APPENDICES FOR: TELERAG: EFFICIENT RETRIEVAL-AUGMENTED GENERATION INFERENCE WITH LOOKAHEAD RETRIEVAL

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code, evaluation scripts, and datasets for TeleRAG, an efficient retrieval-augmented generation (RAG) inference system that reduces latency and improves throughput via *lookahead retrieval*, a prefetching mechanism that predicts required data and transfers them from CPU to GPU in parallel with LLM generation. The artifact reproduces Table 3 and Figures 9–14 of the paper, covering single-GPU latency (RTX4090), single-GPU throughput (H100), multi-GPU scaling (H200), and prefetch cluster hit rates. All experiments are automated using GNU Make and run within a Docker container to ensure reproducibility.

A.2 Artifact check-list (meta-information)

- **Algorithm:** IVF cluster-based lookahead retrieval with prefetching, greedy cache-aware batch scheduling.
- **Program:** Python (`ragacc` library) and shell scripts.
- **Compilation:** Docker build.
- **Data set:** wiki_dpr dataset (Karpukhin et al., 2020), NaturalQuestions (Kwiatkowski et al., 2019), HotpotQA (Yang et al., 2018), and TriviaQA (Joshi et al., 2017).
- **Run-time environment:** Docker (CUDA 12.8, Ubuntu 22.04, Python 3), NVIDIA Container Toolkit
- **Hardware:** RTX 4090 (24 GB) for single-GPU latency; H100 (80 GB) for single-GPU throughput; up to 8×H200 (140 GB each) for multi-GPU scaling. 64+ GB system RAM for single-GPU, 900+ GB for 8-GPU experiments.
- **Run-time state:** No other GPU-intensive processes should be running during evaluation (PCIe bandwidth contention affects results).
- **Execution:** Automated via GNU Make targets.
- **Metrics:** End-to-end latency (ms), throughput (queries/s), retrieval speedup, prefetch cluster hit rate (%).
- **Output:** CSV files, PDF plots, JSON hit rate results.
- **How much disk space required (approximately)?:** ∼400 GB (200 GB dataset + models + index).
- **How much time is needed to prepare workflow (approximately)?:** 1–2 hours (download models/dataset, build Docker image).
- **How much time is needed to complete experiments (approximately)?:** up to 24 hours per GPU configuration.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache 2.0.
- **Workflow framework used?:** GNU Make and Bash.
- **Archived (provide DOI)?:** `https://doi.org/10.5281/zenodo.19361856`.

A.3 Description

A.3.1 How delivered

The artifact is available as a GitHub repository (`https://github.com/uw-syfi/TeleRAG`) that contains the `ragacc` library, evaluation scripts, and a Dockerfile. Models and datasets are downloaded separately from HuggingFace.[1] Detailed setup instructions are provided in `docs/artifact-evaluation.md`.

A.3.2 Hardware dependencies

For Single-GPU experiments, 1× NVIDIA RTX 4090 or 1× H100 with 64+ GB system RAM and PCIe Gen4/Gen5 is required. For Multi-GPU experiments, 8× NVIDIA H200 with 900+ GB system RAM and PCIe Gen5 is required. See Table 2 for details.

A.3.3 Software dependencies

The artifact requires Docker with NVIDIA Container Toolkit. The Dockerfile installs all software dependencies. A Hugging Face account with access to gated Llama 3 and Mistral models is required to download models.

A.3.4 Data sets

The published dataset contains the vector index trained for the wiki_dpr dataset (Karpukhin et al., 2020), and pregenerated RAG pipeline traces for three datasets (NaturalQuestions, HotpotQA, and TriviaQA (Kwiatkowski et al.,

[1] https://huggingface.co/datasets/lauyeeyu/TeleRAG-Dataset

2019; Yang et al., 2018; Joshi et al., 2017)) with GPT-3.5-Turbo (OpenAI, 2023).

A.4 Installation

Set up TELERAG by first cloning the TeleRAG repository.

```
$ git clone \
  https://github.com/uw-syfi/TeleRAG.git
```

Download the required Hugging Face models and datasets as described in `docs/artifact-evaluation.md`. Some models, including Llama and Mistral, are gated and require accepting their licenses before download.

```
$ hf download <model name>
```

Build the Docker image for TELERAG.

```
$ docker build -t telerag .
```

Launch a detached GPU-enabled Docker container with the required capabilities and directory mounts for the repository, models, and dataset.

```
$ docker run --gpus all \
  --cap-add=SYS_NICE \
  -d -it --name telerag-ae \
  -v "$(pwd)":/app \
  -v <path to models>:/hf_models \
  -v <path to datasets>:/data/ \
  telerag:latest
```

Then enter the running container.

```
$ docker exec -it telerag-ae /bin/bash
```

Inside the Docker container, run the smoke test to verify the setup.

```
$ make smoke-test
```

A.5 Experiment workflow

All experiments are orchestrated via GNU Make. Do not run experiments in parallel (`-j`) due to PCIe bandwidth contention.

1. **Single-GPU latency (RTX 4090):** `make 4090` runs FAISS and TeleRAG on 6 RAG pipelines across 3 datasets with Llama-3.2-3B and Llama-3-8B. `make 4090-plots` generates Figure 9.
2. **Single-GPU throughput (H100):** `make h100` runs hit rate calculation (Table 3), batch evaluation with Llama-3-8B and Mistral-Small-22B. `make h100-plots` generates Figures 10 and 12.
3. **Multi-GPU scaling (H200):** `make h200` runs multi-GPU evaluation with 1–8 H200 GPUs plus scheduling ablations. `make h200-plots` generates Figures 11, 13, and 14.

Results are saved as CSV files in `evaluation/` and PDF plots in `figure/`.

A.6 Evaluation and expected result

The generated plots should be compared with the corresponding figures in the paper:

- **Table 3:** Prefetch cluster hit rates per pipeline on NQ.
- **Figure 9:** TeleRAG should show consistent latency speedup over FAISS across all 6 pipelines and 3 datasets on RTX 4090.
- **Figure 10:** TeleRAG should achieve higher throughput than FAISS across batch sizes on H100.
- **Figure 11:** Throughput should scale with GPU count (1–8 H200 GPUs) across all datasets.
- **Figure 12:** Latency breakdown should show retrieval time reduction due to prefetching overlap.
- **Figure 13:** Cache-aware scheduling should improve throughput compared to without cache.
- **Figure 14:** Scheduling overhead should be small relative to total latency.

Performance numbers may vary across runs due to system load, thermal throttling, and PCIe contention. Ensure no other GPU-intensive processes are running. If results differ significantly, restart the Docker container and re-run.

A.7 Experiment customization

Key parameters can be adjusted in the shell scripts under `artifact_evaluation/`:

- `N_SAMPLES`: Number of evaluation samples (default: 1024 for single-GPU, 512 for multi-GPU).
- `N_RUNS`: Number of benchmark runs (default: 5).
- `NPROBE`: Number of IVF clusters to probe (default: 256).
- `VM_SIZE`: Prefetch buffer size in GB.

Prefetch budgets per pipeline/GPU/dataset are defined in `ragacc/pipeline_budgets.py`.

B Background: Inverted File Index (IVF) for Vector Search

The *vector index* is a crucial component of RAG applications that retrieves similar items from large datasets of high-dimensional vectors. Given the query vector $x \in \mathrm{R}^D$ and vector database $Y = \{y_0, \ldots, y_{N-1}\} \subset \mathrm{R}^D$, which comprises N vectors, the vector index aims to find the k nearest neighbors of x from the database:

$$\text{k-argmin}_{i=0:N} d(x, y_i),$$

where D is the dimensionality of the vector determined by the embedding model, and d denotes the distance function, which is typically the L2-norm or inner product (Johnson et al., 2019).

To improve search efficiency, the inverted file index (IVF) algorithm is widely used for large-scale vector databases due to its simplicity and effectiveness. IVF partitions the datastore into *clusters* and restricts searches to the most relevant clusters, reducing the computational cost. To obtain the cluster assignment of each stored vector, it performs a clustering algorithm such as k-means (Norouzi & Fleet, 2013) and partitions the database into N_c clusters:

$$\{C_1, C_2, \ldots, C_{N_c}\} = k\text{-means}(Y, N_c),$$

where C_j is the set of vectors assigned to the j-th cluster, and the cluster centroids $\{c_1, c_2, \ldots, c_{N_c}\}$ are obtained by taking the mean of vectors in $\{C_1, C_2, \ldots, C_{N_c}\}$. Then, each database vector $y_i \in Y$ is assigned to the nearest cluster center:

$$\text{Cluster}(y_i) = \text{argmin}_{j=1:N_c} d(y_i, c_j).$$

With the trained centroids and cluster assignment, we can perform a more efficient index search. There are two steps involved in the IVF index search. First, it will identify the nearest L cluster centroids of a query vector x:

$$\{c_{j_1}, c_{j_2}, \ldots, c_{j_L}\} = L\text{-argmin}_{j=1:C}\ d(x, c_j).$$

This step is also referred to as the coarse-grained search in IVF, as it identifies candidates at the cluster level. Second, the fine-grained search is performed in the L nearest clusters and identifies the k closest vectors of x:

$$k\text{-argmin}_{y_i \in \cup_{l=1}^{L} C_{j_l}} d(x, y_i).$$

This involves sorting. In this way, IVF reduces search space and accelerates the retrieval process. Here, the hyperparameter L from the first step is also referred to as `nprobe` (Douze et al., 2024). When `nprobe` is larger, the IVF retrieval will be more accurate as it searches more clusters. However, the retrieval latency is longer as more computation and data are needed.

C Finding Prefetching Amount

A key challenge in TeleRAG is to balance the benefit of reducing retrieval latency by prefetching data against the overhead of CPU-GPU transfers. Prefetching more clusters reduces the subsequent retrieval time but can also extend the transfer phase; if it extends beyond the LLM generation window, we lose the advantage of overlap and potentially introduce additional delay. Still, additional delays in data transfer may be worthwhile if they substantially reduce retrieval latency. To guide the choice of the optimal amount of data to prefetch, we develop a mathematical model based on profile information.

Mathematical model. Here, we denote b_p as the number of bytes to prefetch and B as the CPU-GPU bandwidth. The optimal amount of data to prefetch is denoted as b_p^*. To start, we let t_1 represent the combined time of prefetching and pre-retrieval LLM generation, and t_2 represent the retrieval time. We have: $t_1 = \max(t_{\text{LLM}}, t_p)$ and $t_2 = \max(t_c, t_g)$, where t_{LLM} is the time for LLM generation, t_p is prefetching time, t_c is the CPU retrieval time, and t_g is the GPU retrieval time. The objective is to minimize $t_1 + t_2$.

Since prefetching time t_p is proportional to b_p, we express t_1 as a piecewise function:

$$t_1 = \begin{cases} t_{\text{LLM}}, & \text{if } b_p \le B \cdot t_{\text{LLM}}, \\ \frac{b_p}{B}, & \text{if } b_p > B \cdot t_{\text{LLM}}, \end{cases} \tag{1}$$

From Eq. 1, if we prefetch fewer bytes than can be transferred during LLM generation, t_1 is effectively just t_{LLM} because the transfer overlaps completely with LLM execution.

As shown in the analysis section, GPU retrieval (t_g) is generally much faster than CPU retrieval (t_c). Thus, we assume $t_c \gg t_g$. As CPU has a limited parallelism, t_c usually grows proportionally with the number of clusters to be processed (Jiang et al., 2023):

$$t_2 = t_c = r_{\text{miss}} \times n_{\text{probe}} \times t_{cc}, \tag{2}$$

where r_{miss} is the miss rate (the percentage of IVF clusters that are not cached on the GPU), n_{probe} is the total number of clusters to search, and t_{cc} is the CPU time to search a single cluster. Increasing b_p can only decrease or maintain the current miss rate, *i.e.*, $\frac{\mathrm{d}r_{\text{miss}}}{\mathrm{d}b_p} \le 0$. Moreover, because clusters are prefetched in order of descending likelihood, we assume r_{miss} is either linear or a concave up function[2] of b_p, *i.e.*, $\frac{\mathrm{d}^2 r_{\text{miss}}}{\mathrm{d}b_p^2} \ge 0$.

We now examine two cases:

[2] An upward U-shaped function whose second derivative is positive.

- **Case 1:** $b_p^* \leq B \cdot t_{\text{LLM}}$. Here, $t_1 = t_{\text{LLM}}$ is constant because prefetching is fully overlapped with LLM generation. Since increasing b_p in this regime will not increase t_1 and cannot worsen the miss rate, pushing b_p to the boundary $B \cdot t_{\text{LLM}}$ minimizes $t_1 + t_2$. Hence,

$$b_p^* = B \cdot t_{\text{LLM}}. \tag{3}$$

- **Case 2:** $b_p^* > B \cdot t_{\text{LLM}}$. In this region, t_1 grows linearly with b_p, and we have: $\frac{\mathrm{d}^2}{\mathrm{d}b_p^2}(t_1 + t_2) = \frac{\mathrm{d}^2 r_{\text{miss}}}{\mathrm{d}b_p^2} \cdot n_{\text{probe}} \cdot t_{cc} \geq 0$. Therefore, $t_1 + t_2$ is concave up, allowing at most one minimum. At the minimum point, we have:

$$\frac{\mathrm{d}}{\mathrm{d}b_p}(t_1 + t_2) = 0 \implies \frac{1}{B} + \frac{\mathrm{d}r_{\text{miss}}}{\mathrm{d}b_p} \cdot n_{\text{probe}} \cdot t_{cc} = 0. \tag{4}$$

  From this, we obtain:

$$b_p^* = B \times n_{\text{probe}} \times t_{cc} \times \Delta r_{\text{miss}}, \tag{5}$$

  where Δr_{miss} is the decrement of the miss rate for this round. If b_p^* is indeed larger than $B \cdot t_{\text{LLM}}$, it becomes the global minimum; otherwise, the solution reverts to Case 1.

In summary, our analysis shows that the optimal prefetch amount b_p^* can only lie at one of two points: (1) Prefetch until LLM generation completes (*i.e.*, $b_p = B \cdot t_{\text{LLM}}$). (2) A point determined by Eq. 5. However, under typical CPU-GPU bandwidth (*e.g.*, 64 GB/s on PCIe 5), the time spent loading additional clusters often outweighs any retrieval latency reduction from lowering r_{miss}. Consequently, the second scenario in Eq. 5 becomes nearly infeasible in practice. Therefore, on current hardware, *prefetching exactly until LLM execution ends* is generally the most effective choice.

D Implementation Details

TeleRAG is implemented in Python and also leverages PyTorch's (Ansel et al., 2024) operator ecosystem for efficient computation. The datastore index is initially constructed using FAISS (Douze et al., 2024), and its data structures, such as IVF centroids and cluster data, are converted to PyTorch tensors.

At runtime, cluster data is loaded into a contiguous pinned memory region on the CPU, enabling non-blocking memory copies to the GPU. A fixed-size contiguous buffer on the GPU is allocated based on the user's configuration or GPU memory capacity during runtime.

We use separate processes for the schedulers and services in TeleRAG to avoid bottlenecks in Python global interpreter lock. For the communication between the scheduler and the services, we also leverage ZeroMQ (The ZeroMQ authors, 2025) as the communication framework. The message is encoded and decoded with Pickle in Python. For the LLM service, we use SGLang (Zheng et al., 2024) as the LLM backend. For the retrieval service, to implement the index search with GPU-CPU cooperation, on GPU, we use a single matrix-vector multiplication that computes distances for all prefetched vectors; on CPU, we utilize `multithreading` in Python to parallelize similarity searches across clusters. We then move the distances computed from CPU to GPU, merge with results on GPU and perform global sorting on GPU.

To monitor the hotness of cached clusters better, we use a **decay-based** approach to track the hotness of cached clusters. Every cached cluster has a hotness value. Upon fetching to the GPU, the cluster is essentially cached and will be assigned an initial value $h_0 = h_{\text{init}}$. Then, when a new round of hotness update starts, the hotness will be divided by a decay factor d, and if the cluster is used in the previous round, a hotness update value h_{inc} will be added to the hotness. Therefore, the transition of hotness from the round r to round $r + 1$ can be given as:

$$h_{r+1} = \begin{cases} h_r/d, & \text{if not used in round } r, \\ h_r/d + h_{\text{inc}}, & \text{if used in round } r. \end{cases} \tag{6}$$

In this way, the process can be parallelized easily on GPU, minimizing the overhead of hotness tracking in terms of both the time and computation. We also set a cache fraction, which means if the cache exceeds the fraction, we need to evict some clusters. The cache fraction is set to 0.5 by default. To make evaluation results stable and reproducible, we do cache eviction for the excessive clusters and consolidate the GPU memory after serving each batch. This makes sure that we will not over-evaluate the performance of the caching system.

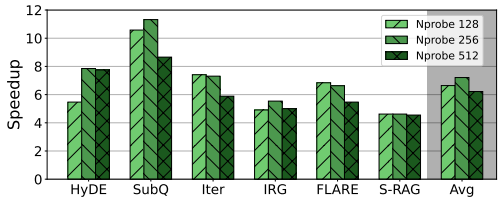
Both schedulers use greedy search to minimize overhead. The prefetching scheduler groups similar queries into micro-batches, and feeds them into the cache-aware scheduler to assign micro-batches to GPUs. The cache-aware scheduler will get the number of overlapped clusters on each GPU for each micro-batch, and assign the micro-batch to the GPU with the most overlapped clusters.

E Detailed Index Configurations

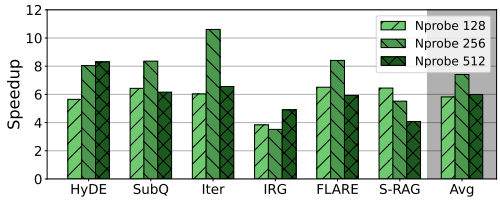
Table 4 shows the detailed configurations of our retrieval index used in the evaluation.

Specification	Value
Dataset	wiki_dpr (Karpukhin et al., 2020)
Dataset size	2.1 billion tokens
# of chunks	21 million
# of IVF cluster	4096
Embed model	Contriever (Izacard et al., 2021)
Embed dimension	768
Index type	`FLAT`[3]
Distance metric	`Inner Product`
Index size	61 GB

Table 4. Detailed configurations of our retrieval index.



(a) Llama-3.2-3B with an RTX4090 GPU.



(b) Llama-3-8B with a H100 GPU.

Figure 15. Single-query retrieval speedup on NQ with different `nprobe` values.

F Additional Evaluation Results

F.1 Retrieval Speedups Across `nprobe`

Figure 15 shows the retrieval latency reduction on NQ. We observe consistent speedups for all `nprobe` values. The greatest speedups are achieved at `nprobe` 256, with average speedups of 7.21× and 7.41× on RTX4090 and H100, respectively. As a larger `nprobe` value is used, the retrieval performance of TeleRAG becomes constrained by missed clusters on the CPU, given our fixed prefetch budget across varying `nprobe` values. However, RAG inference with higher `nprobe` will result in longer retrieval latency, and hence TeleRAG still has a significant latency improvement over the CPU retrieval baseline.

[3]Original embedding without compression for the best retrieval precision.

References

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C. K., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M. Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., and Chintala, S. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pp. 929–947, New York, NY, USA, 2024. URL https://doi.org/10.1145/3620665.3640366.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

Izacard, G., Caron, M., Hosseini, L., Riedel, S., Bojanowski, P., Joulin, A., and Grave, E. Unsupervised dense information retrieval with contrastive learning. *arXiv preprint arXiv:2112.09118*, 2021.

Jiang, W., Li, S., Zhu, Y., de Fine Licht, J., He, Z., Shi, R., Renggli, C., Zhang, S., Rekatsinas, T., Hoefler, T., et al. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2023.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7 (3):535–547, 2019.

Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, 2017.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., and Yih, W.-t. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., et al. Natural questions: A benchmark for question answering research. *Transactions of the*

Association for Computational Linguistics, 7:452–466, 2019.

Norouzi, M. and Fleet, D. J. Cartesian k-means. In *Proceedings of the IEEE Conference on computer Vision and Pattern Recognition*, pp. 3017–3024, 2013.

OpenAI. Gpt-3.5 turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo, 2023. Accessed: 2026-04-19.

The ZeroMQ authors. ZeroMQ. https://zeromq.org/, 2025.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.