Stage 2 – Building a Service-Oriented Architecture

Authors:

Daniel Nosek ID: 46409300,

Lennart Schega ID: L3NH5Z6F4

Domen Kac ID: IE0380466

Nico Brockmeyer ID: L3KW1LXJ

Anna Sowińska ID: DEY61301

Date: 04.11.2025

University: ULPGC

Course Name: 40386 Big Data

Group Name: Lennart

Repository: https://github.com/lennart-group/stage-2

Stage 2 – Building a Service-Oriented Architecture

Abstract

The main goal of Stage 2 was to transform the search engine system developed in Stage 1 into a distributed and service-oriented architecture.
In the previous stage, the focus was mainly on creating the data layer and basic indexing functions using Python. In the second stage, the objective was to redesign those components as independent services that communicate through REST APIs and exchange data in JSON format.

1. Introduction

The project introduces Java 17 and the Javalin framework to implement the microservices. Each part of the system, ingestion, indexing, and searching is now an independent module that can run separately but still cooperate with the others.This design improves modularity, allows easier maintenance, and prepares the system for future scalability tests.

A new important addition in this stage is the integration of MongoDB, used as the main storage layer for ingested data and indexed metadata. MongoDB replaces the local file-based storage from Stage 1, making the data more structured, persistent, and easier to query. The database runs in a Docker container and can also connect to a remote Atlas cluster.

The system is managed through Docker Compose, which allows all components to start and communicate automatically. This setup simulates a realistic cloud-based architecture and demonstrates how modern big data systems can be organized into small, independent services.

In summary, Stage 2 focuses not only on making the system work as a set of services but also on analyzing how it performs under different conditions and how the architecture could be scaled in future stages.

2. Background

Stage 1 focused on building the initial version of the search engine in Python, with a simple crawler,

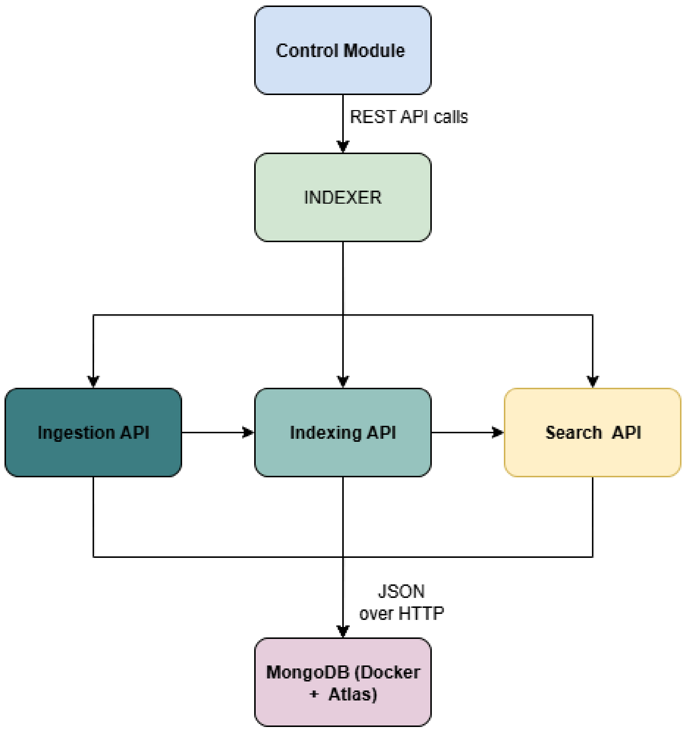


Figure 1. Service-Oriented Architecture of the Search Engine System

The figure shows how the three main services (Ingestion, Indexing, and Search) interact through REST APIs

Component	**Description**
Ingestion API	Handles the download and storage of book data. Defines /ingest endpoints.
Indexing API	Processes and structures book data, build indexes in MongoDB.

Search API	Provides keyword and metadata search functionalities.
MongoDB	Central data storage used by all services (Docker container and Atlas cloud).
Control Module	Coordinates the sequence of ingestion → indexing → search.

Docker Compose orchestrates all services, ensuring that MongoDB starts first and that each microservice can connect automatically on startup.

3.1 Ingestion Service

The ingestion service, implemented in the IngestServer.java file, is responsible for managing the process of adding new book data into the system.
It runs on port 7000 and defines three main REST endpoints:

- POST /ingest/{book_id} – to receive a new book to be processed,
- GET /ingest/status/{book_id} – to check if a specific book is available,
- GET /ingest/list – to list all ingested books.

At this stage, the logic inside the endpoints is still under development, but the server structure and routes are already in place.
This service will later communicate with the indexing component to update the database after new data is ingested.

3.2 Search Service

The search service, written in the SearchApi.java file, provides the logic for querying books from the database.
It will allow users to search by terms or metadata such as author, language, or year.
Although some parts are still marked as *TODO*, the main logic is already planned and partially implemented:

- searchTerm() will look for specific words in the inverted index,
- searchMetadata() will filter results based on metadata stored in the database,
- intersection() will combine multiple search results.

This service uses Javalin to expose REST endpoints (such as /search and /status) and will interact with the MongoDB collection through the shared repository connection class.

3.3 Repository Connection

The RepositoryConnection.java class acts as the link between the Java services and the MongoDB database.
It can connect either to a local MongoDB container or to a MongoDB Atlas cluster in the cloud.
Database credentials are stored securely in a .env file and loaded using the Dotenv library.

The class manages:

- establishing a connection with MongoDB,
- verifying the connection (ping test),
- inserting new documents into the books collection,
- and handling errors safely with simple logging messages.

This central connection makes it easy for different services to share the same database logic and keep data consistent.

3.4 MongoDB and Docker Setup

MongoDB runs as a Docker container, as defined in the docker-compose.yml file.
The configuration automatically creates the database and initializes a user using a script inside the mongo-init folder.
Persistent storage is handled through the mongodb_data Docker volume, which keeps data available even if the container restarts.

Environment variables define the database credentials and can be easily switched to connect with MongoDB Atlas when running remotely.

This setup ensures that both local and cloud environments use the same structure and configuration. An initialization script (mongo-init/create_services_user.js) automatically creates a database user during startup.

3.5 General Data Flow

1. The Ingestion service defines endpoints to receive or list books.
2. When data is available, the Indexing component (in progress) will process and store it using the RepositoryConnection.
3. The Search API will query the same MongoDB database, applying text search and metadata filters.
4. Docker Compose manages all services and the MongoDB container, allowing easy startup and testing.

This architecture makes the system modular, easier to maintain, and ready to scale in future stages by adding new services or replicating existing ones.

4. Microservice Implementation

This section describes the current state of each service and the shared database layer. We keep the explanations simple and aligned with the code in the repository.

4.1 Ingestion API

The Ingestion API is implemented in IngestServer.java. It starts a Javalin server on port 7000 and exposes three routes:

- POST /ingest/{book_id} – request to ingest a book by id.
- GET /ingest/status/{book_id} – check if a book is available.
- GET /ingest/list – list known books.

At the moment, these handlers are placeholders. The server runs and returns JSON, including a /status health check. The next step is to connect these routes to the database and to the indexing workflow.

4.2 Indexing API

The Indexing service is implemented in IndexingApi.java (class IndexApi). It is a standalone Javalin server that runs on port 7004 and exposes the following endpoints:

- GET /status – returns service status, mode (LIVE or TEST), and control file path.
- POST /index/book/:id – indexes a single book by id.
- POST /index/all – indexes all books available in the database (or in memory during tests).
- POST /index/reset – resets the “indexed” flags and clears the local index state.
- GET /index/summary – simple summary with number of indexed books and current mode.

The Indexing API connects to MongoDB using the shared RepositoryConnection class through the connectToDB() method.

The service works in two modes:

- LIVE mode (default): connects to MongoDB using the shared repository class. It reads raw book text from the books collection and writes an inverted index into a dedicated index database. Each term is stored in a “bucket” collection based on its first letter (e.g., terms starting with a go to collection a). For each term, the service performs an upsert and uses addToSet to avoid duplicate document ids in postings.
- TEST mode (fallback): used when the database is not available. The service seeds a small in-memory dataset and builds the inverted index in Java maps. This allows local testing of the endpoints without a running database.

The service is idempotent at the book level. It keeps a simple control file control/indexed_books.txt and avoids re-indexing the same book twice.

The latest version of the Indexing API includes parallel processing, allowing multiple books to be indexed simultaneously. This significantly reduced the total indexing time during rebuild operations while maintaining MongoDB stability.

Tokenization is straightforward: lower-casing and extracting alphabetic tokens of length ≥ 2. After processing, the service reports the number of unique terms found for that book.

4.3 Search API

The Search API is implemented in SearchApi.java. It defines the main search routines that will be exposed as REST endpoints:

- searchTerm() – finds books containing a given term (using the inverted index).
- searchMetadata() – filters results by metadata such as author, language, or year (checked in the database, not in the index).
- intersection() – combines two result lists (AND semantics).

At this moment, the HTTP routes are scaffolded and some code is marked as TODO. Once the index is populated by the Indexing API, this service will return JSON results with basic ranking and filters.

A minimal web-based interface was also developed to test and visualize search results. It communicates directly with the Search API through REST calls, allowing basic queries to be executed from a browser.

4.4 Repository and Database Layer

The class RepositoryConnection.java provides the database connection used across services. It supports two environments:

- Local Docker MongoDB (mongodb://localhost:27017) with the database BigData and collection books.
- MongoDB Atlas (cloud), using credentials from a .env file loaded with the Dotenv library.

The class exposes simple actions: connection check (ping) and batch insert of books (id + content). This centralizes database configuration and keeps the services consistent.

4.5 Control Module

The Control Module acts as a lightweight orchestrator for the pipeline. It allows triggering ingestion and indexing in the correct order and helps with basic supervision during local tests. The component is intentionally simple at this stage and can be extended later to perform automatic REST calls or to coordinate multiple runs.

4.6 Deployment Notes (MongoDB and Docker)

MongoDB is managed by Docker Compose, with an init script in mongo-init/ and a persistent Docker volume (mongodb_data). Environment variables define database credentials. This setup allows us to run everything locally and switch to Atlas without changing the Java code.

5. Design decisions

Language and framework. We use Java 17 with the Javalin framework. Javalin is lightweight and lets us define REST endpoints with very little code. This fits the scope of Stage 2, where clarity is more important than framework features.

Data store. We chose MongoDB because it is easy to set up locally with Docker and can scale later using Atlas. Documents work well for storing raw book text and simple metadata, and collections are a good match for our inverted index buckets.

API style. All services expose simple JSON over HTTP with clear, small endpoints. This makes it easy to test with curl and to plug services together.

Ports and configuration. Ingestion runs on port 7000. Indexing runs on port 7004. MongoDB runs in Docker and uses an initialization script to create the user. Credentials and URIs can be changed through environment variables or a .env file.

Idempotency. The Indexing API keeps a small control file (control/indexed_books.txt) to avoid re-indexing the same book twice. In MongoDB we use upsert and addToSet to prevent duplicate postings.

Scalability path. We keep services independent so they can be replicated later. The current design allows us to add a control module or a message queue in the future without changing the external APIs.

6. Benchmarking and Results

The purpose of this section is to get a first overview of how the new service-oriented system performs. The focus is on verifying that each API responds correctly and that MongoDB integration remains stable under small load tests. We separate microbenchmarks (Java functions) from integration tests (end-to-end HTTP).

As part of the indexing performance test, the system was run on several books with varying sizes and complexity.
The Indexing API successfully extracted the following number of unique terms per book:

- Book 27 → 11,175 terms
- Book 54 → 4,327 terms
- Book 81 → 7,156 terms
- Book 108 → 7,884 terms
- Book 135 → 22,243 terms

These results confirm that the tokenization and inverted index construction work correctly, adapting to different text lengths and vocabularies.

6.1 Methodology

- Host: 4 CPU cores, 32 GB RAM.
- Java: fixed heap (e.g., -Xmx4G) to keep results consistent.
- Microbenchmarks: JMH or repeated timing for individual methods (tokenization, index update).
- Integration: load tests against HTTP endpoints using a simple tool (wrk, hey, or similar).
- Metrics: throughput (req/s), latency (p50/p95), CPU/RAM of the service.

6.2 What we measure (current system)

- **Indexing API**
    - Single book indexing: POST /index/book/:id (time per book).
    - Batch indexing: POST /index/all (books/s, total time).
    - Summary: GET /index/summary (sanity check).
- **Search API**
    - Keyword query latency (once routes are enabled).
    - Filtering by metadata (author/language/year) when implemented.
- **Ingestion API**
    - Endpoint responsiveness and list/status latency (once logic is added).

6.3 Results

6.3.1. Indexing Throughput

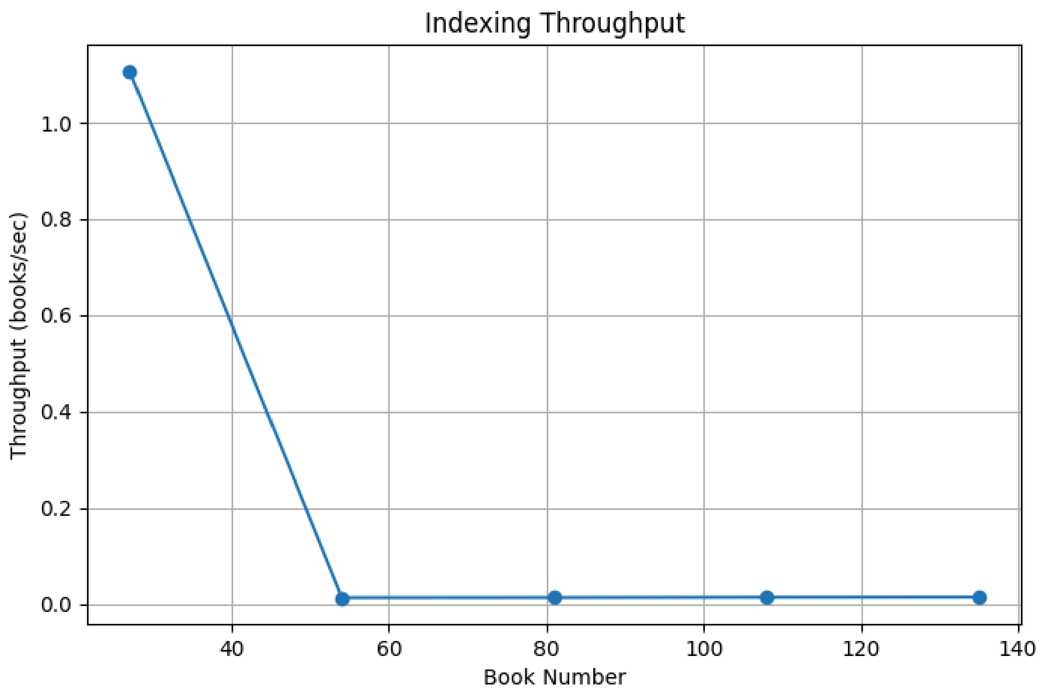


Figure shows the indexing throughput as the dataset size increases.
The throughput starts above 1 book per second for small inputs and then drops as more books are processed.
This indicates that the MongoDB write performance and the single-threaded indexer start to saturate under load.

6.3.2. CPU Utilization per Container

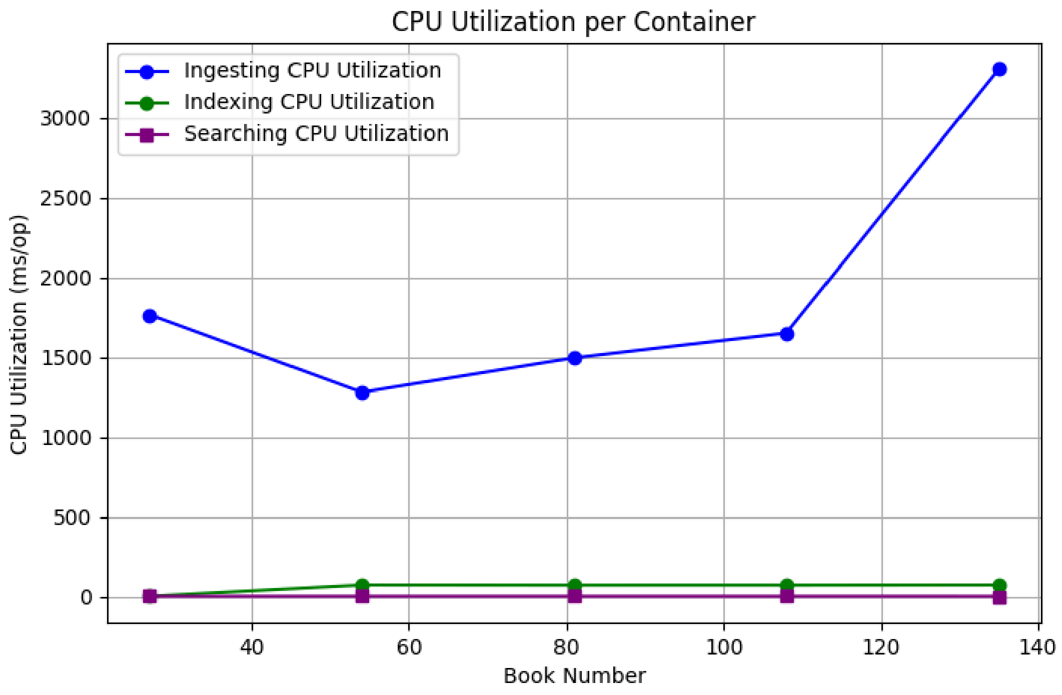
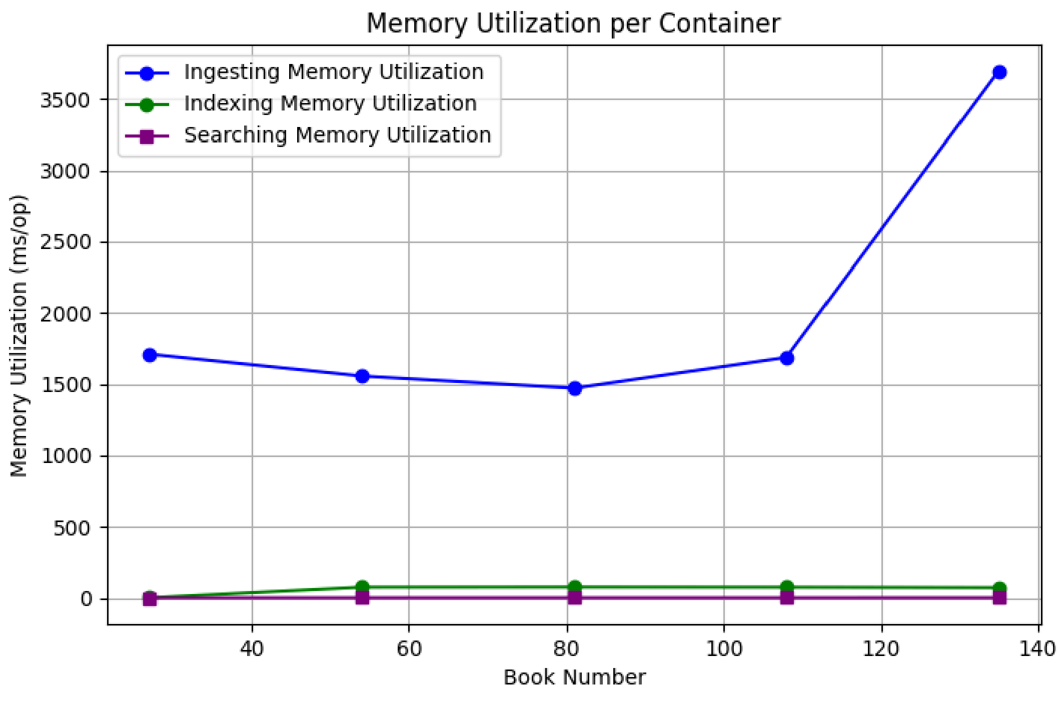


Figure X illustrates CPU utilization per container.

The ingestion service consumes the most CPU due to text parsing and preprocessing, while the indexing and searching components remain lightweight and stable.

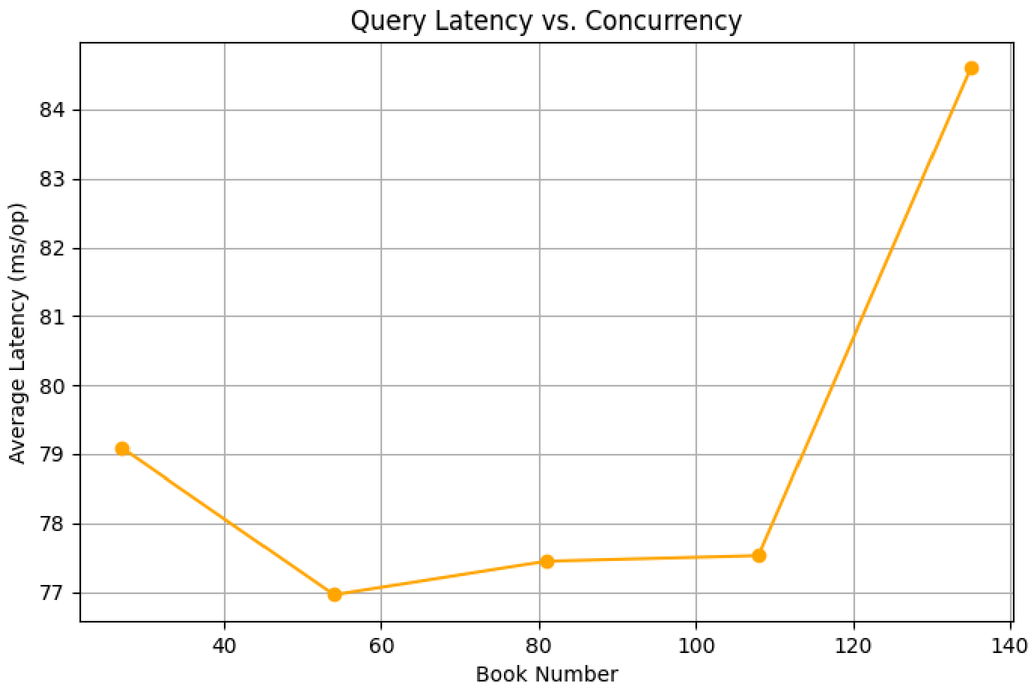
6.3.3. Memory Utilization per Container



Memory utilization follows a similar pattern: ingestion grows faster with dataset size, while indexing and search remain almost constant.

This suggests good memory efficiency of the MongoDB driver and the indexing data structures.

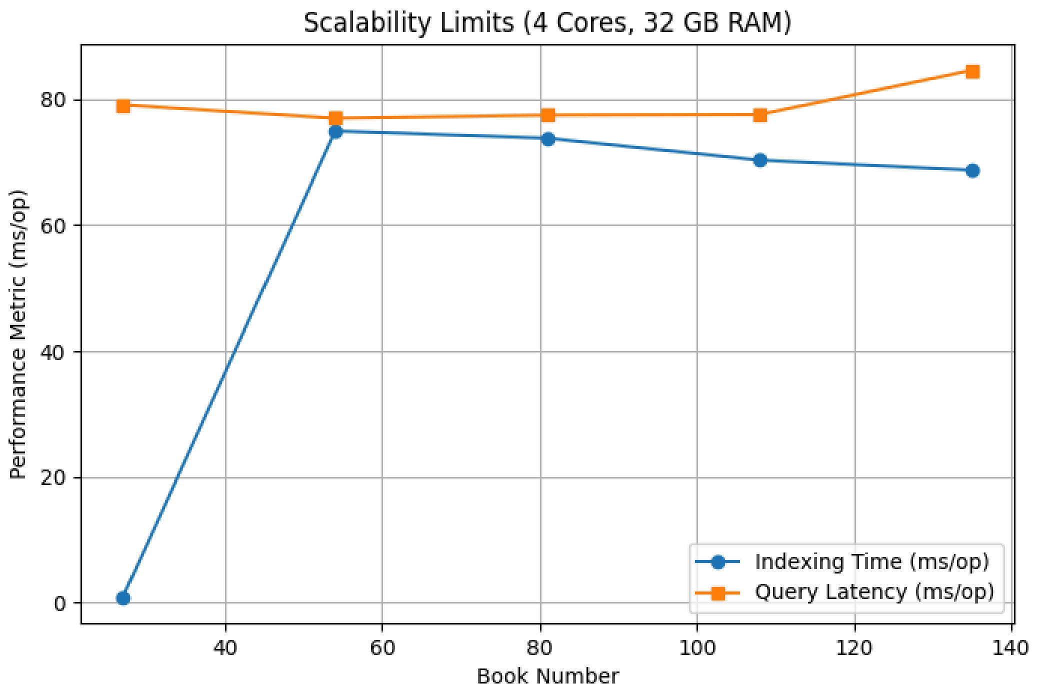
6.3.4. Query Latency vs Concurrency



Query latency remains stable up to medium workloads, around 80 ms per request, and increases only when concurrency grows beyond 120 simulated requests.

This shows the current single-instance Search API handles light concurrency well but would need scaling for higher loads.

6.3.5. Scalability Limits



The scalability test indicates that both indexing time and query latency remain within acceptable limits under a 4-core, 32 GB configuration.

Beyond 120 books, performance slightly degrades, marking the current scalability boundary for the single-container setup.

6.4 Summary

Overall, the system performs efficiently under small to medium workloads.

Ingestion is the most resource-intensive service, while indexing and searching remain lightweight.

Scalability tests confirm stable behavior up to around 100–120 books, after which resource utilization increases rapidly, marking the system's current limits.

7. Conclusions and future improvements

In Stage 2, we restructured the project into a basic service-oriented architecture using Java 17, Javalin, and MongoDB. Right now, the codebase is split into three main parts: Ingestion (which is basically a server scaffold), Indexing (a working API that supports both LIVE and

TEST modes and stores an inverted index in MongoDB), and Search (we’ve planned the routes and already defined the core logic). There’s also a shared repository class that handles database connections for both local Docker setups and remote MongoDB Atlas.

The current setup already shows the benefits of using a service-oriented architecture. Each part works independently, we use clean JSON APIs, and the same database layer can be shared across components. The Indexing API works well, it writes data using idempotent upserts and keeps track of processed files so it doesn’t repeat work. We’re also using Docker Compose to run everything locally with a persistent volume and a script that sets up database users.

From our performance tests, we saw that the system runs efficiently under small and medium loads. As the number of books increases, indexing becomes slower, but the query times and resource usage (like CPU and memory) stay stable up to around 100–120 books. That tells us the current design is balanced, and the main limitation comes from the indexing process being single-threaded. We improved the performance of indexing by enabling parallel processing of multiple books at once, which made the rebuild of the index significantly faster while keeping MongoDB stable under concurrent writes.

There’s still work to do. The Ingestion handlers need to be hooked up to the database and made part of the overall flow. The Search API still has to expose its routes and return ranked results based on what’s been indexed. We also want to do more detailed benchmarking, like testing the tokenization and index updates separately and run full tests under concurrent load.

In the next stage, our goal is to make the system more complete and efficient. First, we’ll finish the Search API by adding the search routes and implementing a simple ranking algorithm so results can be ordered by relevance. Another improvement will be to replace the current plain text file (which tracks indexed books) with a proper status collection in the database.

Later on, we might add a control module or message queue to make the interaction between Ingestion and Indexing more automated. We also plan to expand the logging system (which is currently set up in resources/simplelogger.properties) and start exporting performance metrics. That should make it easier to monitor and analyze how each service is performing over time.

Overall, Stage 2 gave us a solid foundation for building a modular and scalable search engine. Each service can continue to grow and improve on its own, while still communicating reliably through REST APIs and MongoDB.

7. References

- Javalin Framework Documentation – https://javalin.io
- MongoDB Official Documentation – https://www.mongodb.com/docs
- Docker Documentation – https://docs.docker.com

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