MLS REPORT

Anonymous authors
Paper under double-blind review

ABSTRACT

This report focuses on creating an efficient, GPU-accelerated ML system for efficient information retrieval and scalable model serving. We implement optimized KNN, K-Means, and ANN algorithms using Pytorch and CUDA. GPU optimizations help in achieving up to 16x speedup over CPU and a recall rate of over 0.7. Further, we developed an efficient model serving system with request queuing and batching for Retrieval-Augmented Generation (RAG) tasks. Techniques like adaptive batching and parallel request handling were used, reducing latency by 35–50%. Together, these solutions create a scalable, efficient system suitable for real-world applications.

1 INFORMATION RETRIEVAL ON GPU (TASK 1)

1.1 MOTIVATION

Information Retrieval (IR) is a multidisciplinary field focused on extracting relevant information from large datasets in response to user queries, involving stages like document collection Zhang et al. (2001), indexing, query processing, ranking, and clustering. Clustering enhances IR by grouping similar documents, uncovering semantic patterns, and improving precision, especially for complex or ambiguous queries. However, scaling clustering algorithms like K-means to high-dimensional data introduces computational and memory challenges. GPU acceleration addresses these issues by enabling parallel distance computations and high-throughput memory access, significantly reducing latency and runtime. This makes large-scale clustering practical and impactful across domains like finance, healthcare, and AI.[1]

1.2 OVERVIEW

This system is a modular, GPU-accelerated clustering framework with three core functions: accurate k nearest neighbor (KNN) search, k-means clustering, and ANN retrieval with guaranteed recall. The system's data processing pipeline starts with generating data of different sizes and dimensions and converting them into PyTorch tensors. In different tasks, the system uses vectorized operations on tensors for calculations: for example, KNN calculates distances in batches and performs top-k selection; k-means iteratively updates cluster centers, relying on `argmin` distribution and `scatter` aggregation; and ANN combines clustering results with density-weighted scores to select candidate points from candidate clusters. The system performs runtime evaluations for all combinations of methods, distance metrics, and datasets, and calculates GPU speedup over CPU.

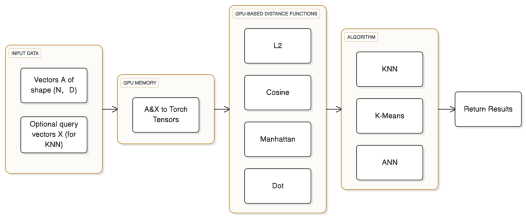

Figure 1: Workflow of GPU-accelerated vector search with distance computation.

[1] Ben Nancholas, "Clustering and cluster analysis: an explainer"

In the design process, this system follows several key design principles. Firstly, to achieve hardware independence, the system automatically allocates tensors to the CPU or GPU based on available resources. What's more, through batch processing and memory-aware execution strategies, the system can be extended to high-dimensional and even large-scale data. In addition, by modularizing the distance function and initialization process, good reusability Qi et al. (2023) and adaptability are achieved. Finally, the system adopts a centralized control process to ensure the repeatability of the evaluation process, so that the operation and comparative analysis under different test settings can be uniformly managed.

1.3 DISTANCE FUNCTIONS

Problem Definition:In large-scale machine learning systems, distance functions are essential for measuring similarity between data points, especially in unsupervised tasks like KNN, K-Means, and recommendation systems, where decision-making relies entirely on metrics such as L2, cosine, dot, and Manhattan distances. Given two vector sets $X[D]$ and $Y[D]$, the goal is to compute the pairwise distance matrix. However, this presents several system-level challenges: high computational cost (e.g., $N = 10{,}000$, $D = 1024$), memory bandwidth limitations during high-dimensional tensor operations, and significant CPU–GPU data transfer overheads that hinder GPU acceleration. Additionally, Python `for` loops are inefficient for large-scale data processing, making it difficult to meet performance demands.

Design Choice: In this task, we implement four distinct distance functions to compute the similarity between two vectors X and Y in a high-dimensional feature space. All functions are implemented using PyTorch for efficient GPU/CPU support. **L2 distance** (see equation1) measures straight-line distance and is effective for normalized features, commonly used in clustering like K-Means. **Manhattan distance** (L1, see equation 2) sums absolute differences, offering robustness to outliers and suitability for sparse data. **Dot product** (see equation 3) evaluates vector alignment, used as a similarity measure by negating its value, ideal for embedding-based retrieval. **Cosine distance** (see Equation 4) captures angular similarity, independent of vector magnitude, and is widely applied in NLP and recommendation systems.

Implementation Details: All distance functions were implemented using PyTorch tensor operations to fully exploit GPU-based parallelism B.The L2 distance is vectorized using the expanded formula $(x_i - y_i)^2$, leveraging `torch.matmul` for dot products and broadcasting to avoid explicit loops. Cosine distance is computed by normalizing vectors using `torch.nn.functional.normalize`, then calculating the dot similarity and converting it to distance via $1 -$ similarity. Dot distance is calculated by simply negating the dot product to transform similarity into a distance measure. Manhattan distance is computed using `torch.cdist` with $p = 1$, and for large-scale inputs, batch processing is employed to prevent memory overflow.

1.4 TOP-K RETRIEVAL

Problem Definition: The main problem addressed here is the efficient search for Top-k nearest vectors in higher-dimensional datasets. As the dataset grows in size, the process becomes more expensive and complicated to accurately determine the nearest neighbor. In situations like these, **ANN or Approximate Nearest Neighbor** is used. This function makes the process faster and more scalable in real-world situations while compromising on accuracy. It finds vectors close to the query without comparing with all database vectors. Some important terms include recall, which refers to the fraction of true nearest neighbors found by the ANN algorithm (higher recall indicates better performance), clustering, which involves grouping vectors to limit the search space, and GPU acceleration, which leverages parallel computing to efficiently handle large-scale operations.

Design Choice: To address the problem at hand, we used a clustering-based ANN approach, which was an optimized variant of the K-means clustering algorithm for indexing along with a priority-based search strategy. We use K-means as it narrows the search space and is well-supported in both CPU (scikit-learn) and GPU (cuML). All operations were implemented in **PyTorch**, and GPU acceleration was achieved via using **CuPy**, which utilises the **built-in CUDA** for performance boosts. The principle behind this approach is to: Partition the dataset into clusters using K-means, identify the most promising clusters for a query vector based on distance to centroids, search only within

these selected clusters reducing computation and use a mathematical formula $L \geq \frac{K-1}{1-r}$ to guarantee a minimum recall rate 'r'

Implementation Details: The implementation consists of two main components:

1. Clustering Functions - We used `our_kmeans_modified` with some basic optimizations for clustering. PyTorch tensors were used on CUDA devices when available for GPU acceleration. For large datasets, **batch processing** was used,i.e., data was processed in batches to avoid memory issues. Scatter operations for centroid updates were used rather than creating large intermediate matrices to address memory consumption with large datasets

2. Top-K search pipeline: This was implemented using `our_ann_improved`(improved is used because of the optimizations done to the function). Dimensionality scaling was used, wherein the function automatically adjusts parameters based on data dimensionality to handle the "curse of dimensionality". From each priority cluster, top-K2 candidates were selected (where K2 adapts to dimensionality). Torch.topk() was implemented here for efficient selection of top elements. It is highly optimized for GPU execution, avoids the need to sort an array, and returns values and indices. One of the biggest challenges faced was that adaptive parameters were used rather than fixed ones to balance recall guarantees with performance, which maintained a high recall.

Evaluation: `test_ann_with_guarantees()` was used as a framework to run benchmarks across different configurations. The techniques that we used for evaluation included **GPU vs CPU performances, actual vs target recall rates, and speed comparison with KNN**. The target recall rate of 0.7 was set, and testing was done on different dataset configurations using L2 and cosine as distance metrics.

Results and Analysis: The actual recall is **always higher than 0.7** (target recall), but varies depending on the distance metric and dataset. The GPU performance was significantly higher than the CPU at all times. Using a small dataset (D = 2), the GPU is 0.09x and 0.36x faster than the CPU in L2 and cosine respectfully, whereas for the large dataset (D = 1024), it is 16.30x and 6.84x faster C for L2 and cosine, respectfully. While these values can change depending on the systems used, it is very clear that no matter what, the GPU will be faster than the CPU. In the larger dataset especially, **the GPU is at least 6-10x faster than CPU** 2, if not more. Moreover, the speed at which ANN processes is not much different from KNN. The extrapolation tests suggest that for very large datasets (4M vectors), the ANN approach would be hundreds of times faster than exact KNN while maintaining acceptable recall rates. The implementation successfully addresses the challenges of efficient nearest neighbor search in high-dimensional spaces, the GPU acceleration through PyTorch enhances performance, and the mathematical foundation ensures consistent results.

1.5 Advanced Optimization

For KNN optimization: The implementation utilizes PyTorch's `torch.cuda` functions to parallelize computations on the GPU, significantly accelerating distance calculations, which are the primary bottleneck in KNN. functions to parallelize computations, significantly accelerating the distance calculations that are the core bottleneck in KNN. All operations utilize PyTorch tensors, not NumPy arrays, with explicit tensor transfers to the appropriate device using `.to(device)` to ensure GPU acceleration. Memory management is handled by deleting large intermediate tensors when no longer needed and invoking `torch.cuda.empty_cache()` to release unused GPU memory and reduce fragmentation during batch processing. Distance calculations are vectorized for efficiency: for L2 distance, `torch_clamp()` is used to mitigate numerical instability from floating-point errors; for cosine distance, clamping ensures values remain in the valid $[-1, 1]$ range; for Manhattan distance, `torch_cdist()` with `p=1` improves performance. Additionally, a custom TopK implementation finds nearest neighbors efficiently using `torch.min()` to identify minimal distances, with an alternative clone-and-mask approach that preserves the original distance tensor.

For ANN optimization: The algorithm uses KMeans++ for well-spaced centroid initialization, with `torch.multinomial()` for weighted sampling. Parameters such as `num_clusters`, K1, and K2 are adaptively tuned based on dataset size and recall targets. A two-tier search retrieves top-K1 clusters using priority scores, then top-K2 points within them. Fallback to exact search ensures complete results when candidate count is insufficient. Performance on large datasets is extrapolated from smaller-scale tests to save computational resources.

1.6 K-MEANS

Problem Definition: K-means clustering is widely used in large-scale machine learning systems for unsupervised learning tasks such as document clustering, feature compression, and initialization steps. The core goal is to partition the data into clusters such that the similarity of samples within groups is maximized and the differences between groups are maximized.

Design Choice: The algorithm follows the classic K-Means process: each sample is assigned to the nearest centroid using L2 or cosine distance, $\text{label}_i = \arg\min_k \ d(x_i, \mu_k)$, and centroids are updated as the mean of their assigned samples, $\mu_k^{\text{new}} = \frac{1}{N_k} \sum_{x_i \in C_k} x_i$. The implementation leverages fully vectorized PyTorch tensor operations—using `argmin` for label assignment and `scatter_add_` for centroid updates—to avoid Python loops. GPU acceleration is enabled via PyTorch's CUDA support, with `pin_memory()` and `non_blocking=True` used to reduce CPU–GPU transfer overhead. The distance module is modularized to support efficient switching between metrics.

Implementation Details: This experiment implements a scalable, modular K-Means algorithm in PyTorch with support for multiple distance metrics and CPU/GPU execution. It uses the standard K-Means loop—random centroid initialization, label assignment via `argmin`, centroid updates via averaging, and early stopping based on ℓ_2 shift—fully vectorized with `argmin` and `scatter_add_` to avoid Python loops. The distance module supports flexible switching between ℓ_2 and cosine metrics, while GPU efficiency is enhanced using `pin_memory()` and `non_blocking=True` for asynchronous data transfer.

Evaluation: Two datasets were constructed to reflect realistic use cases: a **small dataset** with $N = 100$, $D = 2$, $K = 5$ for lightweight tasks, and a **large dataset** with $N = 10000$, $D = 1024$, $K = 20$ to stress test computational scalability. The K-Means algorithm was tested under four configurations: `CPU+L2`, `CPU+cosine`, `GPU+L2`, and `GPU+cosine`, with total runtime and convergence steps recorded for each. Evaluation was based on two key metrics: **runtime**, measuring the wall-clock time for the clustering process, and **convergence steps**, indicating the number of iterations needed to meet the convergence threshold.

Results and Analysis:On the **small dataset** ($N = 100$, $D = 2$), the **CPU consistently outperforms the GPU** in both L2 and Cosine metrics. This is due to the fixed overhead of data transfer and kernel launch on the GPU, which dominates when the data volume is small. On the **large dataset** ($N = 10000$, $D = 1024$), **GPU significantly outperforms CPU**, showing a speedup of more than 10× across all configurations. This validates the advantage of GPU parallelism for high-dimensional and large-scale tasks.

2 MODEL SERVING SYSTEMS (TASK 2)

2.1 MOTIVATION

Machine learning systems require efficient queueing and batch processing to achieve scalable and responsive model servining. In real-time systems, unpredictable request surges can lead to latency spikes and service degradation. Request queues help manage load by ensuring orderly processing, while batch processing leverages GPU parallelism Akoush et al. (2022) to optimize throughput and resource utilization. However, managing online inference brings challenges—excessive queuing can delay responses in latency-sensitive applications, and balancing batch size with real-time constraints is complex, as larger batches improve efficiency but risk increased response times. Systems like TensorFlow Serving and NVIDIA Triton address these issues through adaptive batching and smart queue management, illustrating the practical benefits of these techniques in delivering reliable, high-performance ML services.

2.2 OVERVIEW

In this system, we implement request queueing and batch processing to handle Retrieval-Augmented Generation (RAG) requests more efficiently by accumulating multiple incoming requests into batches rather than processing each individually, thereby reducing overhead and improving throughput under high concurrency. The implementation resides in `serving_rag_M.py`, with `load_test_m.py` used for testing. We expose two FastAPI endpoints: `/rag`, which processes

requests in batches using a background thread that monitors a global queue and forms batches based on a maximum size or timeout, and `/rag_no_batch`, which processes requests immediately and serves as a baseline. The batching pipeline involves temporarily storing requests in a thread-safe `queue.Queue`, batching them either when the `MAX_BATCH_SIZE` is reached or `MAX_WAITING_TIME` elapses, then calling the LLM with the batch of prompts (each consisting of a query and retrieved documents), and finally writing responses back to individual "future" objects to unblock the FastAPI threads.

The figure 2 below depicts this architecture at a glance. At a high level, multiple incoming requests that arrive at /rag can be processed together in a single forward pass of the LLM, thus improving throughput and reducing repeated overhead. Under lighter loads, the waiting time for the batch is minimal, ensuring requests do not idle excessively.

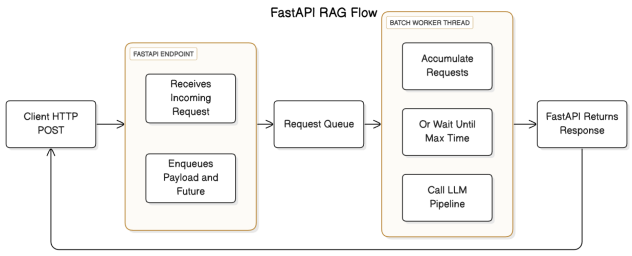

Figure 2: Workflow of Request Queue and Batch Processing Pipeline

2.3 System Measurement

- **Testing Methods:** The testing methods apply a systematic load-testing approach to evaluate the performance of a Retrieval-Augmented-Generation(RAG) system in both batched and non-batched modes. It compares responses from two endpoints `/rag` for batched processing and `/rag_no_batch` for individual requests under varying load conditions. Tests are run at multiple requests-per-seconds(RPS) levels(0.5,1,2,5 and 10) to observe system behavior across a range of conditions. The load generation process leverages asynchronous programming, utilizing Python's `asyncio` and `aiohttp`, which enables efficient and scalable load generation. Each test runs for a fixed duration of 10 seconds to maintain consistency across scenarios. Additionally, all tests use the same query ("Which animals can hover in the air?") to eliminate variation from differing query complexity, ensuring fair performance comparisons.
- **Metrics Measured:** The testing framework measures key performance metrics, including latency (average latency, 90th percentile latency, and 99th percentile latency), throughput (requests processed per second), and test completion statistics (total requests processed and total test duration).
- **Rationale for Metrics:** Latency metrics, including average latency and tail latencies (P90, P99), evaluate consistency and user experience under varied conditions. Throughput measures processing capacity, identifying saturation points and efficiency differences between batched and non-batched modes. Testing across multiple RPS levels highlights the system's scalability, optimal operating range, and performance degradation as load increases.
- **Measurement Techniques:** The test harness employs precise measurement techniques, like the use of `time.perf_counter()` to record individual request latency and calculate end-to-end response times. Aggregate statistics are derived, such as latency percentiles (P90, P99) and throughput, based on the ratio of successful requests to test duration. Performance results are visually represented through `matplotlib` plots, comparing average latency and throughput across varying RPS values. Detailed metrics are saved to JSON files, while plots are stored as timestamped PNG files for further analysis.
- **Challenges and Solutions:** The challenge of slower batched 4 processing due to generation bottlenecks is addressed by optimizing algorithms, adjusting batch sizes dynamically, and monitoring latency trends for improved efficiency

2.4 Request Queue and Batcher

Naively handling each request independently can overload system resources, especially under high concurrency. Each request incurs repeated steps (embedding, retrieval, generation), leading to inefficient GPU usage and latency spikes.

- **Rationale for Request Queue and Batcher:** By introducing a batching mechanism, we can: Aggregate multiple requests for shared processing (single forward pass), reduce per-request overhead (e.g., tokenization), and smooth load spikes by queuing and batching.
- **Implementation Details:**We use Python's thread-safe `queue.Queue` to store incoming requests, each paired with a "future" dictionary containing a `threading.Event` to signal completion and a placeholder for the response. A dedicated background thread (`batch_worker`) handles batching by accumulating requests until either `MAX_BATCH_SIZE`=4 is reached or `MAX_WAITING_TIME`=1s elapses. It then embeds the queries, retrieves the top-k documents, constructs prompts, performs batched inference using the LLM, sets the results in the respective futures, and signals the waiting threads.
- **Experiments and Metrics Analysis:** We conducted load tests on two endpoints, `/rag (Batched)` and `/rag_no_batch` (Non-Batched), to evaluate the performance impact of our request queue and batcher. The tests were performed using a Python script (aiohttp-based) with varying request rates (0.5, 1, 2, 5, and 10 requests per second) for 10-second intervals. We measured total requests sent, average latency, 90th/99th percentile latency, and throughput.The test results are shown in the figure below. Detailed results of the test in the Appendix. 5
- **Challenges and Solutions:** Batch inference presents several challenges, including the trade-off between batch size and waiting time, where large batches improve efficiency but increase latency, and small batches reduce efficiency. Errors in a single request can disrupt the entire batch, and managing concurrent access to shared resources raises thread safety concerns. Additionally, high request rates risk overloading the system, leading to degraded performance or crashes. To solve these, the system uses a balanced batch size of 4 and a 1-second timeout, error handling via try-except blocks with fallback responses, thread-safe queues with synchronization primitives to ensure thread safety, and frequent batch flushing alongside lightweight models to prevent memory overload and ensure stability under high load.

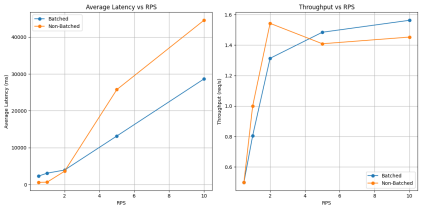

Figure 3: Impact of Request Rate on Latency and Throughput for Batched vs. Non-Batched Processing

3 Discussion of Task Synergy

The GPU kernel optimizations in Task 1 and distributed RAG system improvements in Task 2 highlight the need for joint optimization across layers of a modern Machine Learning pipeline. Vector similarity search forms the foundation for the retrieval stage of RAG systems, and while low-level GPU tuning boosts computation speed, its benefits are limited without efficient system-level batching and queue management. Achieving effective performance requires balancing latency and throughput across components. To fully utilize system potential, designs should incorporate adaptive methods like workload-aware HNSW routing Malkov & Yashunin (2020), dynamic batching (e.g., PriorityBatch), scalable indexing such as DiskANN Jaiswal et al. (2022), and intelligent schedulers like SIEVE Naderan-Tahan et al. (2023) to optimize retrieval, batching, and resource use.

REFERENCES

Sherif Akoush, Andrei Paleyes, Arnaud Van Looveren, and Clive Cox. Desiderata for next generation of ml model serving. In *Proceedings of the NeurIPS Workshop on Challenges in Deploying and Monitoring Machine Learning Systems (DMML)*, 2022. URL https://arxiv.org/abs/2210.14665.

Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries, 2022. arXiv preprint arXiv:2210.12066.

Yu. A. Malkov and D. 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, 2020. doi: 10.1109/TPAMI.2018.2889473.

Mahmood Naderan-Tahan, Hossein SeyyedAghaei, and Lieven Eeckhout. Sieve: Stratified gpu-compute workload sampling. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 224–234, 2023. doi: 10.1109/ISPASS57527.2023.00030.

Binhang Qi, Hailong Sun, Xiang Gao, and Hongyu Zhang. Reusing deep neural network models through model re-engineering. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 870–882. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00090.

Jian Zhang, Jianfeng Gao, Ming Zhou, and Jiaxing Wang. Improving the effectiveness of information retrieval with clustering and fusion. *Computational Linguistics and Chinese Language Processing*, 6(1):109–125, February 2001.

APPENDICES

A DISTANCE FORMULAS

A.1 L2 (EUCLIDEAN) DISTANCE BETWEEN VECTORS X AND Y

$$d_{\text{L2}}(X, Y) = \sqrt{\sum_{i=1}^{D} (X_i - Y_i)^2} \tag{1}$$

A.2 MANHATTAN DISTANCE

$$d_{\text{L1}}(X, Y) = \sum_{i=1}^{D} |X_i - Y_i| \tag{2}$$

A.3 DOT PRODUCT

$$d_{\text{dot}}(X, Y) = -\sum_{i=1}^{D} X_i Y_i \tag{3}$$

A.4 COSINE DISTANCE

$$d_{\cos}(X, Y) = 1 - \frac{X \cdot Y}{\|X\| \cdot \|Y\|} \tag{4}$$

B KNN ALGORITHM PERFORMANCE COMPARISON

Table 1: GPU vs CPU Speedup for KNN with Various Distance Metrics

Dataset	**L2**	**Cosine**	**Dot Product**	**Manhattan**	**Notes**
SMALL	N/A	60.45x	35.98x	6.63x	
MEDIUM	2.76x	0.50x	0.98x	1.02x	
LARGE	1.01x	0.98x	1.12x	1.01x	
LARGE_DIM2	1.67x	1.11x	1.00x	1.00x	Low dimensionality
LARGE_DIM32K	0.86x	0.86x	0.90x	0.92x	High dimensionality
HUGE	1.04x	1.05x	0.99x	1.13x	Extrapolated from 40K vectors

C K-MEANS AND ANN PERFORMANCE COMPARISON

Table 2: GPU vs CPU Speedup for K-Means and ANN

Algorithm	**Dataset**	**L2**	**Cosine**
K-Means	SMALL	0.03x	0.56x
	LARGE	23.55x	17.36x
ANN	SMALL	0.09x	0.36x
	LARGE	16.30x	6.84x

D CHALLENGE IN BATCH PROCESSING

E EXPERIMENTS AND METRICS ANALYSIS

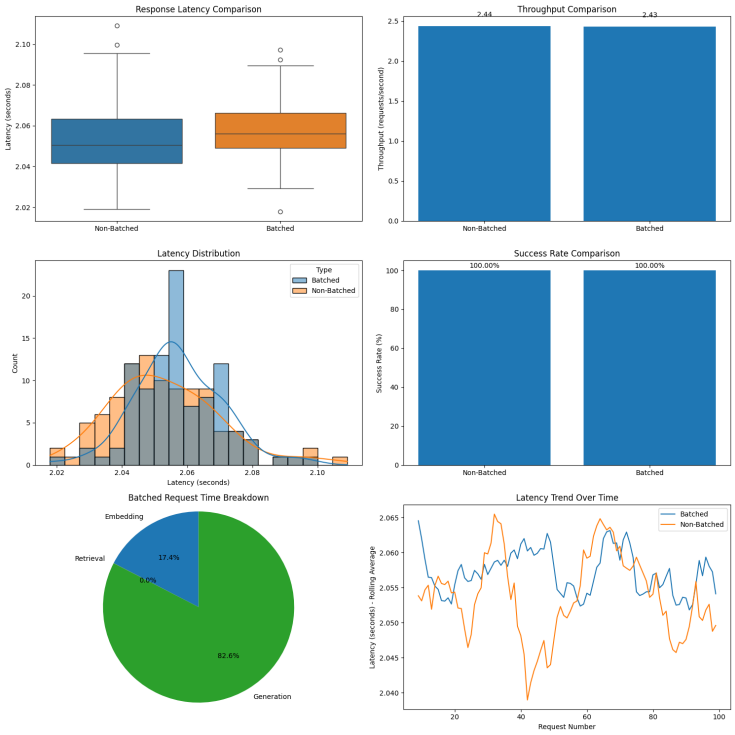

Figure 4: The challenge of slower batched processing due to generation bottlenecks

```
Batched Results:
{'rps': 0.5, 'duration': 10, 'total_requests': 5, 'total_time': 10.006542213999978, 'avg_latency_ms': 2284.9075656000423, 'p90_latency_ms':
2284.9075656000423, 'p99_latency_ms': 2284.9075656000423, 'throughput': 0.49967310316290753}
{'rps': 1, 'duration': 10, 'total_requests': 10, 'total_time': 12.427316115999929, 'avg_latency_ms': 3087.783556999989, 'p90_latency_ms':
4272.928586000035, 'p99_latency_ms': 3087.783556999989, 'throughput': 0.8046789754647984}
{'rps': 2, 'duration': 10, 'total_requests': 20, 'total_time': 15.232083549000095, 'avg_latency_ms': 3929.3932175499945, 'p90_latency_ms':
5708.435625999982, 'p99_latency_ms': 3929.3932175499945, 'throughput': 1.3130180080526734}
{'rps': 5, 'duration': 10, 'total_requests': 50, 'total_time': 33.69177837500001, 'avg_latency_ms': 13165.456110199995, 'p90_latency_ms':
22086.196452999957, 'p99_latency_ms': 13165.456110199995, 'throughput': 1.4840415796246904}
{'rps': 10, 'duration': 10, 'total_requests': 100, 'total_time': 64.00179097800003, 'avg_latency_ms': 28646.823917220005, 'p90_latency_ms':
50323.27377400009, 'p99_latency_ms': 54207.51780299997, 'throughput': 1.5624562761747 4}

Non-Batched Results:
{'rps': 0.5, 'duration': 10, 'total_requests': 5, 'total_time': 10.010016647000043, 'avg_latency_ms': 604.021204799983, 'p90_latency_ms':
604.021204799983, 'p99_latency_ms': 604.021204799983, 'throughput': 0.4994996688140851}
{'rps': 1, 'duration': 10, 'total_requests': 10, 'total_time': 10.010618406000049, 'avg_latency_ms': 646.4244206000103, 'p90_latency_ms':
745.3077870000016, 'p99_latency_ms': 646.4244206000103, 'throughput': 0.9989392857094938}
{'rps': 2, 'duration': 10, 'total_requests': 20, 'total_time': 12.965740544000028, 'avg_latency_ms': 3609.854506550016, 'p90_latency_ms':
6709.532261000049, 'p99_latency_ms': 3609.854506550016, 'throughput': 1.5425266248486762}
{'rps': 5, 'duration': 10, 'total_requests': 50, 'total_time': 35.49675069500006, 'avg_latency_ms': 25759.097037600004, 'p90_latency_ms':
32153.167760999964, 'p99_latency_ms': 25759.097037600004, 'throughput': 1.4085796311222034}
{'rps': 10, 'duration': 10, 'total_requests': 100, 'total_time': 68.85330232399997, 'avg_latency_ms': 44562.76602701, 'p90_latency_ms':
59863.23528200001, 'p99_latency_ms': 60893.27839700002, 'throughput': 1.4523631637802117}
Comparison plot saved to rag_compare_20250413_183636.png
Detailed comparison results saved to rag_compare_20250413_183636.json
```

Figure 5: Test Result for Task 2