CS5416 Final Report

Yaxi Zeng, Morgan Mason, Shichao He, Shengjing Zhang

1. Systems Design Overview

Our system is a distributed, multi-stage retrieval-augmented generation (RAG) pipeline deployed across three nodes. Each node hosts a subset of the overall functionality- including embedding, retrieval, reranking, generation, sentiment classification, and safety filtering-and the nodes communicate via HTTP. This design enables horizontal scalability while isolating computationally expensive and memory intensive stages across specialized hardware.

A central architectural principle of our system is opportunistic batching: rather than enforcing fixed-size batches, each stage dynamically aggregates requests based on either the current batch size or a short timeout. This improves throughput and CPU utilization while keeping tail latency controlled under bursty workloads. Figure 1 provides an overview of this distributed architecture.

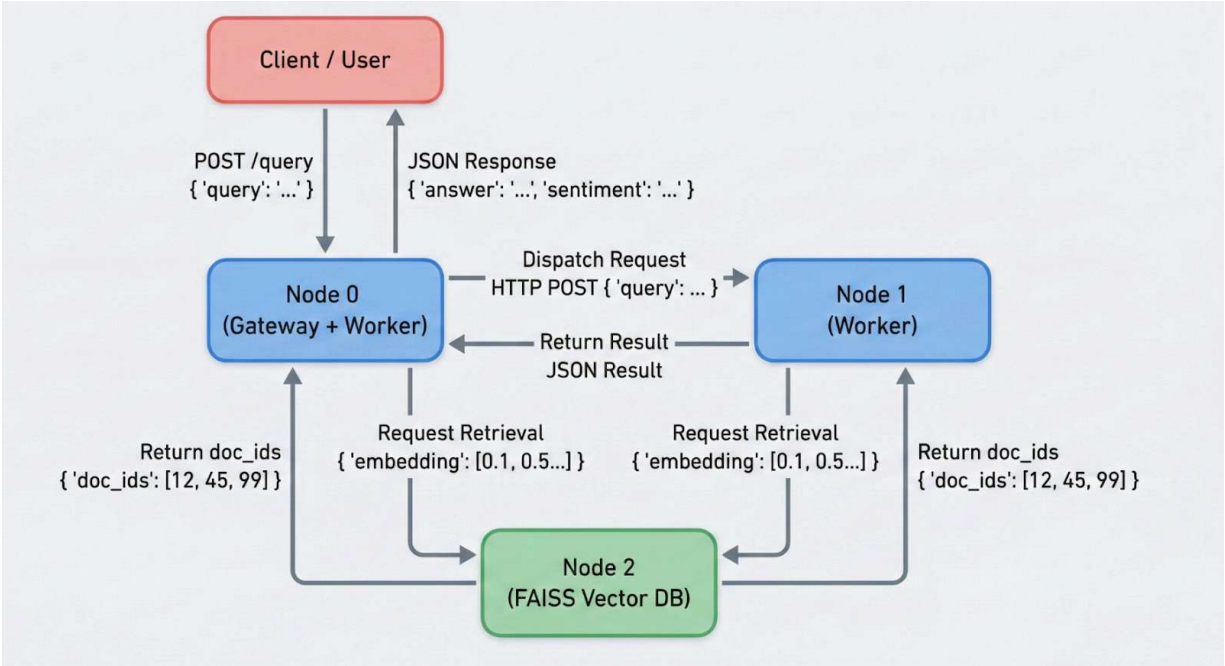


Figure 1. Distributed RAG Architecture.

1.1. Work Distribution Across Three Nodes

Figure 1 illustrates how we partition the RAG pipeline across three nodes with distinct responsibilities. This separation reduces contention, improves parallelism, and allows each stage to scale according to its own bottlenecks while keeping the overall system low-latency and resilient under load.

Node 0 serves as both the gateway and a worker. It receives all incoming client queries, performs lightweight preprocessing, and dispatches requests to compute workers using a round-robin routing strategy, which provides a simple, fair, and low-overhead mechanism for load distribution. Allowing Node 0 to execute the full pipeline locally improves resource utilization and provides robustness under bursty traffic.

Node 1 is also a worker that runs the same embedding, retrieval, reranking, and generation stages as Node 0. This symmetry simplifies orchestration, enables horizontal scaling, and provides predictable performance characteristics. Multiple workers processing requests in parallel directly satisfy the microservices and multi-node execution requirements of the project.

Node 2 hosts the FAISS vector index. Retrieval is memory-intensive and latency-sensitive, so isolating this service avoids interference with compute workloads and prevents memory pressure on worker nodes. Returning only the indexes of the top-k retrieval results minimizes cross-node communication cost and supports efficient batching in downstream stages.

Overall, this three-node architecture satisfies the distributed execution, routing, and microservices requirements of Tier 1 and Tier 2. Opportunistic batching, node specialization, and parallel compute execution further contribute Tier 3-level performance characteristics, such as improved throughput under load and reduced contention across pipeline stages.

1.2. End-to-End Request Flow

A request begins at Node 0, which receives the user query and immediately forwards the request either to itself or to Node 1 using a round-robin scheduling policy. From this point onward, both compute nodes follow the same processing pipeline.

The request first enters the embedding stage, where the input text is converted into a vector representation. The compute node then sends a retrieval request containing this embedding to Node 2, which hosts the FAISS vector index. Node 2 performs approximate nearest neighbor search and returns the top-k document IDs.

These document IDs are fed into the document fetching and reranking components. The document fetcher takes multiple requests and fetches each requested id once before returning each of the documents requested to each thread. The reranker scores candidate documents and selects the most relevant ones, producing the final ordered context used for generation.

The curated context is passed to the LLM generation stage, which produces the model's response. In parallel, the output is sent to the sentiment analysis and safety filtering modules to ensure compliance and content quality. Once all checks pass, Node 0 returns the final response to the user.

Figure 2 illustrates the end-to-end execution path, showing how the RAG pipeline is decomposed into retrieval, ranking, generation, and safety stages that execute across different nodes with clear boundaries and minimal cross-service coordination.

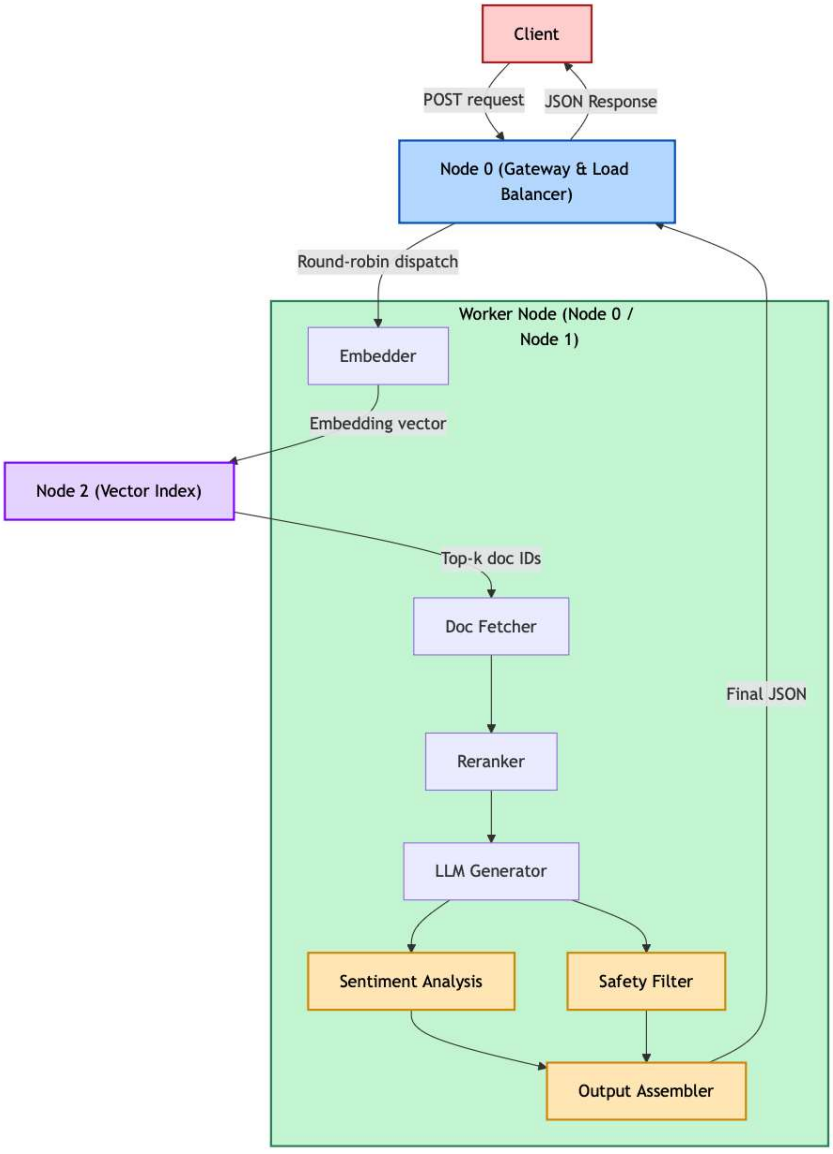


Figure 2. End-to-End Request Flow

A user request lands at the gateway (Node 0), from which it is dispatched - via round-robin scheduling - to either Node 0 (self) or Node 1 (worker). The selected compute node then performs the following pipeline: embedding → vector retrieval (by querying Node 2) → document fetch & reranking → LLM generation → parallel sentiment & safety filtering. Finally, once the response passes all checks, Node 0 returns the final JSON result to the user.

2. System Implementation Details and Optimizations

Below we describe in detail what we implemented or changed beyond the original starter code, covering request routing, asynchronous batching, modularization, service abstraction, inter-node communication, error handling, and runtime mechanisms.

2.1. Modularization of Pipeline Components

- We refactored each core pipeline function - embedding, retrieval, document fetching, reranking, LLM generation, sentiment analysis, safety filtering - into its own service class. Each class encapsulates model/resource loading, inference or processing logic, and maintains a class-specific batching service.
- Each class has a batching service that requests can submit a request to and wait on a concurrent.Future object to get a result from. The BatchingService class handles the batching logic based on _batch_job() as well as max_size (batch size) and max_wait_ms (timeout) parameters.
- This modularization dramatically improves maintainability and extensibility: individual components can be replaced (e.g. swap embedding model, change reranker) without touching other parts of pipeline or infrastructure.
- Batching each service also lets us improve our parallelism as each step can run in parallel to other steps.

2.2. Asynchronous Batching Framework for High Throughput

- Instead of processing each request synchronously and immediately, we built a unified asynchronous batching service. When a request arrives at a stage (e.g. embedding, rerank, generation), it's submitted through an interface returning a concurrent.futures.Future. The caller obtains this Future immediately, making submission non-blocking.
- Internally, each service maintains a thread-safe queue of pending tasks. A background worker thread monitors this queue: when either the number of pending tasks reaches a configured maximum batch size or the waiting time exceeds a configured timeout, the worker pops all queued tasks and invokes the _batch_job on the batch.

- After processing, the service resolves each Future with the corresponding result (or exception), unblocking the upstream logic. Upstream flows treat the result as though each request were processed individually.
- We chose this design (Future + queue + worker thread) because it provides a high-level, robust abstraction over raw thread management. The standard library's concurrent.futures module makes it easier to submit asynchronous tasks, manage result propagation, handle exceptions, and coordinate concurrency.
- Batching parameters (maximum batch size, batching timeout) are configurable - this flexibility lets us balance throughput and latency. For instance, under high load we benefit from large batches (amortizing model overhead), while under low load timeouts prevent excessive waiting.

2.3. Request Forwarding & Distributed Multi-Node Deployment

- We transformed the architecture from a single-node monolithic pipeline into a distributed, multi-node deployment: a gateway node (Node 0) that handles client HTTP requests; one or more compute worker nodes (Node 1, and optionally Node 0) that execute the pipeline stages; and a dedicated index node (Node 2) that hosts the vector index.
- When Node 0 receives a client request, it dispatches it using a round-robin scheduling policy to distribute load across workers evenly. After that, the assigned compute node (whether Node 0 or Node 1) runs the pipeline. Retrieval requests (vector search) are forwarded to Node 2 via inter-node HTTP.
- Inter-node communication uses a HTTP-based interface (JSON over HTTP). While alternatives such as binary RPC (e.g. gRPC) exist and may offer lower latency or more efficient serialization, we opted for HTTP + JSON because of simplicity, broad compatibility, ease of debugging, and because our throughput requirements are largely dominated by model inference and batch execution rather than RPC overhead.
- Deployment parameters - node roles, IP / port, paths (index, document store), batching parameters - are configured via environment variables. This allows flexible deployment configurations: single-machine testing, container-based deployment, or full cluster deployment without code changes.

2.4. End-to-End Pipeline Integration with Batching + Forwarding

- All stages - embedding, retrieval, doc-fetch, reranking, LLM generation, sentiment/safety - are integrated under the unified batching + forwarding + service framework. From the client's perspective, the external API remains unchanged: a JSON request in, JSON response out. Internally, the request may traverse multiple nodes, be batched, be processed asynchronously, but the result is returned only when fully ready.
- This integration preserves the functional contract of the baseline pipeline, while enabling scalability, resource isolation, modularity, concurrency, and high throughput.

2.5. Error Handling, Timeouts, and Robustness Considerations

- Because we rely on asynchronous execution and inter-node HTTP, we added mechanisms to handle timeouts and exceptions. When calling future.result(timeout=...), we detect and catch timeouts or exceptions from worker threads. HTTP calls to other nodes (e.g. retrieval on Node 2) are wrapped in try/except; failures are propagated back via Futures or HTTP error responses, enabling upstream code to handle or log errors.
- This design ensures that a failure in one batch or one stage does not silently hang the system; instead, errors are surfaced and can be logged, retried, or lead to graceful failure (e.g. returning an error JSON to client). This improves system robustness and fault tolerance.
- The use of concurrent.futures' built-in semantics allows clean handling of cancellation, exceptions, and ensures that resources (threads, memory) are not leaked over time - important for long-running servers.

2.6. Key Dependencies & Runtime Mechanisms for the New Architecture

Below are the essential tools / libraries / runtime mechanisms we rely on - specifically because of our architectural and performance enhancements, not merely baseline pipeline logic:

- concurrent.futures + thread-safe queues + background worker threads - forms the core asynchronous batching infrastructure across all stages.
- Lightweight HTTP server / web framework + JSON-over-HTTP - provides both the public gateway API and inter-node communication between gateway, workers, and index node.
- Modular service abstraction layer - encapsulating each pipeline stage into a service class improves decoupling, maintainability, and extensibility under the distributed deployment pattern.
- Environment-variable driven configuration - allows flexible deployment across different environments (single-machine, container, multi-node cluster) without code changes, enabling realistic, scalable deployment.

These dependencies and mechanisms together form the foundation that supports our distributed, batched, high-throughput RAG pipeline, enabling modularity, scalability, robust execution, and efficient resource utilization.

3. Batching Implementation

Our system adopts a unified opportunistic batching framework that wraps every major computational stage in the pipeline. Instead of processing requests synchronously, each stage exposes a non-blocking `submit()` interface backed by a `BatchingService`. Requests are pushed into a per-stage queue, and a background worker aggregates them into batches based on maximum batch size or a timeout threshold, whichever is reached first.

This approach allows us to amortize expensive GPU/CPU computation across multiple requests, reduce redundant model overhead, and significantly improve throughput under load, while still offering controlled latency under light traffic.

3.1 Stages That Use Batching

Every computational stage in our distributed pipeline uses batching:

Stage	Node	Reason for Batching
Embedder	Node 0/1	Light model; batching reduces repeated forward-pass overhead.
FAISS retrieval	Node 2	FAISS can process large matrices efficiently; batching reduces RPC overhead.
Document Fetch	Node 0/1	A single SQL query benefits from merging multiple doc-id sets.
Reranker (Transformer)	Node 0/1	Largest win from batching; large matrix multiplications are highly batch-efficient.
LLM generation	Node 0/1	Hugely expensive; batching improves GPU/CPU utilization but is limited by memory.
Sentiment classifier	Node 0/1	Hugely reduced overhead when batched.
Safety classifier	Node 0/1	Similar to sentiment; benefits from shared tokenizer + model.

3.2 Opportunistic Batching Mechanism

- Each stage runs inside the same batching abstraction:
- A thread-safe queue holds incoming jobs
- Each submission returns a Future, allowing asynchronous execution
- The worker thread waits until either:
- batch size reaches max_size, or
- waiting time exceeds max_wait_ms
- The worker then executes _batch_job(batch) on the entire group
- Results are written back to individual Futures

This provides a clean interface:

```
f = embedder_service.submit(request_id, {"query": query})
embedding = f.result(timeout=300)
```

The upstream logic never deals with batching directly; it simply receives the final result.

Queries are wrapped in a BatchedObject to handle the relation between inputs and futures

```
@dataclass

class BatchItem:

    """Request and a Future object for returning the result."""

    request_id: str

    data: Dict[str, Any]

    future: Future
```

A simplified version of the actual batching worker:

```
item = self.input_queue.get()
current_batch.append(item)

if len(current_batch) >= max_size or wait_time >= timeout:
    outputs = process_batch_func(batch)
    for item, out in zip(batch, outputs):
        item.future.set_result(out)
    reset batch and timer
```

This mechanism allows small batches under low load and full batches during peak load.

3.3 Batch Size Choices

We tuned our batch sizes by looking at the behavior of each component individually. More information is available in 4. Profiling, Experiments, and Conclusions.

Stage	Max Batch Size	Timeout (ms)
Embedder	8	~60 ms
DocFetch	8	~20 ms
Reranker	16	~60 ms
LLM	4	500 ms
Sentiment	8	~200 ms
Safety	4	~80 ms
FAISS	16	500 ms

3.4 Why Opportunistic Batching Works Well

- Under high load: requests accumulate quickly → full batches → excellent throughput.
- Under low load: timeout fires → small batches → low latency.
- Future-based design: upstream pipeline remains non-blocking and resilient to slow stages.
- Stage independence: each microservice can tune its batch sizes differently.

This batching design is essential for scaling to production-style workloads and directly meets the Tier-3 requirement for opportunistic batching.

4. Profiling, Experiments, and Conclusions

We conducted extensive profiling to characterize the memory footprint, throughput, and latency of our distributed pipeline. All profiling was performed on UGClinux machines using the same constraints as the autograder (16 GB RAM, CPU-only unless GPU explicitly enabled).

4.1. Per service tests

For each of our individual services, we tested its performance over several different batch sizes.

Batch_Size	Avg_Time_s	Std_Dev_s	time_per_batch
1	0.038238674	0.005872018	0.038238674
2	0.05840561	0.01005338	0.029202805
4	0.09641348	0.001879496	0.02410337
8	0.154587486	0.020192288	0.019323436
16	0.33663838	0.020024367	0.021039899

Figure 3. Embedder Latency vs Batch Size

Batch_Size	Avg_Time_s	Std_Dev_s	time_per_batch
1	1.309616729	0.025396089	1.309616729
2	1.350990573	0.010203893	0.675495287
4	1.364134877	0.011325094	0.341033719
8	2.72109176	0.035002587	0.34013647
16	4.286062816	0.095409351	0.267878926

Figure 4. FAISS Latency vs Batch Size

Batch_Size	Avg_Time_s	Std_Dev_s	time_per_batch
1	0.010139142	0.021459831	0.010139142
2	0.01854468	0.039594807	0.00927234
4	0.023754777	0.063810402	0.005938694
8	0.04092459	0.114800999	0.005115574
16	0.046170615	0.130429293	0.002885663

Figure 5. Document Fetcher Latency vs Batch Size

Batch_...	Avg_Tim...	Std_Dev_s	time_per_batch
1	0.828848...	1.225480...	0.828848425
2	0.816207...	0.052432...	0.408103871
4	1.824045...	0.131183...	0.456011486
8	3.848616...	0.091092...	0.48107702
16	9.160374...	2.625155...	0.572523386

Figure 6. Reranker Latency vs Batch Size

Batch_Size	Avg_Time_s	Std_Dev_s	time_per_batch
1	68.90587411	0	68.90587411
2	47.75324761	9.772436703	23.8766238
4	64.2407009	1.485047367	16.06017523

Figure 7. LLM Latency vs Batch Size

Batch_Size	Avg_Time_s	Std_Dev_s	time_per_batch
1	0.169626466	0.186846727	0.169626466
2	0.520575307	0.254014321	0.260287653
4	0.585467639	0.482674108	0.14636691
8	0.600464647	0.142849895	0.075058081
16	1.266023805	0.685291169	0.079126488

Figure 8. Sentiment Latency vs Batch Size

Batch_Size	Avg_Time_s	Std_Dev_s	time_per_batch
1	0.087781611	0.006946013	0.087781611
2	0.150550076	0.014414719	0.075275038
4	0.167516672	0.026234812	0.041879168
8	0.711819611	0.042789384	0.088977451
16	1.181253711	0.063632963	0.073828357

Figure 8. Safety Latency vs Batch Size

We chose the batch size that was most efficient for our services and set the timeout to be either half of the difference in the chosen batch size latency and bs=1 latency or 500 ms, whichever was lower. Our choice in batch size was to optimize for processing the most requests per second. Our choice in latency was to prevent requests from waiting so long that they would not benefit from batching.

Furthermore, from these experiments, we can see that the LLM will be the bottleneck for latency: all requests must go through it and it will take an order of magnitude longer to run than the other services. This cannot be improved without better hardware (GPUs). For this reason, we decided to focus on increasing the throughput of our system and duplicating the node/LLM. So that requests won't have to wait on other requests going through the LLM.

4.2. Tests of different request patterns

We wanted to test how our model would fare in different scenarios. We chose to test our pipeline against small numbers of requests to understand the per-request behavior, against

large bursts of requests to understand how it would handle this situation, and request spaced out at regular intervals to see if we would be able to still benefit from batching.

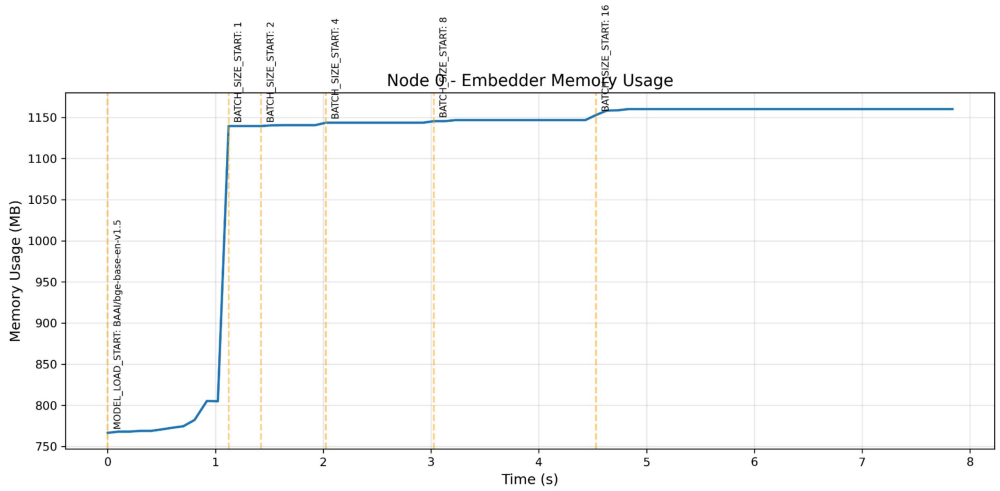
n requests	time between requests (s)	average lateny (s)
1	0	34.73
2	0	35.69
4	25	38.83
8	0	92.25
16	0	138.36
8	5	53.49

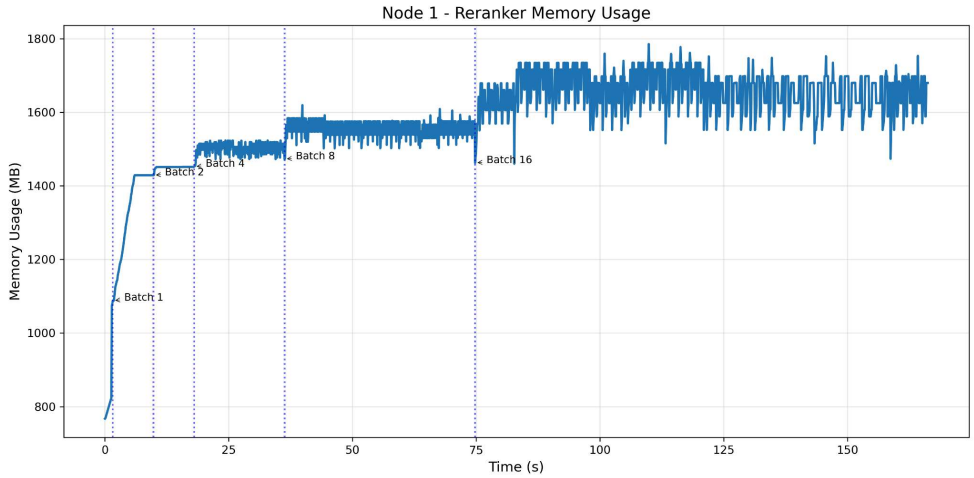
From our first, second, and third tests we see that when the pipeline is not under much stress (because of few / infrequent requests) we are capable of serving requests in about 35 seconds. It is important to make sure that our batching timeouts do not make these infrequent requests wait for too long. From our LLM experiments, finishing in 40 seconds feels like an appropriate amount of time to wait.

From our fourth, fifth, and sixth tests, we see that when large numbers of request come in at once, they frequently get gummed up. When requests are sent without any waiting, this is especially evident as all the requests get stuck at the LLM bottle neck. When there is more time in between requests, each request individually will have to wait less as the LLM will have more time to clear out.

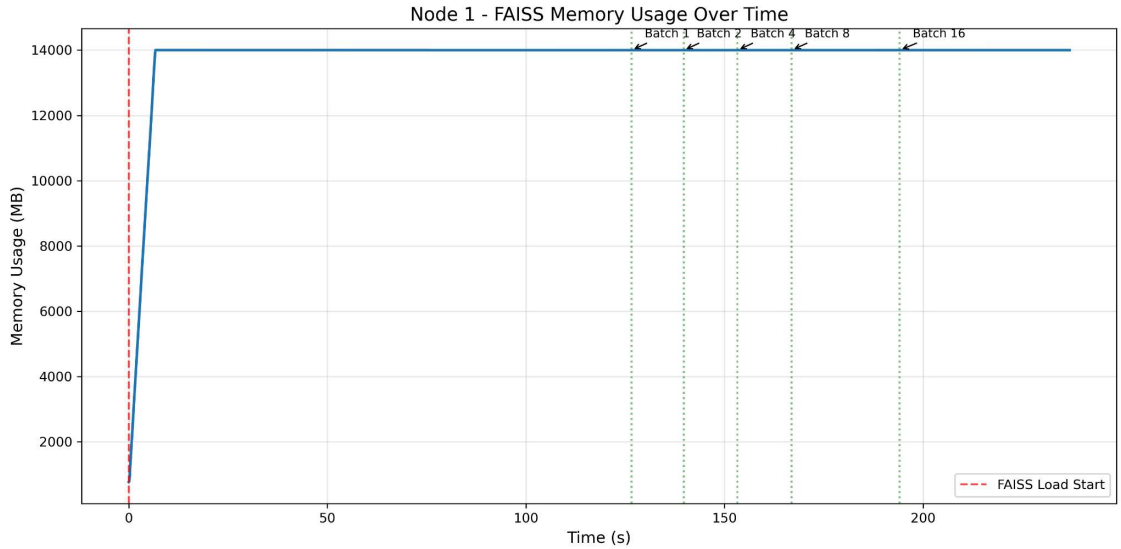
4.3. Memory Usage per Service

During our per service tests, we profiled the memory usage of each individual model for a given batch size. The majority of models showed a modest increase in memory for each batch size, but all (except FAISS) stayed well under 2GB.





However, Faiss ended up using around 14GB of memory to load the model in.



Because FAISS runs quickly and is efficient at high batch sizes, but takes up a large amount of memory, we decided that FAISS needs to be on its own node, but that node could be shared among the other nodes. Likewise, since all our other nodes take up little memory and are mostly sequential (though safety and sentiment can be run in parallel) we would put all other services on a single node that is duplicated to increase throughput.

4.4. Peak Memory usage per service

Memory was measured using:

```
psutil.Process(os.getpid()).memory_info().rss
```

Node	Loaded Components	Peak Memory (GB)	Notes
Node 0	Embedder, Reranker, LLM, Sentiment, Safety	9 GB	LLM accounts for the largest footprint.
Node 1	Same as Node 0	9 GB	Provides parallel compute capacity.
Node 2	FAISS index only	~14 GB	FAISS index dominates; isolated node required.

This reaffirms that FAISS requires its own node to run, but also that we might have had space to run multiple LLM processes to further increase throughput.

4.5. Architecture Test

Early on, we had implemented a 3 stage pipeline, where Node 0, 1, 2 each were a batch service spanning multiple models and pipelines. We later opted for our current architecture to achieve higher parallelism, less communication, and better latency. We tested both our architectures on a burst of requests and steady flow of requests.

n requests	time between requests (s)	double main latency	3 stage latency
8	0	92.25	128.72
8	5	53.49	100.53

As we see, though both services perform worse when a burst of request comes in, our current architecture - double main - performs much better overall.

4.6 Final Conclusion

Profiling reveals two major bottlenecks:

(1) LLM Generation

- Highest latency and compute cost
- Small batch sizes required to avoid memory spikes
- Dominates tail latency and throughput ceilings

(2) FAISS Index Memory Footprint

- ~13GB index forces isolation onto Node 2
- High startup cost and non-negligible paging risk
- Retrieval itself is extremely fast (<10 ms), but memory footprint constrains architecture

These observations justified separating FAISS and LLM onto different nodes and motivated the microservice partitioning in our final architecture.