CS 5416 Final Project Report

Distributed RAG Pipeline Implementation

Riley Bakes - rb972 | Alvis Yan - hy676 | Laurence Yang - yy2394 | Caira Anderson - ca542

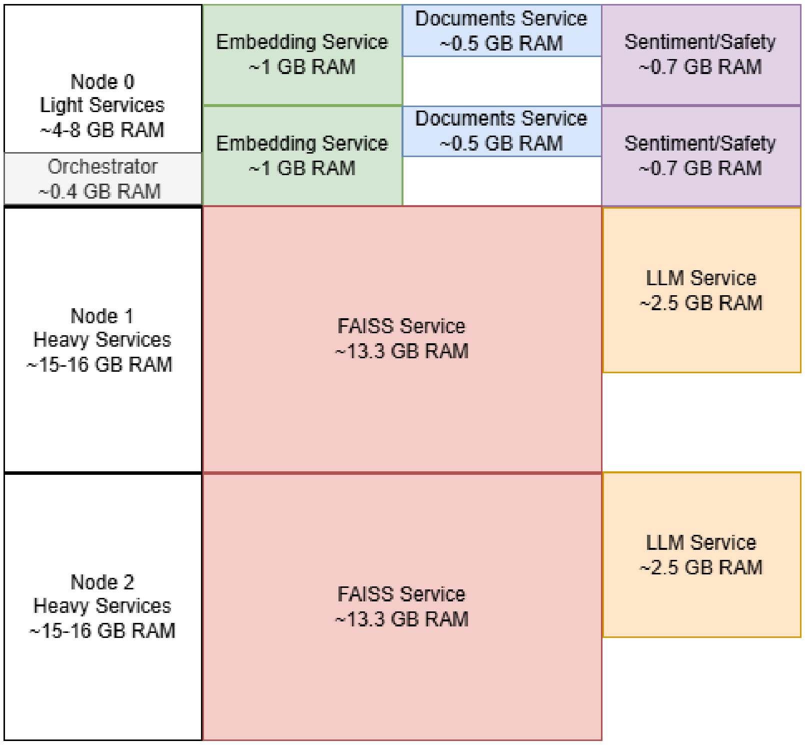
Table of Contents

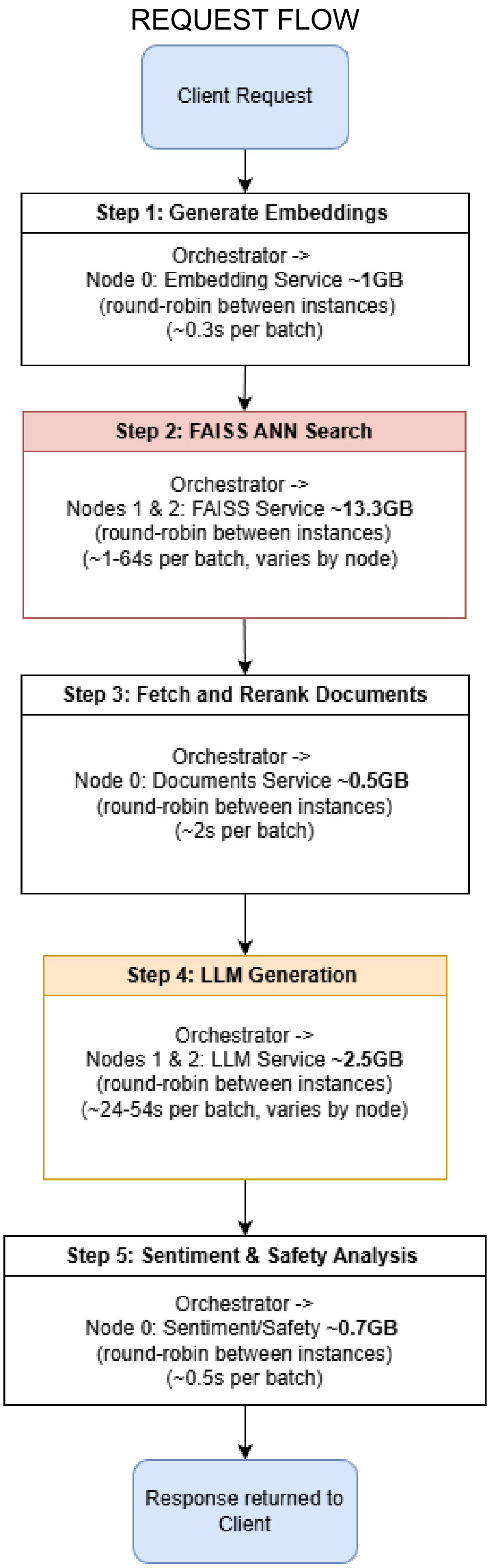
1. System Design Overview

1.1 Work Distribution Across 3 Nodes

Below are diagrams detailing our choice of work distribution across 3 nodes:

MICROSERVICES SETUP




REQUEST FLOW
Client Request
Step 1: Generate Embeddings
Orchestrator ->
Node 0: Embedding Service ~1GB
(round-robin between instances)
(~0.3s per batch)
Step 2: FAISS ANN Search
Orchestrator ->
Nodes 1 & 2: FAISS Service ~13.3GB
(round-robin between instances)
(~1-64s per batch, varies by node)
Step 3: Fetch and Rerank Documents
Orchestrator ->
Node 0: Documents Service ~0.5GB
(round-robin between instances)
(~2s per batch)
Step 4: LLM Generation
Orchestrator ->
Nodes 1 & 2: LLM Service ~2.5GB
(round-robin between instances)
(~24-54s per batch, varies by node)
Step 5: Sentiment & Safety Analysis
Orchestrator ->
Node 0: Sentiment/Safety ~0.7GB
(round-robin between instances)
(~0.5s per batch)
Response returned to Client

1.2 Request Flow Through System

Our app follows a centralized orchestrator pattern with isolated microservices to handle per-pipeline-step processing. The orchestrator for a given batch drives processing for each of the steps found in the original pipeline, and ultimately is responsible for returning results. Microservices and the orchestrator are deployed through flask web-apps responsible for spawning threads to execute requests and exposing simple API's for ingress/egress.

1.3 Concurrency Details

Concurrency is implemented as follows. As per project requirements, Node 0 is the entrypoint for the app lives. While we call this entrypoint the orchestrator, it is really just a flask web-app which handles enqueueing requests into a request queue. From here we have the first parameterization of concurrency, the number of workers pulling requests from the request queue and running the request through the pipeline. The second level of parameterized concurrency is the batch size, where each spawned worker will opportunistically batch requests from the queue and forward to the respective step in bulk.

We then have a third level of concurrency. Each service is itself replicated—allowing for simultaneous batch processing with the orchestrator following a simple round robin load balancer. Additionally, a final level of concurrency exists as each replicated service in the pipeline is itself a flask web-app surrounding the underlying steps execution. No manual worker spawning exists here, rather, the flask web-apps (dev) web-server handles distributing each unique request to a separate thread executing the underlying python code.

2. System Implementation Details

2.1 Functional Requirements Achieved

Our implementation satisfies all Tier 3 requirements, including the mandatory requirement and multiple additional optimizations:

Tier 3 Required Features

1. Opportunistic Batching

- **Implementation:** Implemented in process_requests_worker() function in exp3/pipeline.py
- **Details:**
    - Waits for the first request (blocking)
    - Attempts to collect up to MAX_BATCH_SIZE-1 additional requests
    - Uses MAX_TIMEOUT=0.1s (100ms) per additional request to balance latency and throughput

    - Processes batch immediately once time limit reached even if not full (doesn't wait indefinitely)
    - Processes batch immediately once the maximum batch size is reached.
- **Benefits:** Dynamically adapts batch sizes based on incoming request rate, optimizing throughput under varying load conditions

Tier 3 Additional Features (Multiple Implemented)

2. GPU Acceleration

- **Implementation:** Automatic GPU detection and utilization in exp3/config.py
- **Details:**
    - Automatic CUDA detection when available
    - All compute-intensive services (embedding, reranker, LLM, sentiment, safety) utilize GPU
    - CPU fallback when GPU unavailable
    - Manual override via ONLY_CPU=true environment variable for CPU-only mode
    - Float16 optimization for LLM on GPU, float32 on CPU
- **Services Using GPU:** Embedding service, Documents service (reranker), LLM service, Sentiment & Safety service
- **What is offloaded to GPU:** Models used in this pipeline are loaded to GPU using .to(DEVICE) when CUDA is available.
- **CPU <-> GPU transfers:** CPU-to-GPU transfers occur when input data is sent to GPU models for inference, and GPU-to-CPU transfers happen when model outputs are returned for further processing (for example, when embeddings converted to NumPy arrays).
- **GPU and CPU used simultaneously:** Different microservices run as separate processes, so GPU-intensive services (embedding, LLM, sentiment) can run in parallel with CPU-based operations.
- **Benefits:** Significant speedup for model inference (proven during office hour).

3. Running Stages Simultaneously When Possible

- **Implementation**: Concurrent batch processing and multi-instance service architecture in exp3/pipeline.py

- **Details:**
  - Multiple worker threads: ORCHESTRATOR_NUM_WORKERS (default 2) process different batches concurrently, allowing different batches to be at different pipeline stages simultaneously
  - Batch processing: Multiple requests are grouped and processed together at each stage (embedding, FAISS, documents, LLM, sentiment), enabling parallel execution within each stage
  - Multi-instance services: Multiple instances of each service (2 embedding, 2 FAISS, 2 documents, 2 LLM, 2 sentiment) allow different batches to use different service instances in parallel
  - Cross-batch stage overlap: While one batch is in Step 1, another batch can be in Step 2 or Step 4, creating natural stage overlap across batches

Tier 2 Requirements (Also Implemented)

1. True Microservices Architecture

- **Implementation:** 5 distinct microservices, each running as separate processes
- **Services:**
  - **Embedding Service** (01_embedding_service.py) - Generates embeddings for queries
  - **FAISS Search Service** (02_faiss_search_service.py) - Performs vector similarity search
  - **Documents Service** (03_documents_service.py) - Fetches documents and performs reranking
  - **LLM Service** (04_llm_service.py) - Generates LLM responses
  - **Sentiment & Safety Service** (05_sentiment_and_safety_service.py) - Performs sentiment analysis and toxicity detection
- **Benefits:** Independent scaling, deployment, and optimization of each pipeline stage

2. Orchestration and Request Routing

- **Implementation:** Centralized orchestrator in exp3/pipeline.py on Node 0
- **Details:**
  - Flask-based orchestrator receives all client requests
  - Routes requests through microservices via HTTP

    - Coordinates the full 5-stage pipeline execution
    - Manages request queuing and batch processing
    - Returns final responses to clients
- **Benefits:** Single entry point, centralized request management, and clear request flow

3. Round-Robin Load Balancing

- **Implementation:** get_service_url() function with thread-safe round-robin counters
- **Details:**
    - Round-robin counter per service type (embedding, FAISS, documents, LLM, sentiment_safety)
- **Benefits:** Even distribution of load across multiple service instances

Additional Optimizations

1. Service Pre-loading of Memory-Intensive Models

- **Implementation:** All models loaded once at service startup, not per-request
- **Details:**
    - Embedding model, reranker model, LLM, sentiment model, and safety model all loaded during service initialization
    - Models remain in memory for all subsequent requests
    - Prevents memory thrashing from repeated model loading/unloading
- **Benefits:** Eliminates per-request model loading overhead, reduces latency, and improves memory efficiency

2. Service Replication Across Nodes

- **Implementation:** Multiple instances of compute-intensive services distributed across 3 nodes
- **Service Distribution:**
    - **Node 0:** Orchestrator + 2×Embedding + 2×Documents + 2×Sentiment/Safety
    - **Node 1:** 1×FAISS + 1×LLM
    - **Node 2:** 1×FAISS + 1×LLM

- **Total Instances:** 2 Embedding, 2 FAISS, 2 Documents, 2 LLM, 2 Sentiment/Safety
- **Benefits:** Horizontal scaling, improved throughput, fault tolerance potentials through redundancy

3. Multi-Threaded Worker Pool

- **Implementation:** Configurable number of worker threads in orchestrator
- **Details:**
  - ORCHESTRATOR_NUM_WORKERS environment variable (default: 2)
  - Each worker thread processes batches independently
  - Enables concurrent batch processing
  - Thread-safe request queue and results storage
- **Benefits:** Parallel batch processing, improved throughput under high load
- **Performance:** Experiments show 3 threads provide faster overall completion time compared to 2 threads

3. Batching Implementation

3.1 Stages Using Batching

For system simplicity, we kept the batching organization at our orchestrator level. Batches are collected and processed together through all pipeline stages. This means:

- **All pipeline stages receive batched inputs:** Embedding, FAISS search, document retrieval, reranking, LLM generation, sentiment analysis, and safety filtering all process batches of requests simultaneously
- **Single batch coordination point:** The orchestrator's process_requests_worker() function collects requests into batches before sending them through the pipeline
- **Consistent batch size across stages:** Once a batch is formed at the orchestrator, it maintains the same size through all 5 microservice stages
- **Opportunistic Batching available:** As mentioned above, opportunistic batching is implemented at orchestrator level

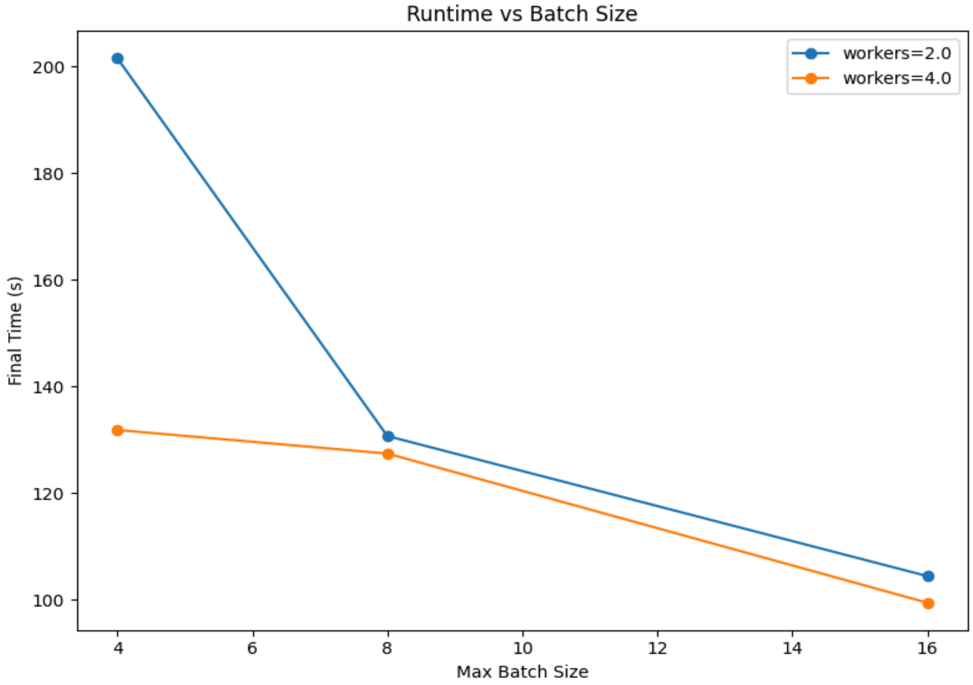
3.2 Batch Size Selection and Experiments

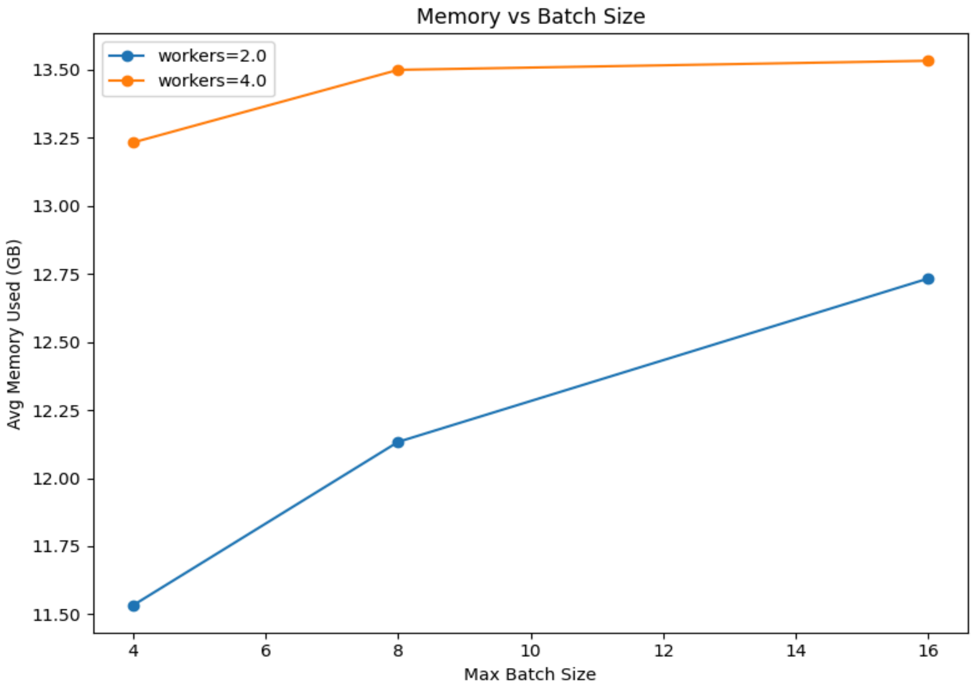
We experimented with different batch sizes to find the optimal balance between latency and memory constraints. Our experiments were informed by initial profiling that identified FAISS search as the primary memory bottleneck. To evaluate the impact of batch on both latency and memory usage, we generated a suite of diagnostic plots from the orchestrator benchmark results. The goal of these visualizations is to understand how batch size influences performance. The actual batch size selection is explained in part 4. The graphs produced are:

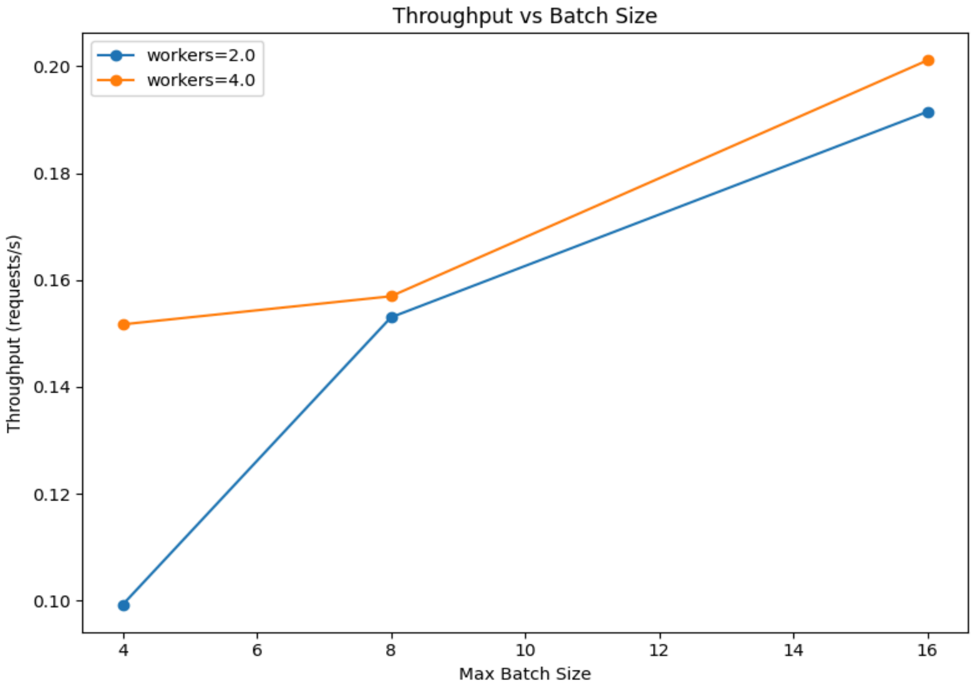
Runtime vs Batch Size — Shows how latency changes as batch size increases for different worker counts.

Memory vs Batch Size — Shows how memory consumption scales with batch size for each worker count.

Throughput vs Batch Size — Shows how throughput changes with batch size for each worker count.




Memory vs Batch Size
workers=2.0
workers=4.0
Avg Memory Used (GB)
13.50
13.25
13.00
12.75
12.50
12.25
12.00
11.75
11.50
4
6
8
10
12
14
16
Max Batch Size


Throughput vs Batch Size
workers=2.0
workers=4.0
Throughput (requests/s)
0.20
0.18
0.16
0.14
0.12
0.10
4
6
8
10
12
14
16
Max Batch Size

4. Profiling

Profiling was achieved through a series of experiments advancing our underlying architecture, we refer to them as experiments 1/2/3 below.

4.1 Experiment 1

In this experiment, we explore the as-given single node pipeline.py performance in order to begin developing a strategy for the passing grade implementation and then future microservice decomposition. In pursuit of this goal, we use memory-profiler to identify memory usage patterns and a simple custom decorator to track per step times.

More concretely, we do uv pip install memory-profiler and then thread the @profile decorator at each of the STEP functions in pipeline.py. We write a custom @profile_with_time function to print a simple time out for each invocation. While execution time is dependent on underlying test hardware, relative execution time metrics per function call are still informative for bottleneck analysis.

We then run python3 pipeline.py >> exp1_memory_profile.log to generate a memory profile log with our simple timing data using the client.py script.

Data

function	Step #	num_calls	abs_mem_min_MiB	abs_mem_max_MiB	abs_mem_avg_max_MiB	mem_min_MiB	mem_max_MiB	mem_avg_MiB	time_min_s	time_max_s	time_avg_s
_generate_embeddings_batch	1	6	443.2	1020.8	953.43	236.7	399.2	289.38	2.88	3.49	3.21
_faiss_search_batch	2	6	691.7	4685.5	4203.92	2364.6	3980.9	3504.1	14.28	22.01	18.09
_fetch_documents_batch	3	6	929.1	945.4	935.85	0.2	1.7	1.27	0.01	0.02	0.01
_rerank_documents_batch	4	6	930.7	1639.8	1624.05	673.7	708.7	688.2	1.84	1.99	1.9
_generate_responses_batch	5	6	684.7	2073.5	2040.67	1322.6	1322.7	1322.63	2.62	9.42	6.26
_analyze_sentiment_batch	6	6	684.7	1088.2	1054.83	329.5	355.6	336.78	1.7	1.9	1.76
_filter_response_safety_batch	7	6	682.8	1080.2	1047.1	326.9	328.2	327.67	1.39	1.47	1.41
process_batch	ENTRY	6	443.1	1119	1106.02	384.3	675.9	449.52	27.77	36.47	32.64

Analysis

From the data pulled into the raw log file we calculate the max and average max across the 6 client.py calls per step-function and overall. We make some immediate observations:

- **_faiss_search_batch is the most time consuming step, and consumes the most memory.**
- ~4.5 GB is the max memory used across the 6 requests at that step.
- ~52% of the total processing time was for that step on average as well.

With this in mind, we can estimate an optimal batch size for the initial replication of the monolith across the 3 nodes for the naive project implementation to be validated with actual benchmarking.

- Base memory: ~800 MiB (Python runtime, previous step outputs)
- FAISS index load: ~323 MiB (line 207, shared across batch)
- Per-query search: ~2873 MiB (line 209, conservatively, assume scales linearly with batch size)
- Total for batch=1: 800 + 323 + 2873 = 3996 MiB

With 16 GB of RAM as our max, and a desire to keep memory consumption at a max of 85%* our max batch size is:

Calculating optimal batch size:

Available = 16 GB × 0.85 = 13.6 GB
Batch=3: 0.8 + 0.323 + ~(2.873 × 3) = 9.74 GB
Batch=4: 0.8 + 0.323 + ~(2.873 × 4) = 12.3 GB
Batch=5: 0.8 + 0.323 + ~(2.873 × 5) = 15.0 GB → (exceeds limit)

***Regarding the 85% limit set.** In a production system, we would likely have autoscaling of compute nodes, triggered on some kind of resource consumption metric such as this memory utilization being hit. Technically, the naive pipeline script only has a single worker so we could push this to 4 to maximize resource utilization and we would be guaranteed safety assuming scaling assumptions are conservative enough, however, in a production system we would not have that guarantee of a single worker. Additionally, it provides a buffer if a particular batch of requests all consume a larger memory quantity.*

From here we moved onto experiments 2 and finally 3 using our data collected here to motivate initial design decisions.

4.2 Experiment 2

Small experiment included for completeness and to verify correctness of per node deployment understanding. Not really used for performance benchmarking. Experiment entailed deploying the same monolith across the three nodes, with a load balancer in front. See folder for code.

Through this experiment we achieve the lowest tier requirements:

- Modify the pipeline to run across 3 nodes.

- Opportunistic batching

4.3 Experiment 3

The transition from experiment 2 to experiment 3 shifts from a distributed monolith to a true microservices architecture, with major improvements in performance and scalability. In experiment 2, three monolithic pipeline instances (one per node) each ran the full pipeline with all models loaded locally, with a simple load balancer doing round-robin routing at the request level. This led to poor resource utilization, no fault tolerance, and limited scalability. In contrast, experiment 3 decomposes the pipeline into five independent microservices (embedding, FAISS search, documents/reranking, LLM generation, and sentiment/safety) distributed across nodes with service replication, enabling independent scaling of bottleneck stages. The system now introduces GPU acceleration with automatic detection, opportunistic batching that sends immediately when batches reach maximum size, per-service round-robin load balancing, a configurable multi-threaded worker pool, and service pre-loading where models load once at startup. For our profiling and experiment for this part, we ran the pipeline with different configurations (number of workers/max batch size) and recorded memory usage on each node and time reported on the client. The resulted data is reported as follows:

orchestrator_num_workers	**max_batch_size**	Max_Ram_Used_Node 0	Max_Ram_Used_Node 1	Max_Ram_Used_Node 2	reported_client_final_time(s)	Successful	Throughput (req/sec)	Throughput (req/min)
2	4	4.2	15.6	14.8	201.59	20	0.0992112704	5.952676224
	8	5.4	14.9	16.1	130.71	20	0.1530104812	9.18062887 3
	16	6.6	15.6	16	104.43	20	0.1915158479	11.49095088
4	4	7.9	15.7	16.1	131.84	20	0.1516990291	9.10194174 8
	8	8.2	16.2	16.1	127.42	20	0.1569612306	9.41767383 5
	16	8.6	15.9	16.1	99.43	20	0.2011465353	12.06879212

8	32	7.9	14.8	16.3	383.45	100	0.2607901943	15.64741166

Cell marked orange shows the peak memory used by the processes per node

Cell marked green shows the performance of the system under low load

For our analysis, we included additional visualizations to help us better understand the relation between different batch size/number of workers and performance metrics. Additional visualizations included are:

Runtime vs Workers — Shows how adding more workers affects end-to-end latency across batch sizes.

Memory vs Workers — Shows how memory usage changes as the number of workers increases for each batch size.

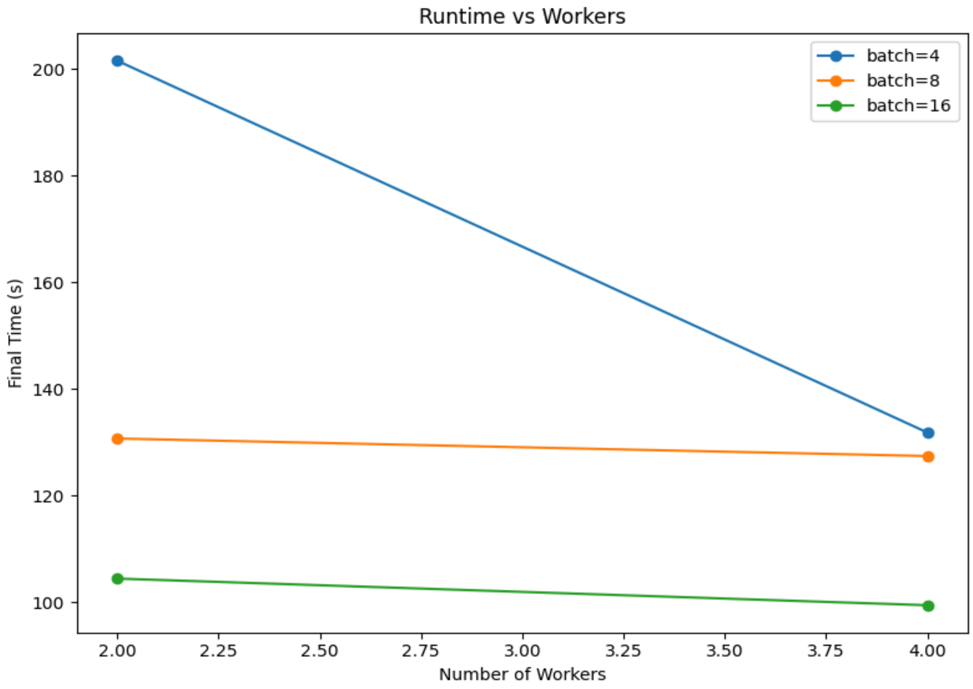
Throughput vs Workers — Shows how throughput changes as the number of workers increases for each batch size.

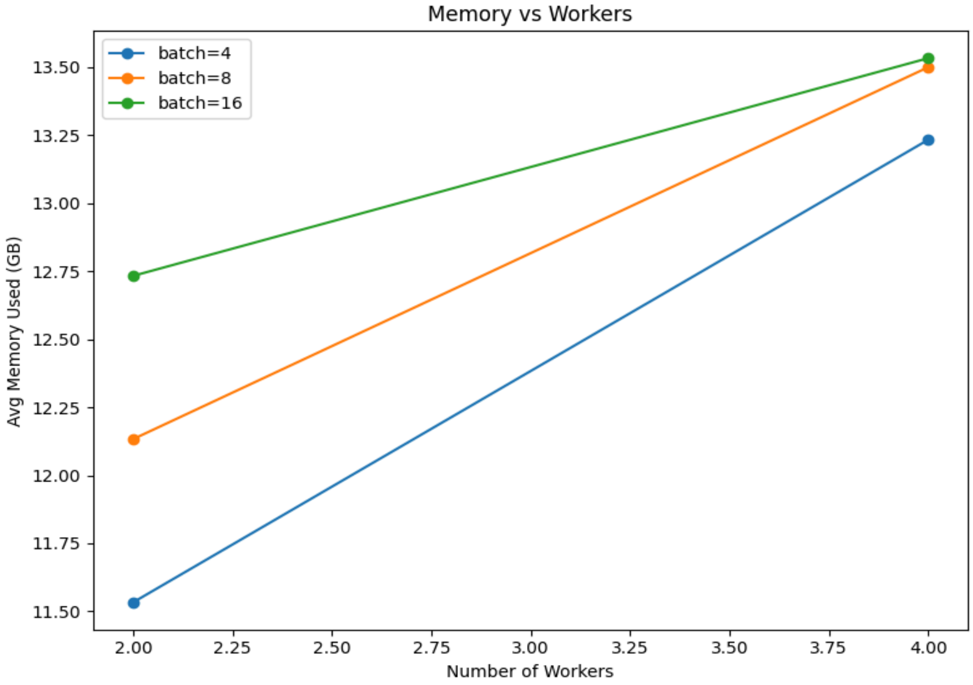
Runtime Heatmap — Visualizes latency across all worker–batch combinations to reveal performance hotspots.

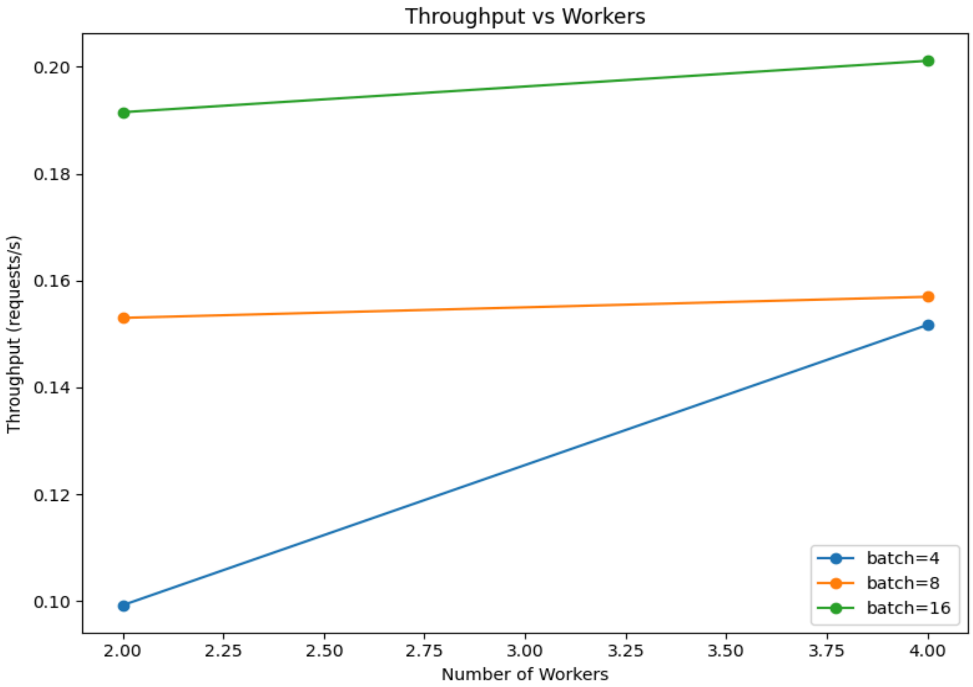
Memory Heatmap — Visualizes average memory usage across configurations to identify high-usage settings.

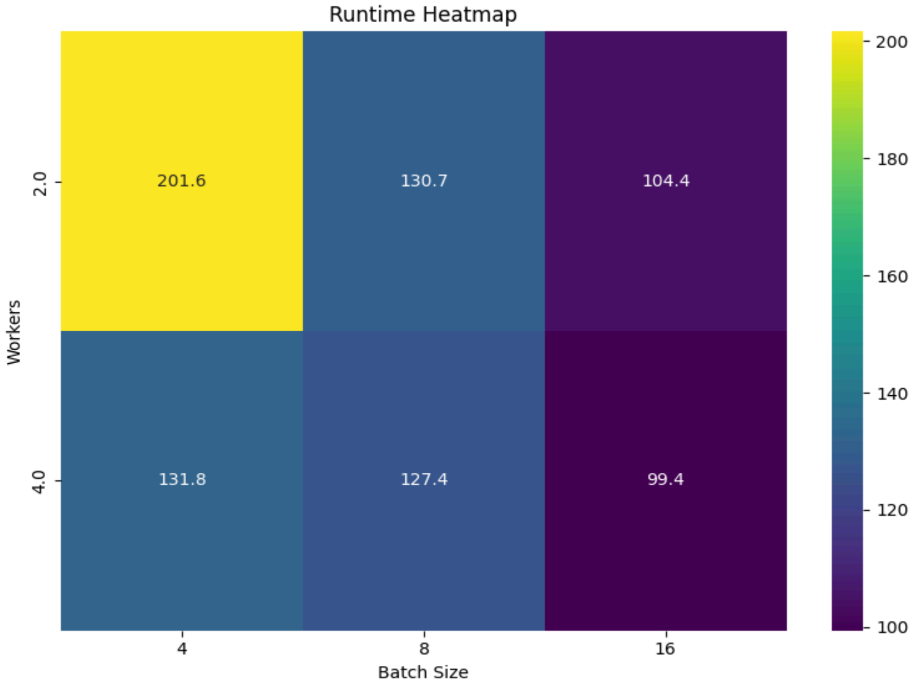
Latency vs Memory (Trade-off) — Shows the latency–memory Pareto trade-off across all configurations.

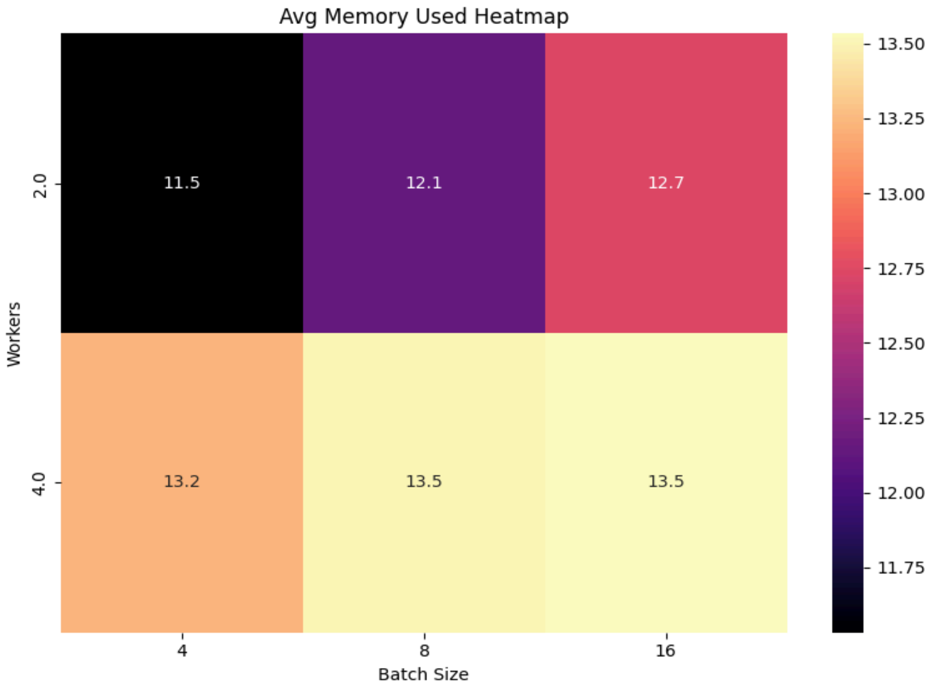
Throughput vs Memory (Trade-off) — Shows how system throughput changes with memory usage to identify configurations that maximize output rate per unit of memory.

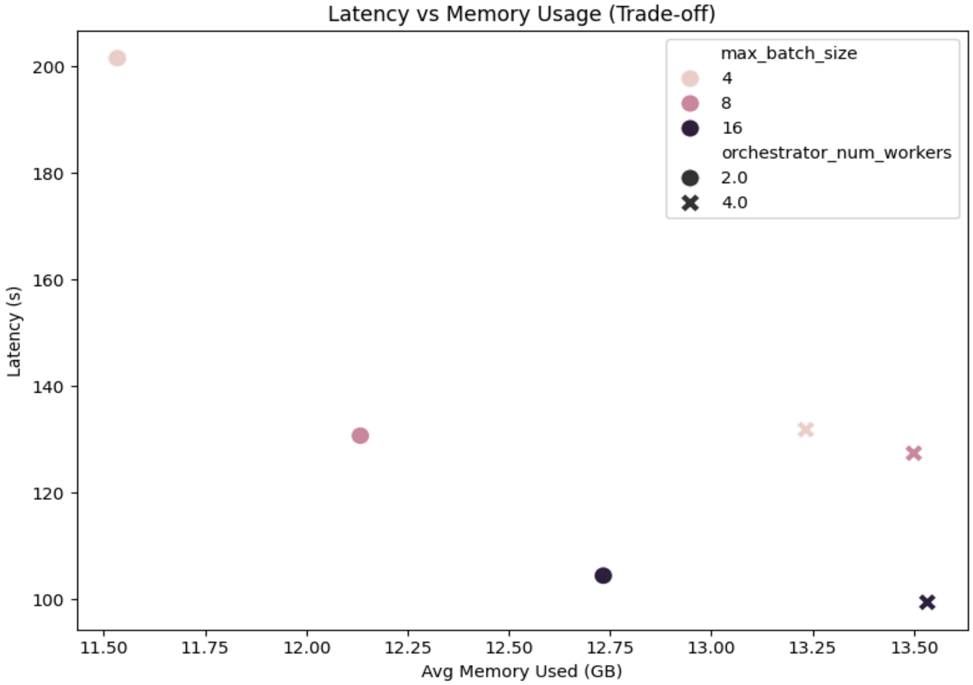

Runtime vs Workers
Final Time (s)
200
180
160
140
120
100
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
Number of Workers
batch=4
batch=8
batch=16

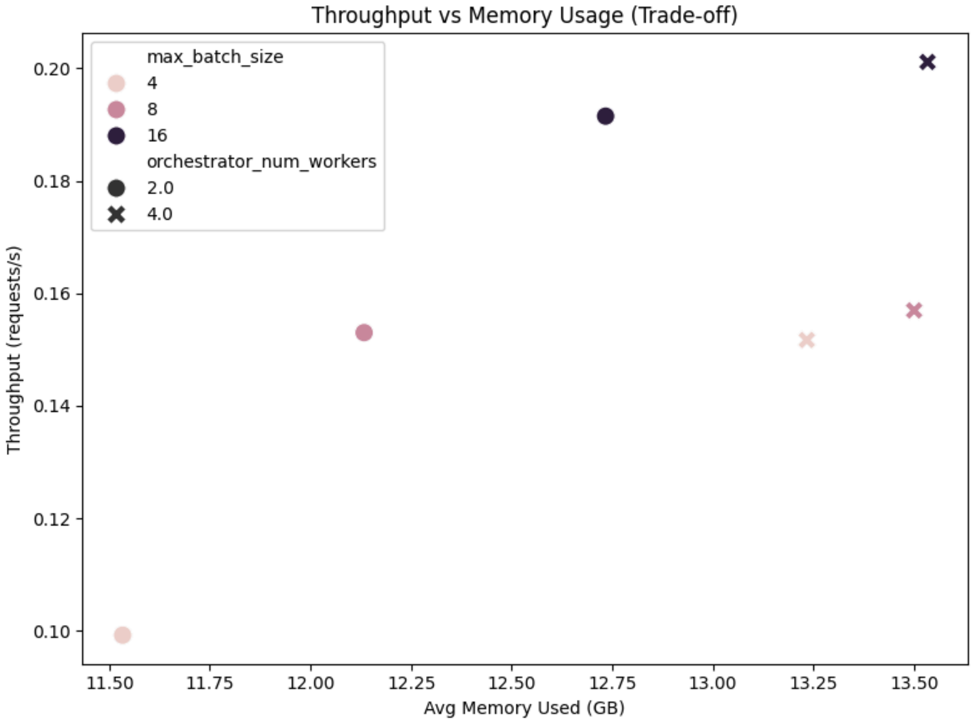

Memory vs Workers
Avg Memory Used (GB)
13.50
13.25
13.00
12.75
12.50
12.25
12.00
11.75
11.50
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
Number of Workers
batch=4
batch=8
batch=16


Throughput vs Workers
Throughput (requests/s)
0.20
0.18
0.16
0.14
0.12
0.10
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
Number of Workers
batch=4
batch=8
batch=16


Runtime Heatmap
Workers
2.0
4.0
201.6
130.7
104.4
131.8
127.4
99.4
4
8
16
Batch Size
200
180
160
140
120
100


Avg Memory Used Heatmap
Workers
2.0
4.0
Batch Size
4
8
16
11.5
12.1
12.7
13.2
13.5
13.5
13.50
13.25
13.00
12.75
12.50
12.25
12.00
11.75


Latency vs Memory Usage (Trade-off)
Latency (s)
200
180
160
140
120
100
Avg Memory Used (GB)
11.50
11.75
12.00
12.25
12.50
12.75
13.00
13.25
13.50
max_batch_size
4
8
16
orchestrator_num_workers
● 2.0
✖ 4.0



The Throughput vs. Memory Usage graph shows the **tradeoffs between throughput and memory usage** across different choices of batch size and number of workers.

To measure the **maximum workload the system can handle**, we ran a separate experiment with number of workers = 8 and batch size = 32, and issued 100 requests.The profile results are saved and numbers reported above.

Under **low load** (2 workers, 4 batch size), our system achieves a latency of 201.59s for 20 requests (avg ~10s per request).

Configuration choices: Across all our experiment runs, only the 2 workers, 4 batch size setting did not pass the 16gb threshold for every node. Even though this may not be the most efficient configuration choice, and the numbers we obtained may not be accurate due to the limitation of UGC, we chose this configuration to be on the safe side.

We want to call out that our group had to use UGC and there was observed non-deterministic resource provisioning occurring. We hope it is not from a misunderstanding of how to use the resource, but documentation was scarce. Without isolated compute environments, meaningful metric collection proved exceedingly difficult. While any server environment would have some variation due to underlying hardware-spec-drift, latency and networking variation, the degree observed on UGC points to literal shared environment resources, e.g., our execution potentially started thrashing from adjacent uncontrolled processes observed using ps from other students. In a typical compute on demand environment, cloud provider SLAs would demand much more stable performance and total environment isolation then what we observed.

Because of that, in general the metrics above are suspect—and the low test size was due to a sheer slowness of execution across multiple attempts to benchmark, albeit we got lucky for one execution occurring at 2 am for an 100 request bulk soft-stress test.

Through several sessions, and some estimation based on averages between them, we arrived at the final design—and performed the final benchmarking comparing number of orchestrator workers (threads) to batch size for 20 requests submitted in 1 ms intervals through a modified client.py. This allowed us to validate our dynamic batching implementation.

5. Conclusion

We successfully transformed the monolithic RAG pipeline into a production-ready distributed system meeting all Tier 3 requirements. Our final implementation features a true microservices architecture with five independent services (embedding, FAISS search, documents/reranking, LLM generation, and sentiment/safety analysis) distributed across three nodes with service replication for horizontal scaling.

Our profiling revealed FAISS search as the primary memory bottleneck. This insight drove our microservice decomposition strategy and informed our batch size selection. The system achieves a maximum throughput of 15.6 requests/minute (8 workers, 32 batch size) while our production configuration (2 workers, 4 batch size) safely operates within the 16GB RAM constraint across all nodes with a throughput of 5.95 requests/minute.