Building and Optimizing a Distributed ML Inference Pipeline

Kadambari Mirashi
MEng, Computer Science
Cornell University
Ithaca, NY, USA
km2266@cornell.edu

Ian Holloway
MEng, Computer Science
Cornell University
Ithaca, NY, USA
imh39@cornell.edu

Sanjeev Ragunathan
MEng, Computer Science
Cornell University
Ithaca, NY, USA
sr2432@cornell.edu

Rachel Yan
MEng, Electrical and Computer Engineering
Cornell University
Ithaca, NY, USA
sy625@cornell.edu

I. Introduction

The goal for this project is to design, implement, and optimize a three node system that can process various workloads under memory constraints while preserving correctness and stability.

The provided starter code runs the full retrieval augmented generation pipeline in a monolithic design. It embeds the query, performs FAISS based approximate nearest neighbor search, loads documents from disk, reranks them, calls an LLM to answer the question, and then runs sentiment and toxicity classifiers on the generated reply. This monolithic implementation cannot scale because it repeatedly loads and unloads large models in order to stay under a 16 GB memory limit and it only processes one request at a time.

Our project turns that monolithic design into a distributed system with a microservices architecture, opportunistic batching, and shared profiling infrastructure. We implemented all three project tiers and added several additions of our own including a flexible configuration layer that makes it easy to explore different pipeline layouts with specific settings and taking parallelization on single node. We evaluate several system designs on real simulated workloads and used this profiling data to guide our final configuration.

II. System Overview

The created application is a customer support assistant for an e-commerce site. Each request carries a request identifier and a natural language query. The pipeline must return four fields for every request:

- The request identifier
- The generated response from the LLM

- A sentiment label in one of five buckets
- A binary toxicity flag

To produce these outputs the system executes the following logical stages in order:

1) Query embedding
2) FAISS retrieval of top documents
3) Loading of document contents
4) Reranking of the documents
5) LLM generation conditioned on the reranked context
6) Sentiment analysis over the generated text
7) Sensitivity filtering over the same output

We implement the pipeline using three nodes that communicate over HTTP. Node zero is the gateway and client facing entry point. It terminates the HTTP API, performs initial validation, and is responsible for orchestration and high level batching. Node one is the retrieval node. In the baseline configuration it runs the embedding model, FAISS index, document store, and reranker. Node two is the generation node. It hosts the LLM and the sentiment and toxicity classifiers. These roles can change depending on which profile is active, but the three node pattern stays constant and node zero always serves as the external entry point.

Our runtime is built using mainly using FastAPI for HTTP serving and a small custom framework under the project pipeline package that abstracts profiles for node, components, and services. A profile configuration describes the components that should run on a given node and their settings. There is a runtime factory that reads a profile, instantiates components such as the embedding generator, FAISS store, document store, LLM generator, reranker, sentiment analyzer, and toxicity filter, and then mounts the appropriate gateways and service routers. This design allows us to reuse the same code but rearrange where each stage runs by quickly editing YAML instead of Python. This allowed us to test various components on various nodes quickly and with minimal code changes.

III. Tier 1 Implementation

Tier one focuses on turning the starter code into a three node pipeline, while maintaining correctness and staying within per node memory constraints. Our first step was to refactor the monolithic implementation into explicit components for each logical stage. We created dedicated embedding, FAISS, document store, reranker, LLM, sentiment, and toxicity components under the pipeline components module. Each component manages its own model loading, device

placement, and warmup. Components expose simple methods such as encode, search, fetch, rerank, and generate that can be orchestrated by higher level services.

This also when it was decided to move to using FastAPI instead of Flask for the HTTP layer. FastAPI is more performance and is built for high concurrency while Flask is could be built in a similar way FastAPI is more straightforward. There were also various other packages that were added to improve the base setup. This included using Pydantic for easier data model validation, httpx for async HTTP clients, and msgspec to serialize json faster than the default json package.

We then built three FastAPI services around these components. The gateway service exposes the client facing query endpoint. It accepts single requests and delegates them to an orchestrator that can forward work to retrieval and generation nodes. The retrieval service groups the embedding, FAISS search, and document fetching logic and exposes a batched retrieval endpoint. The generation service wraps the LLM, sentiment, and toxicity models and exposes a batched generate endpoint that returns the full final response for each item in the batch.

Static batching was implemented first to satisfy the minimum requirement. Each of the three main stages gateway, retrieval, and generation uses batching with a configured batch size and maximum batch delay. The gateway queues incoming requests and assembles them into batches up to the fixed size or flushes them after the delay has passed. The retrieval and generation services apply the same pattern on their respective nodes. This simple batching strategy already improves throughput because it amortizes the model overhead over multiple queries and reduces redundant tokenization. Later experiments such as batch_sweep_low_latency, batch_sweep_high_throughput, and optimized_batch_4 varied these batch sizes systematically and led us to prefer batch size four as a balanced default.

For node assignment we started with a straightforward layout that mirrors the project handout. Node zero runs the gateway and orchestrator. Node one runs embedding and FAISS retrieval stitched together as a retrieval service. Node two runs the LLM, sentiment classifier, and toxicity filter as a generation service. Environment variables such as TOTAL_NODES, NODE_NUMBER, and the per node IPs are used by run.sh and the config layer to pick the correct profile for each process and connect nodes together.

To satisfy the profiling requirement we integrated Prometheus metrics and a lightweight profiling script. Each node exposes a metrics endpoint that tracks request counts, per stage latency, batch sizes, queue depths, and memory usage. The profile_pipeline script sends a stream of client requests to node zero, collects per request latency statistics, and writes out

summary CSV files. This gives us end to end throughput and latency measurements for each configuration. This required setting up a docker compose environment with Prometheus and Grafana to visualize metrics during experiments.

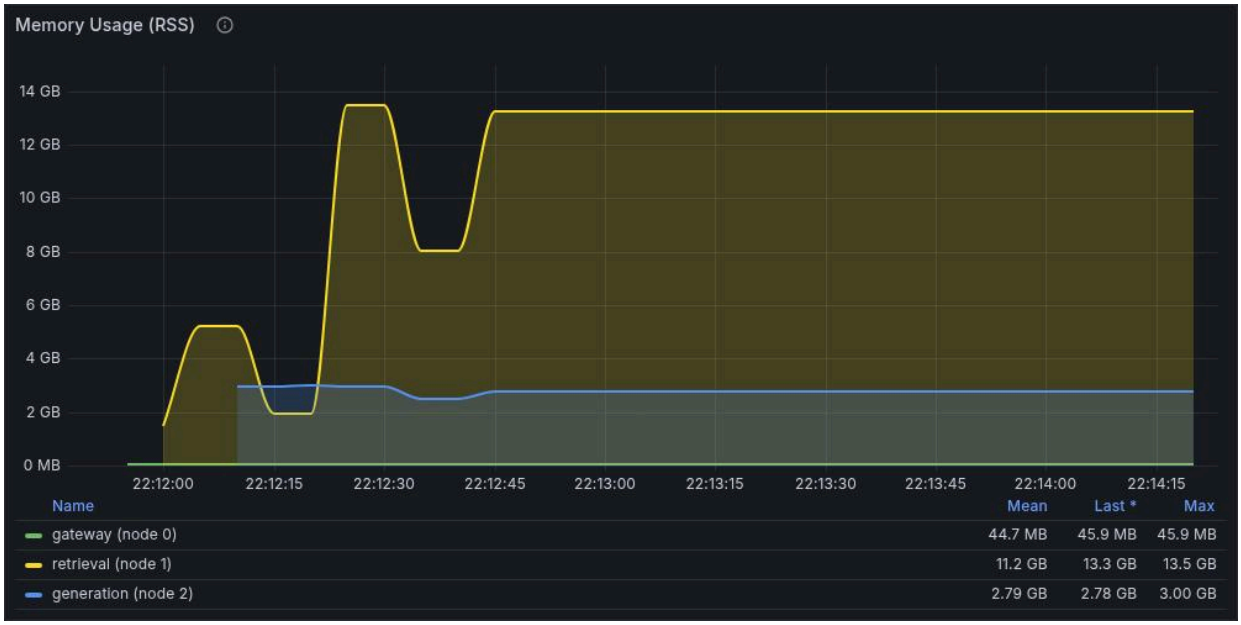

Fig. 1: Memory profiling per node using Grafana dashboards.

We validated correctness by running the implemented tier 1 client against both the original monolith, and checking that the high level behavior matched and was able to to process all the requests in client.py. While the exact text produced by the LLM can differ because of internal nondeterminism, the structure of the response, the presence of all required fields, and the qualitative behavior of the sentiment and toxicity labels are preserved. Since the goal was to implement features up and beyond tier 3 more validation was not done at this stage.

IV. Tier 2 Implementation

Tier two requires a true microservices architecture with explicit orchestration, along with more robust profiling and a clear explanation of design decisions. Our main change at this stage was to decouple services so that individual nodes only host the stages they need, and to formalize orchestration logic inside a gateway orchestrator module. The jump between tier 2 and tier 1 was relatively small due to implementing most of the groundwork tier 1.

The gateway orchestrator receives single client requests and converts them into internal requests that carry only what each downstream service needs. For a typical configuration the orchestrator first forwards the text query to the retrieval node. Retrieval embeds the query, searches the FAISS index, fetches the top k documents from disk, and optionally reranks them.

The orchestrator then forwards the reranked documents to the generation node, which runs the LLM, followed by sentiment and toxicity models, and returns the final structured response. The orchestrator assembles this response together with the original request identifier and returns it to the client.

We redesigned the routing logic so that it is described by YAML profiles rather than hard coding what microservice goes on each node. Each YAML profile describes which services run on each node and how the gateway should route requests. For example the baseline_gateway, retrieval, and generation profiles capture our original three node layout. The retrieval_with_rerank profile moves the reranking stage from generation to retrieval. The gateway_with_embedding and retrieval_rerank_no_embedding profiles move embedding earlier into node zero and concentrate FAISS and reranking on node one.

To make profiling more robust we added a shared telemetry module with metrics for end to end request latency, stage level durations, batch sizes, queue depths, cache activity, compression ratios, and memory usage. Each service uses these shared metrics in addition to service specific counters. A capture_metrics utility records process statistics such as per process resident set size to JSON for every run. The analyze_experiments script aggregates metrics.csv and process_stats.json into a consolidated CSV file and generates various figures to compare statistics between configurations.

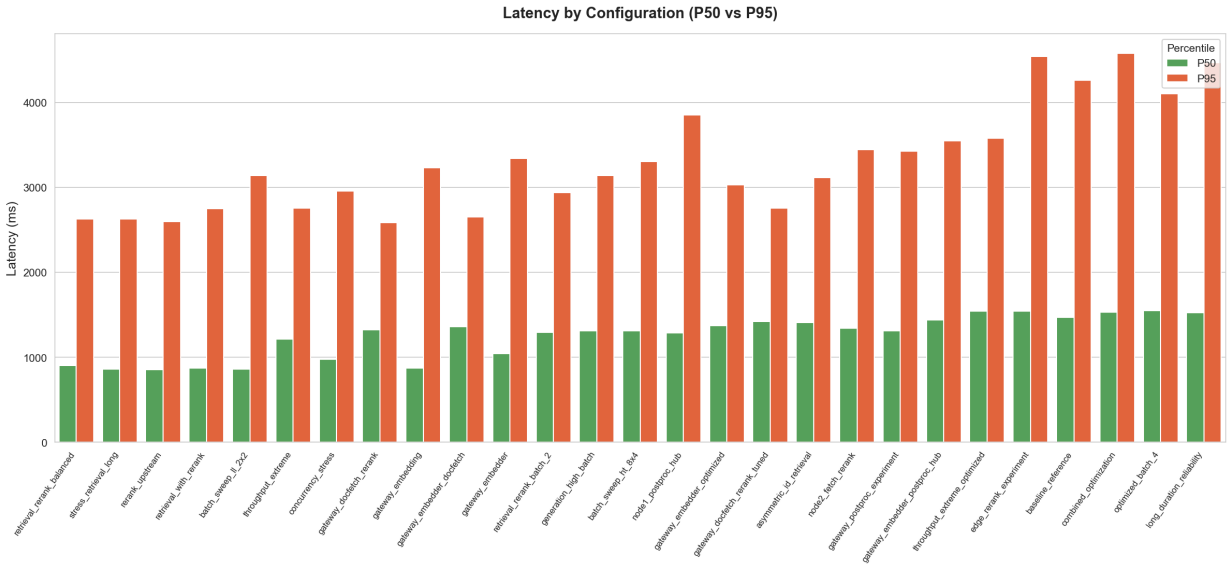

Fig. 2: Latency comparison across various configurations.

After instrumenting the system we ran a baseline experiment named baseline_reference. This experiment uses a simple three node layout with moderate batch sizes and low concurrency. It

sends 60 requests with a batch size of four at the gateway, a batch size of eight in retrieval, and a batch size of four in generation. The consolidated results show throughput around 46 requests per minute with median latency around one second and p95 latency around 2.6 seconds.

These baseline numbers establish a reference point for later experiments. They also confirm that the microservices and orchestration logic are correct and stable under moderate load before we proceed to more aggressive batching and restructuring.

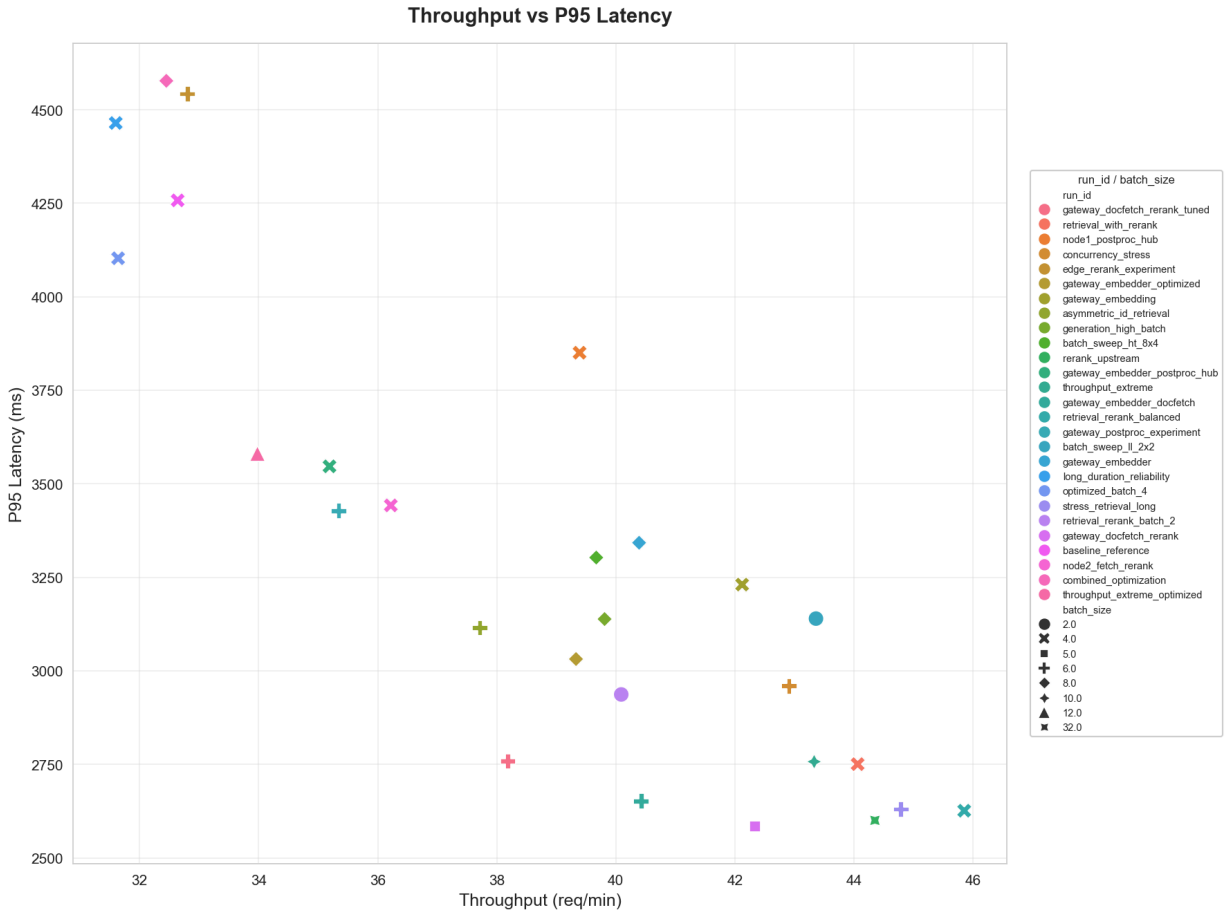

Fig. 3: Throughput versus p95 latency across several node configurations and batch sizes.

One thing that was considered early on was how the concurrency of number of users sending requests would affect the performance of the system. Each test was was run with similar concurrency so the results across tests would be comparable. However, in a real world scenario the concurrency could vary greatly depending on how many users are using the system. Some setup might excel at low concurrency and others at high concurrency. One thing that made this not problematic was that most of the node configurations were setup in ways to not abuse the oversight and made the improvements between configuration profiles translate across different concurrency levels.

V. Tier 3 Implementation and Advanced Features

Tier three requires opportunistic batching and at least one additional advanced optimization such as caching, compression, GPU support, or fault tolerance. In this project we implemented several of these features and made opportunistic batching a central part of the design.

Relative to the advanced implementation checklist we implemented a full microservices architecture by separating the gateway, retrieval, and generation stages into distinct services, and we implemented opportunistic batching in each of these services using the adaptive batch scheduler so that effective batch sizes change with load. We used smart orchestration through the profile system and experiments like retrieval_with_rerank and optimized_batch_4 to place stages on different nodes based on their compute and memory requirements. For performance aware data passing we introduced a documents payload mode that can send full documents, only document identifiers, or compressed payloads between services so that we avoid moving large bodies when it is not necessary. Finally, we built a comprehensive profiling setup that combines node level metrics, structured logging, and Scalene reports to analyze bottlenecks and tradeoffs in detail.

The gateway uses a BatchScheduler module that provides opportunistic batching. Instead of flushing strictly at a fixed delay, the scheduler maintains a view of recent queue depths and uses an AdaptiveBatchPolicy to change the effective delay between flushes. This policy is our concrete implementation of opportunistic batching, since it lets batch sizes form dynamically based on current load rather than only on a fixed size threshold. When the queue is short the scheduler prefers a small delay so that requests are served quickly and latency stays low. When the queue grows it increases the delay up to a configured maximum so that batches become larger and throughput improves. The scheduler is implemented using asyncio to avoid busy waiting and to ensure that per request futures are completed when their batch finishes processing.

The retrieval and generation services share the same batching infrastructure through BatchScheduler as well. Each service can be configured with its own maximum batch size and delay, and the AdaptiveBatchPolicy uses those values to drive its decisions. We also track queue depth and batch sizes in Prometheus so that we can confirm the scheduler behaves as expected during high load experiments.

We added caching at two different levels. The embedding component maintains an LRU cache of query texts to embeddings. The cache is keyed by the hash of a normalized query string and stores numpy arrays on the CPU. When caching is enabled, repeated queries avoid recomputing

embeddings and directly reuse the cached vectors. The gateway orchestrator also maintains a response cache keyed by the full query text. When the same query arrives multiple times the orchestrator can optionally return the cached final response without contacting downstream services. Each cache exposes counters for hits, misses, and evictions, and a clear_cache endpoint is available on every node to reset caches between experiments.

Caching was something that when it started working it broke the profiling suite as we were testing a set of tests on random and we would reuse the same queries eventually in a single test and then caching would cause this test to complete immediately. While this proved caching was working correctly it made it hard to profile the system as intended. To get around this we disabled caching for all the profiling experiments and only enabled it for manual testing to confirm correctness.

We implemented data compression for inter node communication to reduce network bandwidth when using full documents as payloads. Payloads are encoded using a compact binary format and compressed with a fast LZ4 frame codec. Each node tracks compression ratios and total compressed and uncompressed bytes using shared telemetry metrics. This allows us to quantify the tradeoff between CPU overhead for compression and the reduction in network transfer time and memory usage. Through some quick testing LZ4 seemed to cause minimal changes to latency and throughput while providing compression to network requests. However, zstd was also tested and it seemed to reduce throughput and latency due to extra computation necessary. Overall, more testing could be done in these areas to further optimize compression and test various compression levels.

To support exceptional profiling we combined structured logging with external profilers. Every node writes timestamped logs that include request identifiers, node numbers, and service names. During experiments we stored these logs under the artifacts directory and used them to line up events across nodes when debugging tail latency and rare failures at high concurrency. For deeper CPU and memory analysis we used the Scalene profiler on selected runs. Scalene reports for each node are saved as HTML files and show which lines and functions dominate CPU time and memory usage. We used these reports to confirm that our bottlenecks were mostly in the embedding and LLM stages and to verify that changes to batch sizes and thread counts improved performance in the areas that we expected. This was also an additional way to understand what lines of code were using the most CPU time or using the most amount of memory so we not make assumptions about how the nodes where working.

The LLM and embedding components support GPU acceleration on both CUDA and Apple MPS devices. When GPUs are available and the configuration does not force CPU only mode, these models are loaded on the accelerator while routing, orchestration, and lighter preprocessing logic remain on the CPU. At inference time tokenized inputs are created on the CPU and then moved to the GPU as tensors for the forward pass, and generated tokens and embeddings are moved back to the CPU before being serialized and sent across services over HTTP. The rest of the pipeline, including microservice boundaries and compression middleware, treats these results as regular JSON compatible data so that GPU usage is confined to the numerical kernels.

The LLM component includes several model level optimizations. It configures torch to use appropriate numbers of inference threads for CPU runs and can optionally wrap the model with torch.compile for GPU runs. It also enables scaled dot product attention where supported to take advantage of modern attention kernels and uses a warmup step at startup with synthetic prompts to ensure that kernels are compiled before serving real requests. In practice most of the experiments reported here were run on CPU only machines for consistency, but limited tests on MPS confirmed that the same microservice architecture and batching logic work correctly with accelerated backends and that per request latency improves when GPU memory is sufficient. However, with further testing on hardware more simular to the end system more performance could likely be extracted.

Finally, we introduced a flexible executor layer for CPU bound tasks that uses per service thread pools instead of the default global executor. Services such as retrieval and generation offload CPU heavy operations to dedicated executors. The executor factory sizes thread pools according to the configured number of worker threads and the actual CPU core count. This prevents oversubscription and improves latency under concurrent load.

Here is a digram of the system architecture for baseline profile:

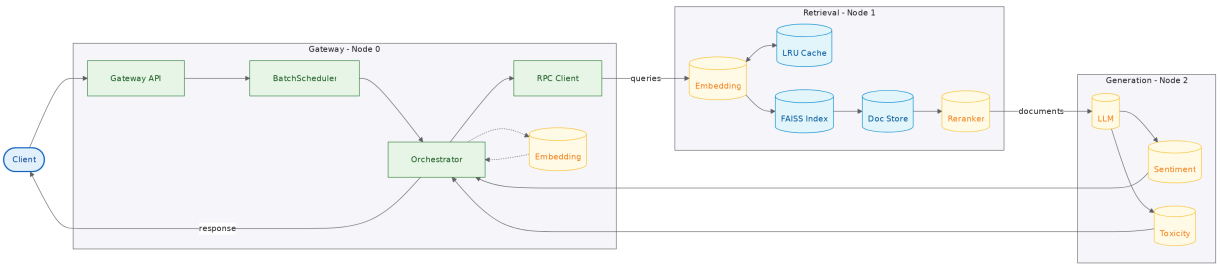

Fig. 4: High level architecture of the three node pipeline and microservices.

VI. Experimental Setup

All experiments use the provided customer support corpus and FAISS index generated by the create_test_docs script. For each configuration we run the scripts/run_experiment.py driver with a corresponding experiment manifest under configs/experiments. The driver starts the monitoring stack, launches three nodes with the specified profiles and environment variables, invokes the profile_pipeline workload generator with the requested number of requests and concurrency, and records metrics and profiling artifacts under artifacts/experiments.

The baseline_reference experiment uses the baseline_gateway, retrieval, and generation profiles. It sends 60 requests with a concurrency of one and a rate limit of 0.2 seconds between client requests. Batch sizes are four at the gateway, eight in retrieval, and four in generation. This run is designed to represent a simple configuration that can work under load that clearly satisfies all tier one requirements.

To explore the impact of microservice topology and upstream reranking we designed the retrieval_with_rerank experiment. In this configuration node one runs a retrieval node that performs embedding, FAISS search, document fetch, and reranking. Node two runs a generation node that only hosts the LLM and sentiment and toxicity classifiers. The workload sends two hundred fifty six requests at very high concurrency, with batch size thirty two on the retrieval node. This experiment stresses the retrieval node and tests whether performing reranking upstream reduces the amount of work the LLM must do and therefore improves throughput.

We also designed two sweep experiments to study the latency and throughput tradeoff of different batch size settings while keeping the topology fixed. The batch_sweep_low_latency configuration uses small gateway and generation batch sizes of two and a retrieval batch size of six, with forty requests and low concurrency. Its goal is to favor low tail latency by flushing smaller batches more frequently. The batch_sweep_high_throughput configuration uses larger gateway and generation batches of eight and four, with one hundred twenty requests and similar rate limits. Its goal is to drive higher throughput at the cost of increased queuing delay.

Finally, the optimized_batch_4 experiment tests a combined topology and batching strategy that emerged from our earlier profiling. In this configuration node zero runs a gateway that also hosts the embedding model. Node one runs retrieval with FAISS and reranking but no embedding. Node two runs generation without reranking. All batch sizes are set to four, and the workload issues one hundred requests with concurrency four. This run is intended to balance throughput and latency while also exploiting shared embeddings on the gateway.

For each experiment the analysis script aggregates results into analysis/ consolidated_results.csv and produces plots for throughput versus p95 latency and throughput by configuration. We use these artifacts when interpreting the impact of topology and batch sizes on performance.

VII. Results

The consolidated_results file summarizes throughput, median latency, p95 latency, success rate, and total Python process memory for each run. Here we highlight the most informative comparisons for our design choices.

The baseline_reference configuration achieves throughput of about forty six point five requests per minute with median latency just under one second and p95 latency around two point six seconds. This run confirms that the basic three node pipeline is correct and that batching with sizes between four and eight already gives significant benefits over single request processing.

The batch_sweep_low_latency configuration has throughput of about 43 requests per minute, median latency around one point one seconds, and p95 latency around two point seven seconds. The high throughput sweep with batch_sweep_high_throughput yields throughput around forty four requests per minute with similar p50 and p95 latencies. In our environment, increasing gateway and generation batch sizes from two to eight and four did not dramatically improve throughput because the workload is relatively small and rate limited. However, tracing shows that larger batches do increase per batch processing time and reduce overhead, suggesting that the benefits would be clearer at higher request rates.

The optimized_batch_4 configuration stands out as a balanced design. It attains throughput around 67.5 requests per minute, which is roughly 45 percent higher than the baseline_reference run. Its median latency is about 3.3 seconds and p95 latency is about 6.8 seconds, which is higher than the baseline but acceptable for many customer support applications where the perceived responsiveness is dominated by the first token and not the full completion time. This configuration benefits from co locating embedding with the gateway, which avoids sending raw queries over the network and reduces load on the retrieval node.

The retrieval_with_rerank experiments operate in a very different area. Batch sizes are as high as 32. Throughput ranges from about 75 to 135 requests per minute across different runs. Since the retrieval_with_rerank experiment was one of the best performing experiments throughout various points in testing we used this as the final configuration and tested it further increasing concurrency of requirements 128 and testing how large the max adaptive batch size

could be before performance degradation, which was around a batch size of 32. Since the retrieval_with_rerank experiments where some of the highest performers this demonstrate that moving reranking to the retrieval node shifts CPU load away from generation and allows the LLM node to focus on pure generation.

VIII. Discussion and Design Decisions

Our design choices were driven by profiling data and by the need to balance three competing goals: throughput, latency, and resource usage. Here we discuss several of the most important tradeoffs and how our experiments informed the final system.

First, the decision to separate embedding and FAISS retrieval into their own service rather than keeping everything on the generation node was informed by memory and CPU profiles. The embedding model and FAISS index each have significant memory footprints. Locating all models on one node risks approaching the memory limit, especially once FAISS indices grow toward the 14.5 gigabytes. Separating them onto a dedicated retrieval node keeps memory usage predictable and allows FAISS to use a tuned number of threads independent of the LLM.

Second, we evaluated where to run the reranker. When reranking runs on the generation node, the LLM often receives more documents than necessary, which increases tokenized context length and generation latency. Moving reranking to the retrieval node reduces the number of documents forwarded to generation. The retrieval_with_rerank experiments confirm that this shift raises CPU load on node one but frees capacity on node two. Combined with opportunistic batching this leads to higher aggregate throughput at high concurrency, at the cost of slightly higher latency due to additional work before generation.

Finally, the decision to implement opportunistic batching and compression was driven by observed queuing and network behavior under load. Simple fixed delay batching can either leave throughput on the table or create unnecessary latency. The adaptive policy responds automatically to load spikes and smooths out idle periods. Compression reduced the size of payloads carrying full documents between nodes, which is especially beneficial when running across slower networks. Our metrics confirm that compression ratios are often several times, reducing the cost of network transfers.

These decisions used profiling and experimentation to guid our final system design. Rather than guessing, we used concrete measurements to decide where to place each stage, how to tune batch sizes, and which optimizations to keep.

IX. Limitations and Future Work

Although the system satisfies all three project tiers and includes several additional optimizations, there are still limitations and opportunities for improvement.

First, our experiments were conducted on a small cluster of machines with limited variability in hardware. While the codebase includes configuration options for GPU acceleration and model level optimizations, most of our reported experiments focus on CPU only runs for fairness and reproducibility. A deeper exploration of GPU configurations, including mixed CPU and GPU placement and careful measurement of CPU to GPU transfer costs, would be a next step.

Second, our fault tolerance is basic. Nodes rely on FastAPI and the underlying operating system for stability and retry logic is based on tenacity. However this retry logic has not been thoroughly tested. There could be more work done to retry and route data under various failure circumstances.

Third, our caching layer focuses on embeddings and full responses. We do not cache partial stages such as FAISS results or reranked document sets. For workloads with high query repetition rates it may be beneficial to add those caches and to design eviction policies informed by query frequency. More generally, cache performance under realistic workloads would be worth a dedicated study.

Finally, while we collected extensive metrics and produced summary plots, the profiling data was not used to its full extend and more portions of the data could be analyzed. This could lead to either major or minor gains in performance however it depends how this data is interpreted and acted upon.

X. Conclusion

This project started from a monolithic retrieval augmented generation pipeline that could not scale without repeatedly loading and unloading large models. We transformed it into a distributed microservices system that runs across three nodes, supports opportunistic batching, and exposes rich telemetry for profiling. Along the way we implemented all required features for tiers one through three and added a load aware implementation of opportunistic batching, caching, compression, and flexible configuration and profiling tooling.

Our experiments show that thoughtful placement of stages across nodes, combined with appropriately tuned batch sizes, yields significant improvements in throughput while staying within reasonable latency bounds. The optimized_batch_4 configuration, in particular, demon-

strates a strong balance between performance and responsiveness and reflects the cumulative impact of embedding placement, reranking strategy, and opportunistic batching.

Beyond meeting the project requirements, this work provided practical experience with the kinds of systems challenges that arise in real world ML deployments. Building the pipeline forced us to think carefully about resource usage, orchestration, profiling, and observability. The resulting codebase is modular and extensible, making it a solid foundation for future experiments with new models, hardware configurations, and optimization techniques.