accurately capturing row buffer hit rate would require instruction-level simulation; therefore, we instead report a power range by considering two extremes, 0% and 100% hit rates. We present results for the representative balanced configurations discussed in Q5: a small embedding model with HNSW_0, IVF-PQ_2, and an 8B LLM with Top-10 fetched documents, while saturating throughput in each component.

Observations Table VII shows that Milvus HNSW_0 and IVFPQ_2 consume similar levels of CPU power, while HNSW_0 exhibits higher DRAM power, due to higher memory bandwidth utilization. This may be related to the fact that HNSW fetches entire vectors from memory when computing distances. In contrast, pgvector’s HNSW consumes significantly less CPU and DRAM power, which can be attributed to its lower performance and smaller memory footprint. The embedding model requires less GPU power due to its significantly smaller model size compared to LLMs. As expected, LLM inference is the most power-hungry stage, reaching over 350 W on the H100 GPU. In practice, LLM inference will account for an even larger share of the power budget, since it is also the performance bottleneck of the overall pipeline.

V. EXAMPLE CASE STUDY USING ATHENA

We showcase one example using Athena on long-context processing. The details of this scenario are discussed in Section II-E. At a high level, there are three different ways of building a LLM system that allows long-context processing: 1) directly inject the long document into LLM prompts as a context window, 2) pre-process the document, insert it into the vectorDB, build an index, and prompt the LLM using the original question augmented with retrieval information, and

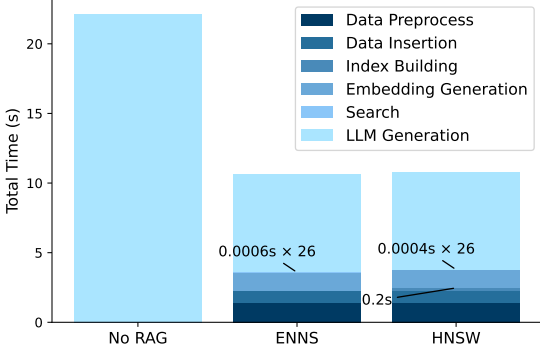


Fig. 10: End-to-end latency breakdown of a long-context processing scenario with and without RAG

3) use the second option but do not build an index (i.e., use an ENNS algorithm).

The first option is straightforward, but the large number of tokens in the LLM prompt dilutes the density of useful information presented and significantly increases the prefill time. The second option follows the same pipeline as regular RAG for knowledge-intensive tasks on large datasets. However, building the index in real time adds more latency, so it may fail the minimal response time requirement. Option 3 has no index building time involved and gives perfect recall on nearest neighbor search, but incurs additional retrieval latency.

Experimental Setup We construct all three RAG systems using Athena. We use the most efficient combination of system components of small vector embedding, Llama3.1-8B model, and HNSW_0 with Top10 chunks as discussed in Section IV. We chose Postgres as our backend database for benchmarking instead of Milvus, as Milvus implements some segment compaction optimizations that may degrade performance.

We build a micro dataset from Google NQ by collecting Wikipedia articles on NFL Super Bowls such that the total number of tokens exceeds 100K, and apply the same data preprocessing as discussed previously. We extract 26 questions and corresponding golden answers. To study the influence of different context lengths, we further extend the micro dataset to 1 million tokens ($10\times$) by inserting random documents from our dataset.

Observations As shown in Figure 10, the LLM response time of directly processing the long document (i.e., Option 1) is $3\times$ longer than that of using the Top10 retrieved documents (i.e., Option 2) because of the additional number of input tokens. Using RAG only incurs a one-time additional data processing time of 2.27s (1.39s for embedding the entire documents and 0.88s for inserting the data into Postgres). At search time, an additional 0.05s is spent converting the query plain-text to a vector embedding and performing the search. Building an HNSW takes 0.2s and gives a 92% recall rate. However, it only brings a negligible sub-millisecond benefit per query. Based on our previous scalability analysis, an ENNS algorithm

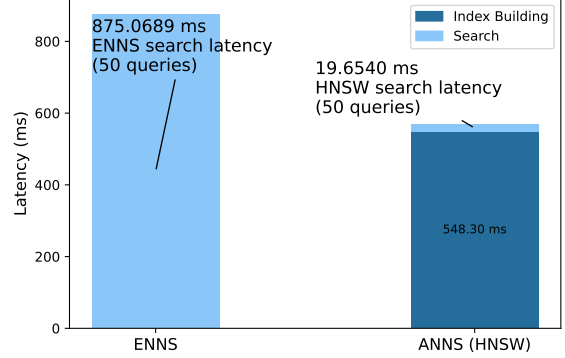


Fig. 11: At a larger scale long-context processing, building an HNSW index beats the performance of brute-force ENNS.

is sufficient as long as this system does not serve multiple users. On our micro-dataset, long-context processing with RAG achieves 65% generation accuracy. Although the LLM can answer most questions by directly injecting the documents, its response indicates that the answer is not found in the provided text, making Option 1 undesirable.

As shown in Figure 11, we also present a case in which choosing ANNS is beneficial. We perform the same set of experiments on the extended dataset and find that with enough number of queries (50), ANNS using HNSW results in a $1.53\times$ speedup compared to brute-force ENNS. The existence of this turning point highlights the importance of Athena, which can guide developers in selecting the appropriate strategy for different workloads.

VI. RELATED WORK

Prior works on RAG mainly optimize some components relevant to RAG, whereas Athena provides the first benchmark framework for RAG system design. We highlight those that emphasize systematic analysis and design from both software and hardware perspectives.

RAGO [20] designs a system optimization framework for efficient RAG serving on Google server infrastructures. As one of the first works that analyzes RAG systematically, the authors present RAGSchema, which provides a modular abstraction that defines RAG systems for different use cases. RAGO also proposes optimizations on task placement, resource allocation, and batching policy. CHASE [37] is a query search engine optimized for vector databases that execute hybrid queries. CHASE rewrites hybrid query plans generated by a traditional optimizer, introduces new vector database-specific operators, and incorporates data-centric code generation [24] to accelerate query execution.

Chameleon [21] is a heterogeneous accelerator system for RAG with both LLM and vector search. It implements an IVF-PQ accelerator on an FPGA and serves LLM on a GPU cluster. The work disaggregates parts of IVF-PQ scan and executes them on the GPUs. IKS [51] is a CXL-enabled Process Near Memory (PNM) accelerator system designed for efficient RAG inference. The work promotes ENNS over

ANNS due to its ease of acceleration and perfect recall rate, which supports better generation accuracy. It integrates multiple CXL Type-2 devices, each equipped with an ENNS accelerator, communicating with the host via a cache-coherent CXL.mem/cache interface to improve scalability.

VII. CONCLUSION

To support current and future research on Retrieval-Augmented Generation (RAG) systems, there is an increasing need for tools that offer systematic insights into the vast design space. We introduce Athena, an end-to-end RAG system advisor and benchmarking framework with plug-and-play capabilities. Athena supports a wide range of components commonly used in modern RAG pipelines and is highly configurable to accommodate arbitrary new designs. Through a variety of RAG pipelines built with Athena, we characterize key system behaviors, such as latency, throughput, recall rate, memory usage, power consumption, and generation accuracy, providing a deeper understanding of RAG performance tradeoffs. Additionally, we demonstrate how Athena can be easily adapted to support diverse design objectives through a long-context processing use case. We believe this work lays a foundation for more systematic and informed RAG system design in the future.

REFERENCES

- [1] M. AI, "Llama 3: Open foundation and instruction models," <https://ai.meta.com/llama>, 2024, accessed: 2025-06-30.
- [2] AMD, "Amd epyc™ 4th generation 9004 & 8004 series server processors," <https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series.html>, 2024, accessed: 2025-06-30.
- [3] Beijing Academy of Artificial Intelligence (BAAI), "Bge models: Baai general embeddings," 2024, accessed: 2025-06-30. [Online]. Available: <https://bge-model.com/>
- [4] I. Beltagy, K. Lo, and A. Cohan, "Scibert: A pretrained language model for scientific text," *arXiv preprint arXiv:1903.10676*, 2019.
- [5] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. van den Driessche, J.-B. Lespiau, B. Damoc, A. Clark, D. de Las Casas, A. Guy, J. Menick, R. Ring, T. Hennigan, S. Huang, L. Maggiore, C. Jones, A. Cassirer, A. Brock, M. Paganini, G. Irving, O. Vinyals, S. Osindero, K. Simonyan, J. W. Rae, E. Elsen, and L. Sifre, "Improving language models by retrieving from trillions of tokens," 2022. [Online]. Available: <https://arxiv.org/abs/2112.04426>
- [6] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. B. Van Den Driessche, J.-B. Lespiau, B. Damoc, A. Clark, D. De Las Casas, A. Guy, J. Menick, R. Ring, T. Hennigan, S. Huang, L. Maggiore, C. Jones, A. Cassirer, A. Brock, M. Paganini, G. Irving, O. Vinyals, S. Osindero, K. Simonyan, J. Rae, E. Elsen, and L. Sifre, "Improving language models by retrieving from trillions of tokens," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 2206–2240. [Online]. Available: <https://proceedings.mlr.press/v162/borgeaud22a.html>
- [7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: <https://arxiv.org/abs/2005.14165>
- [9] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.
- [10] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: <https://doi.org/10.18653/v1/n19-1423>
- [11] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations (ICLR)*, 2021. [Online]. Available: <https://openreview.net/forum?id=YicbFdNTTy>
- [12] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.
- [13] D. Edge, H. Trinh, N. Cheng, J. Bradley, A. Chao, A. Mody, S. Truitt, and J. Larson, "From local to global: A graph rag approach to query-focused summarization," *arXiv preprint arXiv:2404.16130*, 2024.
- [14] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang, "Retrieval-augmented generation for large language models: A survey," 2024. [Online]. Available: <https://arxiv.org/abs/2312.10997>
- [15] M. Glass, G. Rossiello, M. F. M. Chowdhury, A. R. Naik, P. Cai, and A. Gliozzo, "Re2g: Retrieve, rerank, generate," *arXiv preprint arXiv:2207.06300*, 2022. [Online]. Available: <https://arxiv.org/abs/2207.06300>
- [16] K. Guu, K. Lee, Z. Tung, P. Pasupat, and M.-W. Chang, "Realm: Retrieval-augmented language model pre-training," *arXiv preprint arXiv:2002.08909*, 2020.
- [17] Hugging Face, "Transformers: State-of-the-art machine learning for pytorch, tensorflow, and jax," 2024, accessed: 2025-06-30. [Online]. Available: <https://github.com/huggingface/transformers>
- [18] G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, P. Merlo, J. Tiedemann, and R. Tsarfaty, Eds. Online: Association for Computational Linguistics, Apr. 2021, pp. 874–880. [Online]. Available: <https://aclanthology.org/2021.eacl-main.74/>
- [19] G. Izacard, P. Lewis, M. Lomeli, L. Hosseini, F. Petroni, T. Schick, J. Dwivedi-Yu, A. Joulin, S. Riedel, and E. Grave, "Atlas: few-shot learning with retrieval augmented language models," *J. Mach. Learn. Res.*, vol. 24, no. 1, Jan. 2023.
- [20] W. Jiang, S. Subramanian, C. Graves, G. Alonso, A. Yazdanbakhsh, V. Dadu, and S. Subramanian, "RAGO: A systematic framework for designand optimization of retrieval-augmented generation serving," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2025.
- [21] W. Jiang, M. Zeller, R. Waleffe, T. Hoefler, and G. Alonso, "Chameleon: A heterogeneous and disaggregated accelerator system for retrieval-augmented language models," vol. 18, no. 1, p. 42–52, Sep. 2024. [Online]. Available: <https://doi.org/10.14778/3696435.3696439>
- [22] Y. Jin, Y. Wu, W. Hu, B. M. Maggs, X. Zhang, and D. Zhuo, "Curator: Efficient indexing for multi-tenant vector databases," 2024. [Online]. Available: <https://arxiv.org/abs/2401.07119>
- [23] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," 2017, accessed: 2025-06-30. [Online]. Available: <https://github.com/facebookresearch/faiss>
- [24] M. Jungmaier, A. Kohn, and J. Giceva, "Designing an open framework for query optimization and compilation," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2389–2401, 2022. [Online]. Available: <https://www.vldb.org/pvldb/vol15/p2389-jungmaier.pdf>
- [25] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 1, pp. 117–128, 2011.
- [26] V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih, "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association

- for Computational Linguistics, Nov. 2020, pp. 6769–6781. [Online]. Available: <https://aclanthology.org/2020.emnlp-main.550/>
- [27] T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, M. Kelcey, J. Devlin, K. Lee, K. N. Toutanova, L. Jones, M.-W. Chang, A. Dai, J. Uszkoreit, Q. Le, and S. Petrov, “Natural questions: A benchmark for question answering research,” *Transactions of the Association of Computational Linguistics*, 2019.
- [28] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, “Efficient memory management for large language model serving with pagedattention,” 2023. [Online]. Available: <https://arxiv.org/abs/2309.06180>
- [29] C. Lee, R. Roy, M. Xu, J. Raiman, M. Shoeybi, B. Catanzaro, and W. Ping, “Nv-embed: Improved techniques for training llms as generalist embedding models,” 2025. [Online]. Available: <https://arxiv.org/abs/2405.17428>
- [30] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang, “BioBERT: A pre-trained biomedical language representation model for biomedical text mining,” *Bioinformatics*, 2020.
- [31] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive nlp tasks,” 2021. [Online]. Available: <https://arxiv.org/abs/2005.11401>
- [32] J. Li, Y. Yuan, and Z. Zhang, “Enhancing llm factual accuracy with rag to counter hallucinations: A case study on domain-specific queries in private knowledge-bases,” 2024. [Online]. Available: <https://arxiv.org/abs/2403.10446>
- [33] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, “Competition-level code generation with alphacode,” *Science*, 2022.
- [34] C.-Y. Lin, “ROUGE: A package for automatic evaluation of summaries,” in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. [Online]. Available: <https://aclanthology.org/W04-1013/>
- [35] C. Liu, H. Liu, D. Chen, Y. Huang, Y. Zhang, W. Xiao, X. Liao, and H. Jin, “Heterag: Heterogeneous processing-in-memory acceleration for retrieval-augmented generation,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 884–898. [Online]. Available: <https://doi.org/10.1145/3695053.3731089>
- [36] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
- [37] R. Ma, K. Zhang, Z. He, Y. Jing, X. S. Wang, and Z. Chen, “Chase: A native relational database for hybrid queries on structured and unstructured data,” 2025. [Online]. Available: <https://arxiv.org/abs/2501.05006>
- [38] X. Ma, Y. Gong, P. He, H. Zhao, and N. Duan, “Query rewriting for retrieval-augmented large language models,” *arXiv preprint arXiv:2305.14283*, 2023. [Online]. Available: <https://arxiv.org/abs/2305.14283>
- [39] Y. A. Malkov and D. A. Yashunin, “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, p. 824–836, Apr. 2020. [Online]. Available: <https://doi.org/10.1109/TPAMI.2018.2889473>
- [40] Micron Technology, Inc., “Micron dram power calculator,” <https://www.micron.com/sales-support/design-tools/dram-power-calculator>, 2025, accessed: 2025-09-04.
- [41] —, “Micron mt40a2g4sa-075-e ddr4 sdram 8 gb 2 g × 4,” <https://www.digikey.com/en/products/detail/micron-technology-inc/MT40A2G4SA-075-E/7597789>, 2025, part details retrieved from Digi-Key; device specifics include DDR4, 8 Gbit, 2 G × 4, 1.33 GHz, 1.14–1.26 V supply :contentReference[oaicite:0]index=0.
- [42] Microsoft, “Azure virtual machines ncads h100 v5 series,” 2024, accessed: 2025-06-30. [Online]. Available: <https://learn.microsoft.com/en-us/azure/virtual-machines/ncads-h100-v5>
- [43] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, “Mteb: Massive text embedding benchmark,” *arXiv preprint arXiv:2210.07316*, 2022. [Online]. Available: <https://arxiv.org/abs/2210.07316>
- [44] —, “Mteb: Massive text embedding benchmark,” 2023. [Online]. Available: <https://arxiv.org/abs/2210.07316>
- [45] Ollama, “Ollama: Run large language models locally,” 2023, accessed: 2025-06-29. [Online]. Available: <https://github.com/ollama/ollama>
- [46] Ollama, “Llama-3.1,” 2025, accessed: 2025-06-30. [Online]. Available: <https://ollama.com/library/llama3.1>
- [47] OpenAI, “Gpt-4 technical report,” <https://openai.com/research/gpt-4>, 2023, accessed: 2025-06-30.
- [48] F. Petroni, A. Piktus, A. Fan, P. Lewis, M. Yazdani, N. De Cao, J. Thorne, Y. Jernite, V. Karpukhin, J. Maillard, V. Plachouras, T. Rocktäschel, and S. Riedel, “KILT: a benchmark for knowledge intensive language tasks,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds. Online: Association for Computational Linguistics, Jun. 2021, pp. 2523–2544. [Online]. Available: <https://aclanthology.org/2021.naacl-main.200/>
- [49] pgvector Contributors, “pgvector: Open-source vector similarity search for postgres,” 2025, accessed: 2025-04-22. [Online]. Available: <https://github.com/pgvector/pgvector>
- [50] G. Poesia, O. Polozov, V. Le, A. Tiwari, G. Soares, C. Meek, and S. Gulwani, “SynchroMesh: Reliable code generation from pre-trained language models,” *arXiv preprint arXiv:2201*.
- [51] D. Quinn, M. Nouri, N. Patel, J. Salihu, A. Salemi, S. Lee, H. Zamani, and M. Alian, “Accelerating retrieval-augmented generation,” in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 15–32. [Online]. Available: <https://doi.org/10.1145/3669940.3707264>
- [52] B. Roziere, G. Izacard, A. Piktus, D. Hessel, G. Lample, F. Simig, Y. Ru, R. McHardy, V. Kocijian, N. Roberts *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2307.03172*, 2023.
- [53] R. Shao, J. He, A. Asai, W. Shi, T. Dettmers, S. Min, L. Zettlemoyer, and P. W. Koh, “Scaling retrieval-based language models with a trillion-token datastore,” 2024. [Online]. Available: <https://arxiv.org/abs/2407.12854>
- [54] W. Shi, S. Min, M. Yasunaga, M. Seo, R. James, M. Lewis, L. Zettlemoyer, and W. tau Yih, “REPLUG: Retrieval-augmented black-box language models,” in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for Computational Linguistics, June 2024, pp. 8371–8384. [Online]. Available: <https://aclanthology.org/2024.naacl-long.463/>
- [55] I. S. Singh, R. Aggarwal, I. Allahverdiyev, M. Taha, A. Akalin, K. Zhu, and S. O’Brien, “Chunkrag: Novel llm-chunk filtering method for rag systems,” 2024. [Online]. Available: <https://arxiv.org/abs/2410.19572>
- [56] J. Stojkovic, E. Choukse, C. Zhang, I. Goiri, and J. Torrellas, “Towards greener llms: Bringing energy-efficiency to the forefront of llm inference,” 2024. [Online]. Available: <https://arxiv.org/abs/2403.20306>
- [57] S. J. Subramanya, Devvrit, R. Kadekodi, R. Krishaswamy, and H. V. Simhadri, *DiskANN: fast accurate billion-point nearest neighbor search on a single node*. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [58] R. Taylor, M. Kardas, G. Cucurull, T. Scialom, A. Hartshorn, E. Saravia, A. Poulton, V. Kerkez, and R. Stojnic, “Galactica: A large language model for science,” *arXiv preprint arXiv:2211.09085*, 2022.
- [59] N. Thakur, N. Reimers, A. Rücklé, A. Srivastava, and I. Gurevych, “Beir: A heterogeneous benchmark for zero-shot evaluation of information retrieval models,” 2021. [Online]. Available: <https://arxiv.org/abs/2104.08663>
- [60] The PostgreSQL Global Development Group, *PostgreSQL 17.4 Documentation*, PostgreSQL Global Development Group, 2025, accessed: 2025-04-22. [Online]. Available: <https://www.postgresql.org/docs/current/>
- [61] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, L. Gutierrez, T. F. Tan, and D. S. W. Ting, “Large language models in medicine,” *Nature Medicine*, vol. 29, no. 3, pp. 535–544, 2023.
- [62] B. Wang, W. Ping, L. McAfee, P. Xu, B. Li, M. Shoeybi, and B. Catanzaro, “Instructretro: Instruction tuning post retrieval-augmented pretraining,” 2024. [Online]. Available: <https://arxiv.org/abs/2310.07713>
- [63] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, “Milvus: A purpose-built vector data management system,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD 21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2614–2627. [Online]. Available: <https://doi.org/10.1145/3448016.3457550>
- [64] Y. Wang, S. Li, Q. Zheng, L. Song, Z. Li, A. Chang, H. H. Li, and Y. Chen, “Ndsearch: Accelerating graph-traversal-based approximate nearest neighbor search through near data processing,” in *2024 ACM/IEEE 51st*

- Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 368–381.
- [65] Wikipedia contributors, “Curse of dimensionality,” accessed: 2025-06-30. [Online]. Available: https://en.wikipedia.org/wiki/Curse_of_dimensionality
 - [66] J. Yang, J. Wan, Y. Yao, W. Chu, Y. Xu, and Y. Qi, “Infretrieverv11.5b,” 2025, revision 5f469d7; Accessed 2025-06-30. [Online]. Available: <https://huggingface.co/infly/inf-retriever-v1-1.5b>
 - [67] W. Yang, T. Li, G. Fang, and H. Wei, “Pase: Postgresql ultra-high-dimensional approximate nearest neighbor search extension,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2241–2253. [Online]. Available: <https://doi.org/10.1145/3318464.3386131>
 - [68] N.-A. Ypsilantis, K. Chen, B. Cao, M. Lipovský, P. Dogan-Schönberger, G. Makosa, B. Bluntschli, M. Seyedhosseini, O. Chum, and A. Araujo, “Towards universal image embeddings: A large-scale dataset and challenge for generic image representations,” 2023. [Online]. Available: <https://arxiv.org/abs/2309.01858>
 - [69] Z. Yue, H. Zhuang, A. Bai, K. Hui, R. Jagerman, H. Zeng, Z. Qin, D. Wang, X. Wang, and M. Bendersky, “Inference scaling for long-context retrieval augmented generation,” 2025. [Online]. Available: <https://arxiv.org/abs/2410.04343>
 - [70] Q. Zhang, S. Xu, Q. Chen, G. Sui, J. Xie, Z. Cai, Y. Chen, Y. He, Y. Yang, F. Yang, M. Yang, and L. Zhou, “VBASE: Unifying online vector similarity search and relational queries via relaxed monotonicity,” in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA, USA: USENIX Association, 2023, pp. 377–395. [Online]. Available: <https://www.usenix.org/conference/osdi23/presentation/zhang-qianxi>