PipeRAG: Fast Retrieval-Augmented Generation via Adaptive Pipeline Parallelism

Wenqi Jiang[†]
ETH Zurich
Zurich, Switzerland

Shuai Zhang
Amazon Web Services
Santa Clara, USA

Boran Han
Amazon Web Services
Santa Clara, USA

Jie Wang[†]
Meta
Santa Clara, USA

Bernie Wang
Amazon Web Services
Menlo Park, USA

Tim Kraska
MIT and AWS
Boston, USA

Abstract

Retrieval-augmented generation (RAG) can enhance the generation quality of large language models (LLMs) by incorporating external token databases. However, retrievals from large databases can constitute a substantial portion of the overall generation time, particularly when retrievals are periodically performed to align the retrieved content with the latest states of generation. In this paper, we introduce PipeRAG, a novel algorithm-system co-design approach to reduce generation latency and enhance generation quality. PipeRAG integrates (1) pipeline parallelism to enable concurrent retrieval and generation processes, (2) flexible retrieval intervals to maximize the efficiency of pipeline parallelism, and (3) a performance model to automatically balance retrieval quality and latency based on the generation states and underlying hardware. Our evaluation shows that, by combining the three aforementioned methods, PipeRAG achieves up to 2.6× speedup in end-to-end generation latency while improving generation quality. These promising results showcase the effectiveness of co-designing algorithms with underlying systems, paving the way for the adoption of PipeRAG in future RAG systems.

ACM Reference Format:
Wenqi Jiang[†], Shuai Zhang, Boran Han, Jie Wang[†], Bernie Wang, and Tim Kraska. 2025. PipeRAG: Fast Retrieval-Augmented Generation via Adaptive Pipeline Parallelism. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3690624.3709194

1 Introduction

Retrieval-augmented generation (RAG) enhances auto-regressive large language models (LLMs) by conditioning on contextually relevant content retrieved from external databases. While one retrieval prior to the generation process can be enough when generating short sequences [13, 28], a more general approach involves periodic retrievals throughout the generation [3, 19, 34, 37, 42]. This necessity arises due to the potential shift in the generation context, such as changes in topics. Therefore, periodic retrievals ensure the retrieved content remains relevant to the latest context of the generation (Appendix A showcases a concrete example). A popular example of this category is Retro [3], which tailors the transformer neural network architecture to support the integration of retrieved content at regular intervals.

However, periodic retrievals on large databases, potentially comprising trillions of tokens [3], can significantly slow down the sequence generation. *We ask: can we optimize the system performance of RAG while preserving or even improving generation quality?*

We propose PipeRAG, a pioneering approach to improve RAG efficiency via a collaborative algorithm-system co-design — including a system-aware RAG algorithm and an algorithm-aware retrieval system as overviewed in Figure 1.

The foundation of PipeRAG is established on three observations centered on performance. Firstly, the dependencies between retrievals and LLM inferences lead to hardware underutilization, with either the inference or retrieval system being idle at any given time during the generation process (**O1**). Secondly, the inference latency per token increases with sequence lengths, due to the growing workloads of the attention mechanism in transformer neural networks (**O2**). Lastly, the retrieval process, particularly the approximate nearest neighbor search, exhibits a trade-off between search latency and search quality (**O3**).

The key idea of PipeRAG is to prefetch content from databases to facilitate pipeline parallelism between the inference and retrieval systems. This solution reduces end-to-end generation latency by allowing simultaneous inference and retrievals, effectively addressing the hardware inefficiencies identified in O1 (**S1**). We then enhance this key idea with two additional solutions. On the model side, PipeRAG modifies Retro's attention mechanism to support flexible retrieval intervals (invoke a retrieval after generating a certain number of tokens), because the intervals must be carefully tuned to capitalize the efficiency of pipeline parallelism (**S2**). On the system side, the retrieval system adopts a performance model informed by O2 and O3 to dynamically adjust the retrieval search space (the amount of database vectors to scan per query) according to the latency expectation of the upcoming token inferences in the pipeline, thereby optimizing search quality without increasing end-to-end generation latency (**S3**).

Our evaluation of PipeRAG, involving various evaluation datasets and using large databases based with up to 200 billion tokens,

[†]Work done while at Amazon Web Services. Correspondence to: Wenqi Jiang <wenqi.jiang@inf.ethz.ch> and Shuai Zhang <shuaizs@amazon.com>.

KDD '25, August 3–7, 2025, Toronto, ON, Canada

ACM ISBN 979-8-4007-1245-6/25/08
https://doi.org/10.1145/3690624.3709194

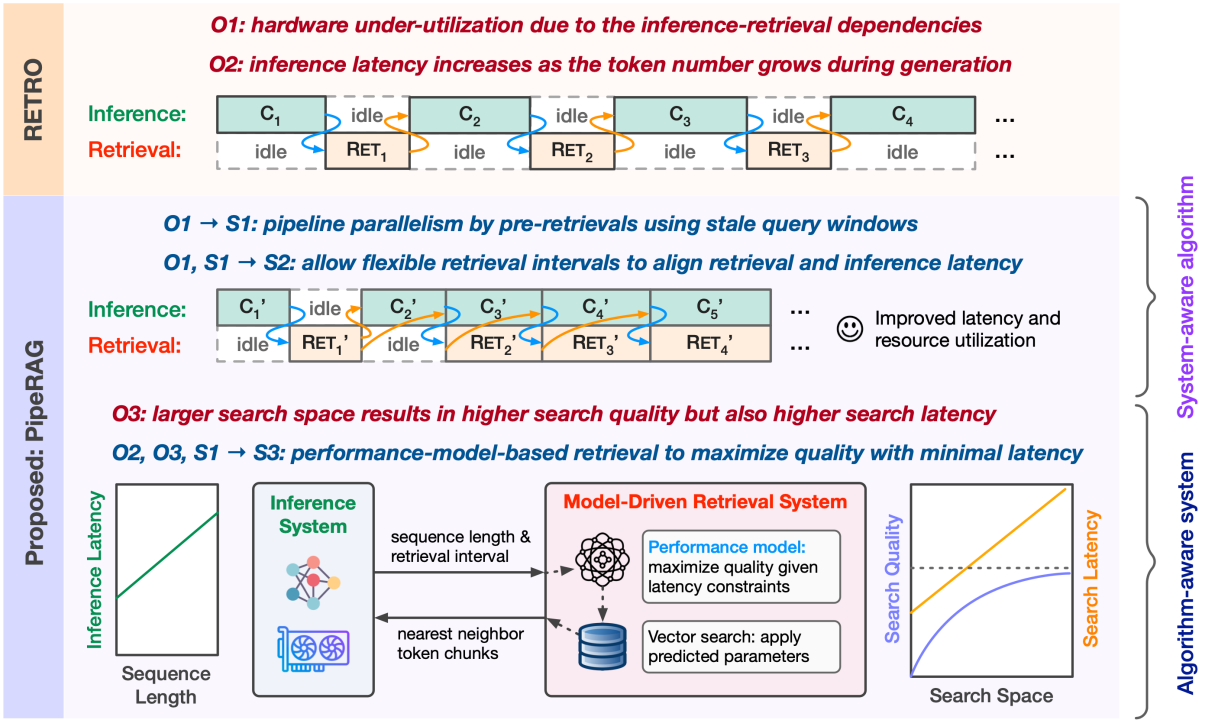

Figure 1: Based on three performance-centric observations (O1~O3), PipeRAG combines a system-aware algorithm integrating pipeline parallelism (S1) with flexible retrieval intervals (S2) and an algorithm-aware retrieval system guided by a performance model (S3).

clearly illustrates its efficiency in both generation performance (latency) and generation quality (perplexity). Specifically, the quality-performance Pareto frontier of PipeRAG significantly outperforms that of Retro: PipeRAG can achieve up to 2.6× speedup in latency without compromising perplexity; alternatively, maintaining the same latency allows PipeRAG to reduce perplexity by as much as 0.93 compared to Retro. These encouraging results highlight the importance of algorithm-system co-design in retrieval-augmented generation, paving the way for deploying PipeRAG in future RAG systems.

Contributions: We propose PipeRAG, an algorithm-system co-design approach aimed at improving retrieval-augmented generation efficiency. Specifically:

- We design a system-aware RAG algorithm that leverages pipeline parallelism, whose efficiency is further improved by supporting flexible retrieval intervals.
- We propose an algorithm-aware retrieval system that uses performance models to dynamically balance search quality and performance.
- We showcase the impressive performance of PipeRAG in various datasets, demonstrating the importance of algorithm-system co-design in optimizing RAG.

2 Background and Motivation

Sequence generation quality of LLMs can be improved through periodically retrieving from large token databases [3, 34, 37]. Here, periodic retrievals, instead of retrieving only once, are essential in handling potential contextual shifts during generation, such as topic changes, ensuring alignments between the retrieved content and the evolving generation context (a concrete example can be found in Appendix A). Retro is a representative model in this category [3]. As illustrated in Figure 2, Retro integrates a retrieval system with an inference system for token generation. It employs an encoder for incorporating retrieved tokens and a decoder for token generation.

Database construction. A Retro database comprises a large collection of documents segmented into n chunks of tokens $S = (S_1, \ldots, S_n)$, where each chunk S_i spans m tokens. These token chunks are each converted into vector representations $R(S)$. The database is then structured as a key-value store, with keys being the vector representations $R(S)$ and values corresponding to the original token chunks S, along with F, in which F_i representing the immediately following token chunks of each chunk S_i. Given a query vector q, the database performs an approximate nearest neighbor (ANN) search to retrieve k closest token chunks and their immediately following chunks.

Retrieval process. Retro performs retrievals at regular intervals during the generation phase. Specifically, when generating a sequence of t tokens $X = (x_1, \ldots, x_t)$, Retro partitions X into l chunks $(C_1, \ldots, C_l)$, each consisting of m tokens. Consequently, token x_i belongs to chunk $C_{\lceil \frac{i}{m} \rceil}$. For the generation of chunk C_i,

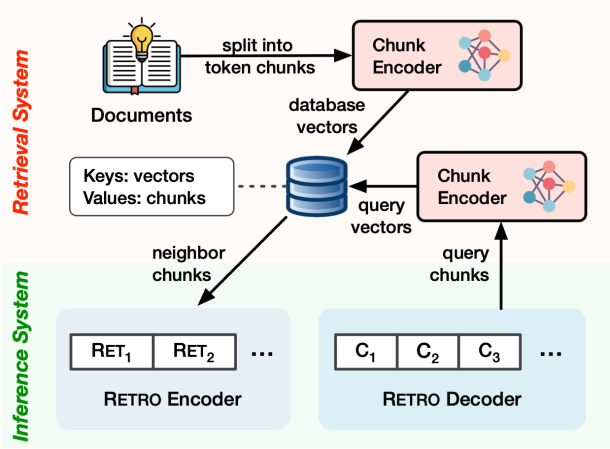

Figure 2: Retrieval augmentation with RETRO.

RETRO employs the preceding chunk C_{i-1} as the query to retrieve k nearest neighbors $\text{RET}(C_{i-1})$ from the database.

Attention mechanisms. RETRO involves both decoder-to-encoder and encoder-to-decoder attention mechanisms. The decoder within RETRO utilizes chunked cross-attention to integrate the retrieved information encoded by the encoder. To preserve causality, the generation of a chunk C_i incorporates the retrieved tokens $\text{RET}(C_{i-1})$ by integrating the encoder states $\text{ENC}(\text{RET}(C_{i-1}))$. On the other hand, the RETRO encoder states $\text{ENC}(\text{RET}(C_{i-1}))$ integrates the decoder's states of the $\text{DEC}(C_{i-1})$ via a standard cross-attention (CA) mechanism, such that the encoder can blend the retrieved information with the generation context. Because both decoder-to-encoder and encoder-to-decoder attention mechanisms operate on a chunk-wise basis, RETRO avoids the excessive computational demands of attending to all previous retrieval and generation states.

Motivation: improving RAG efficiency. Although periodically retrieving tokens from a large database can effectively improve the generation quality of LLMs, frequent retrievals can account for a considerable portion of the total generation time, thereby significantly slowing down the end-to-end generation process.

In this paper, we ask the following question: **is it possible to further enhance the efficiency of retrieval augmented generation?** Here, we conceptualize *RAG efficiency* as a Pareto frontier considering two objectives: *generation quality* and *system performance*. Specifically, given a quality requirement (achieving certain perplexity), can we optimize RAG's system performance (reducing generation latency)? On the other hand, given a system performance requirement, can we improve the quality of generation?

3 Our Approach: PipeRAG

We propose PipeRAG, a novel retrieval augmented generation approach to improve the performance-quality Pareto frontier through an in-depth algorithm-system co-design. The development of PipeRAG stems from performance-centric observations revealing (1) the *fundamental* system inefficiencies in existing RAG algorithms and (2) the distinct performance characteristics of LLM inference and retrieval systems. Based on these observations, PipeRAG includes (1) a system-aware RAG algorithm to address the system inefficiencies and (2) an algorithm-aware retrieval system to dynamically balance retrieval quality and latency.

3.1 Performance-Centric Observations in RAG

O1: Hardware inefficiency due to RAG dependencies. A conventional RAG process introduces dependencies between retrievals and inferences: the current generation context is used as a query to retrieve relevant token chunks stored in the database; the inference process must wait for the retrieval to finish before it can continue generating a few more tokens, until the next retrieval is triggered.

A RAG system typically comprises two sub-systems: the retrieval system and the inference system, each hosted on separate hardware platforms. AI accelerators such as GPUs and TPUs are the ideal hardware platforms for LLM inference due to the high demands for computation and memory bandwidth during inference. On the other hand, the retrieval systems consisting of large databases are usually not based on GPUs. This is because (1) the limited memory capacity of individual GPUs (GPUs adopt high-bandwidth memory that is fast but limited in capacity) makes the hosting of large databases cost-prohibitive, necessitating the setup comprising many GPUs, and (2) the communication bandwidth between the CPU and GPU is significantly lower compared to GPU's device memory bandwidth, thus the CPU-GPU solution, in which database vectors are stored in CPU-side memory and then transferred to GPUs at query time, could be exceedingly slow. Given the capacity requirements, the retrieval system is typically CPU-based [3, 28], with the database either held in substantial main memory (DRAM), or, in more budget-friendly setups, stored on disks.

Given that the two systems are based on separate hardware, the dependencies between retrievals and inferences in RAG result in significant underutilization of hardware resources. Figure 1 illustrates this inefficiency using RETRO as a representative example: due to the dependencies, either the inference or retrieval system is idle at any given time during the generation process, leading to hardware inefficiencies.

O2: Increasing inference time with sequence length. In a standard transformer neural network [43], the cost of generating each new token correlates with the sequence length, rather than remaining a constant. This is due to the attention mechanism in transformers: although the workload of the fully-connected layers remains constant throughout the generation process, the cost of attention layers increases with the sequence length [2]. Specifically, for each new token generated, the query states (Q) of the most recent token are compared against the key states (K) of all preceding tokens to calculate relevance scores. These scores are then utilized for a weighted sum over the value states (V) (note that the queries, keys, and values mentioned here under the context of transformers are distinct from those terms in RAG systems). Consequently, the inference cost per token can be approximated as a linear function to sequence length.

O3: Trade-offs between retrieval quality and latency. Large-scale vector search in RAG employs approximate nearest neighbor (ANN) search instead of exact nearest neighbor search due to the latter's prohibitive cost on large databases. In ANN search, database vectors are indexed, with popular choices including clustering-based inverted-file (IVF) indexes [41] and graph-based indexes [30,

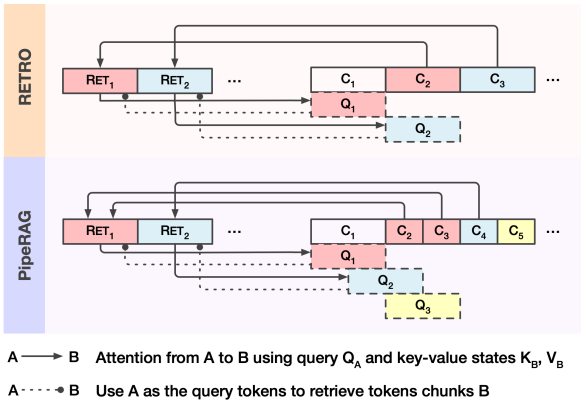

Figure 3: PipeRAG's attention mechanism.

31]. Optionally, database vectors may also be compressed via product quantization (PQ) [15] to shrink database sizes and reduce memory bandwidth usage at query time at the expense of search accuracy. During a search, a query vector is only compared against a subset of database vectors selected by the index.

Regardless of the index types, there exists a *fundamental* trade-off between search quality and latency in ANN search. Typically, the index first directs the search towards those database vectors that are most likely to be the nearest neighbors of the query vector, and then gradually expands the search space. The number of database vectors scanned per query can be directly or indirectly controlled by ANN search hyper-parameters. Expanding the search space would enhance the probability of finding the query vector's true nearest neighbors in the database (improved search quality), but also would also lead to higher latency (lower search performance) due to the greater number of comparisons between query vectors and database vectors.

Figure 1 visualizes the relationship between search quality and latency [15]. As the search space expands (number of scanned database vectors), the search quality (recall of the retrieval) gradually improves until reaching a plateau where the nearest neighbors are likely found. Simultaneously, the search cost (latency) grows linearly with the search space, with an initial cost of scanning the index (which could be zero in some graph-based indexes).

3.2 Algorithm-System Co-deisgn in PipeRAG

Given the aforementioned performance-centric observations, we propose PipeRAG, an algorithm-system co-design approach aimed at enhancing RAG's performance-quality Pareto frontier. PipeRAG addresses the *fundamental* issue of hardware inefficiency (O1) by employing pipeline parallelism (S1) and allowing flexible retrieval intervals (S2). Leveraging the distinct performance characteristics of the inference and retrieval sub-systems (O2, O3), PipeRAG further offers an option to enable automatic search space selection within the retrieval system, facilitating high-quality generation without introducing additional generation latency.

S1: Pipeline parallelism across RAG sub-systems. Because the hardware under-utilization issue in RAG is caused by dependencies between retrievals and inferences, our first solution is about revisiting RAG algorithms to enable pipeline parallelism: the retrievals and inferences should be executed concurrently, thus overlapping their execution latency and improving hardware utilization.

To facilitate pipeline parallelism, we relax the RAG dependencies as illustrated in Figure 1: instead of depending on the content retrieved using the query representing the most recent generation context (the latest generated tokens), the inference process can utilize a slightly older, or *stale*, query window to *prefetch* content from the database. The intuition here is that if the stale query window closely aligns with the latest generation context, it is likely to retrieve content similar to that obtained using the most recent query tokens. Once the dependency constraint is relaxed, retrievals can be proactively initiated to *prefetch* content from the database, thus enabling pipeline parallelism as shown in Figure 1.

Formally, when generating token chunk C_{j+1}, PipeRAG does not use the immediately preceding chunk as the query $Q = C_j = (x_{jm}, \ldots, x_{jm+m-1})$ to retrieve $\textsc{Ret}(Q)$. Instead, it opts for a stale token window $\hat{Q} = (x_{jm-s}, \ldots, x_{jm+m-1-s})$ as an approximate query, offset by s tokens from the latest query window. Subsequently, $\hat{\textsc{Ret}}(Q) = \textsc{Shift}(\textsc{Ret}(\hat{Q}), s)$ serves as the approximation of $\textsc{Ret}(Q)$. Given that the stale query is s tokens behind the most recent generation context, the retrieved results $\textsc{Ret}(\hat{Q})$ are correspondingly left-shifted by s tokens. This shift ensures that the first s retrieved tokens, which are likely less relevant for the upcoming generation due to staleness, are excluded while maintaining the overall length of retrieval tokens. Note that the concept of stale query windows does not apply for the initial retrieval, which is conducted using the first chunk C_1, as illustrated in Figure 1.

S2: Flexible retrieval intervals. Retro utilizes a fixed retrieval interval of $m = 64$, aligning with the generation chunk size, database token chunk size, and query window size. However, the effectiveness of pipeline parallelism (S1) is maximized when the retrieval and inference subsystems have similar latencies — generating $m = 64$ tokens does not always consume similar time as one retrieval.

In order to improve the effectiveness of pipeline parallelism, PipeRAG supports alternative retrieval intervals m' and modifies Retro's attention mechanism accordingly. Here, m' remains constant during a single generation process but can vary from the default value of 64. When using shorter intervals, such as $m' = 32$, the staleness of queries is also reduced ($s = 32$, thereby improving the quality of the retrieved content to more closely resemble that obtained from a non-stale query. Figure 3 illustrates the differences in retrievals and attention mechanisms between Retro and PipeRAG, taking $m' = 32$ as an example. As shown in the figure, while a query Q_i still has a window size of $m = 64$ tokens, the retrieval interval is halved. This necessitates adjustments in the attention regions to align with these modified intervals. For encoder-to-decoder attention, the attention is directed from the retrieved chunk to the query window whose position is different from that of Retro. For decoder-to-encoder attention, the generation of chunk C_{j+1} of length m' applies chunked cross-attention on $\textsc{Ret}(Q_{j-1})$.

S3: Performance-model-driven retrievals. PipeRAG has the potential to match the generation latency of LLMs that do not introduce retrievals, especially when the retrievals and inferences

are completely overlapped in the pipeline. However, achieving this ideal overlap is challenging because of the distinct performance characteristics of the retrieval and inference systems as introduced in O2 and O3.

To address this, we propose a performance-model-driven retrieval system to automatically enable perfectly overlapped pipeline windows. In this context, a performance model refers to any model (not limited to neural networks) designed to predict the performance characteristics of a system. Specifically, the retrieval system takes the generation states as inputs and automatically adjusts the search space using performance models, ensuring that the retrieval latency can be hidden by the generation latency of the next token chunk. By maximizing the search space under the latency constraint, the retrieval quality is also maximized without incurring extra generation latency.

The inference performance can be modeled as follows. The time required to generate a token chunk can be represented by $T_C = T_{\text{ENC}} + T_{\text{DEC}}$. The latency of encoder inference is related to the number of retrieved neighbors and the number of tokens per neighbor, while the decoder inference latency depends on the current sequence length and the chunk size (O2).

On the other hand, retrieval latency can be represented modeled as $T_{\text{RET}} = T_{Network} + T_{EncQuery} + T_{ScanIndex} + T_{ScanVec}$, encompassing the time spent on network communications, encoding the query tokens as vectors, scanning the vector index, and scanning a subset of database vectors. In this paper, we apply the widely-adopted IVF-PQ vector search algorithm [15] that combines a clustering-based inverted-file (IVF) index with product quantization (PQ). The IVF index clusters the database to *nlist* IVF lists. At query time, *nprobe* out of the *nlist* IVF lists are selected to scan (database vectors within the selected lists are compared to the query vectors).

As the performance of both retrievals and inferences are related to hardware, we measure and model their performance on the deployment hardware. We record the time consumption of both encoder and decoder inferences with various input sequence lengths. For retrieval, we model the relationship between *nprobe* and search latency using linear regression, given that *nprobe* is approximately proportional to the number of scanned database vectors.

The retrieval system then leverages these performance models to predict the maximal search space, indicated by *nlist*, given the latency constraint for generating the next token chunk, ensuring that $T_{\text{RET}} \leq T(C)$. Since the $T(C)$ can be easily obtained from the recorded performance numbers, we can then derive the maximal *nprobe* during the search based on the retrieval performance model.

While an alternative approach to achieve a perfectly overlapped pipeline is adjusting the retrieval intervals in the inference system, we rule out this option due to generalizability concerns. In future deployment scenarios, a retrieval system may serve multiple inference systems. Thus, the retrieval performance is impacted by the number of concurrent queries being processed. In this case, it could be challenging for the inference system to accurately predict the retrieval latency, as it lacks the information about the retrieval system's workload at the moment. Therefore, it is the retrieval system, instead of the inference system, that should be responsible for constructing a perfectly overlapped pipeline via performance modeling.

4 Evaluation

We evaluate PipeRAG in various aspects, showing its effectiveness in both generation quality and generation latency.

4.1 Experimental Setup

Database. Our token database was constructed from the C4 corpus with deduplicated English documents. We did not choose the Pile dataset as previous work [3] due to its current copyright issues. Adhering to [3], we segmented the documents into chunks of $m = 64$ tokens, yielding a total of three billion chunks. Following [34], we transformed each token chunk into a 384-dimensional vector using a sentence transformer[38] checkpoint *all-MiniLM-L6-v2*.

Model. We developed PipeRAG based on the Retro checkpoint with 582M parameters provided by [34], the only available pretrained Retro model when we conducted the experiments.

Evaluation Set. To evaluate language modeling quality, we used the Wikipedia dataset [1], the RealNews subset of the C4 dataset, and C4's English document subset [6, 36].

Software. Our implementation of the PipeRAG model is based on a Retro baseline obtained from [34], which is built on top of PyTorch. To enhance inference performance, we supported the caching of key-value states in the transformer and converted the model to ONNX format, enabling model inference by ONNX runtime. With the above optimizations, the inference latency on GPU is improved by around 3× over the original Pytorch implementation. We maintained the fp32 (32-bit floating point) precision of the model. For the retrieval system, we used the Faiss library [21], which is known for its efficient product-quantization-based vector search implementation. We adopted the IVF-PQ vector search algorithm, setting the number of IVF list centroids to $nlist = 16384$ and quantizing each 384-dimensional vector into 64 bytes of PQ code. During retrievals, we set the number of nearest neighbors as $k = 2$. The communication between the inference and retrieval systems was managed via the gRPC library.

Hardware. For model inference, we utilized an NVIDIA A100 GPU (40 GB). The retrieval process was handled by a server equipped with dual-socket Intel(R) Xeon(R) Platinum 8259CL CPUs @2.50GHz (48 cores and 96 threads) and 384 GB memory. The retrieval and inference servers were interconnected through a network, with a round-trip time of around 1 ms.

4.2 Perplexity Evaluation

We report the language modeling quality of PipeRAG as the main quality metric, because most QA benchmarks only contain short answers, which cannot exhibit the latency benefit of PipeRAG when generating longer sequences (some QA results are in Appendix C).

Figure 4 shows the impact of various retrieval strategies across different database sizes. This comparison includes PipeRAG, Retro, retrieval-augmented generation with only one retrieval at the beginning of generation, and generation without retrieval. For the last two strategies, Retro still serves as the base model. As indicated in the figure, retrieval, especially on large databases, plays a crucial role in improving generation quality (lower perplexity is better). Across all evaluated datasets, generation without retrieval performs the worst, followed by only retrieving once, showing the effectiveness of periodic retrieval in Retro. Additionally, perplexity

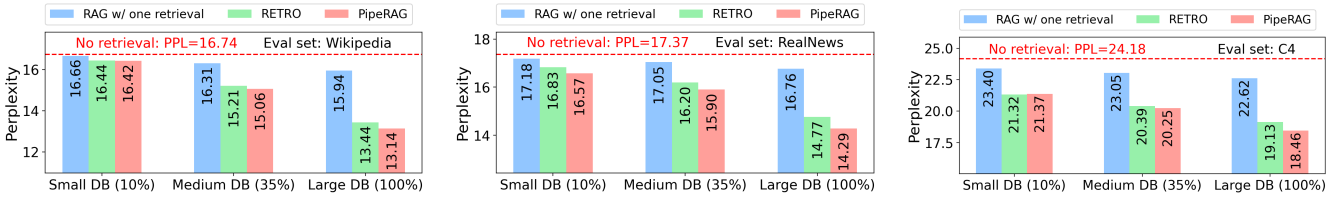

Figure 4: The effect of database sizes and retrieval strategies on language modeling perplexity (lower is better).

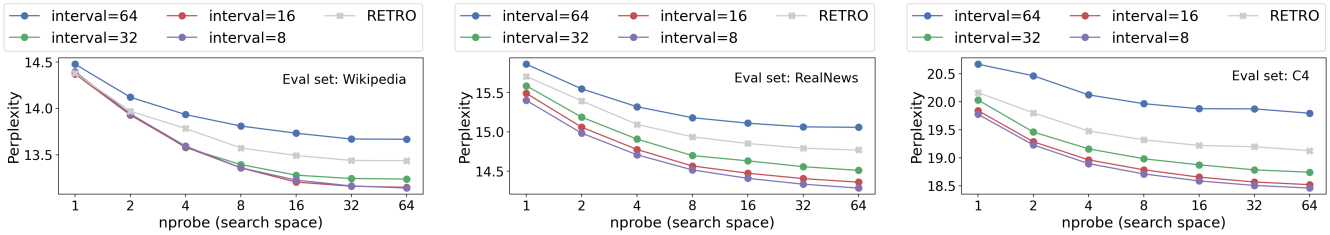

Figure 5: Perplexity of RAG when applying various retrieval intervals and search space configurations (*nprobe*).

decreases as the dataset size increases, highlighting the importance of comprehensive content coverage in the databases. Notably, when pairing with the largest database, PipeRAG outperforms Retro in generation quality, as we will analyze in greater detail later on.

From now on, we report results in generation quality and performance based on the full (largest) database, as using subsets significantly compromises generation quality.

Figure 5 compares the perplexity between PipeRAG and Retro across various retrieval configurations. We assess PipeRAG with different retrieval intervals, setting the search space through *nprobe*, which represents the number of scanned vector lists per query in the IVF index. As shown in Figure 5, both PipeRAG and Retro show reduced perplexity with an expanded search space, which leads to better search quality (O3).

Takeaway 1: The quality of retrieval-augmented generation benefits from higher retrieval quality achieved by expanding the search space during vector search.

Furthermore, PipeRAG demonstrates superior generation quality over Retro, particularly when using shorter retrieval intervals of no more than 32 (Figure 5). This advantage is attributed to PipeRAG's revised attention mechanism. Shorter intervals not only reduce query staleness (equivalent to the interval) but improve the content integration frequency, in contrast to Retro with a fixed interval of 64. The increased retrieval frequency in PipeRAG does not necessarily add to generation latency thanks to the pipeline parallelism, a point we will further elaborate on.

Takeaway 2: PipeRAG can surpass Retro in generation quality when using shorter retrieval intervals backed by PipeRAG's attention mechanism.

4.3 Performance-Quality Pareto Frontier

In this section, we assess the efficiency of PipeRAG. Our primary performance metric is the end-to-end latency to generate a 1024-token sequence, which we reported by taking the median latency of five individual runs.

Figure 6 compares the Pareto frontiers of the performance-quality (latency-perplexity) trade-offs between PipeRAG and Retro. For Retro, we manipulate the search space by tuning *nprobe*. For PipeRAG, we explore a range of retrieval intervals in conjunction with either a fixed search space or the performance-model-driven search space selection (S3). Across all datasets, the Pareto frontier of PipeRAG demonstrates significant advantages over Retro, as shown in Figure 6. For example, PipeRAG can attain up to a 2.6× reduction in latency while maintaining or reducing perplexity relative to Retro; alternatively, under the same latency constraint, PipeRAG can achieve lower perplexity of as much as 0.93 points compared to Retro.

Takeaway 3: PipeRAG shows impressive efficiency, achieving up to 2.6× speedup in latency over Retro without compromising generation quality.

Table 1 demonstrates the effectiveness of the proposed performance-model-driven retrieval system. The objective of the performance model is to dynamically maximize search quality while minimizing additional performance costs. To evaluate this, we compare the generation latency and quality of PipeRAG applying performance-model-driven retrievals to that of Retro as well as the same base Retro model without invoking retrievals. As shown in Table 1, PipeRAG achieves a notable reduction in perplexity (2.50~4.82) with a minor increase in performance overhead (merely 10.6%~13.2% in latency overhead), outperformance Retro in both latency and perplexity. This slight increase in latency is attributed to the extra

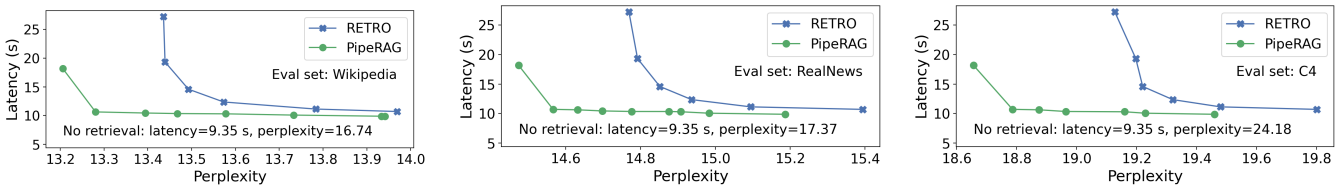

Figure 6: PipeRAG significantly outperforms RETRO on latency-perplexity trade-offs (lower latency and perplexity are better).

Table 1: Performance-driven retrieval (S3) facilitates latency comparable to non-retrieval models while reducing perplexity. Values in parentheses indicate the difference to the baseline model without retrieval (lower latency and perplexity are better).

Eval Set	Latency (s)			Perplexity		
	No retrieval	RETRO	Ours (S3)	No retrieval	RETRO	Ours (S3)
Wikipedia	9.35	14.59 (+5.23)	10.34 (**+0.99**)	16.74	13.49 (-3.25)	13.47 (**-3.28**)
RealNews	9.35	12.36 (+3.00)	10.58 (**+1.22**)	17.37	14.94 (-2.43)	14.87 (**-2.50**)
C4	9.35	11.13 (+1.78)	10.58 (**+1.22**)	24.18	19.48 (-4.70)	19.36 (**-4.82**)

Table 2: Average retrieval latency in milliseconds with varying search spaces.

nprobe	1	2	4	8	16	32	64	128
Latency (ms)	26.06	48.12	92.59	178.43	344.49	666.43	1298.52	2544.44

Table 3: Average generation latency per token as the sequence length (number of tokens) increases, with merged intervals.

Token ID	0 ~ 127	128 ~ 255	256 ~ 383	384 ~ 511
Latency (ms)	7.24	7.29	7.71	8.34
Token ID	512 ~ 639	640 ~ 767	768 ~ 895	896 ~ 1023
Latency (ms)	9.31	10.47	11.84	13.19

computational workload of the cross-attention mechanism when integrating the retrieved content from the encoder.

Takeaway 4: Leveraging performance-model-driven retrievals, PipeRAG can achieve comparable latency to models without retrievals while significantly improving generation quality.

4.4 Serving Performance on Various Hardware

We emphasize that PipeRAG is versatile across various deployment scenarios, which may involve a wide range of models, database scales, search algorithms, retrieval intervals, and hardware configurations. This versatility is essential, as it allows PipeRAG to adapt to environments where the latency balance between retrieval and generation may differ significantly.

To estimate PipeRAG's efficiency on various hardware, we model its performance using hypothetical hardware with enhanced inference and/or retrieval performance. We enable this by scaling the current latencies meansured in Table 2 and 3, which show the latencies for retrievals (with different search spaces indicated by *nprobe*, the number of lists to scan per query) and generation (where per-token inference latency increases with the number of tokens generated).

For RETRO, the end-to-end latency is the sum of inference and retrieval time. In PipeRAG, due to the parallelism, the latency for generating a chunk of tokens is determined by the maximum value of the inference and retrieval latency of that chunk, except for the first chunk where the pipeline is not yet active (see Figure 1).

We then input the measured performance of inference and retrievals into the performance model. This allows us to simulate performance scaling, such as a 4× improvement in retrieval or a 16× enhancement in inference. The result generation latency as well as the respective conclusions are included in Section 4. The model's accuracy is then verified by comparing these projected results against actual experimental data, with deviations found to be within a reasonable range (the median difference is only 5.7%).

Figure 7 illustrates the projected performance trends of PipeRAG across a range of system and hardware configurations. Considering the rapid advancements in hardware accelerators, we expect shifts in performance of both retrieval and inference systems over years. To analyze PipeRAG's effectiveness on future hardware, we model the latency of PipeRAG and RETRO when using faster retrieval or inference systems, with the methodology described in Appendix B. The first row of Figure 7 demonstrates the generation latency when the inference system becomes 4× and 16× faster, while the second row examines the effects of accelerated retrieval. Across all scenarios, PipeRAG achieves superior efficiency compared to RETRO. When either system experiences an order of magnitude speedup (e.g., 16×), however, the benefits of applying PipeRAG become less significant. This trend aligns with our expectations, as the effectiveness of pipeline parallelism peaks when both system components have comparable latencies and diminishes when one component significantly outpaces the other.

Table 4: Cosine similarity between content retrieved by stale and non-stale queries. The results indicate that stale queries are still highly effective in identifying relevant token chunks from the database.

	No staleness	Staleness (number of stale tokens in the query)						
		1	2	4	8	16	32	64
Wikipedia	1.0000	0.9262	0.9204	0.9138	0.9062	0.8990	0.8921	0.8875
RealNews	1.0000	0.9219	0.9147	0.9073	0.8996	0.8925	0.8850	0.8794
C4	1.0000	0.9323	0.9263	0.9193	0.9127	0.9052	0.8980	0.8929

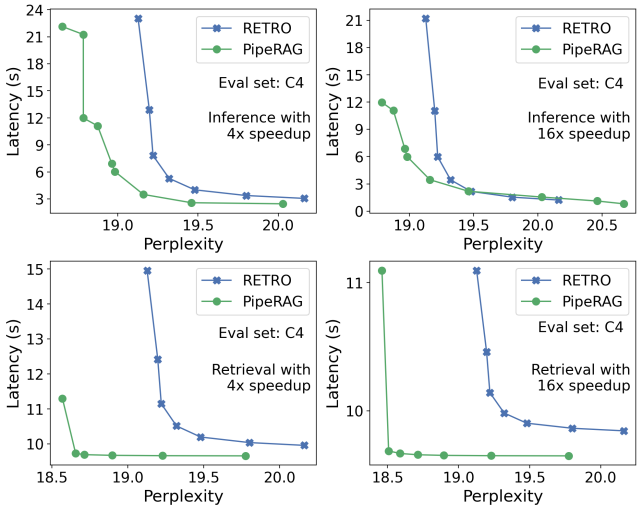

Figure 7: Trends in PipeRAG efficiency when deployed on future hardware that enables faster retrieval or inference.

Takeaway 5: PipeRAG outperforms Retro in efficiency across different hardware, though the extent of improvements depends on sub-system performance.

4.5 Ablation Study

The effectiveness of retrievals using stale queries. We investigate the fundamental applicability of prefetching content using stale queries. For this purpose, we compare the $k = 1$ nearest neighbors retrieved by non-stale queries in our evaluation set with their staleness versions. Same as Section 4, we use the largest C4 database, which consists of three billion token chunks, and set $nprobe = 64$ to ensure high retrieval quality. We then employ the *msmarco-bert-base-dot-v5* checkpoint from sentence transformers [38] to evaluate the cosine similarity between contents retrieved by stale and non-stale queries.

Table 4 presents the retrieval quality using stale queries. Here, we use different degrees of staleness, ranging from 1 token to 64 tokens, while maintaining a consistent retrieval interval of $m = 64$. The results indicate that, despite the staleness, the retrieved content closely resembles that obtained through non-stale queries, with around 90% cosine similarity across datasets. As expected, this similarity shows a gradual decline as the staleness increases.

Enable flexible retrieval interval in Retro baseline. Since PipeRAG not only introduces pipeline parallelism but also modifies Retro's attention mechanism to maximize the effectiveness of pipelining, it is natural to ask how a baseline model would perform if it integrates the same attention mechanism. To illustrate the effectiveness of pipeline parallelism itself, we compare PipeRAG with an enhanced variant of Retro, named Retro+, which also supports flexible retrieval intervals by integrating PipeRAG's attention mechanism.

Figure 8 compares the performance-quality Pareto-frontier between PipeRAG and Retro+. Both models use retrieval intervals ranging from 8 to 64. While Retro+, benefiting from flexible intervals, matches PipeRAG in perplexity, PipeRAG consistently achieves lower latency given the same perplexity. This is attributed to the proposed pipeline parallelism: PipeRAG effectively hides the retrieval latencies by overlapping them with generation latencies, whereas for Retro+, more frequent retrievals lead to increased total generation latency. More detailed comparisons between PipeRAG and Retro+ under identical retrieval intervals (corresponding to the same number of database requests) can be found in Appendix C.

Takeaway 6: Pipeline parallelism is essential for RAG efficiency, as PipeRAG outperforms Retro+ that supports flexible retrieval intervals using PipeRAG's attention mechanism.

5 Broader Applicability of PipeRAG

The idea of improving RAG efficiency through pipeline parallelism is broadly applicable across various RAG configurations, as long as they include periodic retrievals. In this paper, we have focused on improving RAG efficiency based on the Retro model and evaluated generation performance using specific hardware and software setups described in Section 4. In the future, RAG can evolve in several ways: models may adopt a decoder-only transformer architecture [4, 35] although the high cost of periodically appending the retrieved content has to be addressed [19, 37]; retrieval engines could incorporate LLM-based or BM25-based result reranking [7, 29, 33], instead of solely relying on vector-level similarity; and hardware may evolve to include dedicated retrieval accelerators [16–18]. However, regardless of these potential advancements in algorithms and hardware (detailed in Appendix B), the dependencies between retrievals and inferences in RAG systems — especially when retrievals are periodic — remains a *fundamental* obstacle to fully leveraging hardware resources and achieving maximal inference efficiency. Thus, whenever the time consumption of one retrieval and multiple steps of inferences are on a similar scale, pipeline parallelism by *prefetching* content from databases, which

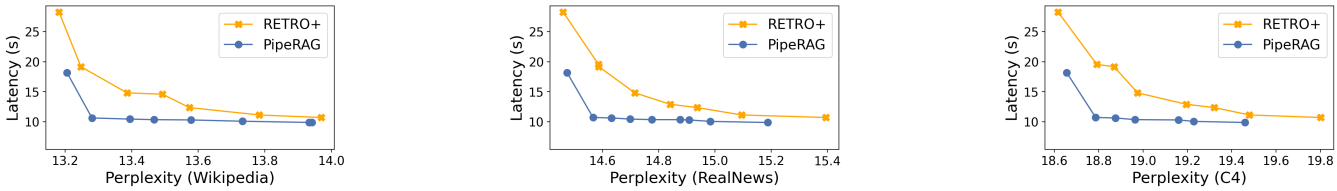

Figure 8: Even if the baseline model supports flexible retrieval intervals (RETRO+), PipeRAG still significantly outperforms it in efficiency thanks to the proposed pipeline parallelism.

can also be combined with retrieval caches [20, 46], should be a great option to improve generation efficiency.

Prefetching content from databases using stale queries is applicable regardless of the specific models used for generation. To demonstrate this, we show that using a stale query window can retrieve content very similar to that obtained via a regular query window, with detailed results included in Appendix C. These findings address a potential limitation in our evaluation, as our experimentation with PipeRAG was conducted using the RETRO checkpoint provided by [34], which was the only available RETRO checkpoint at the time of our research.

6 Related Work

To our knowledge, PipeRAG is a pioneer work to enhance RAG efficiency through an in-depth algorithm-system co-design, diverging from existing RAG research that mainly focuses on improving generation quality. We now briefly introduce these related works.

Since knowledge is primarily retrieved rather than encoded in the LLM's parameters, RALMs, even with LLMs of one to two orders of magnitude fewer parameters, can achieve superior or comparable performance to conventional LLMs on various natural language processing (NLP) tasks [11, 14, 26, 27]. While the generation may only involve a single passage retrieval at the beginning [13, 28, 39], the generated sequence may gradually diverge from the initially retrieved contents as the sequence grows longer. Thus, a more general RAG approach involves multiple retrievals during text generation to improve token generation quality [3, 37].

Another line of RAG research emphasizes token-level retrievals, exemplified by kNN-LM [24] and subsequent works [23, 32, 45]. In these models, during each token generation step, the hidden state of the last layer is used as a query to retrieve contextually similar passages as well as their subsequent tokens (with a retrieval interval of one). The next token of the current context is then predicted by interpolating the model's next-token probability distribution with that of the retrieved contents. There are also arguments suggesting that token-level content integration may not be as effective as integrating longer passages [44].

7 Conclusion

Iterative retrieval-augmented generation presents both opportunities and efficiency challenges, due to the overheads of retrieval on large databases. We propose PipeRAG, which improves generation efficiency by adopting pipeline parallelism, allowing flexible retrieval intervals, and dynamically adjusting retrieval quality via performance modeling. PipeRAG achieves up to 2.6× speedup over RETRO without compromising generation quality. This not only establishes a solid foundation for integrating pipeline parallelism in future RAG systems but also showcasing future research opportunities in optimizing RAG through algorithm-system co-design.

References

[1] The wikipedia dataset. https://www.tensorflow.org/datasets/community_catalog/huggingface/wikipedia.
[2] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.
[3] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pages 2206–2240. PMLR, 2022.
[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
[5] Zhuyun Dai and Jamie Callan. Deeper text understanding for ir with contextual neural language modeling. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 985–988, 2019.
[6] Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. *arXiv preprint arXiv:2104.08758*, 2021.
[7] Ehsan Doostmohammadi, Tobias Norlund, Marco Kuhlmann, and Richard Johansson. Surface-based retrieval reduces perplexity of retrieval-augmented language models. *arXiv preprint arXiv:2305.16243*, 2023.
[8] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR, 2022.
[9] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1):5232–5270, 2022.
[10] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
[11] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. Retrieval augmented language model pre-training. In *International conference on machine learning*, pages 3929–3938. PMLR, 2020.
[12] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2333–2338, 2013.
[13] Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open domain question answering. *arXiv preprint arXiv:2007.01282*, 2020.
[14] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299*, 2022.
[15] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.
[16] Wenqi Jiang, Hang Hu, Torsten Hoefler, and Gustavo Alonso. Accelerating graph-based vector search via delayed-synchronization traversal. *arXiv preprint arXiv:2406.12385*, 2024.

[17] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2023.

[18] Wenqi Jiang, Marco Zeller, Roger Waleffe, Torsten Hoefler, and Gustavo Alonso. Chameleon: a heterogeneous and disaggregated accelerator system for retrieval-augmented language models. *arXiv preprint arXiv:2310.09949*, 2023.

[19] Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation. *arXiv preprint arXiv:2305.06983*, 2023.

[20] Chao Jin, Zili Zhang, Xuanlin Jiang, Fangyue Liu, Xin Liu, Xuanzhe Liu, and Xin Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457*, 2024.

[21] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[22] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*, 2020.

[23] Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710*, 2020.

[24] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172*, 2019.

[25] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 39–48, 2020.

[26] Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. *arXiv preprint arXiv:2107.07566*, 2021.

[27] Mike Lewis, Marjan Ghazvininejad, Gargi Ghosh, Armen Aghajanyan, Sida Wang, and Luke Zettlemoyer. Pre-training via paraphrasing. *Advances in Neural Information Processing Systems*, 33:18470–18481, 2020.

[28] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[29] Sean MacAvaney, Andrew Yates, Arman Cohan, and Nazli Goharian. Cedr: Contextualized embeddings for document ranking. In *Proceedings of the 42nd international ACM SIGIR conference on research and development in information retrieval*, pages 1101–1104, 2019.

[30] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[31] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[32] Yuxian Meng, Xiaoya Li, Xiayu Zheng, Fei Wu, Xiaofei Sun, Tianwei Zhang, and Jiwei Li. Fast nearest neighbor machine translation. *arXiv preprint arXiv:2105.14528*, 2021.

[33] Rodrigo Nogueira and Kyunghyun Cho. Passage re-ranking with bert. *arXiv preprint arXiv:1901.04085*, 2019.

[34] Tobias Norlund, Ehsan Doostmohammadi, Richard Johansson, and Marco Kuhlmann. On the generalization ability of retrieval-enhanced transformers. *arXiv preprint arXiv:2302.12128*, 2023.

[35] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

[37] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. *arXiv preprint arXiv:2302.00083*, 2023.

[38] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[39] Devendra Singh Sachan, Mostofa Patwary, Mohammad Shoeybi, Neel Kant, Wei Ping, William L Hamilton, and Bryan Catanzaro. End-to-end training of neural retrievers for open-domain question answering. *arXiv preprint arXiv:2101.00408*, 2021.

[40] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. Colbertv2: Effective and efficient retrieval via lightweight late interaction. *arXiv preprint arXiv:2112.01488*, 2021.

[41] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *Computer Vision, IEEE International Conference on*, volume 3, pages 1470–1470. IEEE Computer Society, 2003.

[42] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. *arXiv preprint arXiv:2212.10509*, 2022.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[44] Shufan Wang, Yixiao Song, Andrew Drozdov, Aparna Garimella, Varun Manjunatha, and Mohit Iyyer. Knn-lm does not improve open-ended text generation. *arXiv preprint arXiv:2305.14625*, 2023.

[45] Frank F Xu, Uri Alon, and Graham Neubig. Why do nearest neighbor language models work? *arXiv preprint arXiv:2301.02828*, 2023.

[46] Zhihao Zhang, Alan Zhu, Lijie Yang, Yihua Xu, Lanting Li, Phitchaya Mangpo Phothilimthana, and Zhihao Jia. Accelerating retrieval-augmented language model serving with speculation. *arXiv preprint arXiv:2401.14021*, 2024.

A A Motivating Example of Periodic Retrievals

In this section, we present a concrete example demonstrating the effectiveness of *periodic retrievals* during sequence generation, a strategy that has been proven to significantly enhance the quality of language modeling [3, 34, 37].

Figure 9 illustrates the example, wherein the model is asked to describe a high-impact machine learning paper. In crafting its response, the model uses the Transformer neural network [43] as the target paper, covering several aspects related to the paper. The narrative evolves from a brief introduction of the model, through its impacts on various natural language processing tasks, to its influence on subsequent research, its cross-disciplinary applications, and ultimately, to emerging trends in research. Given these shifts in topic, the content initially retrieved about the Transformer architecture might lose relevance in the context of discussing future research trends. Therefore, periodic retrievals, in this instance, are vital to ensure that the retrieved content remains pertinent to the current context of generation.

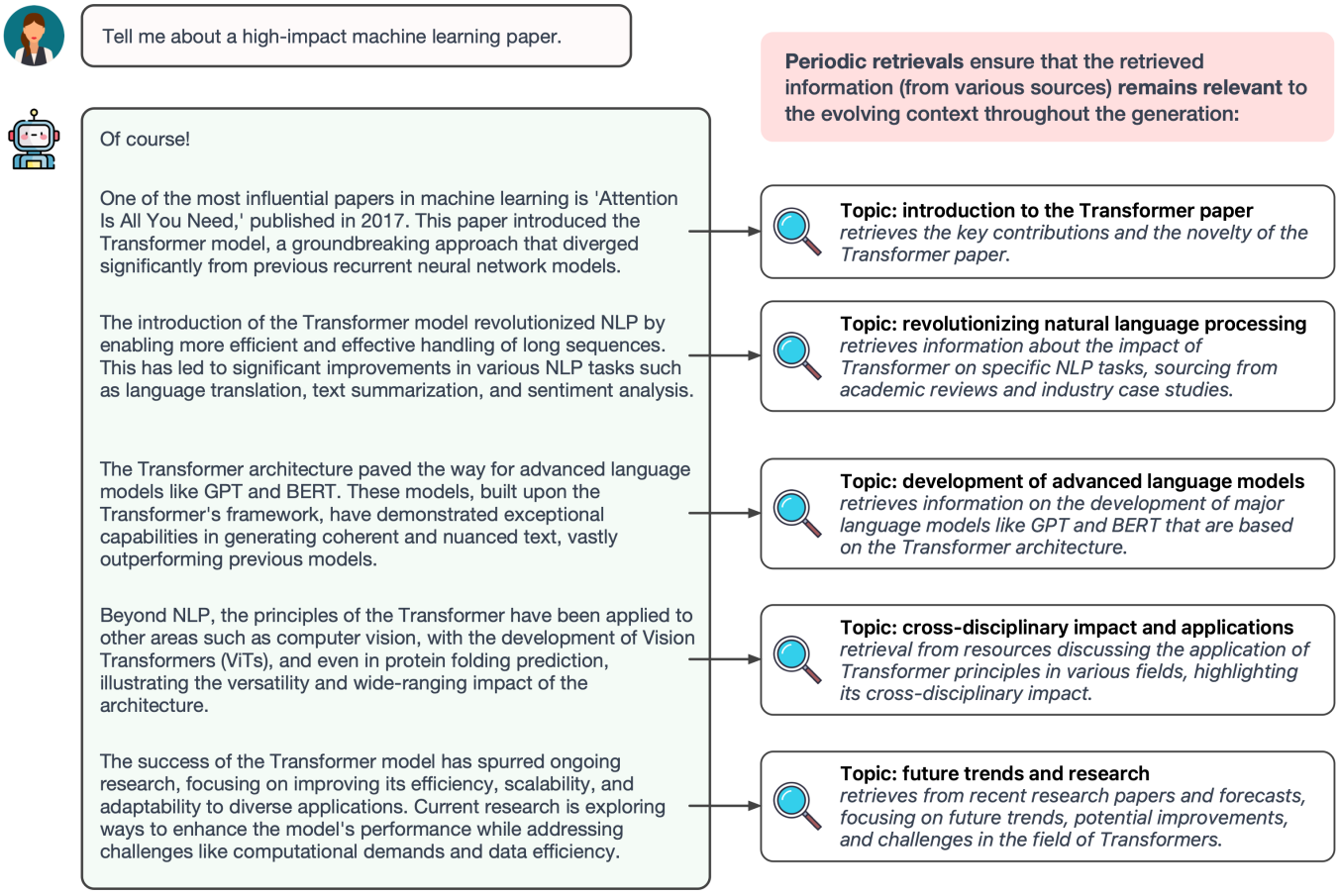

Figure 9: A motivating example of utilizing periodic retrievals during sequence generation.

B Performance Trends on Evolving Hardware

In this section, we begin by enumerating the factors that influence retrieval and inference performance. We then introduce the performance modeling methodology employed in Section 4, which projects PipeRAG's efficiency on future hardware configurations.

B.1 Factors Influencing Retrieval and Inference Performance

Retrieval performance depends on the following factors:

- **Hardware.** The memory bandwidth and computational capacity of the hardware used for retrieval are key factors influencing performance. It is worth noticing that there are emerging hardware accelerators that are specialized for retrievals [17] and integrated into RAG systems [18], offering impressive retrieval performance as well as cost efficiency.

- **Document numbers.** The total number of documents, along with encoding granularity as introduced below, determines the vector count in the database.
- **Encoding granularity.** Documents can be encoded in various granularities by LLMs, ranging from one vector per document [12, 22] to one vector per passage [5, 38] or even per token [25, 40].
- **Dimensionality.** The dimensionality of the database vectors, as well as the compression ratio when employing product quantization, are critical to retrieval performance.
- **Indexes.** The selection of indexes, such as IVF or graph-based ones, and their parameter configurations are crucial for retrieval efficiency.
- **Reranking.** Optionally, the retrieved content can be reranked using LLMs, which often yields better ranking quality than relying solely on vector similarity [33].

LLM inference performance is influenced by the following factors:

- **Hardware.** The performance of inference is heavily dependent on the hardware, particularly its memory bandwidth and computational capacity. LLM accelerators such as GPUs are evolving rapidly in these metrics.
- **Software.** The choice of software for inference also plays a significant role. For instance, PyTorch's eager execution mode might not fully exploit hardware accelerators due to the slow execution speed of Python programs. In such cases, software overhead could exceed the GPU kernel execution time.
- **Quantization**. Quantizing models to lower precisions can markedly reduce inference time, thanks to reduced memory footprint and bandwidth usage. For instance, converting models to 3-bit precision can lead to a 3~5× speedup compared to 16-bit floating point formats [10].
- **Sparsity**. Techniques like mixture-of-experts allow for scaling LLMs without proportionate increases in computational costs [8, 9], because only a small subset of neurons are activated during inference.

C Additional Experimental Results

C.1 Question Answering Quality

While the model checkpoint we used [34] was relatively small and was not fine-tuned on QA datasets, we still conducted QA experiments to show the effectiveness of PipeRAG over the baseline. Specifically, we conducted QA experiments on the open-domain version of the Natural Questions dataset. However, the ground truth answers are typically short (less than five tokens). In order to compare not only the QA quality but also the generation latency between PipeRAG and the baseline, we extended the model's output to sequences of 256 tokens, involving multiple retrievals. We evaluated several configurations: (1) no retrieval, which resulted in low latency, (2) RETRO with retrieval, which showed high latency, and (3) PipeRAG with retrieval, achieving low latency. For PipeRAG, the staleness was set as 64 tokens.

Table 5: Summary of recall and latency for different retrieval settings in QA experiments.

Setting	Average Recall	Latency (ms)
No retrieval	0.098	1859.2
RETRO	0.150	3237.16
PipeRAG	0.148	1920.6

Table 5 summarizes the recall and latency results. Retrieval-augmented settings, such as those employed by PipeRAG and the baseline, demonstrated significant improvements in generation quality compared to the no-retrieval configuration. Notably, even with a staleness of 64 tokens, PipeRAG achieved recall comparable to RETRO while delivering 1.64× lower latency.