In-Storage Acceleration of Retrieval Augmented Generation as a Service

Rohan Mahapatra
University of California San Diego
La Jolla, California, USA
rmahapat@ucsd.edu

Harsha Santhanam
University of California San Diego
La Jolla, California, USA
hsanthanam@ucsd.edu

Christopher Priebe
University of California San Diego
La Jolla, California, USA
cpriebe@ucsd.edu

Hanyang Xu
University of California San Diego
La Jolla, California, USA
hanyang@ucsd.edu

Hadi Esmaeilzadeh
University of California San Diego
La Jolla, California, USA
hadi@ucsd.edu

Abstract

Retrieval-augmented generation (RAG) services are rapidly gaining adoption in enterprise settings as they combine information retrieval systems (e.g., databases) with large language models (LLMs) to enhance response generation and reduce hallucinations. By augmenting an LLM's fixed pre-trained knowledge with real-time information retrieval, RAG enables models to effectively extend their context to large knowledge bases by selectively retrieving only the most relevant information. As a result, RAG provides the effect of dynamic updates to the LLM's knowledge without requiring expensive and time-consuming retraining. While some deployments keep the entire database in memory, RAG services are increasingly shifting toward persistent storage to accommodate ever-growing knowledge bases, enhance utility, and improve cost-efficiency. However, this transition fundamentally reshapes the system's performance profile: empirical analysis reveals that the *Search & Retrieval* phase emerges as the dominant contributor to end-to-end latency. This phase typically involves (1) running a smaller language model to generate query embeddings, (2) executing similarity and relevance checks over varying data structures, and (3) performing frequent, long-latency accesses to persistent storage. To address this triad of challenges, we propose a metamorphic in-storage accelerator architecture that provides the necessary programmability to support diverse RAG algorithms, dynamic data structures, and varying computational patterns. The architecture also supports in-storage execution of smaller language models for query embedding generation while final LLM generation is executed on DGX A100 systems. Experimental results show up to 4.3× and 1.5× improvement in end-to-end throughput compared to conventional retrieval pipelines using Xeon CPUs with NVMe storage and A100 GPUs with DRAM, respectively.

CCS Concepts

• **Information systems** → *Information retrieval*; *Cloud based storage*; • **Computer systems organization** → **Architectures**; **Cloud computing**; • **Computing methodologies** → *Artificial intelligence*; • **Hardware** → *Hardware accelerators.*

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731032

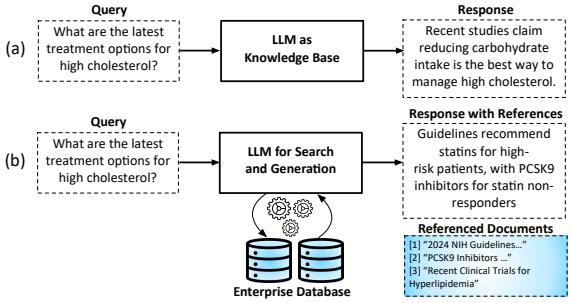


Figure 1: (a) Traditional LLMs generate responses using only the knowledge encoded in their training data, which limits contextual relevance and may lead to hallucinations, whereas (b) retrieval-augmented generation (RAG) enhances LLMs with real-time search and retrieval from enterprise databases, enabling extended context and up-to-date responses without requiring retraining.

Keywords

Retrieval-Augmented Generation, RAG, Large Language Models, LLM, In-Storage Acceleration, Specialized Accelerators

ACM Reference Format:
Rohan Mahapatra, Harsha Santhanam, Christopher Priebe, Hanyang Xu, and Hadi Esmaeilzadeh. 2025. In-Storage Acceleration of Retrieval Augmented Generation as a Service. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3695053.3731032

1 Introduction

Retrieval-augmented generation (RAG) [44] combines the generative power [84] of large language models (LLMs) with the precision and contextual relevance of information retrieval systems such as databases, knowledge bases, and web pages. As such, RAG services (referred to as RAGs) enable LLMs to provide verifiable responses that come with citations to specific sources in the database and reduce the risk of hallucinations [26, 83] that could cause serious legal, financial, and intellectual issues in enterprise applications. Furthermore, RAGs augment LLMs with an organization's internal database and help maintain timeliness by providing them with up-to-date information without the high cost and challenges of fine-tuning or retraining [66]. These advantages have fueled the rapid adoption of RAGs in healthcare [59, 94], finance [93], law [68], and scientific publishing [1], where staying factual and current with evolving information is essential.

In a sense, retrieval-augmented generation enables LLMs to effectively extend their context window with the content of a complete database. The context window is the range of tokens that an LLM can consider when generating its response. Increasing the context window length is costly, as the transformer layer, the core component of an LLM, has a time complexity that increases quadratically with the context window length [85]. As such, simply expanding the context window to fit the entire database's contents becomes computationally infeasible. Although recent advances (e.g., Google's Gemini 1.5 Pro) have extended context lengths to two million tokens [21], long-context LLMs remain inefficient as context size increases, due to high computational costs and declining accuracy [31], making them unsuitable for reasoning over knowledge bases. Thus, RAGs continue to play a critical role in integrating external knowledge into generative applications. Rather than physically increasing the context window, RAGs search the database and only inject relevant information into the context window to construct an augmented query. Some deployments even pay the high cost of keeping the entire database in memory [5, 72] for responsiveness. However, RAGs are increasingly shifting toward using persistent storage to accommodate ever-growing knowledge bases, enhance utility, and significantly improve cost-efficiency [3, 5, 9, 22, 30, 38, 75, 78, 87, 91].

While much of the recent research has focused on accelerating LLM inference [8, 14, 25, 27, 28, 34, 43, 67, 70, 71, 92], our analysis of end-to-end RAGs on Amazon Web Services indicates a different bottleneck. Based on experiments using NVIDIA DGX A100 GPUs with the PubMed dataset (5 million documents) and various similarity metrics and retrieval algorithms, we find that, on average, 61% of the total runtime is spent in the *Search & Retrieval* phase rather than LLM inference. This phase typically consists of: (1) generating query embeddings using a language model to enable relevance evaluation; (2) employing various data structures and algorithms for similarity and relevance checks, which may differ across datacenter deployments; and (3) interleaving these computations with frequent, long-latency storage accesses, which can dominate the overall RAG pipeline.

To address this triad of challenges, we propose a shape-shifting metamorphic in-storage accelerator architecture to provide the necessary programmability and dynamism to support various RAG algorithms and their dynamic data structures and shapes. This architecture can also run the embedding generation that requires inferencing with a language model while adhering to the rather stringent thermal and power constraints of the storage device [23, 42, 55, 56]. We refer to this in-storage programmable acceleration approach for disaggregated datacenters as RAGX. This design is proposed to serve a variety of RAGs that utilize different algorithms and data structures for the similarity checks. Some rely on various distance calculations between the embedding of the original query and the entries in a vector database, which are also in the form of embeddings. Others count the frequency of keywords between the query and the entries using hash tables and inverted indices. In addition, these algorithms require data structures with data-dependent, dynamic shapes that are contingent on the contents of the query. To support dynamic algorithms and metrics in RAGs, we also propose a novel Metadata Navigation Unit that directly loads data from the NAND arrays of the storage devices into the accelerator on-chip memory.

While also offering strategies for the co-location of compute and data, we evaluated RAGX using five diverse end-to-end RAG deployments that utilize LLAMA2 (34B). Because both the content of the database and the nature of the queries can significantly affect retrieval and response quality, we use a realistic dataset containing PubMed [24, 62] documents with 50 million passages and 3,800 BioASQ [39] queries. RAGX achieves up to 4.3× end-to-end throughput improvement over a conventional deployment that uses a Xeon CPU with NVMe storage for retrieval, search, and augmentation, and DGX A100 GPUs for LLM inferencing. When the Xeon CPU with NVMe is replaced by an A100 GPU with DRAM, RAGX provides up to 1.5× end-to-end throughput improvement. We perform rigorous sensitivity studies that change the LLM from LLAMA2 (13B) to LLAMA2 (70B) and also consider different database sizes (0.5M, 5M, 50M, and 500M passages). The results show that benefits from RAGX grow as the size of the database increases, even when the LLM is largest.

2 RAG: Concept to Datacenter Deployment

Figure 2 illustrates the typical RAG pipeline on Amazon Web Services (AWS) [60, 80, 81] and its three major phases: *Search & Retrieval*, *Augmentation for Query Reconstruction*, and *Referenced Generation*. To ground the query response with concrete information in the database, the first phase searches the database to identify the most relevant entries using a similarity score that varies across RAG deployments and algorithms. Some classes of RAG algorithms use embeddings [18, 57], while others utilize keyword frequencies [16, 76]. Figure 2 represents deployment of the former class on AWS. The latter class is alike in execution with some differences discussed in §2.3. In the context of the depicted pipeline, this section offers an intuitive perspective on how the interplay between iterative storage accesses and similarity score computation affects end-to-end RAG execution and performance in a disaggregated datacenter.

2.1 Offline Vector Database Generation

Prerequisites and graph metadata: To perform similarity checks efficiently, RAGs generate a vector database representation of documents in the enterprise knowledge base [24]. This offline process is performed once and involves segmenting each document from the database into passages and embedding them using a transformer-based neural network such as ColBERT [36] or GTR [63]. These embeddings represent each passage of the documents in the database as high-dimensional dense vectors that preserve the semantics of the text, which is essential for similarity checks. These embeddings are stored in a vector database that uses graphs to expedite the search [12, 30, 47, 57]. These graphs, which we refer to as metadata, consist of vertices pointing to the actual embeddings and are typically stored in DRAM for fast navigation. While each vertex of the graph points to an embedding, the existence of edges denotes semantic proximity between the corresponding embeddings. As such, when searching to identify the most relevant and similar passages to the query, the metadata graph enables quick navigation of the vector database by skipping irrelevant portions and exploring the most semantically relevant entries.

On the other hand, the embeddings themselves, which are much larger, are typically stored on low-latency NVMe storage drives in cloud environments [3, 5, 9, 22, 30, 38, 75, 78, 87, 91]. The original passages are stored separately in object storage such as Amazon S3, leveraging its cost-effectiveness for large-scale unstructured data. This multi-tiered approach balances performance with cost, enabling RAGs to efficiently handle large volumes of data. The vector database

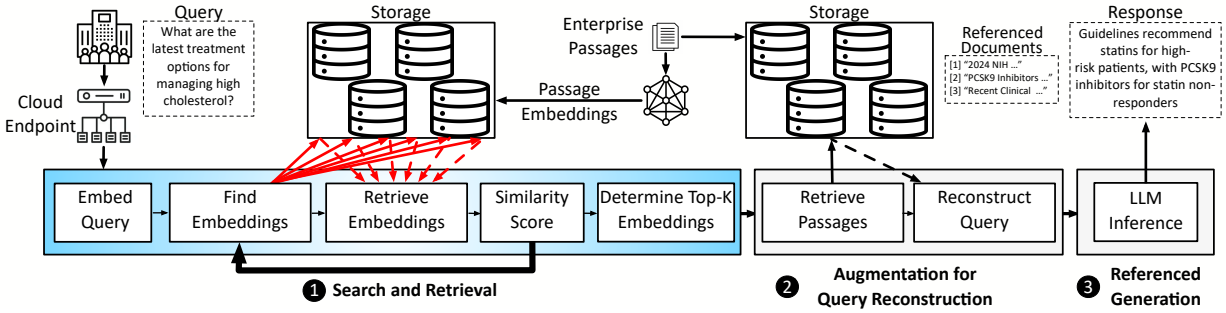


Figure 2: RAG query workflow in datacenter. The *Search & Retrieval* phase embeds the query using a language model and exhibits an iterative storage access pattern where each embedding retrieval is contingent upon prior distance computations.

and metadata graph are algorithmic techniques designed to expedite RAGs that perform similarity checks intertwined with frequent, iterative storage accesses when servicing queries.

2.2 Online Query Processing

As Figure 2 shows, the RAG workflow, deployed as a service in a disaggregated datacenter, begins when a customer submits a query to the datacenter gateway, which acts as the entry point. After performing basic integrity checks and load balancing, the query is dispatched to specialized nodes for the *Search & Retrieval* process.

❶ ***Search & Retrieval.*** This phase begins by transforming the query into a vector representation using the same neural embedding model, which is a language model like ColBERT [36] or GTR [63], applied to the passages in the offline phase. This transformation is crucial and enables direct comparison between the query and passage representations using vector distance computation functions (e.g., cosine similarity, ℓ^2-norm). The similarity search then proceeds by utilizing the metadata stored in DRAM to efficiently traverse the database and identify embeddings that are most similar to the query. Using the metadata graph, only a subset of the embeddings is retrieved from the vector database in an iterative manner. Each iteration retrieves a subset of passage embeddings from NVMe storage and calculates a similarity score for each of them. Then, based on these calculations and the structure of the graph metadata, the next set of embeddings to be retrieved from the storage is determined. This process is inherently iterative, involving multiple rounds of interleaved graph traversal, storage access, and score calculation.

In disaggregated datacenters, these storage requests are managed through a storage access layer. Despite using high-performance NVMe storage, which offers best-in-class low-latency local storage [90], each access still incurs latency due to PCIe interface traversal and data retrieval operations. The sequential nature of this process creates a critical bottleneck, as each iteration depends on the results of the previous, limiting opportunities for parallelization or prefetching. This iterative process manifests as a series of alternating storage accesses and similarity computations. The cumulative effect of these repeated sequential storage operations substantially increases the overall system latency. Hence, RAGs face significant challenges when increasing to larger databases in production environments, as detailed in §2.4.

❷ ***Augmentation for Query Reconstruction.*** The objective of the *Search & Retrieval* stage is to identify the top-k passages that have the highest similarity score to the query at hand. Once identified, these top-k passage IDs are sent over the network to an *Augmentation for Query Reconstruction* node where the main document database is accessed to obtain the original corresponding text. It is common for the original text to be stored in key-value storage for unstructured data (e.g., AWS S3). Although this tier has higher latency compared to NVMe storage, this final retrieval step occurs only once per query, in contrast to the iterative accesses to storage during *Search & Retrieval*. The multiplicity of retrieved passages is used to augment the original query to create a new one with the context from the database. As such, with this similarity-based search and augmentation, RAG aims to effectively extend the context of LLMs with an entire database.

❸ ***Referenced Generation.*** In the final stage of the RAG workflow, the augmented query is sent to a high-performance server instance over the network, such as a DGX A100 GPU cluster, which runs the LLM inference to generate a response. This stage leverages the computational capabilities of GPUs to process the augmented and context-rich query efficiently. The generated response also now has concrete citations to a source document in the database. As such, the response can be traced back and verified, crucial for the majority of enterprise applications that cannot tolerate hallucinations.

2.3 Storage-Compute Interplay in Retrievers

Having established the deployment architecture and workflow of RAGs in disaggregated datacenters, we now dive deeper into the retriever component. The retriever is the module that uses the metadata to search and identify the most relevant passages for query augmentation, which involves iterative accesses to the storage and, thus, incurs their overhead. Retrievers used in RAG can be broadly categorized into two types: embedding-based and keyword-based. Despite their algorithmic differences, our analysis reveals that both types share similar structural patterns in their implementation and face comparable challenges in terms of storage access and computation.

To better understand the interactions and bottlenecks in both types of retrievers, we conceptualize their operation across three planes: the representation plane for retrieval (located in direct-attached NVMe storage), the metadata plane for search (stored in DRAM), and the compute plane for search and retrieval. This conceptual separation allows us to clearly identify where and how storage access bottlenecks occur in both keyword-based and embedding-based retrieval systems.

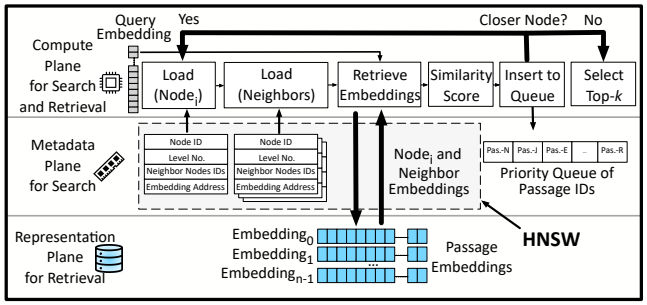


Figure 3: Operational flow of embedding-based retrievers.

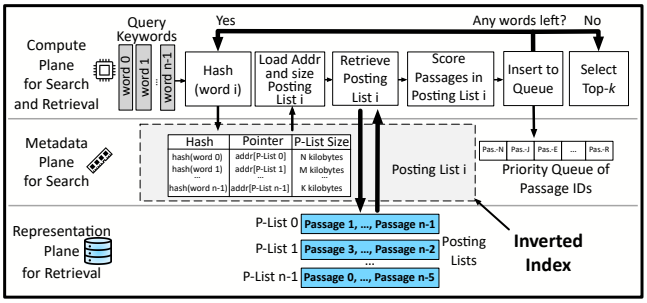


Figure 4: Operational flow of keyword-based retrievers.

Embedding-based retrievers. These neural networks, typically the encoder portion of a language model [10, 63], encode queries and passages into dense vector representations (i.e., embeddings) that capture semantic meaning. The use of embeddings enables matching the queries and the passages according to their conceptual similarity rather than the amount of keyword overlap, which is used in keyword-based retrievers. Figure 3 depicts the *Search & Retrieval* workflow across the aforementioned three conceptual planes. The passage embeddings live in the representation plane (located in the NVMe). Without graph metadata, the *Search & Retrieval* phase would need to exhaustively retrieve and score each passage's embedding, incurring immense overheads. In this paper, we use the HNSW metadata graph, which is currently one of the most popular options [57]. As discussed, this metadata graph is constructed once offline, captures the semantic similarity of the passages in the database, and represents their relationships. As such, it is an algorithmic optimization to reduce storage accesses and to enable the use of a graph search algorithm for efficient traversal and identification of highly similar passages to the query.

The retrieval process starts when a query arrives and is embedded by the same transformer-based encoder. Next, an initial vertex is designated as the current vertex from the graph, which serves as a starting point for the search. Structurally, HNSW is designed such that neighboring vertices are the most similar semantically. Each vertex contains a pointer to its corresponding embedding in the representation plane in the NVMe storage device, as Figure 3 shows. Using these pointers, the compute plane fetches the embeddings from NVMe for the current vertex and its neighbors. Then, it computes the similarity score using a distance metric (e.g., cosine similarity or ℓ^2-norm) to score all the neighbors. These distance calculations determine the similarity between the query and potential matching passages. The retriever then chooses a subset of the neighbors that have the smallest distances (i.e., the highest similarity) to continue its search. The neighbors with the highest similarity score are the most promising vertices in the graph to explore. Each neighbor now becomes a current vertex, and its neighbors' embeddings will be retrieved from the storage and examined next through distance calculations. This interleaved storage-compute search cycle using the graph is repeated until the top-k most similar passages to the query are identified.

As discussed, each step in the search process for the embedding-based retriever depends on the previous step, introducing a sequential dependency and making the storage accesses for embeddings a significant bottleneck. This characteristic makes embedding-based retrieval particularly sensitive to storage access latencies, especially when dealing with large databases containing copious passages. Despite the graph metadata that prevents retrieving and examining every embedding from storage, making the search more sample-efficient, the search still comprises iterative storage accesses with a sequential dependency to computation. This major bottleneck calls for innovations to manage storage access overheads, even when local NVMe and algorithmic optimizations are rigorously utilized.

Keyword-based retrievers. These encode passages as sparse vectors where each element corresponds to the presence of a specific keyword. These vectors are used to match keywords from the query with passages containing a high frequency of that same keyword. Figure 4 illustrates the *Search & Retrieval* workflow across the three conceptual planes of compute, metadata (memory), and representation (storage). In the representation plane that lives in the storage, each keyword is assigned to a posting list. A posting list contains all the pointers to the passages in the document database that contain the keyword and also records the frequency with which the keyword appears in the corresponding passage. In the metadata plane, keyword-based retrievers utilize an inverted index, which is a hash from the keywords to their pointers that can be used to retrieve their posting list from the storage. This metadata enables a constant time lookup of the frequencies for a given keyword in all passages in the database.

The search process aims to find relevant documents and begins when a query arrives and undergoes a vector generation process that depends on the retrieval algorithm. One class of more traditional algorithms (e.g., BM25 [76]) tokenizes the query into individual keywords. The other more modern class (e.g., SPLADEv2 [16]) passes the query through a transformer-based encoder to generate an embedding. This embedding vector not only captures keywords that are in the query but also represents semantically related keywords that are not explicitly mentioned. Each word in the embedding (keyword or a semantically related word) is then hashed to a pointer that identifies the corresponding posting list in storage. The hash of pointers from either the tokenized keywords or the embeddings constitutes the inverted index, which is metadata for *Search & Retrieval* and is kept in memory. Given the inverted index, the keyword's posting list is retrieved from NVMe storage in the representation plane, which contains passage IDs and keyword frequencies.

For each keyword in the query, the retriever uses the inverted index to find and locate all the posting lists that contain the keywords or their semantically related words. Each passage in the posting list is then scored based on a scoring metric such as TF-IDF (Term Frequency-Inverse Document Frequency) to calculate which passages have the most keyword overlap with the query. These scores are then inserted into a priority queue based on their overlap score.

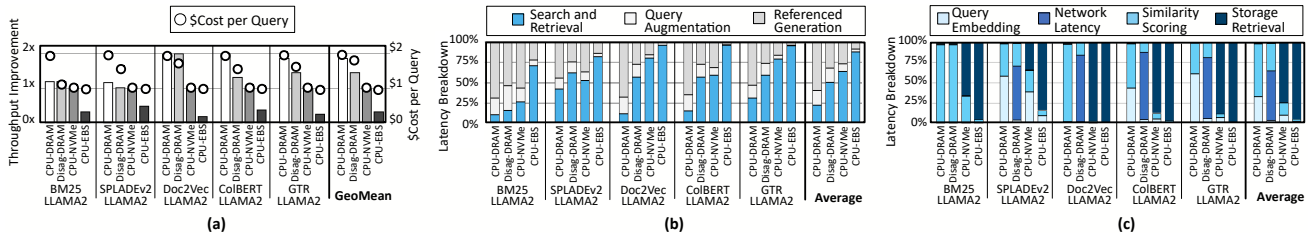


Figure 5: (a) Throughput improvement and $cost per query (normalized to CPU-NVMe). (b) Runtime breakdown for all three phases of various RAGs on AWS with LLAMA2 (34B) running on an A100 GPU. (c) Runtime breakdown of the *Search & Retrieval* phase.

This process repeats for each keyword in the query. Once finished, the top-k passages can be popped from the front of the priority queue that has the highest score in terms of keyword overlap with the query. Although this computation can be performed independently for each keyword, the storage needs to be accessed for each keyword, and the access latency is higher than the computation latency. That is, the disparity between storage access and computation latency makes storage the main bottleneck, causing serialization of the operations.

Structural similarities and challenges. Both keyword-based and embedding-based retrievers exhibit structural similarities in their storage access patterns and computational requirements. They both rely heavily on iterative storage accesses to retrieve representations (embeddings or posting lists) from NVMe storage. Both classes also employ a metadata structure (graph or inverted index) to guide the *Search & Retrieval* process. The key challenge for both classes lies in the interdependence and interleaving of storage and computation. Furthermore, as the database grows, the associated metadata and representations (embeddings or posting lists) grow proportionately. As such, finding the most relevant passages requires a larger number of storage accesses, making the bottleneck more pronounced.

2.4 Characterization of RAG in Datacenters

To empirically quantify the performance characteristics and bottlenecks of RAGs, we conduct comprehensive experiments on AWS using five diverse RAGs with various retrieval algorithms (see Table 1).

Performance-cost trade-offs in RAGs. Figure 5(a) reports the normalized throughput and $cost per query for various retrieval configurations using 5 million passages from the PubMed dataset [24, 62] relative to a CPU-NVMe baseline. The CPU-DRAM configuration, which stores all passage embeddings in DRAM and performs query embedding on the CPU, achieves the highest throughput–1.9× that of the baseline–by avoiding high-latency storage accesses. However, this performance gain incurs a 117% increase in $cost per query, making it impractical for large-scale deployments with cost constraints. This trade-off underscores the increasing adoption of SSD-based storage for representations in both industry [9, 30, 38, 78] and academia [3, 5, 22, 75, 87, 91], which seek to balance throughput with storage capacity and operational cost. Recent studies, such as one from Alibaba [5], reinforce this trend, emphasizing that SSD-based secondary storage is essential for efficiently handling large-scale vector searches in modern services, including RAGs.

***Search & Retrieval* is the primary bottleneck in RAGs.** Figure 5(b) shows the runtime breakdown of the benchmarks with two AWS storage configurations (local NVMe and networked EBS) using 5 million passages. While the AWS NVMe setup uses direct-attached local storage, the AWS EBS employs network-attached storage. In line with current deployments and research [17, 33, 44, 81] that show using GPUs is beneficial for LLM inferencing and not for retrieval and augmentation, we run those phases on a Xeon CPU while the LLAMA2 (34B) is running on A100 GPUs. Even with local NVMe, *Search & Retrieval* accounts for, on average, 61% of the total latency, and this increases to 88% when networked EBS storage is used.

In current systems, retrieval from storage dominates the *Search & Retrieval* phase. Looking into the *Search & Retrieval* phase, we find that the retrieval latency for representations from storage is the main bottleneck in current systems. Figure 5(c) quantifies this impact by showing the runtime breakdown of the *Search & Retrieval* phase. Even in the local NVMe configuration, 74% of the runtime is consumed by storage access, increasing to 94% with networked EBS storage. The use of an HNSW metadata graph reduces the number of storage accesses to 395 with 5 million passages and the ColBERT embedding-based retriever. Nonetheless, storage accesses are still the bottleneck. Each storage access incurs an average latency of 155 μs on a Samsung 970 EVO NVMe SSD, which gets compounded with the iterative nature of the *Search & Retrieval* phase (see §2).

Co-locating query embedding with *Search & Retrieval* is imperative. Given the size of embedding models like ColBERT and Google's GTR (419.62 MB), the potential for accelerated processing becomes apparent. One could imagine delegating the query embedding, which requires inference with a language model, to a GPU. However, in disaggregated datacenters, this would mean performing query embedding on a separate node and incurring an additional network latency to retrieve the embedding (see §5.2.5). To highlight the limitations of using a disaggregated GPU for embedding offloading, we analyze an idealized scenario in which all embedding vectors are stored entirely in DRAM. This setup eliminates storage access latency, allowing us to isolate and emphasize the overheads introduced solely by the disaggregated GPU architecture. As shown in Figure 5(a), this Disag-DRAM system results in a 28% lower throughput for a dataset with 5 million passages (11% for 50 million passages; see Figure 11) compared to the non-disaggregated CPU-DRAM system. This is due to network overhead that adversely affects disaggregated setups like Disag-DRAM, where embedding generation is offloaded to a GPU node. Our measurements show an average of 86 ms latency between two AWS EC2 instances deployed in the same zone (US West), measured at random times over the course of a week. The use of NVMe to store the embeddings would exacerbate the aforementioned results (see Figure 11). The *Search & Retrieval* phase, which also includes query embedding, still contributes 24% and 58%

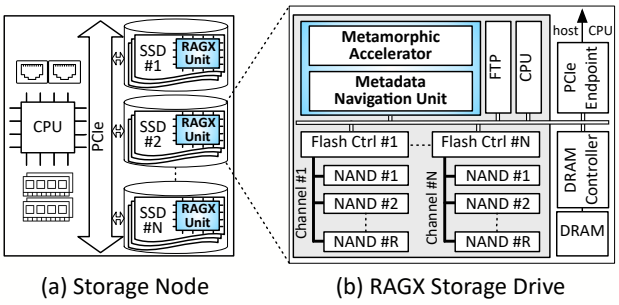


Figure 6: RAGX high-level design showing the integration of the RAGX Unit—comprising the Metamorphic Accelerator and the Metadata Navigation Unit—within the SSD.

of the total runtime for datasets with 5 million and 50 million passages, respectively (see Figure 11). This additional latency negates the performance gains achieved by removing storage overheads. Thus, eliminating storage accesses and co-locating query embedding with *Search & Retrieval* is imperative to eliminate network overheads and ensure efficient query processing in disaggregated datacenters.

3 RAGX: In-Storage Acceleration for RAGs

As discussed, retrievers frequently access NVMe storage, fetch embeddings or posting lists, and use language models to embed the query. To address these requirements, we propose RAGX, which integrates a programmable accelerator within storage devices. This integration enables the accelerator to perform query embedding using language models and directly fetch the embeddings or posting lists from the NAND arrays during the *Search & Retrieval* phase. Our objectives are (1) to minimize data movement through in-storage computation and (2) to efficiently support various embedding generation and retrieval algorithms within the storage device's power limitations. Storage devices are constrained by a strict 15 W power budget [23, 42, 55, 56] due to thermal and reliability issues. To address the latter, we propose a shape-shifting *metamorphic accelerator* discussed in §4. As Figure 6 depicts, this section discusses the in-storage integration of the accelerator within the SSD devices.

Direct NAND array accesses. As depicted in Figure 6(b), we integrate the accelerator alongside the SSD's main controller. By doing so, the accelerator can directly interact with the flash translation layer (FTL) and the flash controller, connecting to the SSD's high-speed internal bus. The accelerator can directly transfer data between the NAND arrays and its on-chip memory, bypassing the SSD's main DRAM buffer when appropriate. The metamorphic accelerator harbors a DMA engine that performs these transfers. This direct path significantly reduces the latency for retrieval operations and minimizes power consumption by avoiding unnecessary data movement. As such, RAGX's metamorphic accelerator benefits from lower-latency data accesses and high-bandwidth communication with the NAND arrays. Furthermore, this in-storage integration alleviates the overheads of data transfer over the PCIe system interconnect.

3.1 System Integration

To leverage the full potential of in-storage accelerators co-located with NAND flash and to address the challenges posed by increasingly large databases, RAGX introduces system primitives for multi-storage acceleration. These primitives facilitate efficient data movement between the storage and the accelerator, manage dynamic reconfiguration of the hardware, and enable direct communication with the host system and other accelerators through extended NVMe commands, all while maintaining compatibility with existing system primitives.

Host interface. To expose RAGX's capabilities to the host system, we extend the NVMe command set with custom admin and I/O commands. These commands allow the host to offload both *query embedding* and the *Search & Retrieval* tasks to RAGX's metamorphic accelerator, configure its operational parameters, and retrieve results. The extended command set is backward compatible with standard NVMe operations, ensuring the SSD can function normally.

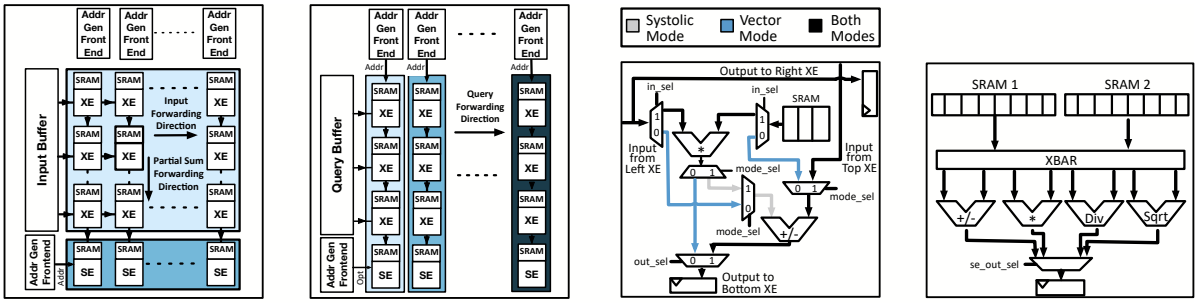
Firmware integration. We extend the SSD's firmware to include a RAGX driver that manages communication between the host, SSD controller, and the metamorphic accelerator. This driver handles task scheduling, resource allocation, and power management for RAGX, integrating its operations seamlessly with the SSD's existing firmware.

Host-device communication. The metamorphic accelerator and the flash storage use the same PCIe links to communicate with the host. A switch in the computational storage routes requests to either the flash storage device or the accelerator based on the request type.

Device-to-device communication. RAGX supports multi-device execution to handle larger dataset sizes. As will be discussed, we use a data placement strategy that prevents inter-device and inter-accelerator communication during retriever execution. However, during the initiation of the execution, there is configuration data that needs to be transferred between different devices and their metamorphic accelerators. Furthermore, in multi-device execution, we use a single RAGX storage drive to embed the query and then broadcast the embedded query to the other RAGX storage drives. To accomplish this, the peer2peer PCIe connection between the storage devices is used and, as such, bypasses the host CPU.

3.2 Data Placement Strategy

Private, smaller HNSW graphs for embedding-based multi-device in-storage acceleration. To manage representation databases that exceed the available capacity of a storage drive, RAGX supports multi-device in-storage acceleration. The data that RAGX deals with is the collection of embeddings and the associated HNSW metadata graph for the embedding-based retrievers. The HNSW metadata lives in the DRAM, but the embeddings need to be stored in the NAND flash. For multi-device execution, the embeddings and the metadata need to be allocated to keep the computation local and avoid inter-device communication. Hence, instead of using a single HNSW for all the data, which is shared across multiple devices, we partition the passage embeddings and generate a dedicated private HNSW for each device, which is smaller. The trade-off here is that the cumulative computation across all devices is greater than in the case of a single, larger HNSW. However, with our setup, there is no costly device-to-device communication. Moreover, the devices perform computation in parallel without dealing with a centralized HNSW, which would have been a bottleneck. In addition, since each device is searching a smaller graph in parallel with other devices, the recall stays the same or may even improve. On the other hand, there is an increased cumulative amount of computation, leading to an increase



(a) Systolic Mode (b) Vector Mode (c) Metamorphic Engine (XE) (d) Scalar Engine (SE)

Figure 7: The two dynamically configurable modes of RAGX's metamorphic accelerator. (a) Systolic mode. (b) Vector mode forms a collection of vector processors that are dynamically constructed from the columns of the systolic array. Microarchitecture of the (c) XE and (d) SE.

in local energy consumption. However, the overall energy consumption decreases. We perform rigorous empirical studies in §5.2 to quantify all the aspects of this data partitioning and its trade-off.

Replicated inverted indices for keyword-based multi-device in-storage acceleration. For keyword-based retrievers, although we evenly partition the posting lists, the metadata (inverted index), is replicated across all devices. The host CPU also holds a copy of the inverted index and identifies which devices need to process which keywords. The CPU initializes the corresponding devices with their list of keywords. As discussed in §2.3, the computation of each keyword with the keyword-based retrievers is independent of the other keywords. Therefore, the participating storage devices do not need to communicate during the execution. After processing their allocated keywords, all devices send their results back to the host CPU that does top-k selection for augmentation and generation. Overall, each drive performs a local similarity search, and the results are aggregated on the CPU for top-k selection. This process ensures that, for a given query, the representation database is sufficiently explored while leveraging the speed of local processing on each drive.

4 Metamorphic Accelerator for In-Storage RAGs

Our objective is to provide a single programmable accelerator that supports both embedding-based and keyword-based retrievers while adhering to the thermal and power constraints of storage devices. For embedding-based retrievers, the query must be transformed into an embedding, a process that involves running a neural network (language models like ColBERT [36] or GTR [63]). Besides running such language models, RAGX's in-storage accelerator needs to support high-throughput computation of similarity scores over high-dimensional embeddings (e.g., cosine similarity or ℓ^2-norm) for embedding-based retrievers and over posting lists using metrics such as TF-IDF [69] or BM25 [76] for keyword-based retrievers. Moreover, modern keyword-based retrievers such as SPLADEv2 [15] utilize a combination of both embedding-based and keyword-based scoring. The accelerator architecture needs to support both running a transformer-based language model as well as diverse forms of data traversal and computation for similarity scoring. Embedding-based retrievers rely on an HNSW graph to identify embeddings for each vertex and its neighbors, while keyword-based retrievers use an inverted index to locate posting lists for query keywords. As discussed in §2, the size and location of the data fetched from storage are determined dynamically based on the query. For embedding-based retrievers, the size depends on the embedding dimensions and the number of neighbors for the current vertex in the HNSW graph. For keyword-based retrievers, the size corresponds to the number of passages in the posting lists for the query keywords. This dynamic, query-dependent nature requires a mechanism to efficiently interpret metadata, fetch representations, and manage the subsequent computation.

4.1 Shape-Shifting Metamorphic Accelerator

To address these challenges, we propose a metamorphic architecture that shape-shifts from a systolic-based neural accelerator to a collection of parallel SIMD units. This is because various distance metrics cannot effectively utilize the 2D systolic array and are more appropriate for single-dimensional vector execution. Fortunately, these computations are performed in disjoint phases, providing an opportunity to reconfigure the same architecture for different forms of execution. To that end, the metamorphic accelerator in Figure 7, with limited modifications, converts the columns of a systolic neural accelerator into a collection of vector processors. This dynamic shape-shifting also matches the insight that the distance computation in RAG both for keyword-based and embedding-based retrieval methods can be decomposed into the following two phases. (1) A series of simple element-wise vector operations that can be efficiently mapped onto the 2D systolic unit. (2) A series of complex operations, such as normalization, that can be handled by a set of vector units. This insight allows us to maintain the efficiency of a 2D systolic array for the bulk of the computations while introducing targeted enhancements to support the full range of RAG operations. To support both modes (systolic and vector) efficiently, the processing element is systematically enhanced with additional arithmetic units and control logic.

4.2 Metamorphic Accelerator: Microarchitecture

As shown in Figure 7, the architecture supports a shape-shifting execution model that dynamically reconfigures a systolic array of metamorphic execution engines (XEs) and a set of scalar engines (SEs) located beneath the array. Each column of the systolic array is equipped with a vector processor front end positioned above the XEs. This front end becomes active when the architecture transitions into vector execution mode. This metamorphic design builds upon conventional systolic and vector execution paradigms. The systolic

mode is optimized for general matrix multiplication (GeMM), while the vector mode targets non-GeMM computations such as distance functions in retrieval workloads. We observe that distance functions can be efficiently mapped to this architecture with minimal modifications to the PE microarchitecture (see Figure 7(c)). In systolic mode, each XE functions as a conventional PE. The internal multiplexers (control path set to one) are configured to enable the operand and partial sum forwarding paths typical of systolic execution. Each PE contains a fused multiply-accumulate (MAC) unit, a local weight buffer, input/output registers, and forwarding logic for propagating data along the systolic wavefront.

When transitioning to vector mode, control logic reconfigures multiplexers to reroute data through a vertical operand pipeline. Each column of XEs is dynamically reinterpreted as a vector engine, where each XE implements one pipeline stage. In this setup, the architecture exploits fine-grained vertical pipelining to achieve vector parallelism. The vector processor front end atop each column fetches instructions, generates addresses, and distributes control signals. Operands are stored in scratchpads within each XE (repurposed from weight buffers used in systolic mode). Vertical registers between adjacent XEs are enhanced to act as pipeline latches, passing data and metadata (e.g., addresses) to the next stage. All XEs in a column execute the same operation, defined by the current vector instruction. However, each column operates independently, maintaining its own program counter and instruction stream via its front end. This design allows different vector engines to execute distinct vector kernels concurrently.

Metamorphic execution engines (XEs). Each XE supports two execution modes: it acts either as a PE in systolic mode or as a pipeline stage in vector mode. The core components of the XE, shown in Figure 7(c), include a MAC unit for GeMM operations, a local buffer that serves as a scratchpad in vector mode, and a set of input/output registers. The shape-shifting functionality is enabled through a network of multiplexers controlled by a mode configuration bit set by the global execution controller. These augmentations allow the XEs to switch between dataflow (systolic) and pipeline (vector) operation with minimal area overhead while reusing much of the existing PE microarchitecture.

Scalar engines (SEs). To support complex scalar operations required by RAG, each column includes an SE positioned below the XE pipeline. These scalar engines, which are shown in Figure 7(d), implement high-latency, resource-intensive functions such as division, square root, and logarithm operations frequently used in retrieval scoring functions. For example, SEs compute square roots for ℓ^2-norm or logarithms for BM25 scoring. In vector mode, each SE operates as an extension of its corresponding vector engine, executing scalar instructions that accompany vector instructions. Specifically, each vector engine executes an instruction pair of the form ⟨`vector`, `scalar`⟩, where the scalar instruction typically follows the vector operation to complete the final computation step. In systolic mode, the SEs are collectively reconfigured to form a standalone horizontal vector processor. The front-end unit for this processor is located at the bottom left of the array, as depicted in Figure 7(a). This horizontal vector engine complements the systolic array during neural network inference, particularly for embedding computations that require vector operations beyond GeMM (e.g., normalization, bias addition).

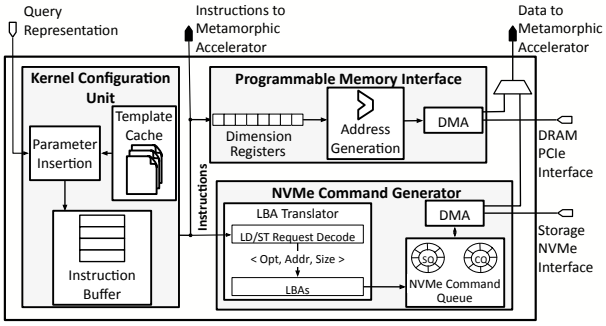


Figure 8: Microarchitecture of Metadata Navigation Unit.

Additionally, in embedding-based HNSW retrieval, SEs play a critical role in graph traversal by sorting vertex scores after each iteration. The sorted results are then written to a DRAM-resident priority queue that guides the selection of the next nodes to evaluate.

4.3 Metadata Navigation Unit

Besides performing the computation, there is a need for a unit that supplies the appropriate data to the metamorphic accelerator. This unit needs to navigate the metadata, which is the HNSW graph for embedding-based retrievers and the inverted indices for the keyword-based retrievers. The important point is that before receiving a query and accessing these metadata structures, the exact size of the data vectors and their locations are unknown. Therefore, there is a need for a unit that first walks over the metadata and determines the number of neighboring embeddings in the case of HNSW and their location in the storage. Then it can load the appropriate data from the NAND arrays of the storage either directly to on-chip memory or the DRAM if necessary. Similar steps need to be taken for keyword-based retrievers, except that they involve the inverted indices and the posting lists. Note that this is just one step of the iterative process of *Search & Retrieval*. After the data sizes are determined, the metamorphic accelerator can be properly configured to perform the computation. To address this need for data-dependent configuration and setup for the metamorphic accelerator, we introduce the Metadata Navigation Unit (MNU), which is shown in Figure 8. The MNU integrates three essential capabilities tailored to address the above challenges. First, it fetches the metadata stored in DRAM to determine the address and size of the representations to be retrieved from storage. Second, the MNU generates NVMe commands to retrieve the required representations through an integrated NVMe command generator. These commands enable efficient data movement to the proposed accelerator's on-chip scratchpads for computation. Finally, the MNU schedules computation using parametric pre-compiled kernel templates. At runtime, the MNU fills in kernel template parameters, such as embedding vector shape or posting list size, to generate customized kernels that match the query's requirements from the kernel templates. This flexible approach avoids the overhead of full runtime compilation while accommodating the diverse workloads of embedding-based and keyword-based retrievers.

Metadata interpretation. The first task of the MNU is to locate and determine the size of the query-dependent representations stored in NVMe. The metadata is retrieved from DRAM and varies based on the retrieval method. For embedding-based retrievers, the

Table 1: Evaluated end-to-end RAG benchmarks.

Benchmark	Retriever	Retriever Type	Token/Embedding Model	Representation Dimensions	Distance Function	Database	Referenced Generation LLM
BM25 LLAMA2	BM25	Keyword	NLTK Tokenizer	Variable Length	BM25	Inverted Index	LLAMA2 34B
SPLADEv2 LLAMA2	SPLADEv2	Embedding, Keyword	DistilBERT	Variable Length	Dot Product	Inverted Index	LLAMA2 34B
Doc2Vec LLAMA2	Doc2Vec	Embedding	Skip-Gram	300	L1	HNSW	LLAMA2 34B
ColBERT LLAMA2	ColBERT	Embedding	Bert-Base	128	Squared L2	HNSW	LLAMA2 34B
GTR LLAMA2	GTR	Embedding	T5-Base	768	Dot Product	HNSW	LLAMA2 34B

metadata identifies the embeddings associated with the current vertex in the graph traversal and its neighbors within the metadata graph. The sizes of the corresponding embeddings are calculated as embedding size×number of neighbors. For keyword-based retrievers, the metadata links keywords to their corresponding posting lists in the inverted index. Here, the size of the posting lists depends on the number of passages containing each keyword. By determining the size of the representations, the MNU ensures that memory access requests are tailored to the query's specific requirements, optimizing retrieval efficiency.

Retriever-specific kernel templates. Next, to handle the diverse retrieval algorithms, the MNU utilizes retriever-specific kernel templates. These templates encode the high-level structure of embedding-based and keyword-based code, with placeholders for parameters such as data sizes and dimensions. In embedding-based retrieval, templates specify parametric tiling and the mapping of embeddings to compute units that will be determined based on runtime data sizes. For keyword-based retrieval, templates describe scoring operations for priority queue management. By precompiling these templates offline and instantiating them at runtime, the MNU avoids the overhead of just-in-time compilation and maintains execution efficiency.

Kernel configuration unit. Expanding on the retriever-specific kernel templates, the MNU incorporates a kernel configuration unit that dynamically creates instances of these templates using parameters obtained from metadata. This unit has three key components: a template cache, parameter insertion logic, and an instruction buffer. The template cache stores pre-compiled templates for common retrieval tasks. The parameter insertion logic replaces placeholders in the templates with runtime values, such as embedding dimensions, neighbor counts, or posting list sizes. The instruction buffer then assembles the instantiated instructions and enables instruction-level parallelism to optimize execution. For example, for embedding-based retrieval, the unit fetches metadata specifying the number of neighboring vertices and embedding dimensions, fills the corresponding kernel template with these parameters, and dispatches it to the compute units.

Programmable memory interface (PMI). To handle the dynamic and variable dimensions of embedding-based and keyword-based representations, the MNU incorporates a programmable memory interface (PMI) that supports multi-dimensional memory access patterns with configurable bounds and strides. The PMI comprises three key components: dimension registers, stride calculators, and address generation units. Dimension registers store runtime-determined data sizes such as the length of posting lists or embedding dimensions. Stride calculators compute memory strides for accessing multi-dimensional data layouts, while address generation units issue memory fetch commands for nested loops and irregular data structures. For embedding-based retrievers, the PMI dynamically accesses embeddings corresponding to a variable number of neighbors. For keyword-based retrievers, it iterates through posting lists of differing lengths, ensuring high-throughput access with minimal overhead.

NVMe command generation. To translate internal fetch requests into standard NVMe read commands, the MNU incorporates an NVMe command generator. This module consists of three key elements: a command queue, an LBA translator, and a DMA engine. The command queue manages pending NVMe requests to maximize storage bandwidth utilization. The LBA translator converts logical addresses derived from metadata into physical logical block addresses (LBAs) for NVMe storage. The DMA engine supports scatter-gather operations, enabling efficient transfer of non-contiguous data regions into internal buffers. By leveraging the internal bandwidth of the storage device and bypassing the traditional PCIe interface, the NVMe command generator reduces data transfer latency and power consumption. For example, when fetching posting lists for a keyword-based retriever, the DMA engine consolidates the required regions into a contiguous on-chip buffer, optimizing data movement.

Mapping templates to the accelerator. The MNU determines how representations are mapped onto the metamorphic accelerator based on the retrieval type and query-specific parameters. For embedding-based retrievers, embeddings of size D for the query and its K neighbors are fetched and mapped as follows: the query embedding is unrolled across the N vector lanes of each XE, while the K neighbor embeddings are distributed across the N vector processors, with each XE processing a segment of the embedding dimension. For keyword-based retrievers, posting lists of length L are fetched and mapped such that term-specific elements (e.g., term frequency, document length normalization) are vectorized across vector lanes, while multiple documents in the posting list are distributed across the N vector processors. This hierarchical mapping ensures efficient utilization of all the XEs for both embedding-based and keyword-based workloads while dynamically adapting to the data layout and query characteristics.

Compiler support. For query embedding via a language model, we rely on prior work [20] that offers an open-source compiler for end-to-end neural acceleration. This is possible because, in the systolic mode, the metamorphic accelerator resembles a conventional neural accelerator. This compiler also provides support for vector execution that we leverage and modify for generating parametric kernels for similarity checks in *Search & Retrieval*. We develop a custom compilation module for the MNU to support both HNSWs and inverted indices.

5 Evaluation

5.1 Methodology

Benchmarks. Table 1 summarizes the five end-to-end RAG benchmarks, representative of the RAG pipeline shown in Figure 2. We use the RAGGED benchmark suite [24], incorporating additional

Table 2: Evaluated Systems.

	CPU-NVMe	CPU-DRAM	Disag-NVMe	Disag-DRAM	GPU-DRAM	RAGX
Chip	Intel Xeon 8175M	Intel Xeon 8175M	Intel Xeon 8175M[1], NVIDIA A100[2]	Intel Xeon 8175M[1], NVIDIA A100[2]	NVIDIA A100	Metamorphic Accelerator
AWS Instance	m7g.4xlarge.search	r7g.8xlarge.search	m7g.4xlarge.search[1], p24.24xlarge[2]	r7g.8xlarge.search[1], p24.24xlarge[2]	p4.24xlarge	N/A
Cost ($/hour)	$1.26	$3.16	$1.26[1], $4.09[2]	$3.16[1], $4.09[2]	$4.09	$1.09*
Cores / PEs	32 Cores	32 Cores	32 Cores[1], 6,912 Cuda Cores[2]	32 Cores[1], 6,912 Cuda Cores[2]	6,912 Cuda Cores	32×32 XEs, 32 Scalar Unit
Memory	64 GB	512 GB	64 GB[1], 80 GB[2]	512 GB[1], 80 GB[2]	80 GB	4 GB (16 MB)
TDP	240 W	240 W	240 W[1], 400 W[2]	240 W[1], 400 W[2]	400 W	13 W
Frequency	2.5 GHz	2.5 GHz	2.5 GHz[1], 1.1 GHz[2]	2.5 GHz[1], 1.1 GHz[2]	1.1 GHz	1 GHz
Storage	Samsung NVMe 970	N/A	Samsung NVMe 970	N/A	N/A	NAND Flash
Storage Bandwidth	4 Gb PCIe Gen3	N/A	4 Gb PCIe Gen3	N/A	N/A	NAND Data Bus 20 Gbps

Note: [1] Used for search and retrieval. [2] Used for query embedding. * Based on ra3.xlplus pricing.

benchmarks to broaden the evaluation scope. In the *Search & Retrieval* phase, we implement the retrievers using unmodified, default versions of Pyserini [48] for BM25, the official GitHub repository for SPLADEv2 [15], and Faiss's implementation of HNSW [13] for embedding-based retrievers, with all default optimizations and multi-threading enabled. For *Referenced Generation*, we use Meta's LLAMA2 (34B) [84] deployed on two A100s with TensorRT-LLM and tensor parallelism. The input consists of an average of 850 tokens, a maximum context length of 4,096 tokens, and a batch size of 1 (up to 256 for sensitivity studies). The query dataset specifies 54 output tokens as the golden answer and is in line with prior studies [24, 46, 73] that recommend retrieval for ≤ 64 output tokens to maintain high recall. We scale to 512 output tokens for sensitivity studies. We set top-k to 100 for all evaluations [75].

Dataset and database generation. We use the PubMed [24, 62] biomedical database with four dataset sizes: 500 million (500M), 50 million (50M), 5 million (5M), and 0.5 million (0.5M) passages. To accommodate the 500M passage dataset, we augmented PubMed's 50M passages with randomly generated embeddings. For embedding-based retrieval, embeddings are generated using GPUs, and HNSW indices (Faiss, $M = 32$, $efconstruction = 100$) are built on CPUs. As per prior work [53, 54], we set *efsearch* to 375, 750, 1,500, and 3,000 for 0.5M, 5M, 50M, and 500M passages, respectively, to maintain a consistent recall as dataset size scales. As for the keyword-based retrievers, BM25 uses Pyserini for inverted index construction, while SPLADEv2 embeds passages before index generation. The vector database size for embedding-based retrieval is computed as $P \times D \times T$, where P is the number of passages, D is the embedding dimension, and T is the datatype size (4 bytes [13]). For example, ColBERT requires $500K \times 128 \times 4 = 256$ MB for 0.5M and $500M \times 128 \times 4 = 256$ GB for 500M, while GTR requires $500K \times 768 \times 4 = 1.5$ GB and $500M \times 768 \times 4 = 1.5$ TB, respectively. For keyword-based retrieval, storage depends on the number of unique tokens and index structures. BM25 0.5M requires 0.36 GB and 500M requires 320 GB. SPLADEv2 requires 0.18 GB for 0.5M and 178 GB for 500M. We evaluate using 3,800 BioASQ [39] queries, ensuring a fair comparison across all systems.

Baseline systems. We compare RAGX with the following systems. Table 2 details the specifications for the *Search & Retrieval* phase of each system. (1) CPU-NVMe (baseline): query embedded on the CPU, with representations (embeddings/posting-lists) stored on direct-attached NVMe SSD, metadata (HNSW/inverted index) cached in DRAM, and retrieval performed by the same CPU. (2) CPU-DRAM: query embedded on the CPU, with representations and metadata cached in DRAM, and retrieval performed by the same CPU. (3) Disag-NVMe: query embedded on a GPU, embeddings transferred to a CPU node over network (100 GbE Ethernet), which performs the *Search & Retrieval* from representations stored on NVMe SSD and metadata cached in DRAM. (4) Disag-DRAM: query embedded on a GPU, embeddings transferred to a CPU node over network, which performs the *Search & Retrieval* from representations stored in DRAM and metadata cached in DRAM. (5) GPU-DRAM: query embedding and retrieval both performed on the same GPU, with all representations and metadata cached in GPU memory (represents idealized zero-delay network communication). For all systems (including RAGX), we use the EC2 instance from Table 2 for the *Search & Retrieval* phase with passages stored on AWS S3 and *Referenced Generation* performed on a DGX-A100 cluster (AWS p4.24xlarge instances) using one, two, and four A100 GPUs for LLAMA2 (13B), LLAMA2 (34B), and LLAMA2 (70B), respectively.

Baseline measurements. We deploy our benchmarks on AWS Sagemaker following the AWS blog and example code [78, 79]. This setup allows us to process a custom dataset (PubMed), generate a vector database, and store the representations on NVMe, DRAM, or EBS. For the *Search & Retrieval* phase, EC2 instances are provisioned (see Table 2), which fetch the top-k passages from AWS S3 and send them to an AWS p4.24xlarge GPU instance for *Referenced Generation*. We perform real measurements by running 3,800 queries from BioASQ, where each query traverses the entire RAG pipeline as described in §2.2. For each phase, we measure runtime using timers integrated into the code to ensure precise and accurate measurements. All results are based on the median latency of 3800 queries to ensure evaluation accuracy.

RTL implementation and prototyping. The metamorphic accelerator is implemented in Verilog and synthesized using Synopsys Design Compiler 2023.09 with the FreePDK 45 nm standard cell library, achieving a clock frequency of 1 GHz.

Metamorphic accelerator simulator. We develop a cycle-level simulator for the RAGX accelerator to evaluate its performance and energy consumption. Our simulator models all the parts of the proposed accelerator (§4) and accounts for all critical components: NVMe flash array reads, control logic for kernel scheduling, execution time on processing units, metadata traversal from DRAM using the MNU, and the network latency to transfer data between each phase since the deployment of RAGs is in a disaggregated datacenter setting. For embedding-based retrieval, we compile the embedding model using the compiler described in §4.3. To simulate the entire

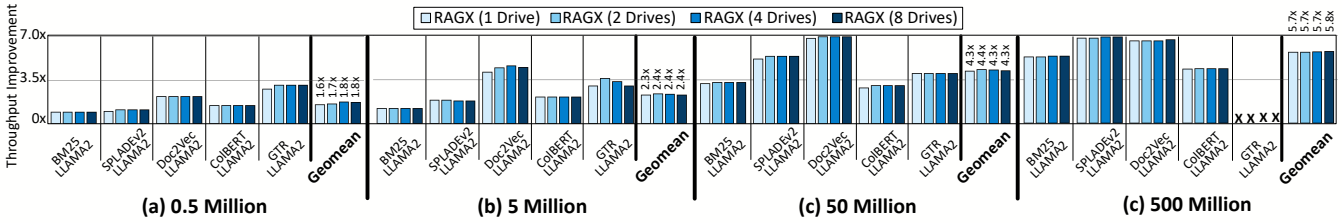


Figure 9: Throughput improvement normalized to CPU-NVMe. 500M analysis for GTR-LLAMA2 was infeasible due to memory requirements.

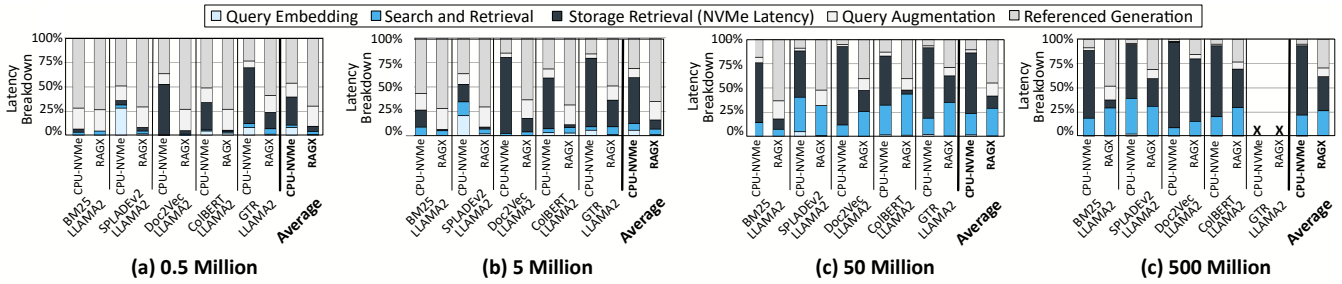


Figure 10: Runtime breakdown comparison with CPU-NVMe for 0.5M, 5M, 50M, 500M passages.

Search & Retrieval phase, we generate traces from baseline runs that capture how the system traverses the HNSW graph (embedding-based) or inverted index (keyword-based). These traces include vertex scoring sequences for embedding-based and posting list accesses for keyword-based retrieval. We use these traces to schedule instructions on our simulated accelerator, instantiating template kernels with actual metadata values. NVMe SSD access latencies are modeled using an open-source simulator [58]. This approach allows us to accurately measure performance and energy consumption for each operation, from data access to computation, providing a thorough assessment of RAGX's capabilities across various scenarios.

RAGX performance measurement. Real deployment of a research chip in the cloud is not currently feasible. We follow standard methods for accelerator research in the cloud [74, 88] that augments real measurements with careful and systematic simulation. To preserve the overheads of a disaggregated datacenter, we systematically replace the embedding, search, and retrieval times with the corresponding operations in the CPU-NVMe baseline on AWS, ensuring that network latencies and variabilities from other phases, such as S3 access, *Augmentation for Query Reconstruction*, and *Referenced Generation*, are included. We repeat this process for all 3,800 BioASQ queries and report the median latency to even capture the impact of outliers.

5.2 Experimental Results

5.2.1 Throughput and Cost Analysis. For throughput measurements, we obtain latencies for all 3,800 BioASQ [39] queries in our dataset for each benchmark. We run baseline systems on AWS, while for RAGX, we simulate the *Search & Retrieval* phase and obtain latencies for *Augmentation for Query Reconstruction* and *Referenced Generation* from our AWS measurements. Then we generate traces per benchmark, simulating Poisson-distributed query arrivals with a consistent seed across all systems. We develop an analytical model to measure throughput, sweeping the query arrival rate λ from 1 to 100 queries/second in 0.1 increments and repeating each experiment 10 times. We report the throughput normalized to CPU-NVMe (baseline), where throughput is defined as the highest λ where the system processes queries without queue accumulation or increased latency.

Figure 9 illustrates the throughput improvements of RAGX compared to CPU-NVMe across various dataset sizes, retriever types, and number of drives. On average, RAGX achieves 1.6×, 2.3×, 4.3×, and 5.7× throughput gains over CPU-NVMe for datasets containing 0.5M, 5M, 50M, and 500M passages, respectively. As dataset size increases, the performance benefits of RAGX become more pronounced, with an average improvement of 3.6× for 500M compared to 0.5M, demonstrating efficiency with larger databases. For embedding-based retrieval, improvements are more significant than for keyword-based retrieval, as RAGX reduces storage access latency, accelerates similarity scoring, and accelerates query embedding. For instance, RAGX achieves a 1.9× to 2.7× performance improvement for GTR-LLAMA2 over BM25-LLAMA2 as the dataset scales from 0.5M to 500M. Additionally, RAGX maintains consistent throughput improvements as the number of drives increases from one to eight, showcasing efficiency for scale-out setups. To understand the sources of RAGX's improvements, we conduct a detailed runtime breakdown analysis, shown in Figure 10.

NVMe retrieval latency reduction. RAGX significantly reduces NVMe read latency, lowering the proportion of runtime spent on storage accesses to 4.1%, 6.7%, 12.8%, and 40% for 0.5M, 5M, 50M, and 500M passages, respectively, compared to CPU-NVMe's 26.6%, 47.7%, 63.2%, and 75%. The remaining storage access latency in RAGX stems from the NAND array's read time. For GTR-LLAMA2, with 50M, NVMe latency decreases from 72% to 27% with the 50M passage dataset, accounting for a 2.9× speedup. ColBERT-LLAMA2 exhibits the largest reduction among the embedding-based retrievers due to its smaller embedding size. For keyword-based retrievers such as BM25-LLAMA2, the impact increases with dataset size. With 50M passages, RAGX reduces NVMe latency from 60.9% to 10.3% of runtime for BM25-LLAMA2 by removing the PCIe latency.

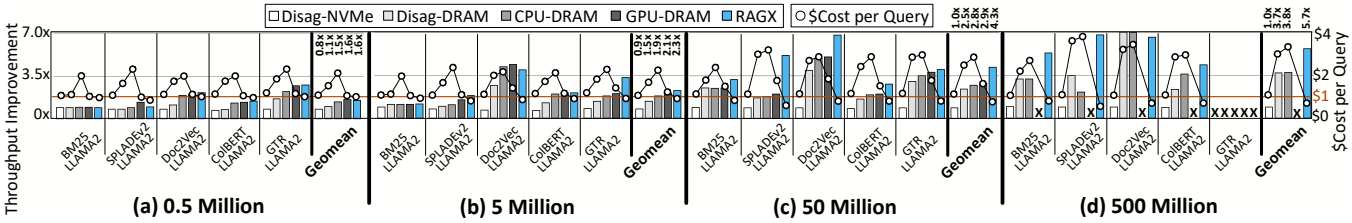


Figure 11: Throughput improvement (shown on the primary y-axis, left) and $cost per query (shown on the secondary y-axis, right) across systems, normalized to CPU-NVMe. GPU-DRAM was infeasible for 500M due to GPU memory limitations.

Performance as dataset size grows. RAGX demonstrates increasing performance benefits as dataset size grows, efficiently handling the higher volume of storage accesses required for searching larger databases. For instance, GTR-LLAMA2 incurs an average of 347 storage accesses per query on 0.5M, which increases to 1493 on 50M, while RAGX 's normalized throughput improves by 1.5× over this scale-up. Similarly, ColBERT-LLAMA2 storage accesses increase from 393 on 0.5M to 3023 on 500M, with RAGX providing a 2.9× throughput improvement. These highlight RAGX 's effectiveness in mitigating storage overhead and accelerating retrieval as database size increases.

Performance across retrievers. RAGX demonstrates better performance across diverse retrievers as dataset size grows, with notable improvements for the embedding-based retrievers. For GTR (5M dataset), RAGX reduces embedding generation time by 12× and similarity computation time by 37.6× compared to CPU-NVMe, providing a 3.1× throughput improvement. Performance gains also extend to keyword-based retrievers such as SPLADEv2, which uses a language model to expand queries. On the 0.5M dataset, *Search & Retrieval* account for 32% of end-to-end time, with 82% spent on query embedding. As dataset size grows (50M), NVMe access and similarity scoring dominate, increasing *Search & Retrieval* time to 93%. With this shifting runtime profile, RAGX delivers growing speedups for SPLADEv2, ranging from 1.1× on 0.5M passages to 6.8× on 500M passages.

Performance trends with more storage drives. RAGX maintains consistent performance when partitioning representations across multiple drives. With 50M passages, RAGX achieves 4.4× throughput improvement with two drives and 4.3× with four or eight drives relative to CPU-NVMe. The proposed data placement strategy is effective for both keyword-based and embedding-based benchmarks, with BM25-LLAMA2 showing 3.2× and GTR-LLAMA2 showing 4.1× improvement using eight drives. This performance stems from RAGX's parallel execution strategy, performing computations in each drive concurrently and accumulating top-k results on the CPU in the augmentation server. Minor variations in embedding-based retriever performance are due to changes in graph structures and traversal patterns.

Benefits from co-locating query embedding with retrieval. In disaggregated datacenters, offloading query embedding to a separate node introduces additional network latency (on average 86 ms) to transfer embeddings (e.g., 128×4 bytes for ColBERT) to the retrieval node, as discussed in §2.4. While GPUs reduce embedding time to just 1% of the total runtime for 50M passages, Disag-NVMe still incurs an additional 11% overhead compared to CPU-NVMe due to network transfer costs. By co-locating embedding and retrieval, RAGX eliminates this overhead. Its specialized design accelerates both query embedding and similarity search by reusing the resources of the metamorphic accelerator, boosting the performance to 4.3× over Disag-NVMe for 50M passages.

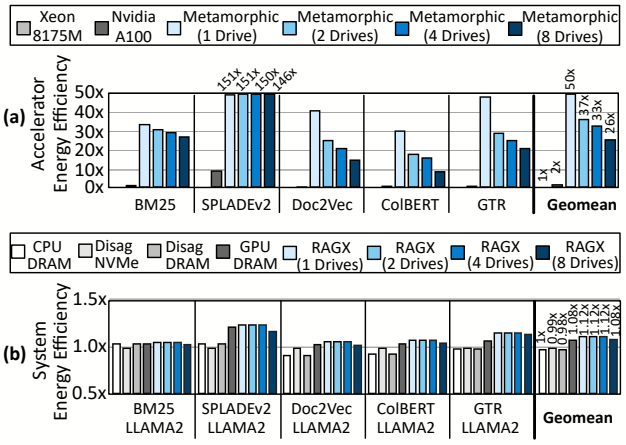


Figure 12: Energy efficiency comparison for 5M passages.

Comparison with additional baselines. Figure 11 shows the throughput improvement of RAGX, CPU-DRAM, Disag-NVMe, Disag-DRAM, and GPU-DRAM normalized to CPU-NVMe. RAGX achieves average throughput improvements of 1.5×, 4.3×, and 1.7×, over CPU-DRAM, Disag-NVMe, and Disag-DRAM, respectively, for 50M passages. Additionally, RAGX provides 1.4× speedup while GPU-DRAM is 119% more expensive per query for 50M passages. Results for 500M dataset could not be obtained for GPU-DRAM because the dataset's embeddings do not fit within the A100's DRAM capacity. RAGX, for the 50M dataset, outperforms CPU-DRAM by 1.5× while CPU-DRAM is 266% more expensive per query. For 500M, the cost difference increases to 391% while the speedup is 1.6×. These results reinforce the substantial cost benefits of using direct-attached NVMe for storing representations.

Cost analysis. Following prior work [40], we compute the $cost per query by summing the costs of each phase in the execution pipeline, where each phase's cost is the product of its runtime and the AWS instance cost (see Table 2). Since RAGX is not currently available on cloud platforms, we estimate its cost using AWS `ra3.xlplus` instances, which provide in-storage database acceleration with integrated storage costs, serving as an effective proxy as per prior studies [55]. We omit data transfer costs. As shown in Figure 11, RAGX not only delivers notable throughput improvements but also achieves the lowest cost among all evaluated systems. As dataset size increases from 0.5M to 500M passages, our results indicate that CPU-DRAM retrieval is between 119% and 391% more expensive per query

than RAGX. This cost trade-off aligns with industry and research trends discussed in §2.4.

5.2.2 Energy Efficiency. We measured energy consumption using Intel's RAPL [77] for CPUs, NVIDIA's SMI [65] for GPUs, and RTL synthesis power results with on-chip memory energy modeled using CACTI [61] for RAGX. We used 1.93 pJ/bit for storage access over PCIe and did not include network energy in our calculations. We computed total energy by multiplying the measured average power for each component with its corresponding runtime for each system configuration.

System and accelerator energy efficiency. Figure 12(b) compares system energy efficiency normalized to CPU-NVMe for various configurations (5M passages). RAGX with a single drive shows energy efficiency improvements of 1.12× over CPU-NVMe, compared to no noticeable improvement for CPU-DRAM and a 1.08× gain for GPU-DRAM. RAGX's limited overall gain is primarily due to the dominant energy consumption of two NVIDIA A100 GPUs (400 W TDP each) used for *Referenced Generation*. However, focusing on the *Search & Retrieval* phase, RAGX's metamorphic accelerator demonstrates significant improvements as shown in Figure 12(a). The proposed accelerator achieves an average 50× energy efficiency gain over CPU-NVMe, compared to GPU-DRAM's 2× improvement. SPLADEv2 benefits most, with up to 150× efficiency increase due to RAGX's acceleration of both embedding and scoring processes. Notably, RAGX maintains a 26× improvement even with eight drives.

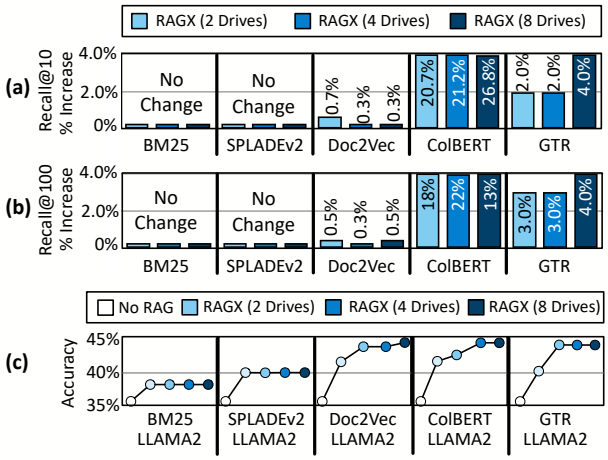


Figure 13: Recall and end-to-end accuracy on 5M passages.

5.2.3 Recall and End-to-End Accuracy. Figure 13(a) and Figure 13(b) show recall@10 and recall@100, respectively, for RAGX configurations with two, four, and eight storage drives. The keyword-based retrievers maintain steady recall as the representation database remains unchanged, replicated, and queried across drives in parallel. In contrast, the embedding-based retrievers use the proposed data placement strategy (see §3.2), which partitions embeddings and assigns each drive a smaller, private HNSW graph. Despite this structural change, the embedding-based retrievers show no degradation; ColBERT and GTR exhibit 13% and 4% improvements in recall@100, respectively, with eight drives. This improvement results from localized computation on smaller graphs and the aggregation of candidates from multiple drives, which expands the effective search space.

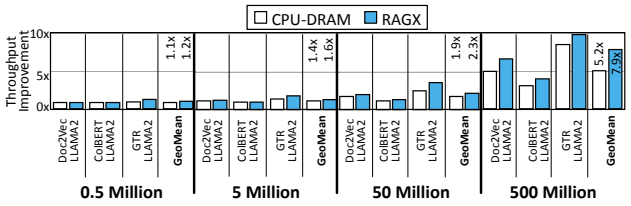


Figure 14: Throughput improvement over IVF.

Figure 13(c) shows the impact of this increased recall on RAGX's end-to-end RAG accuracy (Unigram F_1 [24]). The baseline accuracy is consistent with prior work [6, 72, 86]. The keyword-based retrievers show no change in accuracy when scaling from one to eight drives, while the embedding-based retrievers see a 2–3% increase depending on the retriever.

5.2.4 Comparison with Alternative Vector Databases. We compare RAGX with a non-graph vector database: Inverted Vector File (IVF) [13]. Unlike graph-based databases that rely on greedy traversal, IVF employs a flat index. It partitions the vector space via k-means and assigns each vector to its nearest centroid. At query time, it computes distances to all centroids and probes only a small subset, reducing compute at the expense of recall. We restrict IVF comparisons to embedding-based benchmarks (ColBERT, Doc2Vec, GTR), which align with its flat index structure. We use a single RAGX accelerator to ensure fairness; however, RAGX scales efficiently across multiple devices, making this a conservative baseline. We use two IVF configurations: (1) CPU-DRAM, where the full vector index is cached in DRAM, and (2) CPU-NVMe, where vectors reside in NVMe and only centroids are cached in DRAM. Following [13, 32], we set the number of clusters to the square root of the corpus size and retrieve 0.1% per query. Figure 14 illustrates the throughput normalized to CPU-NVMe. As dataset size increases, RAGX outperforms CPU-NVMe by 20% to 7.9× from 0.5M to 500M due to increasing I/O overheads in IVF that RAGX mitigates via in-storage acceleration. Against CPU-DRAM, RAGX still achieves 12% to 51% higher throughput, enabled by its metamorphic accelerator that executes similarity search efficiently.

5.2.5 Sensitivity Analysis.

LLM inference configurations. Figure 15 illustrates the effects of system and LLM setup changes on the runtime breakdown and throughput improvement. The baseline is CPU-NVMe for *Search & Retrieval* and two A100 GPUs for *Referenced Generation*. Reducing the number of A100 GPUs to one shifts the runtime more towards the *Referenced Generation*, lowering RAGX's benefit from 4.3× to 3.0×. Whereas, using four A100 GPUs or two H100 GPUs boosts improvement to 4.3× and 4.6×, respectively. On a machine with two A100s, if the output token length is 512, the benefits can drop to 1.2×. However, utilizing FlashAttention-2 [7] (a software technique) raises the gains to 1.9×. These trends illustrate that *Search & Retrieval* is a bottleneck in certain configurations, contrary to the assumption that *Referenced Generation* always dominates, and are consistent with prior works [72, 82]. Both software and hardware improvements in inferencing will help the benefits from RAG acceleration and RAGX.

Networked storage drives. Figure 16 demonstrates RAGX's performance with representations distributed across Ethernet-connected storage drives. For eight drives, RAGX maintains 2.5× throughput

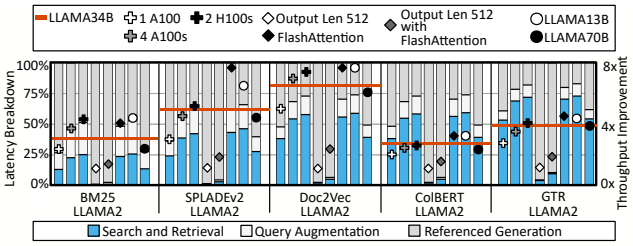


Figure 15: Sensitivity to LLM deployments on 50M (throughput normalized to CPU-NVMe).

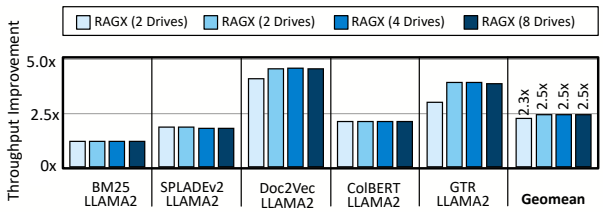


Figure 16: Sensitivity to networked storage drives for 5M passages (normalized to CPU-NVMe).

improvement, consistent with PCIe setups. This stability results from RAGX's elimination of inter-storage drive communication and performing independent computations on each device.

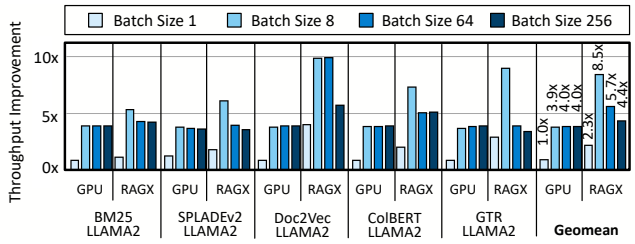


Figure 17: Sensitivity to query batch size for 5M passages (normalized to CPU-NVMe).

Batch size. RAGX supports concurrent query processing via batching. In systolic mode, the batching is performed as it is done conventionally. In vector mode, vector engine columns are multiplexed across concurrent batched queries. Figure 17 compares RAGX and GPU-DRAM performance over CPU-NVMe across batch sizes 8, 64, and 256. RAGX maintains strong speedups: 8.5× for small batches versus GPU-NVMe's 3.9×, and 4.4× for large batches versus GPU-DRAM's 4.0×. As batch size grows, *Referenced Generation* dominates runtime (e.g., 73% at batch size 256), limiting RAGX's benefits. Nevertheless, RAGX still provides gains by accelerating batched embedding generation and reducing NVMe access time for fetching representations.

6 Related Work

Systems for RAG. Prior work on *Search & Retrieval* focuses on accelerating the similarity search process through algorithmic improvements [12, 13, 47, 50, 57], enhancing runtime efficiency through parallelization [13, 48], and near-memory acceleration [32]. These works make the assumption that all representations reside in DRAM. Other works address this by compressing or selectively storing the representations in DRAM, while the entire representation database is stored in a storage drive [4, 19, 30]. However, this setup is prone to significant storage access overheads, as representations still need to be retrieved from a storage drive. For *Referenced Generation*, prior works focus on accelerating LLM inference through system frameworks [29, 51], model parallelism [2], quantization [49], and hardware optimizations [35, 45, 52]. While prior work accelerates individual RAG components, it overlooks *Search & Retrieval*, where storage access is a key bottleneck. To comprehensively accelerate *Search & Retrieval*, RAGX uses an in-storage accelerator that not only significantly reduces storage access overheads but also performs both similarity search and embedding generation for language models.

In-storage acceleration. In-storage acceleration [23, 42] has been applied to diverse fields, including genomics [58], database operations [11], and deep learning [55, 56]. Despite this progress, only a small body of work has explored in-storage acceleration for approximate nearest neighbor [37, 89]. These inspiring works are point solutions that use custom hardware for graph traversal or bitonic sorting. These solutions do not provide a programmable accelerator capable of supporting diverse embedding-based and keyword-based RAGs or executing end-to-end RAG pipelines. They also omit support for embedding generation, a core component in RAGs that rely on language models such as ColBERT (206 MB) or GTR (419.62 MB) to map queries into the same representation space as database entries. RAGX proposes a completely novel metamorphic accelerator for in-storage acceleration of end-to-end RAG execution in datacenters for both keyword-based and embedding-based representations that necessitates accelerating language models for query embedding.

7 Conclusion

Retrieval-augmented generation (RAG) is gaining traction in enterprise applications due to its ability to combine LLMs with real-time information retrieval from databases. As deployments shift to persistent storage to handle larger databases, the *Search & Retrieval* phase–not LLM inference–becomes the primary source of end-to-end latency. To address this, we propose a shape-shifting metamorphic in-storage architecture that supports diverse RAG algorithms, dynamic data structures, and query embedding generation within storage. Evaluation shows that RAGX delivers up to 4.3× and 1.5× improvements in throughput over CPU- and GPU-based retrieval pipelines, respectively. These results show that considering end-to-end execution of LLMs in enterprise applications, such as RAG, calls for innovations that cross systems and architecture.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. We thank Ashwin Rohit Alagiri Rajan for assisting with experimental infrastructure setup. This work was in part supported by generous gifts from Google, Samsung, as well as the National Science Foundation (NSF) award CCF#2107598. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes not withstanding any copyright notation thereon. The views contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of Google, Samsung, NSF, or the U.S. Government.

References

[1] Ahmet Yasin Aytar, Kemal Kilic, and Kamer Kaya. 2024. A Retrieval-Augmented Generation Framework for Academic Literature Navigation in Data Science. *arXiv* (2024).

[2] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. 2024. MEDUSA: Simple LLM inference acceleration framework with multiple decoding heads. In *ICML*.

[3] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. In *NeurIPS*.

[4] Qi Chen, Bing Zhao, Haidong Wang, Mingqin Li, Chuanjie Liu, Zengzhong Li, Mao Yang, and Jingdong Wang. 2021. Spann: Highly-efficient billion-scale approximate nearest neighborhood search. In *NeurIPS*.

[5] Rongxin Cheng, Yifan Peng, Xingda Wei, Hongrui Xie, Rong Chen, Sijie Shen, and Haibo Chen. 2024. Characterizing the Dilemma of Performance and Index Size in Billion-Scale Vector Search and Breaking It with Second-Tier Memory. *arXiv* (2024).

[6] Ha Cho, Tae Jun, Y.H. Kim, Hee Kang, Imjin Ahn, Hansle Gwon, Yunha Kim, Hyeram Seo, Heejung Choi, Minkyoung Kim, JiYe Han, Gaeun Kee, Seohyun Park, and Soyoung Ko. 2024. Task-Specific Transformer-Based Language Models in Health Care: Scoping Review. *JMIR Medical Informatics* 12 (2024).

[7] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *ICLR*.

[8] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *NeurIPS*.

[9] Matthew Davis, Mark Dustin, and Rahul Agarwal. 2023. Accelerate generative AI workloads on Amazon Aurora with optimized reads and pgvector. https://aws.amazon.com/blogs/database/accelerate-generative-ai-workloads-on-amazon-aurora-with-optimized-reads-and-pgvector/.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*.

[11] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *SIGMOD*.

[12] Ishita Doshi, Dhritiman Das, Ashish Bhutani, Rajeev Kumar, Rushi Bhatt, and Niranjan Balasubramanian. 2021. LANNS: a web-scale approximate nearest neighbor lookup system. In *VLDB*.

[13] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv* (2024).

[14] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. 2021. TurboTransformers: an efficient GPU serving system for transformer models. In *PPoPP*.

[15] Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE v2: Sparse Lexical and Expansion Model for Information Retrieval. *arXiv* (2021).

[16] Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. 2021. SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking. In *SIGIR*.

[17] Chris Fregly. 2024. Amazon Bedrock Retrieval-Augmented Generation (RAG) Workshop. https://github.com/aws-samples/amazon-bedrock-rag-workshop.

[18] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2019. Fast approximate nearest neighbor search with the navigating spreading-out graph. In *VLDB*.

[19] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2013. Optimized product quantization for approximate nearest neighbor search. In *CVPR*.

[20] Soroush Ghodrati, Sean Kinzer, Hanyang Xu, Rohan Mahapatra, Yoonsung Kim, Byung Hoon Ahn, Dong Kai Wang, Lavanya Karthikeyan, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, and Hadi Esmaeilzadeh. 2024. Tandem Processor: Grappling with Emerging Operators in Neural Networks. In *ASPLOS*.

[21] Google. 2025. Google AI for Developers. https://ai.google.dev/gemini-api/docs/long-context.

[22] Rentong Guo, Xiaofan Luan, Long Xiang, Xiao Yan, Xiaomeng Yi, Jigao Luo, Qianya Cheng, Weizhi Xu, Jiarui Luo, Frank Liu, Zhenshan Cao, Yanliang Qiao, Ting Wang, Bo Tang, and Charles Xie. 2022. Manu: a cloud native vector database management system. In *VLDB*.

[23] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. 2022. In-storage Processing of I/O Intensive Applications on Computational Storage Drives. In *ISQED*.

[24] Jennifer Hsia, Afreen Shaikh, Zhiruo Wang, and Graham Neubig. 2024. RAGGED: Towards Informed Design of Retrieval Augmented Generation Systems. *arXiv* (2024).

[25] Haiyang Huang, Newsha Ardalani, Anna Sun, Liu Ke, Shruti Bhosale, Hsien-Hsin S. Lee, Carole-Jean Wu, and Benjamin Lee. 2024. Toward Efficient Inference for Mixture of Experts. In *NeurIPS*.

[26] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.* 43, 2, Article 42 (2025).

[27] Suyeon Hur, Seongmin Na, Dongup Kwon, Joonsung Kim, Andrew Boutros, Eriko Nurvitadhi, and Jangwoo Kim. 2023. A Fast and Flexible FPGA-based Accelerator for Natural Language Processing Neural Networks. *ACM Trans. Archit. Code Optim.* 20, 1 (2023).

[28] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. 2024. Pre-gated MoE: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference. In *ISCA*.

[29] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. 2019. Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *ISCA*.

[30] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. 2019. Diskann: Fast accurate billion-point nearest neighbor search on a single node. In *NeurIPS*.

[31] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. In *ACL*.

[32] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. 2024. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. In *VLDB*.

[33] Wenqi Jiang, Shuai Zhang, Boran Han, Jie Wang, Bernie Wang, and Tim Kraska. 2025. PipeRAG: Fast retrieval-augmented generation via algorithm-system co-design. In *KDD*.

[34] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *ISCA*.

[35] Hamza Khan, Asma Khan, Zainab Khan, Lun Bin Huang, Kun Wang, and Lei He. 2021. NPE: An FPGA-based Overlay Processor for Natural Language Processing. In *FPGA*.

[36] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *SIGIR*.

[37] Ji-Hoon Kim, Yeo-Reum Park, Jaeyoung Do, Soo-Young Ji, and Joo-Young Kim. 2022. Accelerating large-scale graph-based nearest neighbor search on a computational storage platform. *IEEE Trans. Comput.* 72, 1 (2022).

[38] Inc. Kioxia America. 2024. How to Accelerate Vector Databases Without High DRAM Costs: Use Disk-Based Vector Indexes with Fast PCIe 5.0 SSDs from Kioxia. https://blog-us.kioxia.com/post/2024/11/12/how-to-accelerate-vector-databases-without-high-dram-costs-use-disk-based-vector-indexes-with-fast-pcie-5-0-ssds-from-kioxia.

[39] Anastasia Krithara, Anastasios Nentidis, Konstantinos Bougiatiotis, and Georgios Paliouras. 2023. BioASQ-QA: A manually curated corpus for Biomedical Question Answering. *Scientific Data* 10, 1 (2023).

[40] Youngeun Kwon and Minsoo Rhu. 2022. Training Personalized Recommendation Systems from (GPU) Scratch: Look Forward Not Backwards. In *ISCA*.

[41] Jey Han Lau and Timothy Baldwin. 2016. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation. In *RepL4NLP*.

[42] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. 2020. SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD. *IEEE Comput. Archit. Lett.* 19, 2 (2020).

[43] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In *OSDI*.

[44] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *NeurIPS*.

[45] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. 2020. Ftrans: energy-efficient acceleration of transformers using fpga. In *ISLPED*.

[46] Siran Li, Linus Stenzel, Carsten Eickhoff, and Seyed Ali Bahrainian. 2025. Enhancing Retrieval-Augmented Generation: A Study of Best Practices. In *COLING*.

[47] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. 2019. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Trans. on Knowl. and Data Eng.* 32, 8 (2019).

[48] Jimmy Lin, Xueguang Ma, Sheng-Chieh Lin, Jheng-Hong Yang, Ronak Pradeep, and Rodrigo Nogueira. 2021. Pyserini: A Python toolkit for reproducible information retrieval research with sparse and dense representations. In *SIGIR*.

[49] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. In *MLSys*.

[50] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. In *ASPLOS*.

[51] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO*.

[52] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. 2020. Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer. In *SOCC*.

[53] Zepu Lu, Jin Chen, Defu Lian, Zaixi Zhang, Yong Ge, and Enhong Chen. 2024. Knowledge distillation for high dimensional search index. In *NeurIPS*.

[54] Xueguang Ma, Tommaso Teofili, and Jimmy Lin. 2023. Anserini Gets Dense Retrieval: Integration of Lucene's HNSW Indexes. In *CIKM*.

[55] Rohan Mahapatra, Soroush Ghodrati, Byung Hoon Ahn, Sean Kinzer, Shu-Ting Wang, Hanyang Xu, Lavanya Karthikeyan, Hardik Sharma, Amir Yazdanbakhsh, Mohammad Alian, and Hadi Esmaeilzadeh. 2024. In-storage domain-specific acceleration for serverless computing. In *ASPLOS*.

[56] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. 2019. DeepStore: In-Storage Acceleration for Intelligent Queries. In *MICRO*.

[57] Yu A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 4 (2020).

[58] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. 2022. GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis. In *ASPLOS*.

[59] Jing Miao, Charat Thongprayoon, Supawadee Suppadungsuk, Oscar A. Garcia Valencia, and Wisit Cheungpasitporn. 2024. Integrating Retrieval-Augmented Generation with Large Language Models in Nephrology: Advancing Practical Applications. *Medicina* 60, 3 (2024).

[60] Microsoft. 2024. RAG and Generative AI - Azure AI Search. https://learn.microsoft.com/en-us/azure/search/retrieval-augmented-generation-overview.

[61] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO*.

[62] National Library of Medicine. 2023. MEDLINE/PubMed Baseline Repository (MBR). https://lhncbc.nlm.nih.gov/ii/information/MBR.html.

[63] Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith Hall, Daniel Cer, and Yinfei Yang. 2022. Sentence-T5: Scalable Sentence Encoders from Pre-trained Text-to-Text Models. In *ACL (Findings)*.

[64] Jianmo Ni, Chen Qu, Jing Lu, Zhuyun Dai, Gustavo Hernandez Abrego, Ji Ma, Vincent Zhao, Yi Luan, Keith Hall, Ming-Wei Chang, and Yinfei Yang. 2022. Large Dual Encoders Are Generalizable Retrievers. In *EMNLP*.

[65] Nvidia. 2024. NVIDIA System Management Interface (nvidia-smi) Documentation. https://docs.nvidia.com/deploy/nvidia-smi/index.html.

[66] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. 2024. Fine-Tuning or Retrieval? Comparing Knowledge Injection in LLMs. In *EMNLP*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.).

[67] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. 2024. AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference. In *ASPLOS*.

[68] Xiao Peng and Liang Chen. 2024. Athena: Retrieval-augmented Legal Judgment Prediction with Large Language Models. *arXiv* (2024).

[69] Shahzad Qaiser and Ramsha Ali. 2018. Text mining: use of TF-IDF to examine the relevance of words to documents. *International Journal of Computer Applications* 181, 1 (2018).

[70] Yubin Qin, Yang Wang, Dazheng Deng, Zhiren Zhao, Xiaolong Yang, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. 2023. FACT: FFN-Attention Co-optimized Transformer Architecture with Eager Correlation Prediction. In *ISCA*.

[71] Yubin Qin, Yang Wang, Zhiren Zhao, Xiaolong Yang, Yang Zhou, Shaojun Wei, Yang Hu, and Shouyi Yin. 2024. MECLA: Memory-Compute-Efficient LLM Accelerator with Scaling Sub-matrix Partition. In *ISCA*.

[72] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating Retrieval-Augmented Generation. In *ASPLOS*.

[73] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. 2023. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics* 11 (2023).

[74] Saeed Rashidi, William Won, Sudarshan Srinivasan, Srinivas Sridharan, and Tushar Krishna. 2022. Themis: a network bandwidth-aware collective scheduling policy for distributed training of DL models. In *ISCA*.

[75] Jie Ren, Minjia Zhang, and Dong Li. 2020. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In *NeurIPS*.

[76] Stephen Robertson and Hugo Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.* 3, 4 (2009).

[77] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (2012).

[78] Amazon SageMaker. 2024. Question Answering with Cohere and LangChain using JumpStart. https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_amazon_algorithms/jumpstart-foundation-models/question_answering_retrieval_augmented_generation/question_answering_Cohere+langchain_jumpstart.html.

[79] Amazon SageMaker. 2024. Question Answering with Pinecone and LLaMA-2 using JumpStart. https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_amazon_algorithms/jumpstart-foundation-models/question_answering_retrieval_augmented_generation/question_answering_pinecone_llama-2_jumpstart.html.

[80] Amazon Web Services. 2024. Amazon Bedrock Knowledge Bases. https://aws.amazon.com/bedrock/knowledge-bases/.

[81] Amazon Web Services. 2024. What is OpenSearch? https://aws.amazon.com/what-is/opensearch/.

[82] Michael Shen, Muhammad Umar, Kiwan Maeng, G. Edward Suh, and Udit Gupta. 2024. Towards Understanding Systems Trade-offs in Retrieval-Augmented Generation Model Inference. *arXiv* (2024).

[83] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval Augmentation Reduces Hallucination in Conversation. In *EMNLP (Findings)*.

[84] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. https://arxiv.org/abs/2307.09288. *arXiv* (2023).

[85] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*.

[86] Hanyin Wang, Chufan Gao, Christopher Dantona, Bryan Hull, and Jimeng Sun. 2024. DRG-LLaMA: tuning LLaMA model to predict diagnosis-related group for hospitalized patients. *npj Digital Medicine* 7, 1 (2024).

[87] Mengzhao Wang, Weizhi Xu, Xiaomeng Yi, Songlin Wu, Zhangyang Peng, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, Rentong Guo, and Charles Xie. 2024. Starling: An I/O-Efficient Disk-Resident Graph Index Framework for High-Dimensional Vector Similarity Search on Data Segment. *Proc. ACM Manag. Data* 2, 1 (2024).

[88] Shu-Ting Wang, Hanyang Xu, Amin Mamandipoor, Rohan Mahapatra, Byung Hoon Ahn, Soroush Ghodrati, Krishnan Kailas, Mohammad Alian, and Hadi Esmaeilzadeh. 2024. Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators. In *HPCA*.

[89] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Hai "Helen" Li, and Yiran Chen. 2024. NDSEARCH: Accelerating Graph-Traversal-Based Approximate Nearest Neighbor Search through Near Data Processing. In *ISCA*.

[90] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *SYSTOR*.

[91] Yuming Xu, Hengyu Liang, Jin Li, Shuotao Xu, Qi Chen, Qianxi Zhang, Cheng Li, Ziyue Yang, Fan Yang, Yuqing Yang, et al. 2023. SPFresh: Incremental In-Place Update for Billion-Scale Vector Search. In *SOSP*.

[92] Zhongkai Yu, Shengwen Liang, Tianyun Ma, Yunke Cai, Ziyuan Nan, Di Huang, Xinkai Song, Yifan Hao, Jie Zhang, Tian Zhi, Yongwei Zhao, Zidong Du, Xing Hu, Qi Guo, and Tianshi Chen. 2024. FlashLLM: A Chiplet-Based In-Flash Computing Architecture to Enable On-Device Inference of 70B LLM. In *MICRO*.

[93] Boyu Zhang, Hongyang Yang, Tianyu Zhou, Muhammad Ali Babar, and Xiao-Yang Liu. 2023. Enhancing Financial Sentiment Analysis via Retrieval Augmented Large Language Models. In *ICAIF*.

[94] Xuejiao Zhao, Siyan Liu, Su-Yin Yang, and Chunyan Miao. 2025. MedRAG: Enhancing Retrieval-augmented Generation with Knowledge Graph-Elicited Reasoning for Healthcare Copilot. In *WWW*.

A Artifact Appendix

A.1 Abstract

This artifact provides scripts that can be executed on various machines, including AWS EC2 instances, to evaluate RAGX. These scripts facilitate database generation for all search and retrieval benchmarks used in our evaluation. This appendix includes detailed instructions for setting up and running benchmarks on the CPU-DRAM baseline, as it is easily accessible and suitable for verifying the process, along with the RAGX simulator.

A.2 Artifact Check-List

- **Program:** For query embedding, we use publicly available implementations of BM25 [48], SPLADEv2 [15, 16], ColBERT [36], Doc2Vec [41], and GTR [64]. To generate databases, we use Faiss [13] for HNSW-based embedding-based retrieval, and Pyserini [48] and SPLADEv2's official GitHub [15] for inverted indices used in keyword-based retrieval.
- **Compilation:** Docker images include all required libraries.
- **Model:** We use encoder-based language models to generate both passage and query embeddings. The loading and usage of these models are integrated into our provided scripts.
- **Dataset:** We use PubMed [62] for passages and BioASQ [39] for queries; download instructions are provided.
- **Run-time environment:** Ubuntu 22.04, CUDA 12.1.0
- **Hardware:** Passage embedding generation was performed using four A10 GPUs. RAGX search and retrieval experiments were run on an AWS `r7g.8xlarge.search` instance. For small datasets (e.g., 0.5M passages), CPU instances with at least 64GB of DRAM can be used.
- **Metrics:** The primary metric is execution time (in seconds).
- **Output:** The artifact generates latency measurements for the query embedding and retrieval phases for both the CPU-DRAM baseline and the RAGX simulator.
- **Experiments:** A Docker container is provided, with all steps detailed in the README.
- **Disk Space:** 100 GB
- **Time Needed - Workflow:** 1 hour
- **Time Needed - Experiments:** 3 hours
- **Publicly Available:** Yes
- **Archived:** 10.5281/zenodo.15092497
- **Publicly available:** https://github.com/rohanmahapatra/ragx

A.3 Description

Following the README instructions, users can set up benchmarks, run the CPU-DRAM baseline for latency, and use the RAGX simulator for embedding, search, and retrieval.

A.3.1 Software Dependencies. The README in the artifact repository provides instructions for manually setting up the software environment. For ease of use, we also provide a Docker container that includes all prebuilt dependencies. We recommend using the Docker setup to ensure consistency and avoid manual installation issues.

A.4 Installation

Follow the steps/instructions provided in the GitHub repository. A brief overview of the steps is also outlined below. First, clone the repository. After cloning, build the Docker image by executing:

- Build the Docker image: `./build_docker.sh`
- Launch with GPU (recommended): `./run_docker_gpu.sh`
- Or launch with CPU: `./run_docker_cpu.sh`

This opens a shell in `/app` with scripts to download data, build databases, measure CPU-DRAM, and run the RAGX simulator.

A.5 Experiment Workflow

Shell scripts for executing the full workflow are located in the `/app` directory of the Docker container. The workflow consists of four scripts:

- `download_datasets.sh`: Downloads the required datasets and builds the 0.5M passage dataset used throughout the artifact.
- `build_databases.sh`: Constructs the search and retrieval databases for BM25, ColBERT, Doc2Vec, and GTR (note: SPLADEv2 is excluded from the functional artifact).[1]
- `run_cpu_dram.sh`: Runs the evaluation for CPU-DRAM and stores the results in `/app/baseline-CPU-DRAM/cpu_dram_results`.
- `run_ragx_simulations.sh`: Runs the evaluation for RAGX and stores the results in `/app/ragx.simulator/ragx-results.csv`.

A.6 Evaluation and Expected Results

After running the experiment workflow, the results for CPU-DRAM will be available in `/app/baseline-CPU-DRAM/cpu_dram_results`. These results report the per-query latency for search and retrieval. For embedding-based retrieval, the latency is further broken down into two components: embedding latency and search latency. The expected CPU-DRAM latencies are listed below. (Note: actual times may vary depending on the system used for evaluation.)

- BM25: total search and retrieval latency – 0.008s
- ColBERT: embedding – 0.009s, search and retrieval – 0.0002s
- Doc2Vec: embedding – 0.0003s, search and retrieval – 0.0007s
- GTR: embedding – 0.07s, search and retrieval – 0.01s

For RAGX, the results are available on `/app/ragx.simulator/ragx-results.csv`. The CSV file provides the median, average, and minimum time (in milliseconds) for a single query. It is important to note that each query point traverses the graph (e.g., HNSW) and scores a varying number of points, which necessitates the compilation of multiple kernels. For this artifact, we provide several precompiled kernel examples that the simulator uses to generate outputs and demonstrate functionality.

A.7 Experiment Customization

The primary point of customization is the dataset. To use a dataset other than PubMed, the user should provide a new dataset in JSONL format. After preparing the new dataset, the user should update the dataset path in the scripts. They can then run the database and CPU-DRAM scripts as usual.

[1] Generating databases requires substantial time and computational resources. For instance, a 50M-entry database requires 48 hours on 4 A10 GPUs with 500 GB memory, while a 500M-entry database takes 96 hours. To ensure feasibility, the artifact is limited to 500K entries. We also provide database traces with query-specific embedded passages and precomputed scores.