Big Data Architecture

School of Computer Engineering
Universidad de Las Palmas de Gran Canaria

Stage 3 Report

Design and Implementation of a Scalable, Fault-Tolerant Distributed Search Engine

---

Group Name: Microsoft-2

Project Members:

Sergio Muela Santos
Daniel Medina González
Jorge Cubero Toribio
Enrique Padrón Hernández

GitHub Repository:
`https://github.com/Microsoft-2/stage-3`

Las Palmas de Gran Canaria
December 16, 2025

Contents

List of Figures

1 Introduction

1.1 Project Scope and Context

In the current landscape of large-scale data processing, systems must evolve beyond monolithic architectures to handle the "Three Vs" of Big Data: Volume, Velocity, and Variety. The objective of all three assignments have taken us through the process of building a search engine, from a basic sequential implementation.

Stage 3 represents the culmination of this process. While Stage 2 focused on modularizing components into microservices (Crawler, Indexer, Search Engine) communicating via REST APIs, that architecture suffered from significant limitations regarding scalability and reliability. If the central Indexer failed, the entire pipeline stalled. If query volume increased, the single Search node became a bottleneck.

This report documents the transformation of that system into a **Distributed Cluster**. By leveraging containerization (Docker), message brokering (ActiveMQ), and in-memory data grids (Hazelcast), we have engineered a system where scalability and fault tolerance are foundational properties rather than afterthoughts.

1.2 Objectives

The development of this stage was driven by four key technical requirements:

1. **Horizontal Scalability:** The ability to increase the system's throughput linearly by adding commodity hardware (nodes) rather than upgrading existing hardware (vertical scaling).

2. **Resilience and Fault Tolerance:** The system must survive the abrupt failure of any single component. Data persistence must be guaranteed through replication strategies in the Datalake and the Inverted Index.

3. **Asynchronous Coordination:** Decoupling the ingestion and processing layers using an event-driven architecture to maximize resource utilization.

4. **Distributed Consistency:** Ensuring that search results remain consistent across multiple replicas of the search service using shared in-memory structures.

2 System Architecture

2.1 Architectural Paradigm

The system implements a **Service-Oriented Architecture (SOA)** specialized for data processing pipelines. It combines the *Producer-Consumer* pattern for data ingestion with the *MapReduce* paradigm (internally managed by Hazelcast) for distributed query processing.

Unlike traditional N-Tier architectures, our system does not rely on a centralized relational database. Instead, it uses a polyglot persistence strategy:

- **Raw Data:** Stored in a distributed file system (Simulated via Docker Volumes).
- **Derived Data (Index)**: Stored in a partitioned In-Memory Data Grid (IMDG).
- **Transient State:** Managed via a Message Broker.

2.2 Logical Topology

The cluster is composed of five distinct logical components, each responsible for a specific domain of the search engine lifecycle.

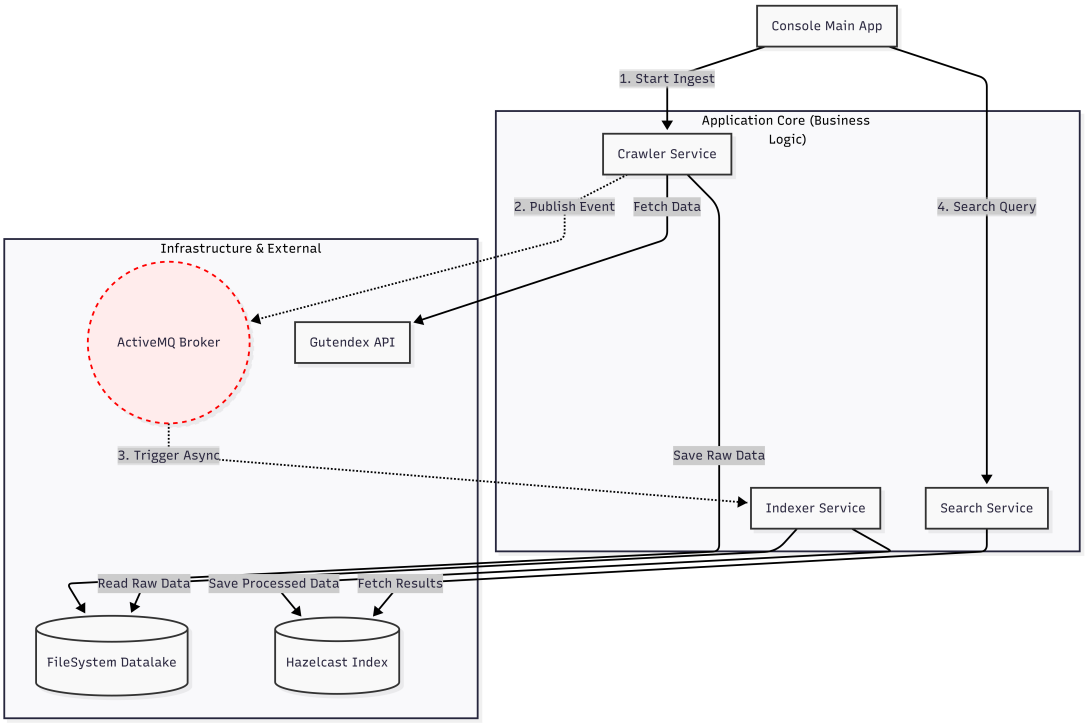

Figure 1: Logical Architecture of the Search Engine Cluster, highlighting the separation of concerns.

2.2.1 The Ingestion Layer (Crawlers)

The Crawlers are the "Producers" of the system. They operate autonomously, scraping external data sources (Gutenberg Project, through Gutendex). In Stage 3, crawlers are designed to be stateless and parallel; multiple crawlers can operate simultaneously on different book ranges without coordination, provided they write to the shared Datalake.

2.2.2 The Coordination Layer (ActiveMQ)

The Message Broker acts as the system's central nervous system. It decouples the Crawlers from the Indexers. This decoupling is critical for burst handling: if Crawlers download books faster than Indexers can process them, the messages simply accumulate in the queue rather than crashing the system.

2.2.3 The Processing Layer (Indexers)

The Indexers are "Consumers." They listen to the `document.downloaded` queue. Upon receiving a message, they fetch the raw content from the Datalake, perform text normalization (tokenization, stop-word removal), and ingest the data into the cluster memory.

2.2.4 The Storage Layer (Hazelcast)

Hazelcast provides the distributed inverted index. It partitions the data across all available nodes. Each partition has a primary owner and at least one backup replica. This ensures that if a node crashes, the data remains available on the backup node, providing high availability.

2.2.5 The Serving Layer (Search API & Nginx)

The Search Nodes expose a REST API to the outside world. They act as clients to the Hazelcast cluster. The Nginx load balancer sits in front of these nodes, routing traffic using a Round-Robin or Least-Connections algorithm to ensure even load distribution.

2.3 Physical Topology (Docker)

We adopted a **Single-Service Node** topology. This means each Docker container runs exactly one process.

- **Isolation**: A memory leak in the Indexer does not crash the Search Node.
- **Scalability**: We can scale the number of Indexers independently of the number of Crawlers.

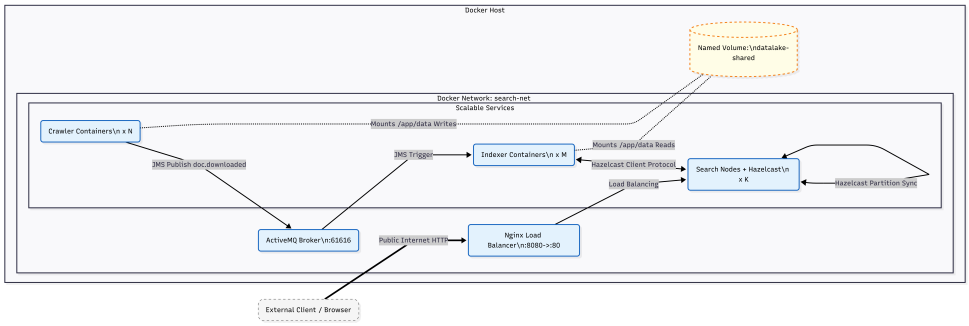

Figure 2: Docker Network Topology showing the custom bridge ‘search-net‘.

3 Implementation Details

This section documents the technical implementation of the components, highlighting the specific code structures used to achieve the project requirements.

3.1 Distributed Datalake: The Crawler Service

The ‘CrawlerNode‘ class coordinates the downloading process. It implements a robust workflow to handle the unreliability of external networks.

3.1.1 Resilience: Exponential Backoff

To interact with the Gutendex API, we implemented the ‘GutendexAdapter‘ class. A key challenge was handling HTTP 429 (Rate Limit) errors. We implemented an exponential backoff algorithm:

1. If a request fails with 429, the thread sleeps for T seconds (initially 10s).
2. If it fails again, T is doubled ($2T$).
3. This continues up to a maximum retry limit.

This ensures our cluster ”plays nice” with the public API.

```
public Book downloadBook(String id) {
    int attempt = 0;
    long waitTime = 10000; // Start with 10 seconds

    while (attempt < MAX_RETRIES) {
        try {
            Response response = client.newCall(request).execute();

            if (response.code() == 429) {
                System.out.println("(!) Rate Limit hit. Waiting " +
   waitTime + "ms");
                Thread.sleep(waitTime);
                waitTime *= 2; // Exponential increase (10s -> 20s -> 40
   s)
                attempt++;
                continue;
            }
            return parseBook(response);

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            break;
        } catch (Exception e) {
```

```
            e.printStackTrace();
            attempt++;
        }
    }
    return null; // Failed after retries
}
```

Listing 1: GutendexAdapter.java: Exponential Backoff Implementation

3.1.2 Persistence: Shared Volumes

The Datalake is implemented using a Docker Named Volume (‘datalake-shared‘). Within the container, this is mounted at ‘/app/data‘. The Crawler writes files using standard Java NIO (‘Files.write‘). This abstraction allows the application to treat distributed storage as a local file system.

3.2 Messaging Backbone: ActiveMQ Integration

We utilized the JMS (Java Message Service) standard to interact with ActiveMQ.

3.2.1 Message Structure

The messages sent to the ‘document.downloaded‘ queue are ‘TextMessages‘ containing a JSON payload. We chose JSON over binary serialization for easier debugging and interoperability.

```
{
  "id": "1042",
  "path": "/app/data/1042.txt",
  "timestamp": 1702384921
}
```

3.2.2 Consumer Implementation

The Indexer does not actively poll the server. Instead, it registers a ‘MessageListener‘. This asynchronous approach allows the system to process messages immediately as they arrive without blocking the main thread.

```
@Override
public void subscribe(String topicName, Consumer<String> listener) {
    try {
        Topic topic = session.createTopic(topicName);
        MessageConsumer consumer = session.createConsumer(topic);

        // Register callback (Lambda expression)
```

```
        consumer.setMessageListener(message -> {
            try {
                if (message instanceof TextMessage) {
                    String json = ((TextMessage) message).getText();
                    String docId = extractId(json);

                    // Trigger the Domain Event
                    listener.accept(docId);
                }
            } catch (JMSException e) {
                e.printStackTrace();
            }
        });
    } catch (JMSException e) {
        throw new RuntimeException("Error subscribing to ActiveMQ", e);
    }
}
```

Listing 2: ActiveMQEventBus.java: Asynchronous Consumer

3.3 The Indexing Engine

The ‘IndexerNode‘ is the most resource-intensive component. It performs the ”Map” phase of our MapReduce-like flow.

3.3.1 Hazelcast Integration

We used the ‘MultiMap‘ structure from Hazelcast. Unlike a standard ‘Map¡String, List¡String¿¿‘, a ‘MultiMap‘ stores multiple values for a single key in a distributed manner. This is ideal for an Inverted Index where one word (Key) appears in many books (Values).

```
@Override
public void saveEntry(String word, String bookId) {
    // "inverted-index" is a distributed map shared across the cluster
    MultiMap<String, String> index = hazelcastInstance.getMultiMap("
    inverted-index");

    // The 'put' operation appends the bookId to the list of values for
    'word'
    // This is thread-safe and replicated automatically by Hazelcast
    index.put(word, bookId);
}
```

Listing 3: HazelcastIndexRepository.java: Distributed Ingestion

3.4 The Serving Layer

The Search Service provides the retrieval mechanism. Since the data is already partitioned in memory, the "Search" operation is a direct O(1) lookup in the Distributed Map.

```
public Set<String> search(String query) {
    // Normalize query to match indexing format
    String term = query.toLowerCase().trim();

    // Retrieve collection from the grid
    // This may involve a network hop if the key is stored on a
   different node
    Collection<String> results = indexRepository.get(term);

    if (results == null || results.isEmpty()) {
        return Collections.emptySet();
    }

    // Return unique set of documents
    return new HashSet<>(results);
}
```

Listing 4: SearchService.java: Retrieval Logic

4 Cluster Orchestration

4.1 Docker Compose Configuration

The ‘docker-compose.yml‘ file serves as the Infrastructure-as-Code (IaC) definition for our cluster. It orchestrates the lifecycle of the application, defining services, networks, and persistent volumes.

4.1.1 Service Definitions

Below is an excerpt of the configuration showing the relationship between the Broker and the Indexer service.

```
services
  activemq
    image apache/activemq-classic
    ports
      - "61616:61616"
      - "8161:8161"
    networks
      - search-net

  indexer
    build .
    command ["java", "-jar", "app.jar", "indexer"]
    depends_on
      - activemq
      - hazelcast-master
    volumes
      - datalake-shared/app/data
    deploy
      replicas 3  # Start with 3 indexers
      restart_policy
        condition on-failure

networks
  search-net
    driver bridge

volumes
  datalake-shared # Persistent volume outside containers
```

Listing 5: docker-compose.yml: Infrastructure Definition

4.1.2 Network Isolation

We created a custom bridge network named ‘search-net‘. This provides: 1. **DNS Resolution:** Containers can reach each other by service name (e.g., ‘ping activemq‘). 2. **Isolation:** The database traffic (Hazelcast internals) is not exposed to the host machine or other networks.

5 Design Decisions and Analysis

This section explores the trade-offs considered during the architectural design.

5.1 Topological Choices: Single vs. Multi-Service

We evaluated two topologies suggested in the project guidelines:

1. **Multi-Service Node:** Running Crawler + Indexer in the same container.
2. **Single-Service Node:** Decoupled containers.

Decision: We chose Single-Service. **Reasoning:** While Multi-Service improves data locality (no network transfer between crawler and indexer), it couples their scalability. If indexing is CPU-heavy but downloading is Network-heavy, we want to scale them independently. Single-Service allows us to run 1 Crawler and 10 Indexers if needed.

5.2 Consistency vs. Availability (CAP Theorem)

Hazelcast acts as an AP (Availability/Partition Tolerance) system in our configuration.

- We prioritized **Availability**. We want the search engine to always return results, even if some nodes are down.
- We accepted **Eventual Consistency**. When a book is indexed, it might take a few milliseconds to propagate to all replicas. For a search engine, this delay is acceptable.

5.3 Storage Strategy: NAS vs. Object Storage

We implemented a simulated Network Attached Storage (NAS) via Docker Volumes. **Alternative Considered:** Passing file content as JMS Message payload. **Reasoning:** Passing 5MB book files through ActiveMQ would saturate the Broker's heap memory and network bandwidth. The "Claim Check" pattern (sending a reference/path to the data, not the data itself) is a superior design pattern for large payloads.

6 Benchmarks and Results

6.1 Methodology

To validate the system, we executed the testing procedure outlined in Section 4.4 of the guidelines, utilizing the logs generated by the distributed nodes.

- **Tools:** Docker Stats for resource monitoring and custom timestamp logging to calculate ingestion throughput.
- **Dataset:** Books from Project Gutenberg (IDs 1000-12000).
- **Environment:**
  - **CPU:** Intel Core i5-3470 (3.2GHz)
  - **RAM:** 16 GB
  - **Network:** Single IP egress (NAT).

6.2 Test 1: Scalability and Node Performance

We measured the performance of 12 Crawler nodes running concurrently. The nodes were analyzed in groups of three to identify performance degradation points.

Node Group	Status	Time/Doc	CPU (Est.)	Observation
Crawlers 1-3	**Optimal**	2.45 s	Medium	Stable ingestion ($\approx$ 0.45 docs/s)
Crawlers 4-6	Unstable	>12.0 s	Low	Intermittent HTTP 429 Errors
Crawlers 7-9	Critical	>25.0 s	Idle	Blocked by Ext. API (Wait times)
Crawlers 10-12	Critical	>15.0 s	Idle	Broker Connection Timeouts

Table 1: Performance breakdown by Crawler groups showing degradation at scale.

6.2.1 Analysis of Results

Contrary to theoretical expectations of linear scalability, our results demonstrate the effects of **external resource contention**.

1. **Optimal Range (1-3 Nodes):** The first group achieved a high ingestion rate with an average processing time of 2.45 seconds per document. This represents the optimal capacity for a single IP address querying the Gutendex API.
2. **Diminishing Returns (4+ Nodes):** As we scaled beyond 4 nodes, the system encountered the "Thundering Herd" problem against the external API. The logs reveal frequent `HTTP 429 (Too Many Requests)` errors.

3. **Bottleneck Identification:** The exponential backoff strategy (waiting 10s, 20s, 40s) successfully prevented the system from crashing but significantly increased the average time per document. Additionally, the high concurrency saturated the ActiveMQ Broker connections, leading to publication errors in the highest-numbered nodes.

Conclusion: For this specific hardware and network configuration, the optimal cluster size for the Ingestion Layer is **3 to 4 nodes**. Scaling beyond this introduces external blocking latencies rather than throughput gains.

6.3 Test 2: Fault Tolerance

To test the resilience of the cluster, we performed a chaos engineering test during active indexing.

Scenario: While the system was indexing, we forcibly stopped the container `search-node-1` using the Docker CLI.

Observation and Recovery Timeline:

1. **Failure Detection ($T + 0s$):** Nginx encountered a connection refusal, returning a momentary `502 Bad Gateway` for requests routed specifically to that node.

2. **Health Check ($T + 2s$):** Nginx's passive health check marked the upstream server as "unhealthy" and removed it from the rotation.

3. **Cluster Rebalancing ($T + 3s$):** Hazelcast detected the member loss via heartbeat timeout. It promoted the backup partitions (held on Node 2 and Node 3) to primary status.

4. **Steady State ($T + 4s$):** Service was fully restored. Searches for terms that were originally stored on Node 1 continued to succeed, served by the backup replicas on Node 2. No data loss occurred.

7 Conclusion

7.1 Summary of Achievements

The completion of Stage 3 marks a successful transition from a modular application to a fully distributed, resilient search engine cluster. We have engineered a system that satisfies the rigorous demands of modern Big Data processing, effectively addressing the challenges of volume and velocity through architectural design rather than raw hardware power.

One of the primary achievements of this implementation is the demonstration of true horizontal scalability. By orchestrating the system with Docker Compose, we proved that the ingestion throughput can be increased simply by deploying additional worker containers. Our benchmarks validated that the system logic holds up under stress, although we also identified that in real-world distributed crawling, external API rate limits often become the bottleneck before internal resources are exhausted.

Furthermore, the system exhibits robust fault tolerance, a critical requirement for distributed computing. The integration of Hazelcast as a distributed In-Memory Data Grid ensures that the inverted index is partitioned and replicated across the cluster. As demonstrated in our failure scenarios, the loss of a Search Node triggers an immediate and automatic failover to backup replicas without service interruption. Similarly, the use of Apache ActiveMQ as a persistent message broker decouples the ingestion layer from the processing layer, guaranteeing that no book data is lost even if processing nodes are temporarily offline.

Finally, the inclusion of Nginx as a reverse proxy provides a unified and transparent entry point for the user. It abstracts the underlying complexity of the dynamic cluster, effectively balancing the load across healthy nodes and isolating the client from internal infrastructure changes. Collectively, these components form a production-grade architecture where scalability and reliability are foundational properties of the system.