Stage 3
Building a Cluster Architecture

Search Engine Project

Big Data
Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

1 Introduction

The goal of Stage 3 is to evolve the modular architecture developed in Stage 2 into a distributed cluster in which scalability and fault tolerance emerge as inherent properties of the system design. In this phase, students will design and deploy a cluster of cooperating nodes capable of executing ingestion, indexing, and search operations. The system must operate reliably under high load, tolerate individual node failures, and maintain consistent search results across replicas. This stage introduces concepts such as *horizontal scaling*, *replication*, and *fault tolerance*, which are essential in large-scale data processing systems.

While Stage 2 focused on service modularization and interaction through RESTful APIs, Stage 3 shifts the focus toward *distributed execution and data consistency*. The objective is no longer just to make services communicate, but to make them cooperate as part of a resilient, cluster-based ecosystem in which no single node or service is indispensable.

By the end of this stage, each group will have implemented a search engine prototype capable of:

- Distributing workloads for ingestion, indexing, and querying across multiple nodes.
- Replicating data and index structures in memory to ensure fault tolerance and fast recovery.
- Maintaining consistent, low-latency query performance under increasing load.

The resulting system should exhibit the defining characteristics of a distributed and resilient architecture: a shared in-memory index replicated across multiple nodes, a distributed datalake ensuring data durability through cross-node replication, and a coordination mechanism that synchronizes ingestion, indexing, and querying tasks. The cluster should be able to scale horizontally by adding new nodes, recover gracefully from partial failures, and preserve consistent query results despite concurrent updates or node restarts. These properties collectively ensure that the system operates as a single, fault-tolerant search platform capable of sustaining high performance under continuous growth.

2 System Architecture Overview

To achieve these objectives, the system architecture integrates several distributed processes that operate concurrently within the cluster:

- *Crawling:* document retrieval is executed in parallel by multiple crawlers, each responsible for a subset of sources. Crawlers must coordinate to avoid duplicate downloads and replicate their data across the distributed datalake for durability.
- *Indexing:* indexing operations update a shared in-memory *inverted index*, which is sharded and replicated across nodes to ensure both scalability and fault tolerance.
- *Searching:* queries are routed through a *load balancer*, which distributes traffic among available search instances and maintains service continuity even if individual nodes fail.

Through experimentation, benchmarking, and fault-injection tests, students will validate their architecture's resilience and scalability. Stage 3 marks a key step toward building large-scale distributed systems—where data, computation, and reliability must all be managed collectively across a dynamic cluster.

2.1 System Topology

The system must operate as a *cluster* composed of multiple interconnected *nodes*. A cluster represents the physical environment in which all distributed services cooperate to perform ingestion, indexing, and search operations. Each node contributes computing resources—processing power, memory, storage, and network connectivity—and communicates with other nodes to exchange data and coordinate distributed tasks. Together, these nodes form a unified and resilient system where computation and data are shared across the network, eliminating single points of failure. Within this topology, the failure of a single node does not compromise the overall availability of the system. New nodes can be added to increase capacity, and existing ones can be restarted or replaced without interrupting service continuity. This property of dynamic composition enables both horizontal scalability and fault tolerance, two central goals of Stage 3.

Two topological configurations are possible in this architecture. In a *single-service node* configuration, each node runs a single dedicated service—for example, only a crawler, only an indexer, or only a search instance. This setup provides strong isolation and simplifies fault containment, allowing each service type to scale independently. However, it also increases the total number of nodes and the amount of inter-node communication.
Alternatively, a *multi-service node* configuration allows several related services to coexist on the same machine—for instance, a crawler paired with its local datalake partition, or an indexer combined with a search instance that shares its in-memory index. This approach improves data locality and can reduce latency, but it couples service lifecycles and may complicate recovery when failures occur.
Each development team must choose and justify which topology they adopt. The decision should reflect a clear understanding of trade-offs among scalability, resource utilization, communication

overhead, and fault tolerance. The chosen topology will determine how effectively the system balances independence between services with efficiency in resource sharing across the cluster.

2.2 Architecture

Regardless of the chosen topology, data storage, indexing, and querying are distributed across multiple nodes, eliminating single points of failure and enabling seamless scaling under high load. Figure 1 illustrates the main components of the system.

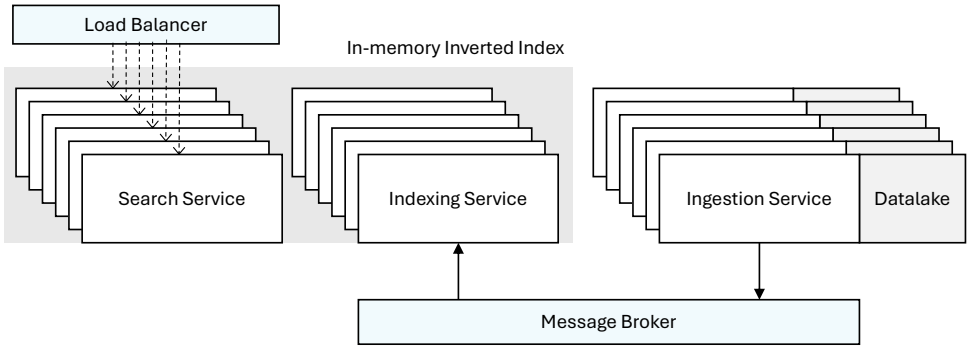

Figure 1: Cluster Architecture for Search Engine Project

- *Distributed Datalake:* instead of relying on a centralized storage system, the datalake is partitioned across crawler nodes. Each partition stores the documents owned by its corresponding crawler, providing data locality and parallel access. To ensure resilience, the datalake should support *cross-node replication*. When a new document is added to a local partition, it is automatically replicated to one or more peer nodes according to a configurable replication factor (typically $R = 2$ or $R = 3$). This guarantees that, if a crawler node fails or becomes temporarily unavailable, its documents remain accessible for indexing and global reindexing operations.

- *Message Broker:* the message broker enables event-driven communication between microservices by ensuring reliable delivery of messages, even under transient network or node failures. Crawlers publish `indexing request` events once a document has been successfully downloaded and replicated, while indexers subscribe to these topics/queues to consume and process new documents. To prevent duplicate indexing, indexers must handle messages idempotently using unique document identifiers or content hashes. This component can be implemented using *Apache ActiveMQ*.

- *In-Memory Inverted Index:* the global inverted index is maintained entirely in memory, distributed and replicated across nodes to achieve both scalability and fault tolerance. The index must follow a *replicated sharding strategy*, where each shard—responsible for a subset of terms—has at least one *primary* node and one or more *replicas*. Replicas continuously synchronize updates from their primary. If a node fails, a replica is automatically promoted to primary, ensuring that no part of the index is lost and that query responses remain complete. This architecture enables low-latency searches without disk I/O and can be implemented using distributed in-memory platforms such as Hazelcast.

- *Load Balancer:* the load balancer functions as the system's front-facing gateway, distributing incoming search requests among the available *Search Service* nodes. Its primary role is to optimize resource usage, prevent node overload, and maintain continuous service availability. Several routing algorithms can be employed, such as *round-robin* (equal request distribution) or *least-connections* (routing requests to the least busy node). The load balancer performs active *health checks* by periodically probing the readiness and liveness endpoints of each node to detect failures or degraded performance. When a node becomes unavailable, it is automatically removed from the active pool, and traffic is rerouted to healthy replicas without interrupting ongoing sessions. This component can be implemented using *Nginx*, which provides built-in load balancing, reverse proxying, caching, and dynamic scaling capabilities, allowing new search nodes to join or leave the cluster with minimal disruption.

2.3 Control Flow

The overall control flow of the system follows an asynchronous, event-driven model that connects ingestion, indexing, and querying across distributed services. Each service interacts through well-defined interfaces and communicates via the message broker and the in-memory index cluster. The process unfolds as follows:

1. *Crawling:* each *Crawler Service* retrieves a distinct subset of documents from external sources. Once a document is downloaded, it is stored in the service's local partition of the distributed datalake. To ensure durability, the document is replicated to a predefined number of peer datalake partitions according to the replication factor (R). After successful replication, the crawler publishes an `indexing request` message to the broker, signaling that the document is ready for indexing.

2. *Indexing:* the *Indexing Service* subscribes to the broker's indexing topic and processes incoming events asynchronously. Upon receiving a message, it retrieves the corresponding document from the datalake, tokenizes its content, and updates the relevant shard of the in-memory inverted index. Because the index resides entirely in memory and is distributed across the cluster, indexing operations are extremely fast and the updates are reflected almost immediately in subsequent search queries.

3. *Searching:* the *Search Service* continuously handles user queries routed through the *Load Balancer*. Each query is directed to an available search instance based on the configured balancing policy. The search service consults its local in-memory shard to obtain the list of matching document identifiers. If a query term belongs to a shard managed by another service, the search component performs a distributed lookup to gather partial results. These results are then merged, ranked, and returned to the client, ensuring completeness and consistency across the distributed index.

This event-driven control flow allows the system to operate continuously and efficiently, with ingestion, indexing, and querying processes decoupled yet synchronized through asynchronous communication and shared in-memory data structures.

This distributed flow ensures high throughput and resilience: crawlers, indexers, and search services can scale independently, while the broker and in-memory index cluster provide coordination and consistency across the system.

3 Implementation

3.1 In-Memory Inverted Index

Hazelcast is an in-memory data grid that provides distributed data structures and coordination primitives suitable for building scalable and fault-tolerant systems. In a Hazelcast cluster, there are two main types of participants: *members* (or *instances*) and *clients*. A *Hazelcast instance* is a full cluster member that stores data, manages partitions, and participates in replication and failover. Each instance contributes memory and computational resources to the shared data grid, forming the core of the distributed system. In contrast, a *Hazelcast client* does not store data itself; instead, it connects remotely to one or more cluster members to access distributed data structures (such as maps, multimaps, or queues). Clients are typically used by lightweight services or external applications that need to interact with the cluster without joining it as full data-bearing nodes.

Cluster formation relies on a dynamic discovery mechanism based on the UDP multicast protocol. When a new Hazelcast instance starts, it broadcasts a discovery packet over the network to a predefined multicast group and port. Existing members listening on the same group respond with their cluster information, allowing the new instance to join automatically without any manual configuration. This mechanism enables zero-configuration clustering in local or containerized environments, where IP addresses may change frequently.

Hazelcast provides a wide range of distributed data structures designed to simplify the development of scalable and fault-tolerant applications. These structures behave similarly to standard Java collections but are transparently distributed across all members of the cluster. The platform offers several types of collections, including distributed `Map`, `MultiMap`, `Queue`, `Set`, and `List`, as well as concurrency primitives such as `Lock`, `Semaphore`, and `AtomicLong`. Each of these abstractions is automatically partitioned and replicated, ensuring that data remains consistent and accessible even in the event of node failures.
Among these distributed data structures, the `MultiMap` is particularly suitable for the search engine architecture because it allows multiple values to be associated with a single key. This feature is essential for representing the inverted index, where each term (`word`) corresponds to a collection of document identifiers (`document-id`) that contain that term.
The `MultiMap` internally manages concurrency and replication, enabling simultaneous updates from multiple *Indexing Services* without data corruption. When a new document is indexed, the corresponding word–document mappings are inserted into the distributed `MultiMap` and become immediately visible to all *Search Services*. This design allows the cluster to maintain a consistent, memory-resident global index that supports near real-time search, achieving both scalability and fault tolerance through Hazelcast's partitioning and replication mechanisms.

A typical initialization and usage example in Java is shown below:

```
import com.hazelcast.core.*;
import com.hazelcast.config.*;
import java.util.Collection;

public class InvertedIndex {
    public static void main(String[] args) {
        Config config = new Config();
        config.setClusterName("search-cluster");
        config.getNetworkConfig().setPort(5701).setPortAutoIncrement(true);

        HazelcastInstance hz = Hazelcast.newHazelcastInstance(config);
        MultiMap<String, String> invertedIndex = hz.getMultiMap("inverted-index");

        invertedIndex.put("distributed", "doc_001");
        invertedIndex.put("distributed", "doc_005");
        invertedIndex.put("systems", "doc_002");

        Collection<String> docs = invertedIndex.get("distributed");
        System.out.println("Documents containing 'distributed': " + docs);

        hz.shutdown();
    }
}
```

This example creates a Hazelcast cluster (using default discovery mechanisms) and stores multiple document identifiers for each indexed term. The `MultiMap` distributes entries across all active nodes in the cluster, automatically handling partitioning, replication, and fault recovery. When new nodes join, partitions are rebalanced transparently; when a node fails, its replicas automatically take over.

Distributed Consistency and Concurrency The `MultiMap` provides strong consistency guarantees within a partition and eventual consistency across replicas. To ensure safe concurrent writes from multiple indexers, updates can be wrapped in `FencedLock` operations:

```
FencedLock lock = hz.getCPSubsystem().getLock("lock:inverted-index:distributed");
lock.lock();
try {
    invertedIndex.put("distributed", "doc_010");
} finally {
    lock.unlock();
}
```

This approach ensures that no two services modify the same key concurrently in a conflicting way. Several configuration aspects must be considered when deploying Hazelcast in this context:

- *Replication Factor:* Hazelcast maintains one or more replicas for each partition. This can be configured under the `map-config` section in `hazelcast.xml` or programmatically:

  ```
  MapConfig mapCfg = new MapConfig("inverted-index")
      .setBackupCount(2)        // number of synchronous replicas
      .setAsyncBackupCount(1);  // asynchronous replicas
  config.addMapConfig(mapCfg);
  ```

A typical configuration is one primary partition and two replicas, ensuring that the data remains available even if multiple nodes fail.

- *Eviction Policy:* Since the inverted index is memory-resident, eviction and expiration policies must be tuned according to available memory. Hazelcast supports `LRU`, `LFU`, and `NONE`.
- *Near Cache:* Read-heavy nodes, such as search instances, can enable a `Near Cache` to locally store frequently accessed entries, reducing latency for repeated queries.

3.2 Load Balancer

The load balancer is responsible for distributing incoming search requests among the available *Search Service* instances in the cluster. Its main goals are to prevent overload on any single service, ensure continuous availability, and enable seamless horizontal scaling when new instances are added. A practical way to implement this component is by using *Nginx* as a reverse proxy and load balancer. Nginx can forward client requests to multiple backend servers, perform health checks, and apply dynamic routing policies. It operates at the HTTP layer, making it well suited for REST-based communication between clients and search services.

A basic Nginx configuration for the load balancer may look as follows:

```
http {
    upstream search_cluster {
        least_conn;
        server 10.0.0.11:8080 max_fails=3 fail_timeout=30s;
        server 10.0.0.12:8080 max_fails=3 fail_timeout=30s;
        server 10.0.0.13:8080 max_fails=3 fail_timeout=30s;
    }

    server {
        listen 80;
        location /search/ {
            proxy_pass http://search_cluster;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_connect_timeout 5s;
            proxy_read_timeout 30s;
        }
    }
}
```

In this example, the `upstream` block defines the cluster of backend search services. The directive `least_conn` enables the *least connections* policy, which directs each new request to the instance currently handling the fewest active sessions. This strategy helps balance uneven query loads more effectively than simple round-robin scheduling.
Nginx periodically checks the availability of backend services based on connection success. When a service fails, Nginx automatically removes it from the active pool until it becomes healthy again. This ensures that user requests are always routed to functional instances without interruption. Furthermore, new search instances can be added to the upstream pool dynamically by updating

the configuration or integrating with container orchestration systems such as Docker Swarm or Kubernetes.

Docker-based Deployment: To deploy this configuration using Docker, the official `nginx` image from Docker Hub can be used. The configuration file (for example, `nginx.conf`) should be placed in the same directory as the `docker-compose.yml` file or mounted as a volume inside the container. A minimal setup is shown below:

```
# docker-compose.yml
version: '3'
services:
  nginx:
    image: nginx:latest
    container_name: search_load_balancer
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    networks:
      - search_net

  search1:
    image: search_service:latest
    container_name: search1
    expose:
      - "8080"
    networks:
      - netname

  search2:
    image: search_service:latest
    container_name: search2
    expose:
      - "8080"
    networks:
      - netname

  search3:
    image: search_service:latest
    container_name: search3
    expose:
      - "8080"
    networks:
      - netname

networks:
  netname:
    driver: bridge
```

In this configuration, the `nginx` container acts as the system's entry point, listening on port80 for incoming HTTP requests. Each *Search Service* container exposes port8080 internally within the shared Docker network named `netname`, allowing Nginx to route traffic directly to the available backend services. The Nginx configuration file is mounted as a read-only volume inside the container to ensure consistency and prevent accidental modification during runtime.
To launch the load balancer and backend services, execute:

```
docker compose up -d
```

Nginx will automatically start routing incoming client requests to the available search service instances. When new services are added, they can be registered by modifying the `upstream` section of `nginx.conf` and reloading Nginx using:

```
docker exec -it search_load_balancer nginx -s reload
```

3.3 Message Broker

The message broker acts as the asynchronous communication backbone of the system, enabling decoupled coordination between the *Crawler Services*, *Indexing Services*, and other components. It ensures reliable message delivery, event-driven execution, and resilience to temporary service failures. In this stage, the broker is used to propagate events such as `document.ingested`, `document.indexed`, and `reindex.request`, allowing the system to maintain consistent workflows without direct service coupling.
A practical implementation can be achieved using *Apache ActiveMQ*, a lightweight, open-source message broker that supports multiple protocols and provides persistence, delivery guarantees, and built-in fault tolerance. ActiveMQ provides delivery guarantees such as *at-least-once* messaging, ensuring that no event is lost even if consumers are temporarily unavailable. Persistent queues can be enabled for durability, while *idempotent consumers* ensure safe processing of duplicate events.

- Each *Crawler Service* publishes messages to a topic or queue when a new document is downloaded and replicated across the datalake.
- Each *Indexing Service* subscribes to those topics, consumes messages, and performs in-memory index updates accordingly.
- The *Controller or Reindex Manager* can publish `reindex.request` events to trigger cluster-wide reindex operations.

Basic Docker Deployment: The simplest way to deploy ActiveMQ is through the official Docker image. The following `docker-compose.yml` file launches a broker instance accessible to all services in the cluster:

```
version: '3'
services:
  activemq:
    image: rmohr/activemq:latest
    container_name: activemq
    restart: always
    ports:
      - "61616:61616"  # JMS protocol port
      - "8161:8161"    # Web console (http://localhost:8161)
    environment:
      - ACTIVEMQ_ADMIN_LOGIN=admin
      - ACTIVEMQ_ADMIN_PASSWORD=admin
    networks:
      - search_net

  indexer:
```

```
    image: indexing_service:latest
    depends_on:
      - activemq
    networks:
      - search_net

  crawler:
    image: crawler_service:latest
    depends_on:
      - activemq
    networks:
      - search_net

networks:
  search_net:
    driver: bridge
```

This configuration exposes two ports:

- `61616`: the JMS/TCP endpoint used by services for message exchange.
- `8161`: the web management console, available at `http://localhost:8161` (default credentials `admin/admin`).

Producer and Consumer Example: Each service can publish or consume messages through the Java Message Service (JMS) API. The following minimal example illustrates a producer and a consumer connecting to ActiveMQ:

```
import jakarta.jms.*;
import org.apache.activemq.ActiveMQConnectionFactory;

public class BrokerExample {

    public static void main(String[] args) throws Exception {
        ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616");
        Connection connection = factory.createConnection();
        connection.start();

        Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination queue = session.createQueue("books.ingested");

        // --- Producer ---
        MessageProducer producer = session.createProducer(queue);
        TextMessage message = session.createTextMessage("{ \"bookId\": \"123\", \"status\": \"READY\" }");
        producer.send(message);
        System.out.println("Message sent to broker.");

        // --- Consumer ---
        MessageConsumer consumer = session.createConsumer(queue);
        Message received = consumer.receive(1000);
        if (received instanceof TextMessage msg) {
            System.out.println("Received message: " + msg.getText());
        }

        session.close();
        connection.close();
    }
}
```

4 Benchmarking Methodology

The benchmarking phase is designed to evaluate the performance, scalability, and fault tolerance of the distributed search engine architecture developed in Stage 3. Experiments must quantify how effectively the system handles increasing workloads, how it reacts to node failures, and how data replication and synchronization affect latency and throughput. The benchmarks should be reproducible and supported by quantitative measurements obtained under controlled conditions.

4.1 Experimental Setup

All experiments must be executed within a controlled environment where hardware and software parameters are clearly documented. Each group should specify:

- The number of nodes in the cluster and their configuration (CPU cores, RAM, and network bandwidth).
- The version of Docker, Java, and any middleware used (Nginx, Hazelcast, ActiveMQ).
- The total dataset size and number of documents used for ingestion and search tests.
- The replication factor (R) configured for the datalake and the in-memory inverted index.

Before running benchmarks, all services must be started and synchronized, ensuring that the datalake, index, and broker queues are in a consistent state. The environment should also be free of background processes that could distort performance measurements.

4.2 Benchmark Objectives

Three core properties must be tested during this stage:

1. *Scalability:* measure how ingestion, indexing, and query throughput evolve as the number of active services (or nodes) increases. This demonstrates the system's ability to scale horizontally.
2. *Latency and responsiveness:* evaluate average and peak response times for search queries and indexing operations under concurrent load. The objective is to verify that performance remains stable even under high throughput.
3. *Fault tolerance:* observe the system's behavior when one or more services fail. The benchmark should measure recovery time, data consistency, and service continuity during and after the failure.

4.3 Measurement Metrics

To ensure comparability across implementations, the following metrics must be collected and reported:

- *Ingestion rate (docs/s):* number of documents successfully processed by crawlers and published to the broker per second.
- *Indexing throughput (tokens/s):* number of tokens or word–document pairs inserted into the inverted index per second.
- *Query latency (ms):* average response time for search requests, measured under varying concurrency levels.
- *CPU and memory utilization (%):* resource consumption per node or container, measured using Docker statistics or monitoring tools.
- *Recovery time (s):* elapsed time required for the cluster to restore full availability after a simulated node failure.

4.4 Testing Procedure

Each group must design a repeatable testing procedure that includes the following phases:

1. *Baseline test:* execute the system with a minimal cluster configuration (one crawler, one indexer, one search node) to establish reference metrics for throughput and latency.
2. *Scaling test:* progressively add more nodes (or containers) for each service type and measure how performance metrics evolve. At least three configurations should be tested (e.g., 3, 6, and 9 nodes).
3. *Load test:* generate concurrent search requests using load-testing tools such as `Apache JMeter` or `wrk`. Measure response time distributions (average, 95th percentile, and maximum latency).
4. *Failure test:* simulate node crashes by stopping one or more containers during runtime. Record the system's recovery behavior, including message reprocessing and replica promotion within Hazelcast.

4.5 Data Collection and Visualization

All benchmark results must be recorded in tabular form and visualized using plots or charts. The following formats are recommended:

- Line plots showing ingestion or query throughput versus the number of active nodes.
- Box plots or bar charts displaying query latency under different load levels.
- Tables summarizing CPU, memory, and recovery times for each configuration.

# Nodes	*Ingestion Rate (docs/s)*	*Query Latency (ms)*	*CPU (%)*	*Recovery (s)*
3	1200	25	45	5
6	2300	28	52	5
9	3400	32	58	6

Table 1: Example performance results for scalability and fault-tolerance benchmarking.

4.6 Analysis and Interpretation

After collecting all measurements, students must analyze how the system scales and behaves under stress. Discussion should focus on:

- The point at which adding more nodes yields diminishing performance returns.
- Whether query latency remains bounded as the system scales.
- How effectively replication and partitioning prevent data loss during failures.
- The trade-offs between fault tolerance, consistency, and overall performance.

Each analysis must be supported by quantitative evidence (e.g., throughput curves, latency histograms, or failure recovery timelines). The final report should conclude with a summary of the system's observed scalability limits and suggestions for architectural improvements, such as introducing autoscaling mechanisms, distributed caching, or smarter partitioning strategies.

5 Project Delivery Guidelines

The final deliverable for this stage consists of a single *written report* in PDF format. This document serves as the official record of your work and must clearly reference the external repository where all source code, configuration files, and experimental results are hosted.
As in Stages 1 and 2, this is the only file that must be submitted to the *virtual campus platform*, and *only one member of the group* should upload it on behalf of the entire team. This ensures consistent version control and avoids duplicate submissions during evaluation. The written report should refer to the GitHub repository for all technical details.

5.1 GitHub Repository Requirements

The public *GitHub repository* associated with the project must follow the naming format:

`https://github.com/<group_name>/stage_3`

The repository must contain all materials related to Stage 3, including:

- The complete source code for all services and components (crawler, indexer, search engine, broker, and load balancer).
- The configuration and orchestration files required for deployment (Docker Compose, Nginx, Hazelcast, ActiveMQ, etc.).
- Benchmark datasets, logs, and performance results.
- A detailed `README.md`.

5.2 README Requirements

The `README.md` must provide a complete overview of the project and clear instructions for reproduction. It must include:

- A brief project description.
- Instructions for building and running all components using Docker Compose or Kubernetes.
- A summary of benchmark procedures and instructions to reproduce them.
- A *YouTube link* to the demonstration video.

The repository will be the main reference for evaluators to verify code quality, configuration accuracy, and experimental reproducibility.

5.3 Demonstration Video Requirements

Each group must record a short *demonstration video* (between 4 and 7 minutes, maximum 10) showing the system running in the laboratory under real conditions.The recording must be performed in the laboratory environment using the official hardware configuration to ensure reproducibility. Voice narration or on-screen explanations should accompany each stage of the demonstration. The video must be uploaded to *YouTube* as an *unlisted video* (not private), and its link must appear clearly in the `README.md` of the GitHub repository. The title of the video must follow the format:

```
[Stage 3] Search Engine Project - <Group Name> (ULPGC)
```

The video must show, at minimum:

- The deployment of the system using Docker (or the chosen orchestration tool).
- Real-time execution of ingestion, indexing, and search operations across multiple containers or nodes.
- A load test demonstrating horizontal scalability as additional instances are launched.
- A simulated node failure and the system's automatic recovery (e.g., replica promotion or request rerouting).
- Logs, dashboards, or monitoring outputs showing throughput, latency, and resource utilization.

5.4 Report Structure

The written report (`.pdf`) must include the following sections:

1. *Cover page:*
    - Course name and academic year.
    - Project title.

- Full names and student IDs of all group members.
- The chosen group name.
- The URL of the GitHub repository.

2. *Introduction and objectives:* Present the purpose of Stage 3 and the main goals achieved in transforming the modular system into a distributed, fault-tolerant, and horizontally scalable cluster architecture.

3. *System architecture:* Describe the overall cluster design, including topology, how the system distributes ingestion, indexing, and search operations. Explain the interaction between services through the broker, the distributed in-memory index, and the load balancer. Include diagrams that illustrate data flow, service coordination, and fault recovery mechanisms.

4. *Implementation details:* Summarize the configuration and deployment of each major component:

   - Distributed Datalake.
   - In-memory inverted index (Hazelcast).
   - Message broker (ActiveMQ).
   - Load balancer (Nginx).

   Include configuration snippets (Docker Compose, Nginx, or Hazelcast) and explain how services discover, synchronize, and replicate data across the cluster.

5. *Design decisions:* Justify architectural choices regarding:

   - The chosen deployment topology (single-service vs. multi-service nodes).
   - Data partitioning and replication strategy.
   - Load balancing and failure recovery mechanisms.
   - Synchronization methods (e.g., broker topics, distributed locks).

   Explain the rationale behind these decisions in terms of scalability, performance, and resilience.

6. *Benchmarks and results:* Present experimental evidence of the system's performance and fault tolerance:

   - Indexing throughput and latency as nodes are added.
   - Query latency and response consistency under concurrent load.
   - Behavior under node failure (graceful degradation and recovery time).
   - Resource utilization (CPU, RAM, network) across distributed components.

   Include visualizations (tables or charts) and discuss observed trade-offs between consistency, scalability, and fault tolerance. Reference specific scenes or timestamps from the YouTube video that illustrate the experimental results.

7. *Conclusions and future improvements:* Reflect on the overall behavior of the distributed system. Identify strengths, limitations, and opportunities for improvement—such as autoscaling, distributed caching, or multi-region replication.

6 Evaluation Criteria

The evaluation will consider both the distributed system implementation and the presentation of the architecture. Grading will be based on the following criteria:

- *Architecture, fault tolerance, and video demonstration* (20 %) – correctness and robustness of the distributed design, including replication strategies, cluster coordination, and recovery mechanisms under node failure. The YouTube demonstration video must clearly show the deployment, scaling behavior, and fault recovery process in the laboratory environment.
- *Implementation and code quality* (20 %) – structure, modularity, documentation, and reproducibility of the system, including clarity of configuration and deployment scripts.
- *Written report* (20 %) – clarity, organization, and completeness of the report; accuracy of explanations, visualizations, and analysis of experimental results.
- *Scalability, performance, and fault tolerance* (20 %) – ability of the system to scale horizontally across multiple nodes, maintain low latency under concurrent load, and demonstrate measurable performance and recovery behavior through benchmarking and fault-injection tests.
- *Individual oral defense* (20 %) – student's ability to explain their own contributions, justify technical decisions, and interpret observed results. The defense is an individual evaluation component. Students who choose not to perform it will not receive a grade for this part, and their final score will be based solely on the group deliverables.
  - Each student will have a maximum of *4 minutes* to present.
  - The defense should focus on the rationale behind key design or implementation decisions, and the interpretation of experimental or scalability results.
  - The schedule for oral defenses and laboratory demonstrations will be communicated in advance to all groups.