# Stage 3: Building a Cluster Architecture

*Search Engine Project*

---

**Authors:** Daniel Nosek ID: 46409300, Lennart Schega ID: L3NH5Z6F4, Domen Kac ID: IE0380466, Nico Brockmeyer ID: L3KW1LXJ, Anna Sowińska ID: DEY61301

**Date:** 18.01.2026

**University:** Universidad de Las Palmas de Gran Canaria (ULPGC)

**Course Name:** 40386 Big Data

**Group Name:** Lennart

**Repository:** https://github.com/lennart-group/stage-3

# 2. Introduction and objectives

Stage 3 builds on the modular setup from Stage 2 by deploying the system in a distributed way, with a focus on making it scalable and fault-tolerant. This version uses Docker Compose to run each part of the system as its own isolated service: IngestAPI, IndexAPI, SearchAPI, a Controller service (which handles the user-facing REST APIs), and an ActiveMQ message broker that helps coordinate tasks asynchronously. Search requests go through an Nginx entrypoint, which routes both **/search** and **/status** to the Search service.

The main goals here were to separate ingestion, indexing, and search into independently deployable services, allow the system to scale horizontally (by spinning up multiple instances of services), and ensure that the system can recover from crashes using health checks and automatic container restarts. We also ran a demo and performance tests to see how well the system handled queries and indexing under load, and how it behaved during intentional service failures.

# 3. System architecture

## 3.1 Cluster topology and service roles

The system is made up of separate services, each running in its own container. Here's what each does:

1. **Controller** – Exposes REST APIs to users and coordinates the processing steps.
2. **Ingest** – Downloads and stores the raw document data.
3. **Index** – Builds and updates an inverted index for searching.
4. **Search** – Handles user queries and returns results based on search terms.
5. **ActiveMQ** – A message broker that allows Ingest and Index to work together without being directly connected.
6. **Nginx** – Routes incoming search-related traffic to the correct backend service.

Every core service (controller, ingest, index, search) has a **/status** endpoint so we can monitor their health and debug if something goes wrong.

### 3.2 Data flow

Ingestion → Persistence → Indexing → Search

**Ingestion and Persistence**

When a user sends a request for a specific book ID, the Controller triggers the Ingest service via HTTP. Ingest downloads the book from Project Gutenberg, extracts the metadata (like title, author, release date, and language), and grabs the main content. It then stores this data in MongoDB, using an idempotent upsert keyed by the book ID so that we avoid duplicate entries.

**Event-Driven Indexing Coordination**

Once the document is saved, Ingest sends a document.ingested event through ActiveMQ. The Index service listens for these events with a background worker, allowing it to start indexing as soon as a new document is ingested without needing a direct call from Ingest. Indexing can also be started manually through its own REST API if needed.

**Building the Inverted Index**

The Index service splits the content into individual terms and updates the inverted index stored in MongoDB. It organizes terms into collections based on the first character (a, b, c, etc.). Each term maps to a list of book IDs where that term appears. The updates use addToSet with upserts to avoid duplicates and to make sure reruns don't mess anything up.

**Query Execution and Filtering**

When a user searches, the request goes through Nginx to the Search service. Search splits the query into terms, looks up each one in the index, and finds the intersection of matching book IDs (i.e., books that match *all* terms). It can also apply optional filters like author, language, or release year by querying the books collection. The response includes the original query, applied filters, number of results, and basic metadata for each matching book.

### 3.3 Service coordination and fault handling mechanisms

**Coordination**

Ingest and Index work together using asynchronous messaging, Ingest sends out events via ActiveMQ, and Index listens for them. The Controller also includes retry logic for HTTP calls between services, so temporary failures don't break the system.

**Health Monitoring and Recovery**

Every core service has a **/status** endpoint, and Docker is configured to automatically restart any service that fails. Nginx acts as a steady entry point for search, forwarding both **/search** and **/status** to the Search service. It also helps detect backend failures. The Controller and Index services keep simple control files (like **processed_books.txt** and **indexed_books.txt**) to avoid reprocessing documents that have already been handled.

## 3.4 Architecture diagrams

1. **Data flow**

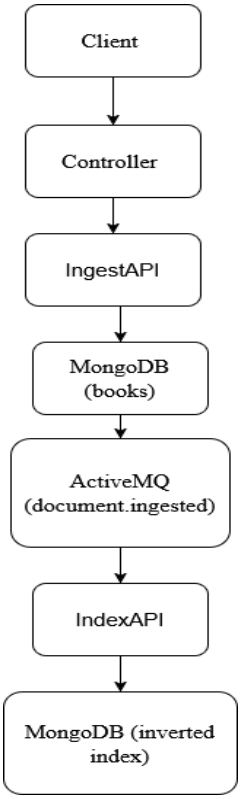


**Figure 1. Indexing pipeline.**

*The controller triggers ingestion for a selected book ID. IngestAPI downloads the book, stores it in MongoDB (books), and publishes a document.ingested event to ActiveMQ. IndexAPI consumes the event (via the indexing worker) and updates the inverted index in MongoDB.*

2. **Search pipeline**

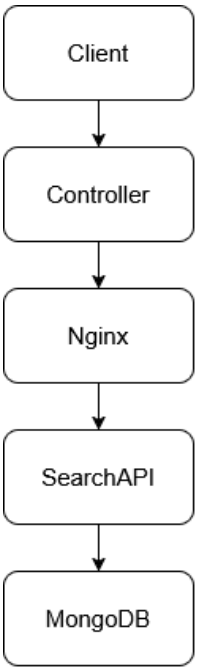


**Figure 2. Search pipeline.**

*The client sends queries to the controller, which forwards them to the Nginx entrypoint. Nginx proxies requests to SearchAPI. SearchAPI retrieves postings from the inverted index and applies optional metadata filters using the books collection in MongoDB.*

# 4. Implementation details

### 4.1 Distributed Datalake / Persistent Storage (MongoDB)

For storing data, we use **MongoDB** with the official Java driver. The connection string gets built dynamically at runtime using environment variables (SERVICES_USER, SERVICES_PASSWORD), these are URL-encoded to make sure they work properly in the connection URL.

Our system connects to **two databases** hosted in the same MongoDB deployment:

- BigData – stores the raw ingested books and their metadata in a books collection.
- invertedIndex – holds the inverted index, organized into multiple collections (buckets) based on the first character of each term.

To make sure everything is set up correctly, we check database connectivity right at startup by pinging both databases. If they're unreachable, the app fails fast instead of hanging.

### 4.2 Message Broker (ActiveMQ)

We use **ActiveMQ Classic** for asynchronous messaging between the Ingest and Index services. This is handled using the **Jakarta JMS API**, with a custom MessageBroker wrapper that sets up a JMS connection using the BROKER_URL. It runs a non-transacted session and uses Session.AUTO_ACKNOWLEDGE for simplicity.

There are three main queues:

- document.ingested – sent by the Ingest service once a book is saved.
- document.indexed – sent by the Index service once indexing is finished.
- reindex.request – queue defined for reindex triggers (hook available in the codebase).

*Messages are sent as **JSON-encoded text**, like:*

***{ "event": "...", "bookId": ... }***

The Index service listens to document.ingested using an **asynchronous MessageListener** called IndexingWorker, which reads the book data from MongoDB and kicks off the indexing process.

### 4.3 Indexing Implementation (Persistent + In-Memory)

**Persistent Inverted Index (MongoDB):**

The Index service tokenizes book content into unique lowercase terms and stores a persistent inverted index in MongoDB, partitioned into bucket collections by the first character of each term. Index updates use addToSet with upsert semantics to avoid duplicates and remain idempotent.

### 4.4 Load Balancer / Entrypoint (Nginx)

All search-related traffic goes through a single **Nginx entrypoint** on port 80. Nginx acts as a single entrypoint for search traffic. In our current Compose setup it proxies **/search** and

**/status** to the SearchAPI service (search:7003). When multiple SearchAPI instances are deployed, Nginx can be configured with an upstream group to distribute requests.

We've configured Nginx with **passive failure detection** using settings like max_fails and fail_timeout, so if the backend becomes unresponsive, it gets flagged. For other services (like the controller), we expose a consistent entrypoint using:

***SEARCH_API=http://nginx:80***

# 5. Design decisions

## 5.1 Deployment Topology (Single-Service Containers)

The system is deployed using **Docker Compose**, with each core component running in its own container that includes the Controller, Ingest, Index, Search services, as well as **ActiveMQ** and **Nginx**. By separating them, each service can be restarted independently (thanks to restart: always) and any issues are easier to isolate and debug. Plus, the setup is clean and reproducible.

To keep everything healthy, each of the main services exposes a /status endpoint, which can be used for monitoring.

## 5.2 Decoupling Ingestion from Indexing Using a Message Broker

Instead of tightly connecting the Ingest and Index services, the system uses **ActiveMQ** to loosely couple them. When Ingest finishes saving a book, it sends out a document.ingested event. The Index service listens for these messages asynchronously using a **JMS MessageListener** (IndexingWorker) and processes them as they come in.

This **event-driven approach** allows for better flexibility—if indexing is delayed or temporarily down, messages can queue up without breaking the system. The messaging session is **non-transacted** and uses AUTO_ACKNOWLEDGE, so we don't need to manually acknowledge messages in our code. This simplifies things and boosts throughput, relying instead on the broker and container recovery to handle failures.

### 5.3 Data Partitioning and Indexing Strategy (Persistent + In-Memory)

We use two types of inverted indexes:

- **Persistent Index (MongoDB):**
  The Index service stores a durable inverted index in the invertedIndex MongoDB database. Terms are grouped into collections based on their first letter (a, b, etc.). Updates use addToSet with upsert, which avoids duplicates and allows us to safely run the process multiple times without side effects.

- **In-Memory Index (Hazelcast):**
  We also have an in-memory version using **Hazelcast**, which provides a distributed **MultiMap** called inverted-index. Hazelcast is set up with backups (backupCount=2, asyncBackupCount=1) to make sure we don't lose data if a node goes down. At the moment, SearchAPI reads from the MongoDB-based index. The Hazelcast-based in-memory index is a prototype component and is not yet used in the live query path.

### 5.4 Synchronization and Concurrent Updates

In the Hazelcast-based prototype, race conditions are prevented using distributed locks per term. These are managed using **FencedLocks** from its CP Subsystem. Locks are applied to keys like ***lock:inverted-index:<term>***, so different terms can be processed in parallel, but updates to the same term are kept safe.

On the control side, both the Controller and Index services use simple tracking files like ***processed_books.txt*** and ***indexed_books.txt*** to keep track of what's already been handled. This avoids reprocessing and keeps things predictable across runs.

### 5.5 Load Balancer and Failure Recovery

All search traffic is routed through a single **Nginx** entrypoint, which forwards **/search** and **/status** requests to the Search service running at search:7003. This gives clients and the Controller a consistent address to use:

***SEARCH_API=http://nginx:80***

Nginx is also set up with **passive failure detection** using max_fails and fail_timeout, so if the backend has issues, it's detected automatically.

Recovery is mainly handled by Docker's **restart policy** (restart: always) and the health check endpoints (/status). Additionally, the Controller includes **retry logic** for internal HTTP calls between services, which helps the system recover smoothly from short-term network hiccups or startup delays.

# 6. Benchmarks and results

The system was tested using a dataset of 118 Project Gutenberg books stored in MongoDB. At the time of evaluation, the BigData database (books and metadata) used about 69 MB, and the inverted index around 11.9 MB. Services were built with Java 17 and deployed via Docker Compose (v3.9). The setup included a replica-set storage backend (replication factor of 3), ActiveMQ Classic (v5.18.3) for async coordination via Jakarta JMS, and an Nginx (v1.29.4) entrypoint for search requests.

Benchmarks were run across three scale-out configurations (S1–S3), adjusting the number of IngestAPI and Index instances. Ingestion requests were routed through a single Nginx endpoint.

**Key Findings:**

- **Ingestion throughput** dropped with scale-out, suggesting that replication didn't help due to shared I/O and backend contention.

- **Indexing throughput** improved with scale, showing benefits from parallelism. However, **indexing latency** first dropped then rose, likely due to coordination overhead.

- **Search latency** was tested using ten random queries. Average latency slightly improved with scaling, though variability remained due to differences in term selectivity and list size.

- **Resource usage** (Figures 5–6) showed stable memory across services, but CPU usage spiked under load, especially for indexing and search workers. Network I/O stayed low, consistent with a CPU-bound workload.

In summary, indexing scaled well, query latency saw minor gains, but ingestion became less efficient. Trade-offs were largely due to shared infrastructure.

Referenced figures include ingestion and indexing performance across nodes, query latency ranges, and Docker resource snapshots. A demo video and benchmark reproduction steps are available in the project README.

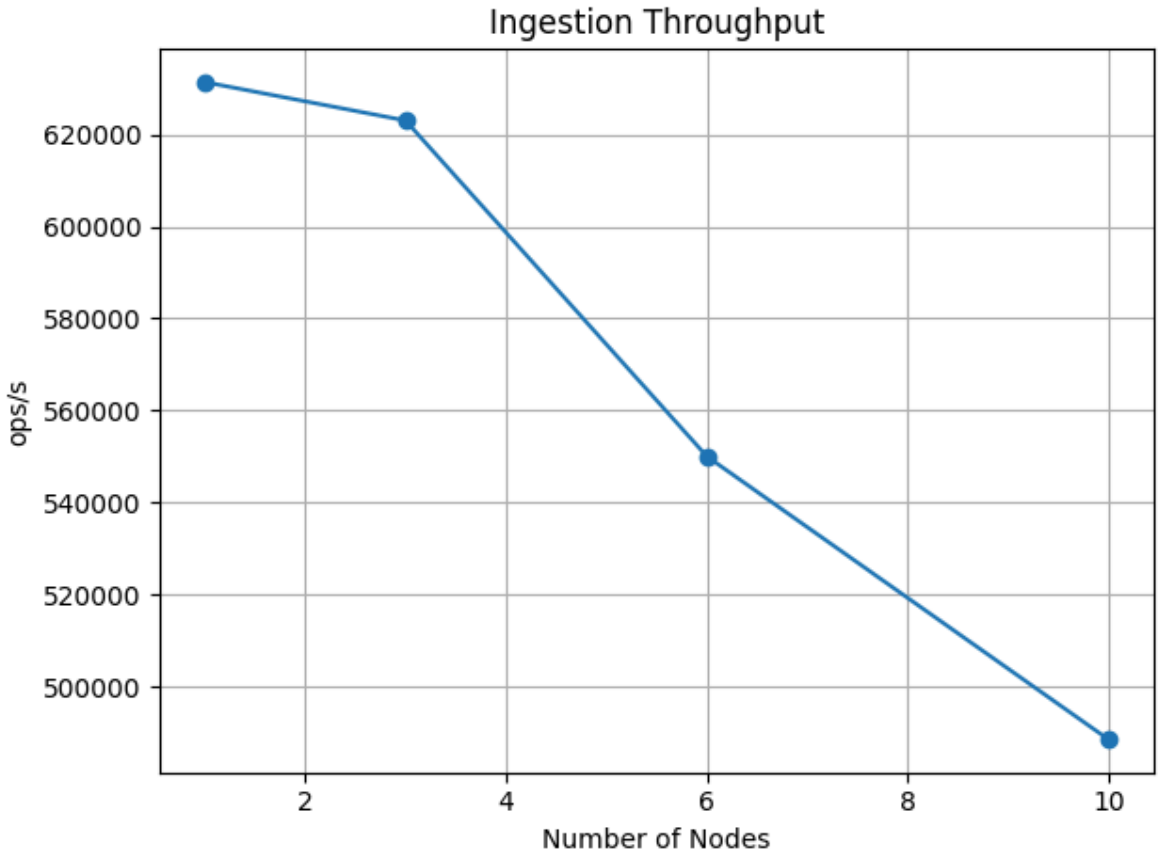


**Figure 1.** Ingestion throughput (ops/s) vs. number of nodes (N = 1, 3, 6, 10).

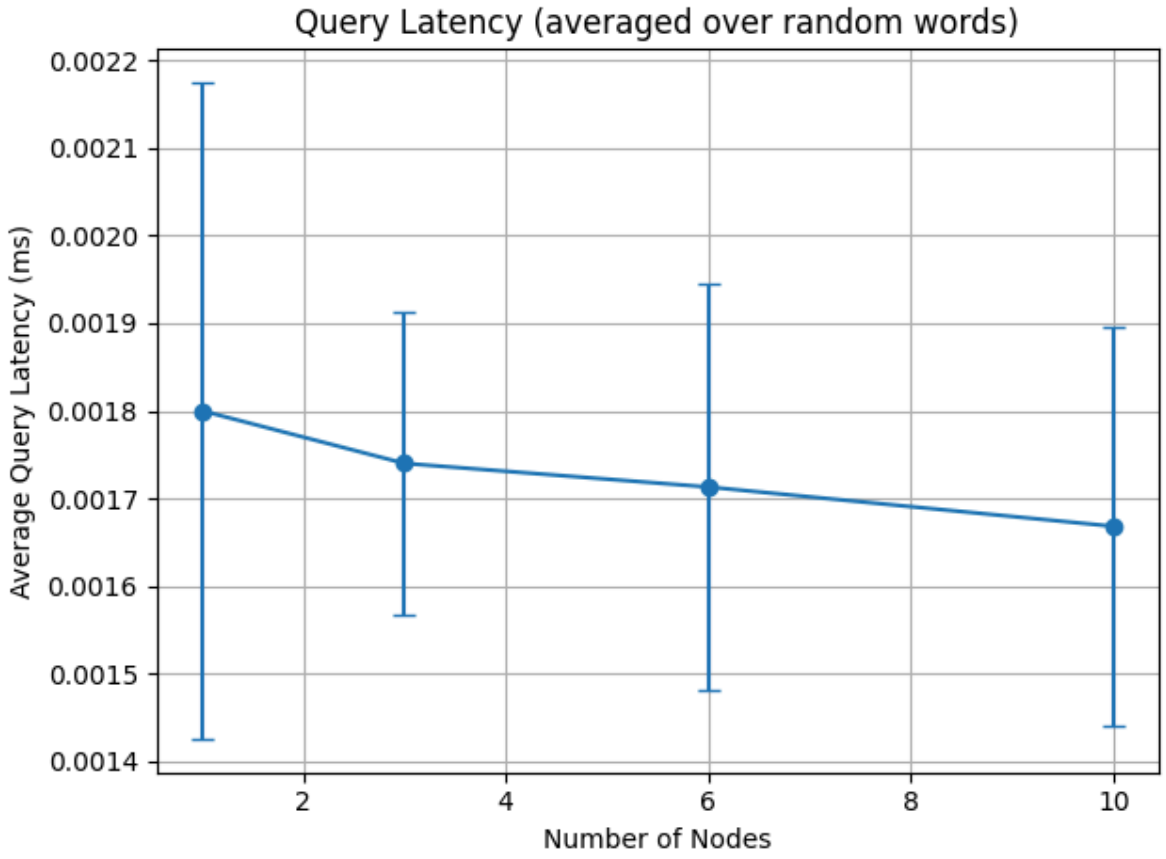


**Figure 2.** Query latency averaged over 10 random query terms (min–max range shown) vs. number of nodes (N = 1, 3, 6, 10).

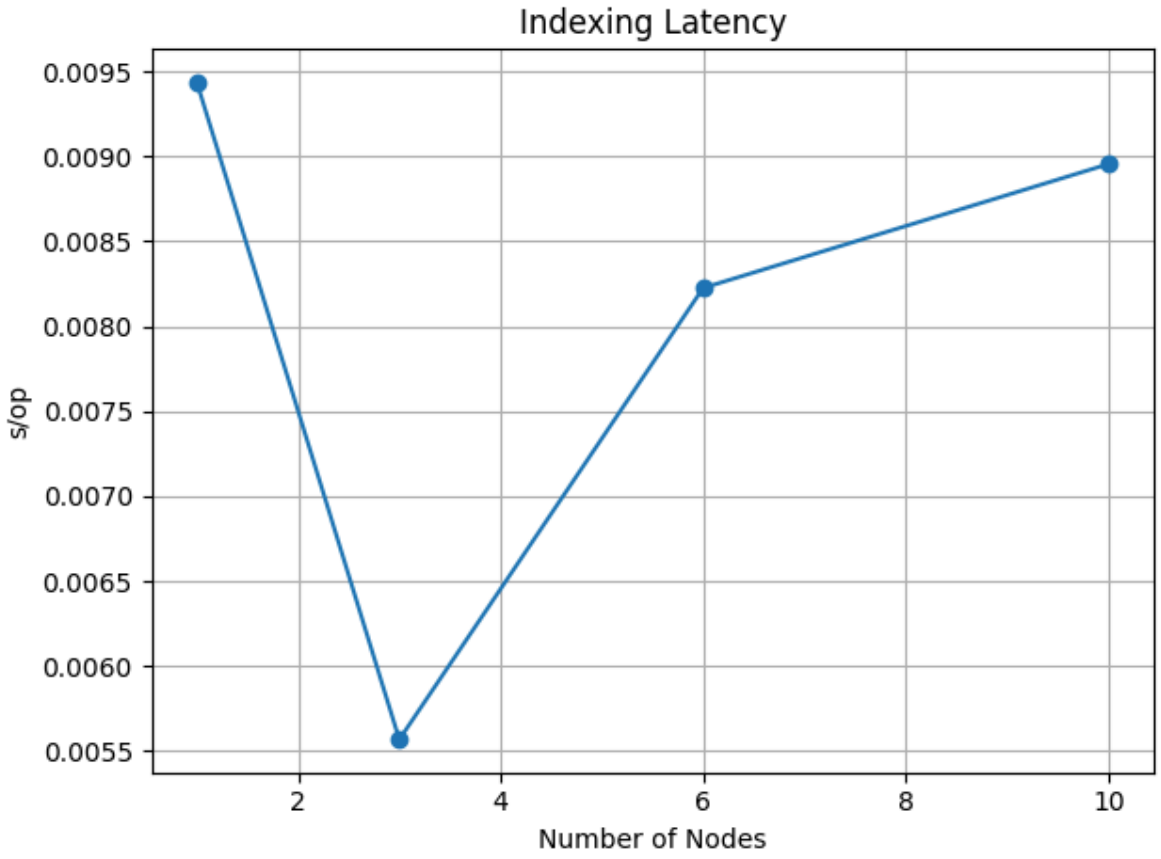


**Figure 3.** Indexing latency (s/op) vs. number of nodes (N = 1, 3, 6, 10).

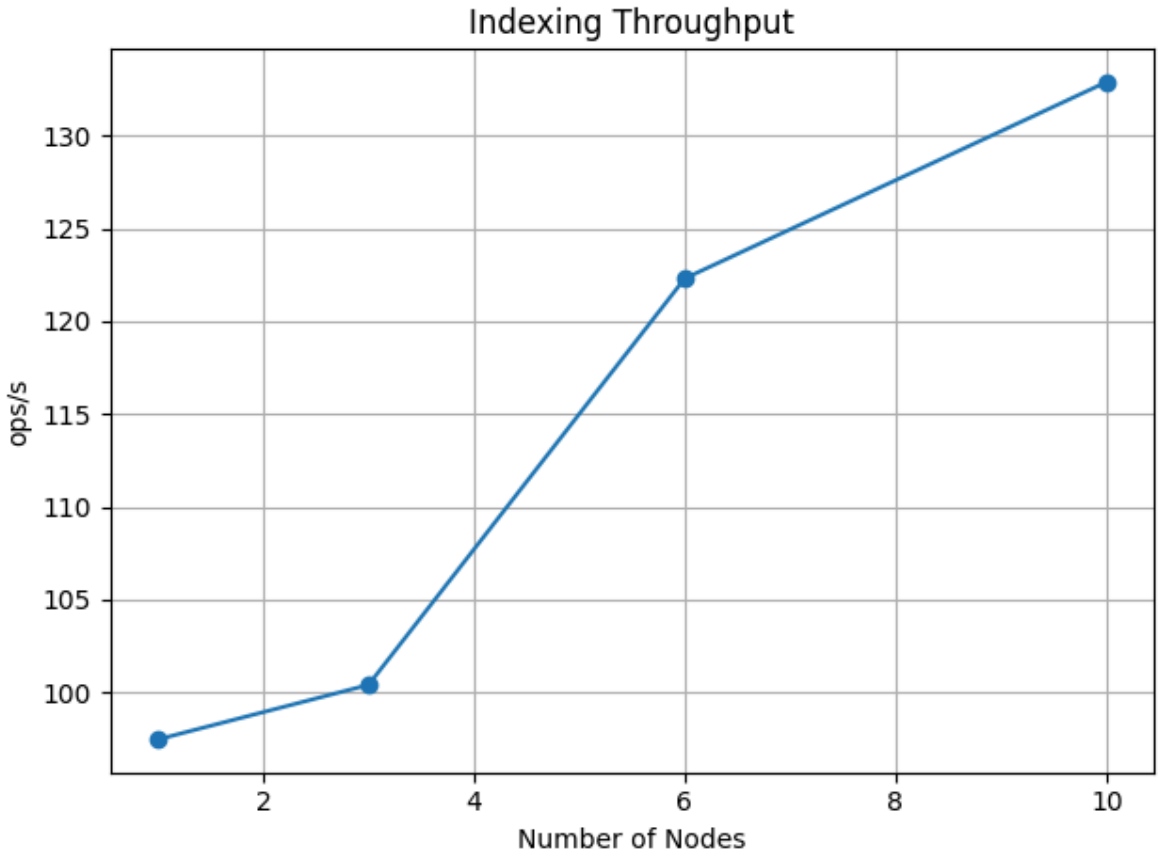


**Figure 4.** Indexing throughput (ops/s) vs. number of nodes (N = 1, 3, 6, 10).

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O | BLOCK I/O | PIDS |
|---|---|---|---|---|---|---|---|
| 60a7af8921a1 | docker-search-4 | 0.33% | 109.9MiB / 15.46GiB | 0.69% | 34.7kB / 15.4kB | 0B / 0B | 56 |
| 4b6bd3daf9c1 | activemq | 535.42% | 317.1MiB / 15.46GiB | 2.00% | 8.04kB / 5.31kB | 0B / 0B | 88 |
| 160423a632dd | docker-search-2 | 0.20% | 105.9MiB / 15.46GiB | 0.67% | 34.5kB / 15.4kB | 0B / 0B | 55 |
| 3521117d03bc | docker-index-worker-3 | 0.13% | 87.08MiB / 15.46GiB | 0.55% | 35.9kB / 17.2kB | 0B / 0B | 43 |
| 15b18838229a | docker-search-1 | 0.13% | 109.8MiB / 15.46GiB | 0.69% | 34.2kB / 15.5kB | 0B / 0B | 54 |
| 21cf5dd12e0e | docker-ingest-1 | 2.90% | 120.5MiB / 15.46GiB | 0.76% | 35.2kB / 16.8kB | 0B / 0B | 58 |
| bf5833afe597 | docker-index-worker-2 | 0.11% | 88.52MiB / 15.46GiB | 0.56% | 31.4kB / 16.9kB | 0B / 0B | 43 |
| 30b74701d68f | docker-search-3 | 17.48% | 117.3MiB / 15.46GiB | 0.74% | 33.9kB / 15.4kB | 0B / 0B | 55 |
| 80170607a01b | docker-ingest-6 | 27.58% | 121.8MiB / 15.46GiB | 0.77% | 34.8kB / 16.8kB | 0B / 0B | 58 |
| 597339bfdcb6 | docker-index-worker-1 | 1.75% | 79.33MiB / 15.46GiB | 0.50% | 28.7kB / 13kB | 0B / 0B | 39 |
| 8a62409c6de0 | docker-search-5 | 2.42% | 77.47MiB / 15.46GiB | 0.49% | 21.4kB / 10.3kB | 0B / 0B | 38 |
| d80d35009064 | docker-ingest-4 | 26.86% | 77.42MiB / 15.46GiB | 0.49% | 21kB / 8.38kB | 0B / 0B | 38 |
| 91328067bc5e | docker-index-worker-4 | 0.25% | 74.76MiB / 15.46GiB | 0.47% | 16.3kB / 5.94kB | 0B / 0B | 36 |
| c0ed3a3310a6 | docker-search-6 | 36.57% | 74.81MiB / 15.46GiB | 0.47% | 11kB / 4.08kB | 0B / 0B | 33 |
| 5e8268de5120 | docker-ingest-5 | 36.08% | 70.69MiB / 15.46GiB | 0.45% | 2.9kB / 994B | 0B / 0B | 31 |
| dffbec363d61 | docker-index-worker-5 | 0.86% | 63MiB / 15.46GiB | 0.40% | 2.86kB / 994B | 0B / 0B | 29 |
| 7e3fd0e8f708 | load_balancer | 0.00% | 2.426MiB / 15.46GiB | 0.02% | 720B / 0B | 0B / 0B | 2 |
| 9fb7b4e211dc | docker-ingest-3 | 0.04% | 70.73MiB / 15.46GiB | 0.45% | 2.44kB / 814B | 0B / 0B | 34 |
| 7d53bb595ac3 | docker-index-worker-6 | 0.04% | 65.39MiB / 15.46GiB | 0.41% | 2.56kB / 772B | 0B / 0B | 29 |
| 192df16baa3a | docker-ingest-2 | 0.04% | 71.17MiB / 15.46GiB | 0.45% | 2.4kB / 772B | 0B / 0B | 34 |
| 7f49a9916cb9 | controller | 0.03% | 62.68MiB / 15.46GiB | 0.40% | 218B / 0B | 0B / 0B | 25 |

**Figure 5.** Docker resource utilization snapshot during the benchmark run (CPU, memory, network I/O per container).

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O | BLOCK I/O | PID |
|---|---|---|---|---|---|---|---|
| 50a7af8921a1 | docker-search-4 | 0.10% | 111.1MiB / 15.46GiB | 0.70% | 34.3kB / 15.4kB | 0B / 0B | 56 |
| 4b6bd3daf9c1 | activemq | 209.92% | 221.1MiB / 15.46GiB | 1.40% | 1.53kB / 96B | 0B / 0B | 36 |
| 160423a632dd | docker-search-2 | 118.48% | 113.2MiB / 15.46GiB | 0.72% | 33.9kB / 15.4kB | 0B / 0B | 55 |
| 3521117d03bc | docker-index-worker-3 | 126.45% | 82.55MiB / 15.46GiB | 0.52% | 33.4kB / 15.1kB | 0B / 0B | 42 |
| 15b18838229a | docker-search-1 | 130.30% | 79.61MiB / 15.46GiB | 0.50% | 32.7kB / 14.4kB | 0B / 0B | 40 |
| 21cf5dd12e0e | docker-ingest-1 | 110.59% | 75.34MiB / 15.46GiB | 0.48% | 16.2kB / 5.94kB | 0B / 0B | 35 |
| bf5833afe597 | docker-index-worker-2 | 139.27% | 74.84MiB / 15.46GiB | 0.47% | 15.5kB / 3.65kB | 0B / 0B | 36 |
| 30b74701d68f | docker-search-3 | 130.87% | 71.32MiB / 15.46GiB | 0.45% | 3.1kB / 970B | 0B / 0B | 34 |
| 30170607a01b | docker-ingest-6 | 154.15% | 72.4MiB / 15.46GiB | 0.46% | 2.74kB / 970B | 0B / 0B | 34 |
| 597339bfdcb6 | docker-index-worker-1 | 0.03% | 55.16MiB / 15.46GiB | 0.35% | 1.15kB / 244B | 0B / 0B | 29 |
| 3a62409c6de0 | docker-search-5 | 0.02% | 38.38MiB / 15.46GiB | 0.24% | 447B / 136B | 0B / 0B | 25 |
| d80d35009064 | docker-ingest-4 | 173.02% | 75.76MiB / 15.46GiB | 0.48% | 15.5kB / 3.65kB | 0B / 0B | 37 |
| 91328067bc5e | docker-index-worker-4 | 0.01% | 28.52MiB / 15.46GiB | 0.18% | 218B / 0B | 0B / 0B | 23 |
| c0ed3a3310a6 | docker-search-6 | 0.04% | 70.66MiB / 15.46GiB | 0.45% | 2.56kB / 772B | 0B / 0B | 34 |
| 5e8268de5120 | docker-ingest-5 | 0.02% | 53.96MiB / 15.46GiB | 0.34% | 1.02kB / 244B | 0B / 0B | 29 |

**Figure 6.** Docker resource utilization snapshot under higher load / different benchmark phase (CPU, memory, network I/O per container).

# 7. Conclusions and future improvements

Stage 3 uses the ingestion-indexing-search pipeline as separate services in Docker Compose. Ingestion and indexing are decoupled via ActiveMQ (document.ingested), while MongoDB holds the book catalog and the durable inverted index maintained by SearchAPI. Basic fault tolerance is added via /status endpoints and container restart policies. The benchmarking data indicates that indexing throughput increases with scale-out, while query latency slightly decreases on average (with variation depending on the query term), and ingestion throughput decreases with scale-out, suggesting that ingestion is bottlenecked by shared external I/O and backend contention. Future work should include setting up Nginx to load balance traffic across multiple SearchAPI instances for search replication, automate benchmarking with fixed workloads and tail-latency reporting (p95/p99), optimize ingestion throughput via throttling/backpressure, incorporate the Hazelcast in-memory index component into the query path in real time while retaining MongoDB as the source of truth, and add fault injection tests with recovery time measurement for services, brokers, and databases.

*Note about using AI: During the development of this project, we made use of AI-based tools to support the coding and documentation process.*