Universidad Politécnica

de Madrid

Escuela Técnica Superior de

Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Local File Search Engine Based on Full-Text Indexing and Metadata Extraction

Autor: Rodrigo Allende Rial

Tutor(a): Víctor Rodríguez Doncel

Madrid, 01/2026

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: *Local File Search Engine Based on Full-Text Indexing and Metadata Extraction*

01/2026

Autor: Rodrigo Allende Rial

Tutor:

Víctor Rodríguez Doncel

Inteligencia Artificial

ETSI Informáticos

Universidad Politécnica de Madrid

Resumen

La búsqueda eficiente en colecciones locales de documentos heterogéneos es un problema recurrente en entornos profesionales donde la privacidad y el control de los datos impiden depender de servicios en la nube, y donde las soluciones nativas del sistema operativo no ofrecen una interfaz abierta ni capacidades de evaluación reproducible. Este Trabajo Fin de Grado diseña e implementa FileSearch, un sistema de búsqueda local orientado a grandes volúmenes documentales, capaz de indexar contenido y metadatos de múltiples formatos y ejecutar consultas de texto completo con tolerancia a errores tipográficos y ordenación por relevancia.

La solución se fundamenta en una arquitectura modular con separación de responsabilidades entre interfaz de usuario y núcleo de procesamiento. El *pipeline* de indexación recorre el sistema de ficheros, detecta el tipo MIME y extrae contenido y metadatos mediante Apache Tika, transformando la información a un modelo documental persistido en un índice invertido en Elasticsearch. Sobre dicho índice se implementa un motor de consultas basado en construcción de consultas multi-campo con *fuzziness* y *scoring*, expuesto a través de una interfaz de línea de comandos para automatización y una interfaz gráfica multiplataforma basada en Electron/React para interacción exploratoria.

El sistema incorpora mecanismos de ingeniería de software orientados a operabilidad y calidad, incluyendo validación de configuración, *logging* estructurado, integración continua y un comando de diagnóstico (*doctor*) que verifica conectividad, estado del clúster, existencia del índice y condiciones operativas relevantes. La evaluación experimental se realiza mediante una metodología de «caja negra» automatizada, ejecutando escenarios reproducibles sobre datasets controlados y midiendo tiempo de indexación, throughput, latencias p50/p95/p99, y sobrecoste de almacenamiento del índice. Los resultados muestran que el enfoque propuesto permite búsquedas interactivas de baja latencia y un rendimiento de indexación estable, manteniendo el procesamiento completamente local y habilitando la comparación objetiva de configuraciones. En conjunto, el trabajo aporta una base técnica reproducible y extensible para sistemas de búsqueda.

Abstract

Efficient search over large local collections of heterogeneous documents remains a recurring challenge in professional environments where data privacy, offline operation, and system control prevent the use of cloud-based solutions. At the same time, operating system–native search tools provide limited extensibility and lack open interfaces for experimentation and evaluation. This Bachelor's Thesis presents the design and implementation of FileSearch, a local document search system aimed at large-scale document collections[1], [2], capable of indexing both content and metadata from multiple file formats and executing full-text queries with typo tolerance and relevance-based ranking.

The proposed solution is based on a modular architecture that clearly separates user interaction from the processing core. The indexing *pipeline* traverses the local file system, automatically detects MIME types, and extracts text and metadata using Apache Tika, transforming the information into a structured document model stored in an inverted index powered by Elasticsearch. On top of this index, a query engine constructs multi-field search queries with configurable fuzziness and scoring mechanisms. The system exposes its functionality through both a command-line interface designed for automation and scripting, and a cross-platform graphical user interface built with Electron and React for interactive exploration.

From a software engineering perspective, the system incorporates mechanisms aimed at robustness and operability, including configuration validation, structured logging, continuous integration, and a dedicated diagnostic command (*doctor*) that verifies connectivity, index integrity, and runtime conditions. The experimental evaluation follows an automated black-box methodology, executing reproducible scenarios over controlled datasets and measuring indexing time, throughput, search latency percentiles (p50, p95, p99), and index storage overhead. The results demonstrate that the proposed approach enables low-latency interactive search and stable indexing performance while maintaining all processing strictly local. Overall, this work provides a reproducible and extensible technical foundation for local document search systems.

Tabla de contenidos

1 Introduction and Objectives **1**
1.1. Context and Motivation 1
1.2. Problem Statement 2
1.3. Objectives of the Project 3
1.4. Scope and Limitations 5
1.5. Structure of the Document 7
2. State of the Art **7**
2.1. Information Retrieval Systems 8
2.2. Local Document Search Systems 9
2.3. Indexing of Heterogeneous Documents 9
2.4. Text and Metadata Extraction Tools 10
2.5. Search Engines and Indexing Technologies 13
2.6. Command-Line Interfaces and Graphical User Interfaces 15
2.7. Experimental Evaluation in Software Systems 15
3. System Design and Architecture **16**
3.1 Functional Requirements 16
3.2. Non-Functional Requirements 18
3.3. Overall System Architecture 19
3.4. Backend Architecture 21
3.5. Frontend Architecture 23
3.6. Data Flow and System Interactions 24
3.7. Elasticsearch Index Design 25
3.8. Key Architectural Decisions 27
4. Implementation **28**
4.1. Backend Implementation 28
4.2. Document Indexing Pipeline 29
4.3. Search Engine Integration 31
4.4. Command-Line Interface Implementation 32
4.5 Graphical User Interface Implementation 34
4.6. Configuration Management and Reproducibility 37
4.7. Error Handling and Logging 37
4.8. Continuous Integration and Code Quality 39
5. Experimental Evaluation **40**
5.1. Experimental Objectives 40
5.2. Experimental Methodology 41
5.3. Experimental Setup and Datasets 42
5.4. Performance Metrics 44
5.5. Benchmark Automation 46

5.6. Experimental Scenarios 47
5.7. Reproducibility and Experiment Automation 49
6. Results and Discussion 49
6.1. Indexing Performance Results 49
6.2. Search Latency Results 51
6.2.1. Cold Search Latency 52
6.2.2. Warm Search Latency 53
6.2.3. Tail Latency Analysis 53
6.2.4. Summary and Interpretation 53
6.3. Index Size and Storage Overhead 54
6.4. Discussion of Trade-offs and Observations 55
6.5 Comparative Analysis with Windows Search 56
7. Conclusions and Future Work 58
7.1. Conclusions 58
7.2. Limitations 59
7.3. Future Work 60
8. Impact Analysis 61
8.1. Technical Impact 61
8.2. Academic / Educational Impact 62
8.3. Industrial Relevance 62
8.4. Ethical and Sustainability Considerations 63
Bibliography 64
Annexes 66
● Annex A. Benchmark Data 66
A.1 Indexing Benchmark Data 67
A.2 Search Latency Measurements 67
A.3 Storage Overhead Measurements 68
● Annex B. Configuration and Reproducibility 69
B.1 Environment Configuration 69
B.2 Experimental Execution Scripts 70
B.3 Scenario Definition Files 71
B.4 Reproducibility Workflow 71
● Annex C. CLI Outputs 72
C.1 System Diagnostic Output 72
C.2 Indexing Command Output 73
C.3 Search Command Output 73

1 Introduction and Objectives

1.1. Context and Motivation

The rapid growth of digital information has led to a significant accumulation of documents in local computing environments. Software developers, researchers, and technical professionals routinely manage collections comprising tens of thousands of files, often exceeding hundreds of gigabytes in size. These collections typically include heterogeneous document formats such as technical documentation (PDF, DOCX), source code repositories, configuration files, and unstructured text datasets distributed across complex directory hierarchies.

Despite this reality, local search capabilities provided by modern operating systems remain limited in terms of transparency, configurability, and extensibility. While native tools such as macOS Spotlight or Windows Search provide fast query response times, they operate as closed systems, offering little to no control over indexing strategies, ranking models, or query behavior. Furthermore, their relevance scoring mechanisms are not inspectable, making them unsuitable for experimental evaluation or reproducible analysis. Empirical studies and practitioner reports indicate that default desktop search tools often struggle with large codebases or mixed-format datasets, exhibiting degraded relevance or incomplete indexing when repositories exceed tens of thousands of files.

Cloud-based search solutions partially address these limitations but introduce additional concerns. Uploading local datasets to external services raises privacy and confidentiality issues, particularly in professional or academic environments where proprietary or sensitive data must remain on-premise. Moreover, reliance on remote services introduces network latency and external dependencies, which conflicts with reproducibility and offline operability requirements.

In this context, there is a clear need for a fully local document search system that combines the performance characteristics of native indexing tools with the flexibility and openness of modern information retrieval technologies. Such a system should support content-based search across multiple document formats, expose its indexing and ranking mechanisms for inspection, and remain entirely self-contained without reliance on external services.

This project is motivated by the objective of bridging this gap. By leveraging mature open-source technologies and adopting a rigorous engineering approach, the proposed system aims to deliver a high-performance local search solution that is extensible, reproducible, and suitable for both practical usage and controlled experimental evaluation.

1.2. Problem Statement

The problem addressed in this project is the absence of an efficient, transparent, and configurable solution for content-based search over large collections of local documents. Existing local search tools typically prioritize ease of use or tight integration with the operating system, often at the expense of openness, extensibility, and inspectability.

From a technical perspective, many native indexing solutions operate as closed systems, exposing neither their internal indexing strategies nor their ranking mechanisms. This lack of transparency prevents advanced users from tuning search behavior, integrating search capabilities into external workflows, or performing systematic performance evaluation. Additionally, such tools rarely provide programmatic interfaces suitable for automation or experimentation.

Alternative approaches based on cloud services introduce additional limitations, particularly in environments where data privacy, regulatory compliance, or offline operation are critical requirements. These constraints make cloud-based solutions unsuitable for many professional and technical use cases.

Furthermore, many academic and experimental implementations of local search systems focus primarily on functional correctness, without addressing performance characterization, scalability analysis, or reproducibility. As a result, it remains difficult to assess how such systems behave under realistic workloads or to compare different architectural and configuration choices.

The core challenge, therefore, is to design and implement a local document search system that combines high performance with architectural clarity, exposes its internal mechanisms through well-defined interfaces, and can be evaluated quantitatively in a reproducible manner. Addressing this challenge requires not only the integration of appropriate technologies but also a disciplined engineering approach grounded in empirical analysis.

1.3. Objectives of the Project

The main objective of this project is to design, implement, and evaluate a local document search system capable of efficiently indexing and retrieving large volumes of heterogeneous documents while operating entirely on-premise.

This primary objective is refined into the following specific objectives:

- O1. System Architecture Design
  To design a modular, extensible, and maintainable system architecture

that clearly separates core indexing and search logic from interface and infrastructure concerns.

- O2. Local Indexing and Search Functionality
  To implement a robust local indexing and full-text search pipeline capable of processing heterogeneous document formats, supporting relevance-based ranking and fuzzy matching without reliance on external services.

- O3. User Interaction and Operability
  To provide both command-line and graphical user interfaces that enable efficient interaction with the system, including indexing control, search execution, and inspection of system state.

- O4. Experimental Evaluation and Reproducibility
  To conduct a quantitative experimental evaluation assessing indexing performance, search latency, throughput, and storage overhead, ensuring reproducibility through controlled configurations and automated benchmark execution.

Together, these objectives reflect an engineering-oriented approach that treats the system not only as a functional application, but also as a platform for systematic performance evaluation and experimentation.

The objectives of the project have been defined to provide a clear and structured view of the intended contributions of the work, covering system design, implementation, and experimental evaluation. This organization facilitates both the development process and the subsequent validation of the proposed solution through quantitative experiments.

Table 1 summarizes the main objectives of the project, together with their classification and the sections where each objective is addressed and validated.

Table 1 – Project Objectives Classification and Validation Mapping

ID	Objective	Type	Evidence (Sections)
F1	Implement a local document indexing engine capable of parsing heterogeneous formats (PDF, DOCX, TXT, HTML) entirely on-premise.	Functional	4.1, 4.2, 6.1
F2	Develop a dual-interface system comprising a CLI for automation and an Electron-based GUI for end-user interaction.	Functional	4.4, 4.5
F3	Integrate full-text search capabilities with fuzzy matching and relevance scoring using an inverted index structure.	Functional	4.3, 6.2
NF1	Ensure low-latency search response times suitable for interactive usage under realistic workloads.	Non-Functional	6.2, Annex A
NF2	Ensure system operability through built-in self-diagnostic tools and structured logging mechanisms.	Non-Functional	4.7, Annex C
NF3	Guarantee experimental reproducibility through automated configuration management and benchmark execution.	Non-Functional	4.6, 5.7
NF4	Maintain data privacy by restricting all processing and storage to the local filesystem without external cloud dependencies.	Non-Functional	3.2, 3.6

Table 1 summarizes the main objectives of the project, distinguishing between functional and non-functional goals. For each objective, the table indicates the sections of the document where its implementation, validation, or experimental evaluation is addressed, ensuring traceability between design intentions and achieved results.

1.4. Scope and Limitations

The scope of this project is focused on the design, implementation, and evaluation of a local document search system intended to operate in single-machine environments. The system targets collections of documents stored on local file systems and emphasizes content-based search capabilities over heterogeneous document formats.

Within this scope, the system supports exactly the following textual document formats: PDF, DOCX, TXT, and HTML. The implementation is primarily oriented towards UNIX-like operating systems, such as Linux and macOS, where file system traversal and process execution models are well suited to the adopted architecture. Although the underlying technology stack is portable, full validation on other platforms is outside the scope of this work.

The project explicitly addresses performance, scalability within a single-node context, configurability, and reproducibility. Particular emphasis is placed on exposing system behavior through programmatic interfaces and quantitative evaluation rather than optimizing for end-user simplicity alone.

Several aspects are intentionally excluded from the scope of this project. These include distributed or clustered deployments of the search engine, real-time indexing of continuously changing document collections and user authentication or access control mechanisms.

These limitations are deliberate and reflect the objective of maintaining a well-defined problem space. By constraining the scope, the project is able to focus on architectural quality, empirical evaluation, and engineering rigor within a realistic and manageable context.

Table 2 – Project Scope Definition: Implemented Features vs Out-of-Scope Aspects.

In Scope (Implemented)	**Out of Scope (Excluded / Future Work)**
Local-only processing: complete indexing and search execution on the local filesystem without external cloud dependencies.	Cloud-based indexing and search, including remote storage integration or web crawling.
Support for common document formats: PDF, DOCX, TXT, and HTML using Apache Tika.	Optical Character Recognition (OCR) for scanned documents or image-based PDFs.
Dual interaction model: a command-line interface for automation and an Electron-based graphical user interface for end users.	Native Windows distribution and platform-specific optimizations outside UNIX-like systems.
Full-text search with fuzzy[4] matching and relevance scoring based on an inverted index.	Development of a proprietary search engine or custom indexing algorithms from scratch.
Automated experimental evaluation of indexing and search performance using external benchmarks.	Multi-user or centralized server-based deployment models.
Experimental reproducibility support through automated configuration and controlled benchmark execution.	Parsing of proprietary or closed-source document formats requiring specialized drivers.

Table 2 summarizes the functional scope of the project by explicitly distinguishing between implemented features and intentionally excluded aspects. This classification helps delimit the problem space and clarifies the design decisions and evaluation focus adopted in this work.

1.5. Structure of the Document

The remainder of this document is organized as follows. Chapter 2 reviews the state of the art in information retrieval systems, local document search solutions, and the technologies relevant to content extraction and indexing. Chapter 3 presents the design and architecture of the proposed system, detailing its main components, architectural principles, and key design decisions.

Chapter 4 describes the implementation of the system, focusing on the realization of the architectural design and the main engineering aspects of the backend, frontend, and supporting infrastructure. Chapter 5 introduces the experimental evaluation methodology, including datasets, metrics, and benchmark automation.

Chapter 6 presents and discusses the experimental results, analyzing indexing and search performance, latency distributions, throughput, and storage requirements. Chapter 7 concludes the work by summarizing the main findings, identifying system limitations, and outlining directions for future work. Finally, Chapter 8 analyzes the technical, academic, and industrial impact of the project, along with ethical and sustainability considerations.

2. State of the Art

This chapter presents a structured review of the state of the art in information retrieval and local document search systems. The analysis is based on a combination of foundational academic literature, industry-standard technical documentation, and widely adopted open-source technologies relevant to the design and implementation of modern search engines.

The review focuses on core information retrieval concepts, indexing and ranking techniques, document content extraction tools, and practical system architectures. Priority is given to approaches that are applicable to on-premise environments and full-text search over heterogeneous document collections, in alignment with the objectives of this project. The selected references provide both the theoretical background and the practical context necessary to justify the architectural and technological decisions presented in subsequent chapters.

2.1. Information Retrieval Systems

Information Retrieval (IR) systems are designed to retrieve relevant information from large collections of unstructured or semi-structured data in response to user queries. Unlike traditional database systems, which operate over structured records and exact matching, IR systems focus on approximate matching, relevance estimation, and ranking.

At the core of most IR systems lies the concept of the inverted index , a data structure that maps terms to the documents in which they appear. This structure enables efficient full-text search over large document collections and serves as the foundation for scalable retrieval systems. During indexing, documents are tokenized, normalized, and transformed into indexable representations that support fast query evaluation.

Query processing in IR systems typically involves parsing user queries, applying analyzers and similarity models, and computing relevance scores for matching documents. Ranking functions, often based on term frequency and inverse document frequency, are used to order results according to their estimated relevance. While the underlying retrieval models may vary, performance considerations such as latency, throughput, and index size play a central role in practical system design.

Modern IR systems emphasize not only retrieval effectiveness but also engineering concerns such as scalability, configurability, and robustness. These systems are commonly deployed as core components in search engines, log analysis platforms, and document management systems. As a result, IR has evolved from a primarily theoretical discipline into a field where architectural decisions and empirical performance evaluation are as important as retrieval models themselves.

In the context of local document search, IR systems must operate under additional constraints [1], including limited hardware resources, heterogeneous document formats, and the need for on-premise execution. These constraints make the selection of appropriate indexing strategies, ranking mechanisms, and evaluation methodologies particularly relevant.

2.2. Local Document Search Systems

Local document search systems are designed to provide content-based retrieval capabilities over files stored on a user's local machine. Most modern operating systems include built-in search functionality that allows users to locate files based on names, metadata, or indexed content. These systems are typically optimized for low-latency interactive use and tight integration with the underlying operating system.

While native solutions achieve high performance, they are commonly implemented as closed systems. Indexing strategies, ranking models, and internal configurations are not exposed to the user, limiting transparency and preventing advanced customization. As a result, it is difficult to adapt these tools to specialized workflows, integrate them into automated processes, or perform systematic performance analysis.

Third-party desktop search tools have attempted to address some of these limitations, but many of them suffer from outdated architectures, limited extensibility, or a lack of active maintenance. Furthermore, few local search solutions provide programmatic interfaces suitable for automation or experimentation, focusing instead on end-user interaction alone.

From an engineering perspective, local document search systems are often treated as lightweight utilities rather than as full-fledged information retrieval platforms. This approach contrasts with the increasing complexity and scale of local document collections managed by technical users. Consequently, there exists a gap between high-performance native search tools and flexible, inspectable systems that expose their internal behavior and can be evaluated quantitatively.

The system proposed in this project is positioned within this gap, aiming to combine the performance characteristics expected from local search tools with the openness, configurability, and evaluability of modern information retrieval systems.

2.3. Indexing of Heterogeneous Documents

Indexing heterogeneous document collections presents a significant challenge for local search systems. Unlike homogeneous datasets, real-world document repositories typically contain files in multiple formats, each with distinct internal structures, encoding schemes, and metadata representations.

Common document formats such as PDF, word processor files, plain text, and HTML differ substantially in how content is stored and accessed. Some formats embed text directly, while others rely on complex layout structures or compressed binary representations. As a result, extracting meaningful textual content in a consistent manner requires specialized parsing strategies and robust error handling.

From an indexing perspective, heterogeneity affects both retrieval quality and system performance. Incomplete or inaccurate extraction can lead to poor search results, while inefficient parsing can become a major bottleneck during large-scale indexing operations. Furthermore, document collections often include partially corrupted files, unsupported formats, or inconsistent encodings, which must be handled gracefully to avoid compromising the overall indexing process.

In addition to textual content, metadata such as file paths, modification timestamps, and document attributes play an important role in filtering, ranking, and result interpretation. Incorporating such metadata into the index requires careful schema design to balance expressiveness, storage overhead, and query performance.

Consequently, the indexing of heterogeneous documents cannot be treated as a trivial preprocessing step. Instead, it constitutes a central component of the system architecture, directly influencing robustness, scalability, and search effectiveness. These considerations motivate the selection of dedicated extraction frameworks capable of handling a wide range of document formats in a unified and reliable manner.

2.4. Text and Metadata Extraction Tools

Text and metadata extraction constitutes a fundamental stage in the document indexing pipeline. The quality, completeness, and efficiency of this stage directly affect both retrieval effectiveness and overall system performance. As a result, the selection of appropriate extraction tools represents a critical design decision.

Several approaches exist for extracting content from documents. Simple solutions based on file extensions or regular expressions can be effective for homogeneous or narrowly scoped datasets, but they fail to scale when dealing with heterogeneous collections. Format-specific parsers, while potentially

efficient, introduce additional complexity in terms of maintenance, extensibility, and error handling.

Apache Tika[5], [16] is a widely adopted content analysis framework designed to address these challenges. One of its key features is the automatic detection of document types based on content rather than file extensions. This capability allows Tika to identify MIME types reliably and select appropriate parsers without requiring manual configuration or format-specific logic.

In addition to text extraction, Tika provides access to a rich set of metadata attributes, including encoding information, content type, and document-specific properties. This unified extraction interface simplifies integration into indexing pipelines and enables consistent handling of diverse document formats.

From an engineering perspective, the use of Apache Tika reduces the complexity of the indexing pipeline by abstracting away low-level parsing details. It also improves robustness by handling malformed or partially corrupted files gracefully. These characteristics make Tika particularly suitable for large-scale indexing scenarios where manual intervention is impractical.

The adoption of Apache Tika version 2.x in this project reflects a deliberate choice to prioritize extensibility, reliability, and long-term maintainability. By leveraging a mature and actively maintained framework, the system benefits from broad format support while remaining focused on higher-level indexing and search concerns.

To justify the selection of Apache Tika as the content extraction component of the system, several alternative approaches were analyzed and compared. The comparison focuses on criteria that are directly relevant to the objectives of this project, particularly the ability to process heterogeneous document formats, robustness against malformed or partially corrupted files, extensibility to support new formats, and long-term maintenance costs.

These criteria reflect practical requirements for building a reliable local indexing pipeline capable of operating autonomously over large and diverse document collections, as required by the functional objectives defined in Chapter 1.

In particular, robustness, extensibility, and suitability for heterogeneous datasets are critical to ensure reproducible experimental evaluation and stable indexing behavior, which are core non-functional objectives of this work.

Table 3 – Comparison of Document Content Extraction Approaches.

Tool	Supported Formats	Metadata Extraction	Robustness to Malformed Files	Extensibility	Maintenance Cost	Suitability for Heterogeneous Datasets
Apache Tika	High: Wide support for common document formats (PDF, DOCX, TXT, HTML) via a unified API.	Comprehensive: Standardized metadata extraction (author, date, MIME type).	High: Built-in exception handling and fallback mechanisms.	High: Modular design with pluggable parsers and detectors.	Low: Actively maintained open-source project.	High: Automatic format detection enables robust heterogeneous processing.
Regex / String Matching	Very Low: Limited to plain or lightly structured text.	None / Manual: Requires ad-hoc rules per format.	Low: Fragile when facing format variations or binary data.	Low: Each new format requires manual logic.	High: Frequent updates needed to maintain correctness.	Poor: Not viable for diverse or binary document collections.
OS-native Indexers	Medium: Depends on OS-installed plugins and extensions.	High: Tight integration with filesystem metadata.	Medium: Reliable for common formats, opaque for edge cases.	Low: Limited customization and portability.	Low: Managed by OS vendor, limited developer control.	Medium: Suitable for personal use, unsuitable for custom systems.
Custom Parsers per Format	Low: Restricted to explicitly implemented formats.	High (Custom): Fine-grained control over extracted fields.	Variable: Depends on implementation quality.	Medium: Requires integrating new libraries per format.	Very High: High long-term maintenance burden.	Low: Poor scalability as format diversity increases.

This table compares alternative approaches for document content extraction according to criteria relevant for heterogeneous local document collections. The analysis highlights the trade-offs between flexibility, robustness, and maintenance cost, and motivates the selection of Apache Tika as the extraction component in the proposed system.

2.5. Search Engines and Indexing Technologies

Search engines designed for information retrieval differ fundamentally from traditional relational database management systems. While databases excel at structured queries and exact matching, search engines are optimized for approximate matching, relevance ranking, and full-text retrieval over large volumes of unstructured data.

Most modern search engines rely on inverted index structures, which enable efficient mapping between terms and the documents in which they appear. This approach allows query execution time to scale with the number of matching terms rather than the size of the entire dataset, making it particularly suitable for large document collections.

Apache Lucene[6] is a widely used open-source library that provides the core indexing and search capabilities underlying many industrial search platforms. It implements advanced features such as tokenization, scoring models, and query parsing, serving as a foundation for higher-level search systems.

Elasticsearch builds upon Lucene and exposes its functionality through a distributed, REST-based architecture[7], [15]. Although commonly deployed in clustered environments, Elasticsearch can also operate effectively in single-node configurations, making it suitable for local search applications. Its flexible schema definition, configurable analyzers, and rich query language enable fine-grained control over indexing and retrieval behavior.

An important feature of modern search engines is support for fuzzy matching[8], which allows the system to tolerate minor misspellings or variations in query terms. This capability improves usability and retrieval effectiveness, particularly in interactive search scenarios. In contrast, traditional database queries based on exact string matching, such as SQL LIKE clauses, provide limited flexibility and poor performance for full-text search tasks.

From an engineering perspective, the adoption of a dedicated search engine enables not only improved retrieval performance but also systematic performance evaluation. Configurable indexing parameters, query models, and execution metrics make it possible to analyze the impact of design choices quantitatively, which is essential for experimental assessment.

Table 4 presents a comparative analysis of different search technologies commonly used for local document retrieval, highlighting their capabilities and limitations with respect to relevance ranking, scalability, performance, and suitability for heterogeneous document collections.

Table 4 – Comparison of Search Technologies for Local Document Retrieval.

Technology	Search Model	Relevance Ranking	Fuzzy / Typo-tolerant	Scalability with Document Volume	Query Performance Characteristics	Suitability for Heterogeneous Documents
Simple File System Search	Linear scan over file paths or contents without a pre-built index.	None: results are binary and ordered by file attributes.	Low: relies on strict matching or complex user-defined expressions.	Very Poor: search time grows linearly with dataset size.	High CPU and I/O usage due to repeated full scans.	Very Low: limited support for complex or binary document formats.
SQL Database (LIKE queries)	Relational table scan over textual columns.	None: exact substring matching without scoring.	Low: edit-distance search requires custom logic or extensions.	Poor: performance degrades rapidly as table size increases.	Slow for text-heavy workloads due to inefficient substring indexing.	Low: requires flattening documents into rigid schemas.
RDBMS with Full-Text Extensions	Token-based inverted index integrated into relational tables.	Medium: basic TF-IDF-style ranking with limited tuning.	Medium: requires additional extensions or complex configuration.	Good: improved over scans, but write performance may degrade.	Moderate: suitable for mixed workloads, less optimized for complex queries.	Medium: adequate for structured data with textual fields.

Inverted Index Engine (Lucene / Elasticsearch)	Dedicated inverted index optimized for token lookup and scoring.	High: native BM25-based relevance scoring with field-level boosts.[3]	High: built-in edit-distance matching configurable at query time.	High: efficient handling of large document collections.	Consistently fast for full-text queries due to index-based execution.	High: flexible document model naturally supports heterogeneous metadata.

This table compares alternative search approaches ranging from simple file system scans to dedicated inverted-index engines. The comparison focuses on characteristics that are directly relevant to the objectives of this project, including relevance ranking quality, fuzzy matching support, scalability with document volume, query performance, and suitability for heterogeneous document collections. The analysis motivates the selection of an inverted index–based engine (Lucene/Elasticsearch) as the core search technology.

2.6. Command-Line Interfaces and Graphical User Interfaces

From a systems engineering perspective, the choice of interaction interfaces has direct implications on automation, reproducibility[11], and experimental evaluation. Rather than focusing on usability aspects, this section discusses the role of different interfaces in supporting distinct operational and evaluation scenarios.

Command-line interfaces (CLI) are particularly suited for automated execution, batch processing, and controlled experimentation. In the context of this project, the CLI constitutes the primary interface for indexing operations, benchmark execution, and system diagnostics, enabling reproducible experiments and precise performance measurements without user interaction variability.

Graphical user interfaces (GUI), in contrast, are intended to facilitate exploratory interaction with the indexed data. The GUI allows manual inspection of search results, relevance ranking behavior, and metadata visualization, supporting qualitative validation of system behavior rather than quantitative benchmarking.

By exposing both interfaces over a shared backend core, the system avoids duplication of logic while enabling complementary usage modes. This design decision supports the dual objectives of experimental evaluation and practical usability, while keeping the core indexing and retrieval mechanisms independent of the interaction layer.

2.7. Experimental Evaluation in Software Systems

Experimental evaluation is a fundamental component of software engineering, particularly in systems where performance, scalability, and resource utilization are critical concerns. Beyond functional correctness, quantitative evaluation provides empirical evidence about how a system behaves under realistic workloads and operating conditions.

In the context of information retrieval systems, evaluation typically focuses on metrics such as indexing time, query latency, throughput, and storage overhead. These metrics allow engineers to assess trade-offs between performance and resource consumption, as well as to compare different architectural or configuration choices objectively.

These evaluation principles are widely adopted in industrial and open-source search platforms such as Apache Solr and Elasticsearch. Both systems emphasize quantitative performance metrics, including indexing throughput, query latency under varying workloads, and storage overhead, and provide built-in instrumentation to support systematic benchmarking and monitoring. Their evaluation methodologies commonly rely on controlled experimental setups, repeatable benchmark executions, and configuration-driven experimentation, reflecting the importance of reproducibility in search system engineering.

A key challenge in experimental evaluation is ensuring reproducibility. Performance measurements are often sensitive to environmental factors, configuration parameters, and execution order. Without controlled experimental setups and automated execution, results may be difficult to reproduce or interpret. Consequently, modern evaluation methodologies emphasize controlled configurations, repeated measurements, and systematic collection of results.

Another important consideration is the distinction between white-box and black-box evaluation approaches. White-box evaluation relies on internal instrumentation and profiling, while black-box evaluation measures system behavior from an external perspective. Black-box approaches are particularly useful for assessing end-to-end performance and avoiding measurement bias introduced by internal instrumentation.

Despite its importance, experimental evaluation is frequently underrepresented in application-oriented software projects, where emphasis is placed primarily on functionality. This limitation reduces the ability to reason about system behavior and to justify design decisions empirically.

In this project, experimental evaluation is treated as a first-class concern. The system is designed not only to provide search functionality but also to serve as a platform for systematic performance analysis, aligning engineering decisions with quantitative evidence.

3. System Design and Architecture

3.1 Functional Requirements

The system is designed to provide advanced local document search capabilities over large collections of heterogeneous files. Based on the objectives defined in Chapter 1 and the analysis of existing solutions presented in Chapter 2, the following functional requirements have been identified.

- **FR1 – Document Indexing**:
  The system shall index documents stored on the local file system by recursively traversing directory structures provided by the user.

- **FR2 – Content and Metadata Extraction**:
  The system shall extract textual content and relevant metadata from supported document formats, enabling content-based search and metadata-based filtering.

- **FR3 – Support for Heterogeneous Formats**:
  The system shall support commonly used textual document formats, including PDF, DOCX, TXT, and HTML files.

- **FR4 – Full-Text Search**:
  The system shall support full-text search queries over indexed content, returning ranked results based on estimated relevance.

- **FR5 – Fuzzy Matching**:
  The system shall tolerate minor variations or misspellings in search queries through fuzzy matching mechanisms.

- **FR6 – Command-Line Interface**:
  The system shall provide a command-line interface that allows users to perform indexing, search, and diagnostic operations in an automated and scriptable manner.

- **FR7 – Graphical User Interface**:
  The system shall provide a graphical user interface that enables interactive search, result exploration, and visualization of system statistics.

- **FR8 – System Diagnostics**:
  The system shall expose diagnostic functionality to verify system

configuration, search engine availability, and index state.

- **FR9 – Index Statistics Retrieval**:
  The system shall allow users to retrieve information about the indexed document collection, including document counts and storage usage.

These functional requirements define the expected behavior of the system and serve as a reference for architectural design and implementation decisions discussed in the following sections.

3.2. Non-Functional Requirements

In addition to functional requirements, the system is designed to satisfy a set of non-functional requirements that define its quality attributes and operational constraints. These requirements guide architectural decisions and serve as evaluation criteria throughout the project.

- **NFR1 – Performance**:
  The system shall provide indexing and search performance suitable for interactive use. Search query latency shall remain within acceptable bounds for typical workloads, and indexing throughput shall scale predictably with dataset size.

- **NFR2 – Scalability (Single-Node)**:
  The system shall scale efficiently with increasing numbers of documents within a single-machine environment, without requiring architectural changes.

- **NFR3 – Reproducibility**:
  Experimental evaluations shall be reproducible across executions, with configurations and datasets explicitly defined and controlled.

- **NFR4 – Robustness**:
  The system shall handle malformed documents, unsupported formats, and partial failures without interrupting indexing or search operations.

- **NFR5 – Modularity and Maintainability**:
  The system architecture shall promote separation of concerns, enabling independent development, testing, and evolution of components.

- **NFR6 – Usability and Developer Experience**:
  The system shall provide both command-line and graphical interfaces

tailored to different user profiles, facilitating ease of use and integration into existing workflows.

- **NFR7 – Observability**:
  The system shall expose sufficient diagnostic information to allow users to inspect system state, verify configurations, and identify operational issues.

- **NFR8 – Portability**:
  The system shall be portable across UNIX-like operating systems, relying on widely supported technologies and standard execution environments.

These non-functional requirements establish the quality objectives of the system and directly influence the architectural design presented in the subsequent sections.

3.3. Overall System Architecture

The system is designed following a simplified hexagonal architecture, also known as the ports and adapters architectural pattern. This approach emphasizes the separation between core domain logic and external interfaces, enabling flexibility, testability, and long-term maintainability.

At the center of the architecture lies the core domain, which encapsulates the fundamental responsibilities of the system, including document indexing coordination, search orchestration, configuration validation, and diagnostic logic. The core domain is deliberately kept independent of specific user interfaces or infrastructure technologies.

Surrounding the core domain are a set of adapters responsible for interacting with external systems and users. These adapters include command-line interfaces, graphical user interfaces, and infrastructure components such as the search engine client. Each adapter communicates with the core domain through well-defined interfaces, ensuring that changes in one adapter do not propagate across the entire system.

This architectural structure enables the coexistence of multiple interaction modalities, such as CLI and GUI, without duplicating business logic. It also facilitates the integration of auxiliary components, including diagnostic commands and experimental evaluation tools, which interact with the core domain through the same abstraction mechanisms.

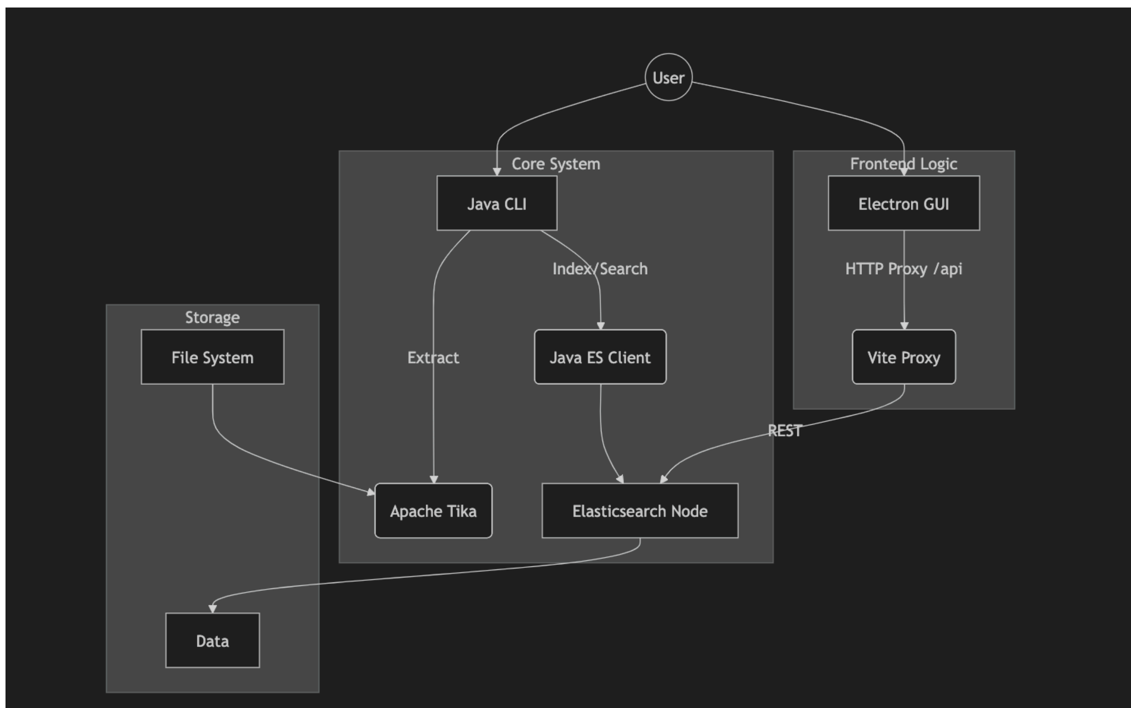
From an engineering perspective, the adoption of a hexagonal architecture supports several non-functional requirements defined in Section 3.2.

Modularity is enhanced by isolating concerns, robustness is improved by containing failures within adapters, and reproducibility is facilitated by providing controlled entry points for automated execution.

Overall, this architectural approach provides a coherent framework that aligns with the project's objectives, supports extensibility, and enables systematic evaluation of system behavior.

Figure 5 provides an overview of the high-level architecture of the proposed system, highlighting the main components and their interactions.

Figure 5 – High-level architecture of the local file search system



The diagram illustrates the overall structure of the system and the interaction between its main components. The core backend layer, implemented in Java, is responsible for document extraction, indexing, and search execution, relying on Apache Tika for content and metadata extraction and Elasticsearch for inverted-index–based search. User interaction is provided through two independent interfaces: a command-line interface (CLI) for automation and experimentation, and an Electron-based graphical user interface (GUI) for interactive exploration. The architecture enforces a clear separation of concerns between core logic, user interfaces, and external dependencies, supporting extensibility, robustness, and reproducible experimental evaluation.

Figure 5 illustrates the high-level architecture of the system and the interactions between its main components. The architecture follows a layered and modular design, clearly separating the core execution logic from user interfaces and external dependencies.

The Core System, implemented in Java, contains the main indexing and search functionality. User interactions are initiated either through the Java-based Command Line Interface (CLI) or through the graphical frontend. Both interfaces interact with the core exclusively through well-defined entry points, ensuring consistency and reproducibility of system behavior.

Document ingestion begins with traversal of the local file system, after which content and metadata are extracted using Apache Tika. The extracted information is then forwarded to the Elasticsearch node via a dedicated Java Elasticsearch client, which handles indexing and query execution through REST-based communication.

The graphical frontend, implemented using Electron, communicates with the backend through an HTTP API exposed by the core system, with a lightweight proxy layer facilitating frontend-backend integration. This design avoids duplication of logic and allows both interfaces to share the same underlying indexing and search mechanisms.

Overall, the architecture emphasizes separation of concerns, robustness, and extensibility, while enabling systematic evaluation and automated execution of indexing and search workflows.

3.4. Backend Architecture

The backend constitutes the core execution layer of the system and is implemented in Java 17. The choice of Java as the backend language is motivated by its mature ecosystem, strong concurrency support, and suitability for high-throughput, long-running processes such as large-scale document indexing.

The backend is structured according to the principles of the hexagonal architecture introduced in Section 3.3. The core domain encapsulates the main application logic, including indexing coordination, search execution, configuration validation, and diagnostic checks. This core remains independent of specific input mechanisms or infrastructure details.

Interaction with users is handled through adapter components. The command-line interface is implemented using a dedicated CLI framework, which maps user commands to application use cases exposed by the core domain. This approach allows commands such as indexing, searching, and system diagnostics to be implemented as thin adapters without duplicating business logic.

Communication with the search engine is managed through a dedicated infrastructure adapter responsible for interacting with Elasticsearch. This adapter encapsulates all low-level details related to HTTP communication, query construction, and response parsing. By isolating these concerns, the backend remains resilient to changes in the search engine client or API.

The backend architecture also incorporates explicit configuration management and validation mechanisms. Configuration parameters are loaded at startup and verified before execution, ensuring that misconfigurations are detected early and reported through diagnostic facilities.

Overall, this architecture promotes modularity, robustness, and testability. It allows backend components to evolve independently, supports multiple interaction modalities, and provides a stable foundation for both operational use and experimental evaluation.

Figure 6 presents a UML-style component diagram detailing the internal structure of the backend application and its execution flow.

Figure 6 – UML component diagram of the backend application.

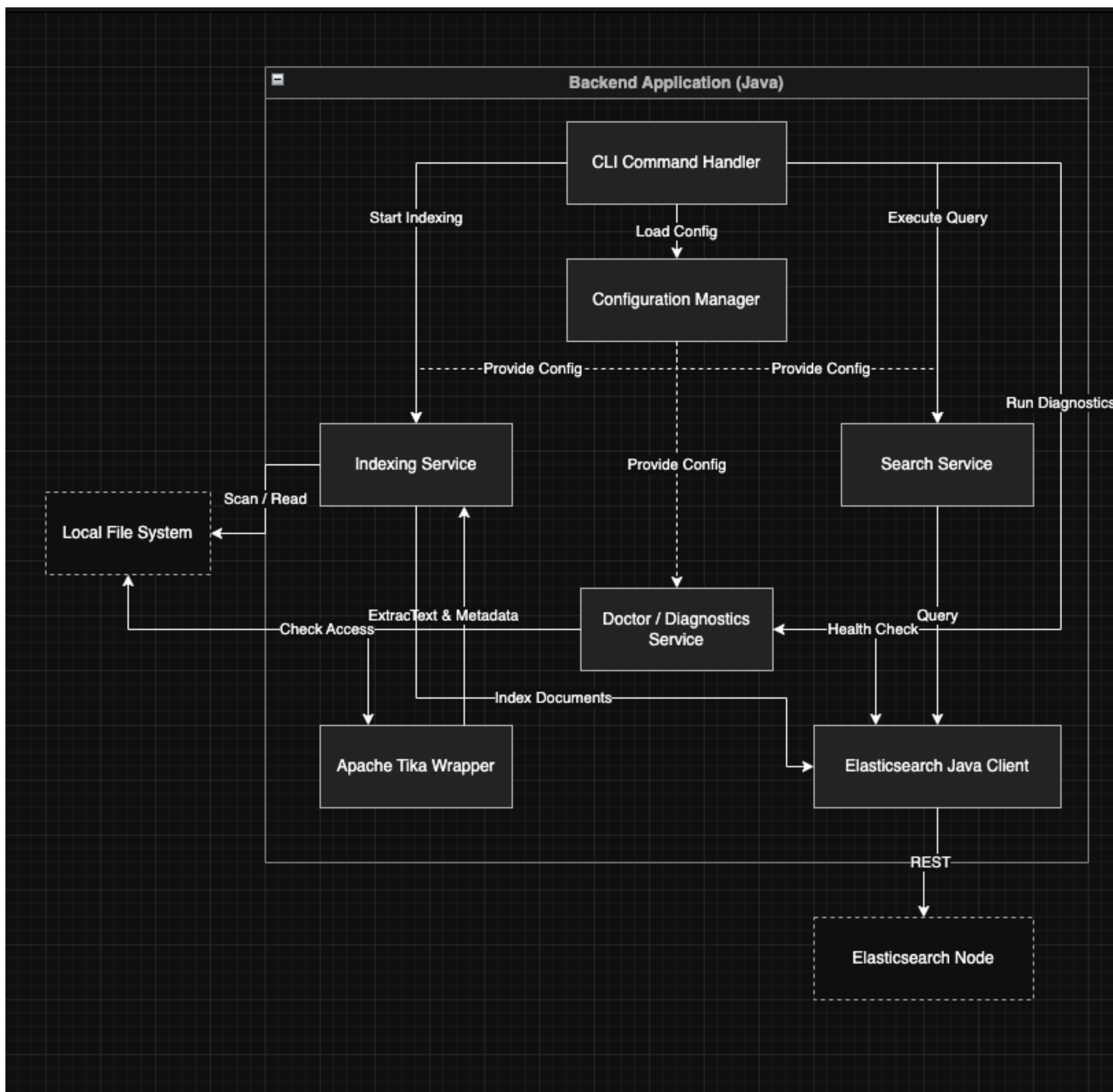


Figure 6 – UML component diagram of the backend application.

The diagram describes the internal organization of the Java-based backend and the interactions between its main components. The CLI Command Handler acts as the primary entry point, triggering indexing and search operations and loading system configuration through the Configuration Manager. The Indexing Service coordinates filesystem access and delegates content and metadata extraction to the Apache Tika wrapper before indexing documents via the Elasticsearch Java Client. The Search Service executes user queries against the Elasticsearch node, while the Doctor/Diagnostics Service performs health checks and validation tasks to detect misconfigurations and runtime issues. This design emphasizes clear responsibility separation, controlled configuration flow, and robust operational diagnostics.

As shown in Figure 6, the backend follows a modular design in which indexing, search, configuration management, and diagnostics are implemented as independent services.

3.5. Frontend Architecture

The graphical user interface (GUI) is implemented as a cross-platform desktop application based on Electron and a React single-page application (SPA). This architecture combines the deployment and integration advantages of desktop applications with the development productivity and UI flexibility of modern web technologies.

The frontend is built with React 18 and TypeScript, adopting a component-based design that separates presentation concerns from data access and application state. User-facing functionality is organized into reusable UI components and higher-level views, enabling independent evolution of the search interface, result visualization, and statistics dashboards.

Electron serves as the application container, providing a consistent runtime across operating systems and enabling distribution as a desktop product. The use of Vite as the build tool supports fast development iteration through hot module replacement (HMR), which improves developer experience and reduces turnaround time during UI development.

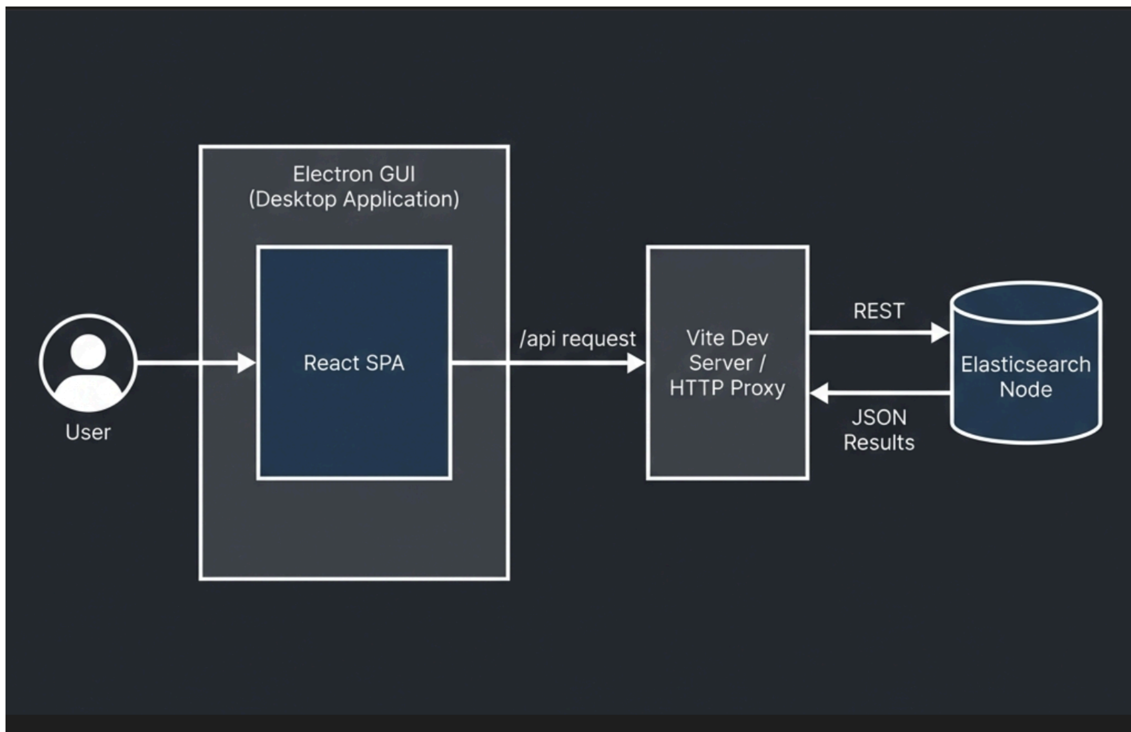
For styling, the system uses a utility-first approach via Tailwind CSS. This enables consistent design across the application while supporting both dark and light modes. Theme selection follows the operating system preference, ensuring a coherent user experience without requiring manual configuration.

Regarding backend communication, the GUI interacts with Elasticsearch through HTTP requests. During development, a proxy mechanism is used to route API calls (e.g., /api) to the Elasticsearch endpoint, effectively simulating a Backend-for-Frontend (BFF) layer without introducing an additional runtime service. This approach simplifies development, reduces coupling, and maintains a clear separation between UI logic and the search engine infrastructure[9], [10].

Overall, the frontend architecture prioritizes usability, maintainability[9], and developer productivity while remaining consistent with the modular principles of the overall system design.

Figure 7 illustrates the architecture of the graphical user interface and its interaction with the backend search engine through a REST-based communication layer.

Figure 7 – Frontend architecture and interaction flow of the graphical user interface.



The diagram shows the structure of the Electron-based desktop application and its communication with the backend search engine. User interactions are handled by a React single-page application (SPA) embedded within the Electron GUI. API requests are routed through a Vite development server acting as an HTTP proxy, which forwards REST queries to the Elasticsearch node and returns JSON-formatted results to the frontend. This architecture decouples the user interface from the search engine, simplifies frontend development, and enables a clear and controlled communication boundary between presentation and data retrieval layers.

3.6. Data Flow and System Interactions

The system is designed around well-defined data flows that govern how documents are indexed, how queries are processed, and how diagnostic information is obtained. These flows reflect the architectural separation between core logic and external interfaces and ensure consistent behavior across different usage scenarios.

The indexing workflow begins with a user-initiated operation, either through the command-line interface or via scripted execution. The backend traverses the file system recursively, identifying candidate files for indexing. For each file, textual content and metadata are extracted using the configured extraction framework and transformed into a structured representation suitable for indexing. Documents are then sent to the search engine using batched requests, optimizing throughput while maintaining fault isolation at the document level.

The search workflow follows a similar interaction pattern. User queries originating from the CLI or GUI are translated into structured search requests and forwarded to the search engine. The search engine evaluates the query against the indexed data and returns ranked results, which are subsequently presented to the user through the corresponding interface. This flow ensures that search behavior remains consistent regardless of the interaction modality.

In addition to indexing and search operations, the system supports diagnostic and inspection workflows. Diagnostic commands interact with the search engine to verify connectivity, cluster status, index availability, and configuration validity. These interactions provide real-time insight into system state and enable early detection of operational issues.

From a data flow perspective, all interactions with the search engine are mediated through dedicated infrastructure components, preserving the independence of the core domain logic. This design simplifies debugging[10], enables controlled experimentation, and supports reproducibility by ensuring that all system interactions follow explicit and inspectable pathways.

Overall, the defined data flows establish a clear and predictable execution model, facilitating both operational use and systematic performance evaluation.

3.7. Elasticsearch Index Design

The effectiveness and performance of a search system are strongly influenced by the design of its underlying index. In this project, a dedicated Elasticsearch index named filesearch is used to store both document content and associated metadata in a structured and searchable form.

The index schema is designed to support efficient full-text search while enabling filtering and result interpretation based on document attributes. To this end, indexed fields are classified according to their intended usage. Fields involved in relevance ranking and free-text search are defined using full-text data types, while fields used for exact matching or filtering are defined using keyword-based representations.

The main indexed fields include the document filename, extracted textual content, file path, and last modification timestamp. The filename field is assigned a higher relevance weight through boosting, reflecting the assumption that matches in file names are often more indicative of user intent than matches occurring exclusively within document content. This design choice aims to improve result quality without introducing complex ranking models.

Textual fields are indexed using analyzers suitable for general-purpose full-text search, enabling tokenization, normalization, and relevance scoring. Metadata fields, such as file paths or extensions, are indexed as keywords to support exact matching and efficient filtering operations.

This schema design balances expressiveness and performance. By separating searchable content from filterable metadata, the index supports a wide range of query patterns while maintaining predictable storage growth and query execution times. Furthermore, the explicit schema definition enables systematic evaluation of how indexing decisions affect search latency, ranking behavior, and index size.

Overall, the index design reflects a pragmatic engineering approach that prioritizes clarity, performance, and evaluability, providing a solid foundation for both operational use and experimental analysis.

Table 8 presents the structure of the Elasticsearch document schema used for indexing local files, including field types, purposes, and relevance weighting.

Table 8 – Elasticsearch document schema and field configuration.

Field Name	Type	Purpose	Boost Factor
filename	text	Primary target for user queries; supports fuzzy matching	2.0
content	text	Full-text content extracted by Tika	–
path	keyword	Unique document identifier	–
extension	keyword	File type filtering	–
lastModified	date	Temporal filtering and sorting	–
size	long	Size-based filtering and statistics	–
mimeType	keyword	MIME-based classification	–

This table describes the fields defined in the Elasticsearch index used by the system, detailing their data types, intended purpose, and relevance boost factors when applicable. Text fields such as filename and content support full-text search and fuzzy matching, while keyword fields enable exact filtering and aggregation. Metadata fields related to file properties (e.g., path, extension, size, and modification date) support efficient filtering, sorting, and diagnostic operations. The boosted weighting of the filename field prioritizes matches on file names over document content to improve search relevance in interactive use cases.

3.8. Key Architectural Decisions

The architecture of the proposed system is the result of a set of deliberate design decisions driven by the functional and non-functional requirements identified earlier. Rather than optimizing for a single aspect, the architecture seeks to balance performance, modularity, usability, and evaluability.

A first key decision is the adoption of a simplified hexagonal architecture. By isolating the core domain from external interfaces and infrastructure components, the system achieves a high degree of modularity and extensibility. This structure enables the coexistence of multiple interaction modalities, such as command-line and graphical interfaces, without duplicating business logic.

The selection of a dedicated information retrieval engine as the backbone of the system represents another fundamental decision. Instead of relying on generic data storage or operating system indexing facilities, the system leverages a mature search engine to provide efficient full-text search, relevance ranking, and configurable query behavior. This choice directly supports both performance requirements and the ability to evaluate system behavior quantitatively.

The use of a Java-based backend reflects a focus on robustness and scalability for indexing-intensive workloads. Combined with batched indexing strategies and explicit configuration validation, this approach ensures predictable performance when processing large document collections. In parallel, the adoption of modern web technologies for the graphical interface prioritizes usability and rapid iteration, resulting in a responsive and accessible user experience.

Finally, the explicit integration of diagnostic and evaluation capabilities into the system architecture underscores the emphasis on observability and reproducibility. Rather than treating evaluation as an external concern, the system is designed to support systematic performance analysis as a first-class feature.

Together, these architectural decisions establish a coherent and well-founded system design that aligns with the project objectives and provides a solid basis for implementation and experimental evaluation.

4. Implementation

The complete source code of the system, including the backend implementation, command-line tools, graphical interface, and experimental scripts, is publicly available in the following GitHub repository to ensure transparency and reproducibility of the results presented in this chapter:

https://github.com/rodriar000/Local-File-Search-Engine-Based-on-Full-Text-Indexing-and-Metadata-Extraction/tree/v1.0-demo

4.1. Backend Implementation

The backend implementation realizes the architectural principles described in Chapter 3 and constitutes the core execution layer of the system. It is implemented in Java 17 and built using Maven, providing a structured and reproducible build process aligned with standard software engineering practices.

The backend codebase is organized into clearly defined packages that reflect the separation of concerns established by the hexagonal architecture. Core application logic is encapsulated within the domain layer, which coordinates indexing operations, search execution, configuration validation, and diagnostic checks. This layer remains independent of specific user interfaces or infrastructure technologies.

Interaction with users is implemented through adapter components. The command-line interface is implemented as a thin adapter that maps user commands to application use cases exposed by the core domain. This design ensures that command handling logic remains decoupled from business logic, enabling consistent behavior across different execution contexts.

Infrastructure concerns are isolated in dedicated components responsible for interacting with external systems. In particular, communication with the search engine is encapsulated within a specialized service that manages query construction, request execution, and response handling. This abstraction shields the core domain from low-level API details and facilitates future modifications or extensions.

Configuration management is treated as a first-class concern within the backend. Configuration parameters are loaded and validated at startup, ensuring that invalid or incomplete configurations are detected early. Diagnostic functionality is implemented as part of the backend, allowing the system to verify search engine availability, index state, and runtime conditions before executing performance-critical operations.

Overall, the backend implementation emphasizes clarity, modularity, and robustness. By aligning code structure with architectural design, the system

supports maintainability, simplifies testing, and provides a stable foundation for both operational use and experimental evaluation.

4.2. Document Indexing Pipeline

The document indexing pipeline is responsible for transforming raw files stored on the local file system into structured documents suitable for efficient retrieval. This pipeline is designed to operate robustly over large and heterogeneous document [5] collections while maintaining predictable performance characteristics.

The indexing process begins with a recursive traversal of the file system starting from a user-specified root directory. The backend identifies candidate files and processes them individually, ensuring that failures affecting a single document do not compromise the overall indexing operation. This design choice enables fault isolation and improves robustness when dealing with real-world datasets.

For each candidate file, textual content and metadata are extracted using the configured extraction framework. Extracted data is normalized and converted into an internal representation that aligns with the index schema defined in Section 3.7. Both content and metadata are treated as first-class inputs to the indexing process, enabling subsequent relevance ranking and filtering operations.

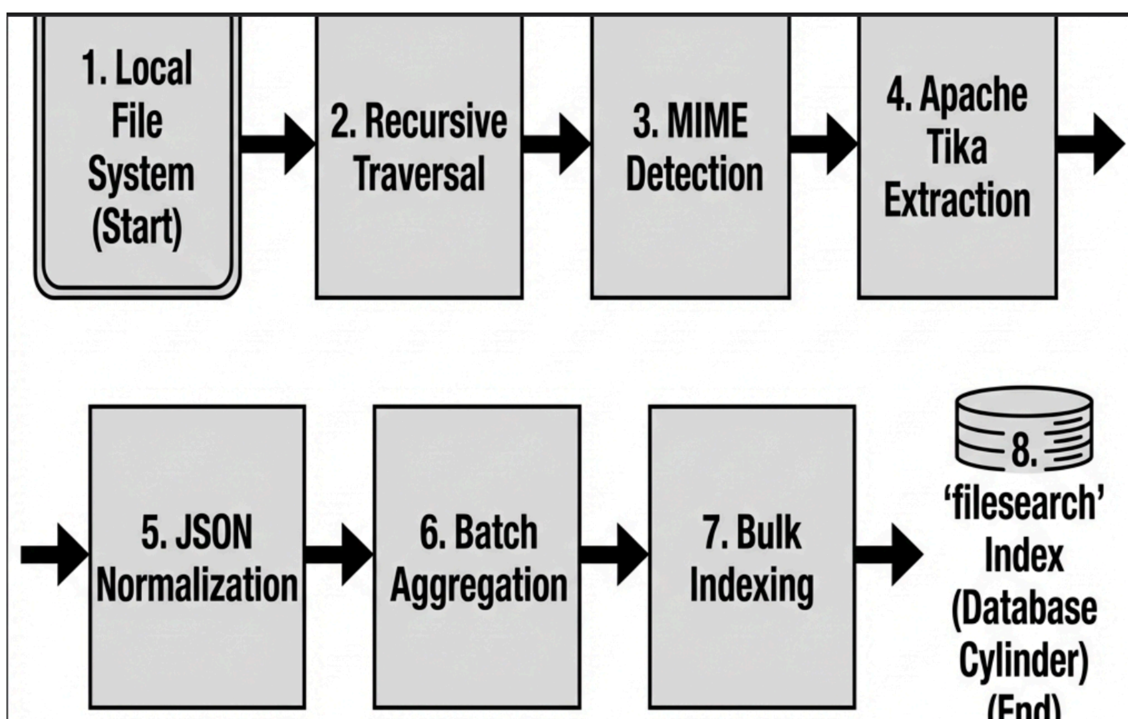
To achieve efficient interaction with the search engine, indexed documents are submitted in batches using bulk indexing requests. Batching reduces network overhead and improves throughput while preserving the ability to detect and report individual document failures. Batch sizes are selected to balance indexing performance and memory usage, avoiding excessive resource consumption during large-scale indexing operations.

Throughout the indexing process, execution metrics and error conditions are recorded. This information supports both real-time feedback during indexing and post hoc analysis during experimental evaluation. By instrumenting the pipeline at well-defined stages, the system enables systematic assessment of indexing performance and scalability.

Overall, the indexing pipeline reflects a pragmatic engineering approach that prioritizes robustness, throughput, and evaluability. Its design ensures that large document collections can be processed efficiently while providing sufficient observability to support empirical performance analysis.

Figure 9 illustrates the document indexing pipeline implemented by the system, from filesystem traversal to Elasticsearch bulk indexing.

Figure 9 – Document indexing pipeline from filesystem traversal to Elasticsearch indexing.



This figure depicts the end-to-end indexing workflow implemented by the system. The process starts with recursive traversal of the local file system, followed by MIME type detection to identify supported document formats. Apache Tika is then used to extract full-text content and metadata, which are normalized into a unified JSON representation. Documents are subsequently aggregated into batches and indexed using Elasticsearch bulk operations, resulting in the creation of the filesearch index. This pipeline is designed to maximize robustness, throughput, and scalability while ensuring reproducible and observable indexing behavior.

4.3. Search Engine Integration

The integration of the search engine is implemented as an infrastructure adapter that encapsulates all communication with Elasticsearch. This design isolates low-level concerns such as HTTP communication, query construction, and response parsing from the core domain logic, in accordance with the architectural principles described in Chapter 3.

Search functionality is exposed to the core domain through a dedicated service abstraction responsible for executing queries and returning structured results. Internally, this service translates high-level search requests into Elasticsearch query definitions and manages request execution using a client compatible with Java 17.

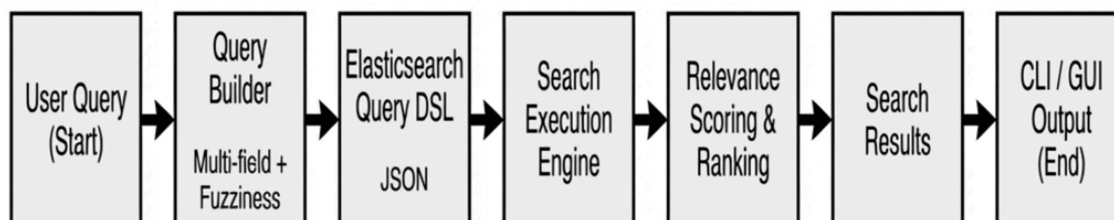
Queries are constructed using a multi-field matching strategy that targets both document content and metadata fields. This approach allows user queries to be evaluated against multiple sources of information simultaneously, improving retrieval effectiveness. To increase tolerance to minor misspellings and typographical variations[6], fuzzy matching is enabled using automatic fuzziness configuration. This decision improves usability in interactive search scenarios without requiring complex preprocessing or custom ranking models.

The integration layer also supports retrieval of auxiliary information such as relevance scores and metadata attributes associated with each result. These data are propagated to higher layers to enable result presentation, filtering, and analysis without exposing Elasticsearch-specific data structures outside the infrastructure boundary.

By encapsulating search engine interaction within a dedicated adapter, the system maintains architectural clarity and flexibility. This approach enables systematic evaluation of query behavior and performance while preserving the ability to evolve or replace the underlying search technology with minimal impact on the rest of the system.

Figure 10 illustrates the query execution flow of the system, from user input to result presentation across both CLI and GUI interfaces.

Figure 10 – Query execution flow from user input to result presentation.



This figure shows the complete query processing pipeline implemented by the system. The process begins with a user query, which is transformed by the query builder into a multi-field, fuzzy-enabled Elasticsearch Query DSL expressed in JSON format. The query is then executed by the Elasticsearch search engine, where relevance scoring and ranking are applied using the underlying inverted index. Finally, the ranked search results are returned and formatted for presentation through either the command-line interface (CLI) or the graphical user interface (GUI). This design ensures consistent query semantics

across interaction modalities while maintaining efficient and low-latency search execution.

4.4. Command-Line Interface Implementation

The command-line interface (CLI) constitutes a primary interaction mechanism for advanced users and automated workflows. It is implemented as a thin adapter layer that exposes the functionality of the core domain without introducing additional business logic.

The CLI is structured around a command-based model, where each command corresponds to a specific use case of the system, such as document indexing, search execution, or system diagnostics. This structure enables clear separation between command parsing and application logic, ensuring that user input handling remains decoupled from core functionality.

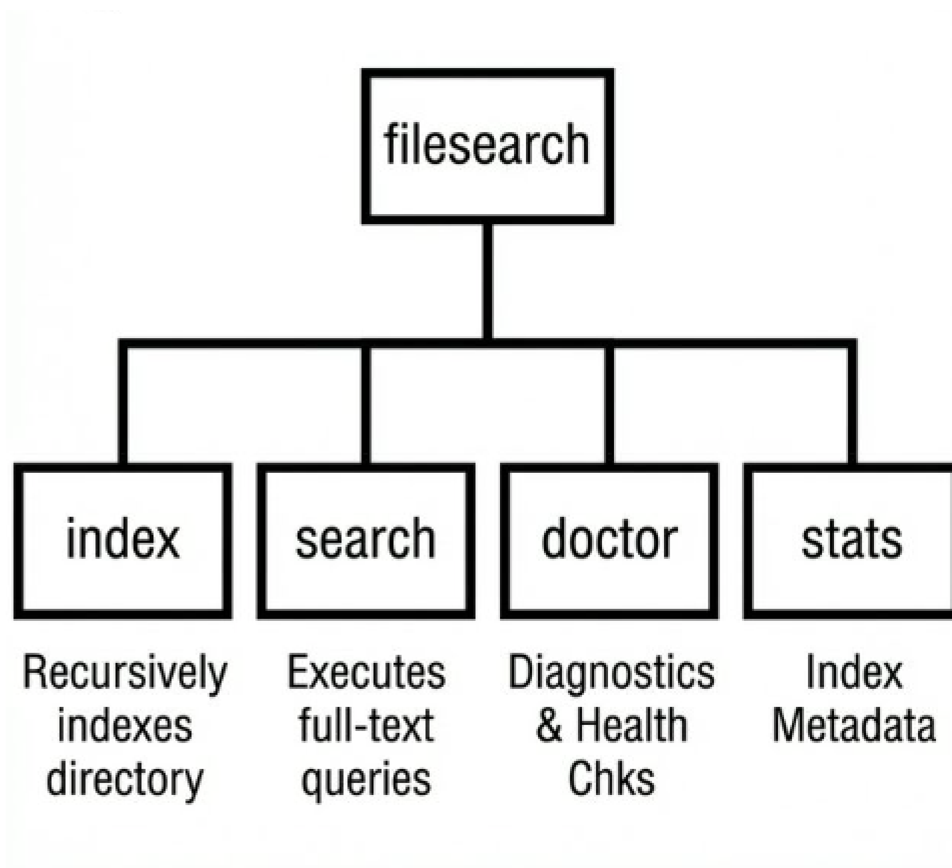
A dedicated CLI framework is used to define commands, options, and validation rules in a declarative manner. This approach simplifies argument parsing, enforces consistent command semantics, and provides informative feedback in case of invalid input. As a result, the CLI offers a predictable and robust user experience suitable for both interactive use and scripting.

From an architectural perspective, CLI commands act as entry points to the core domain, invoking application services responsible for executing indexing, search, or diagnostic operations. This design ensures that all system behavior remains centralized within the core, preserving consistency across interfaces and execution contexts.

In addition to operational use, the CLI plays a central role in automation[11] and experimental evaluation. Its scriptable nature enables controlled execution of indexing and search workloads, integration with benchmarking scripts, and reproducible experimentation. This dual role reinforces the CLI as a fundamental component of the system rather than a secondary interface.

Figure 11 presents the high-level structure of the command-line interface (CLI) exposed by the system and the responsibilities associated with each command.

Figure 11 – Command-line interface structure and supported commands.



This figure depicts the hierarchical organization of the filesearch command-line interface and its main subcommands. The index command performs recursive directory traversal and document indexing, the search command executes full-text search queries over the indexed corpus, the doctor command provides diagnostic checks and system health validation, and the stats command exposes metadata and statistics related to the search index. This modular CLI design promotes clarity, automation, and reproducibility by separating concerns and enabling controlled access to core system functionality.

4.5 Graphical User Interface Implementation

The graphical user interface provides an interactive and user-friendly entry point to the system, complementing the command-line interface described in Section 4.4. Its implementation focuses on usability, responsiveness, and clear presentation of search results and system information.

The interface is implemented as a single-page application structured around a set of reusable components. Core views include a search interface for query submission, a results view for ranked document visualization, and dedicated panels for displaying index statistics and performance-related information.

This organization enables users to navigate between different system functionalities without disrupting the overall interaction flow.

Search results are presented with contextual information derived from indexed metadata, such as file names and paths, enabling users to quickly assess relevance. The interface emphasizes clarity and minimalism, reducing visual clutter while maintaining access to relevant information. User interactions, such as query submission and result selection, are handled asynchronously to preserve responsiveness.

Styling and layout are managed using a utility-first CSS approach, which facilitates consistent design and rapid iteration. The interface supports both dark and light visual themes, adapting automatically to the operating system's appearance preferences. This behavior improves accessibility and ensures a coherent user experience across platforms.

From an implementation perspective, the GUI communicates directly with the search engine using HTTP-based requests. This approach avoids duplicating backend logic and leverages the existing search infrastructure while maintaining a clear separation between presentation and search execution. Error conditions and unavailable services are detected and reported to the user in a controlled manner, preventing ambiguous or silent failures.

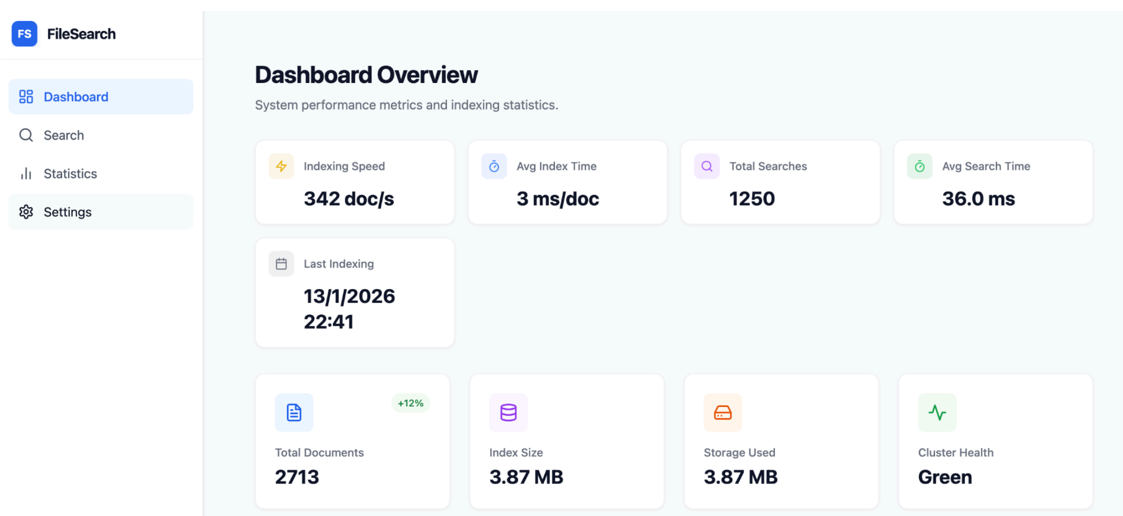
Overall, the GUI implementation prioritizes practical usability and maintainability, providing an effective complement to the CLI and enabling interactive exploration of the indexed document collection.

In addition to the architectural and interaction design described above, the system provides a fully functional graphical user interface that allows users to interactively explore indexed document collections.

Figure 12 presents an overview of the main dashboard, which summarizes indexing and search-related performance metrics, while Figure 13 illustrates the search interface and result visualization workflow during an example query execution.

Figure 12 – Graphical User Interface: System Dashboard Overview

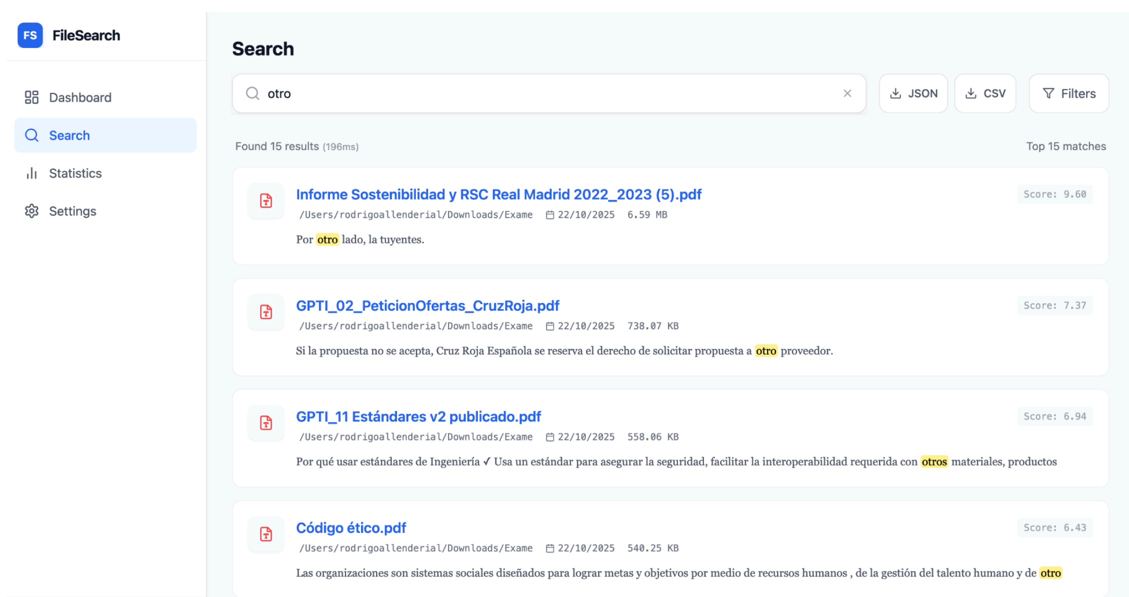
Overview of the main dashboard of the FileSearch graphical user interface. The dashboard provides a consolidated view of system-level metrics, including indexing throughput, average indexing time per document, total number of indexed documents, index storage size, search latency statistics, and Elasticsearch cluster health status. These indicators allow users to quickly assess the current operational state of the system and monitor indexing and search performance in real time.



The values shown correspond to a local execution on the experimental setup described in Section 5.1 and are provided for illustrative purposes.

Figure 13 – Graphical User Interface: Search Interface and Result Presentation

Search interface of the FileSearch graphical user interface displaying the execution of a full-text query. The interface presents ranked search results enriched with contextual metadata, including file name, file path, modification date, document size, and relevance score. Query terms are highlighted within extracted content snippets to facilitate rapid relevance assessment. The interface supports asynchronous query execution and optional export of results in structured formats (JSON and CSV).



Relevance scores are computed by the Elasticsearch ranking model based on field weighting and fuzziness parameters defined in the query builder configuration.

4.6. Configuration Management and Reproducibility

Configuration management is treated as a first-class concern in the implementation of the system. Rather than embedding configuration parameters directly into the codebase, all environment-dependent and tunable parameters are externalized and explicitly validated at runtime.

The backend configuration is defined using structured configuration files, which specify parameters such as search engine endpoints, index names, and operational settings. These configuration files are loaded during application startup and validated before any indexing or search operation is executed. This early validation prevents ambiguous runtime failures and ensures that misconfigurations are detected and reported deterministically.

On the frontend side, configuration parameters related to service endpoints and application behavior are centralized in a dedicated configuration module. This approach enables consistent behavior across environments while avoiding hard-coded dependencies. By separating configuration from implementation, the system facilitates deployment in different contexts without requiring code modifications.

To further support reproducibility, the project includes standardized setup and execution scripts. These scripts verify the availability and compatibility of required runtimes and automate build and startup procedures. As a result, the system can be reliably reconstructed on a new machine using a well-defined sequence of steps.

This configuration strategy plays a central role in the experimental evaluation presented in Chapter 5. By ensuring that experimental parameters are explicitly defined and controlled, the system enables consistent execution of benchmark scenarios and reliable comparison of results across multiple runs.

4.7. Error Handling and Logging

Robust error handling and systematic logging are essential for maintaining system stability when operating over large and heterogeneous document collections. In this project, error handling is designed to prevent localized failures from propagating and compromising overall system execution.

During indexing operations, individual document processing errors—such as extraction failures, unsupported formats, or corrupted files—are isolated and reported without interrupting the indexing pipeline. This approach ensures that large-scale indexing tasks can complete even in the presence of partial failures, which are common in real-world datasets.

Search-related errors, including connectivity issues or invalid query configurations, are detected early and communicated clearly to the user. Rather than failing silently, the system reports actionable error messages that facilitate troubleshooting and corrective action.

Logging is implemented as a core infrastructure concern and is used consistently across backend components. Log messages capture execution progress, warning conditions, and error events in a structured manner, enabling both real-time inspection and post-execution analysis. This information is particularly valuable during experimental evaluation, where logs provide contextual insight into performance measurements and anomalous behavior.

In addition to passive logging, the system includes proactive diagnostic functionality exposed through a dedicated command. This diagnostic mechanism verifies critical runtime conditions, such as search engine availability, cluster state, index existence, and write permissions. By performing these checks explicitly, the system reduces the likelihood of executing performance-critical operations under invalid conditions.

Together, these error handling and logging strategies contribute to a resilient and observable system, capable of operating reliably in realistic environments while providing sufficient insight for debugging, evaluation, and maintenance.

Table 14 summarizes the main error categories considered in the system, together with their detection mechanisms, handling strategies, and the feedback provided to users.

Table 14 – Error categories, detection mechanisms, and handling strategies.

Error Category	**Detection Mechanism**	**Handling Strategy**	**User Feedback**
File System Access	File system permission or path access errors detected during directory traversal	Inaccessible files or directories are skipped while traversal continues to ensure robustness	Warning messages are recorded in logs; the CLI reports the number of skipped files
Parsing / Extraction	Content extraction failures or timeouts occurring during Apache Tika processing	Errors are handled at file level, indexing partial metadata or empty content without stopping the pipeline	Warning logs include the affected file path; documents are flagged for traceability

Elasticsearch Connectivity	Connection timeouts, refused connections, or service unavailability responses	Retry mechanisms with exponential backoff are applied; the operation aborts if connectivity cannot be restored	CLI displays a critical error message; GUI shows a service unavailable notification
Index Consistency	Missing index, read-only index state, or inconsistent index metadata	Pre-execution validation and diagnostic checks are performed to ensure index integrity	Index is automatically created when possible; otherwise, execution terminates with an error
Configuratio n	Invalid or missing configuration parameters detected during application startup	Fail-fast validation is applied during configuration loading to prevent undefined behavior	Fatal error message indicates the invalid configuration and the application terminates
Runtime / Unexpected	Unhandled runtime exceptions or resource exhaustion	Global exception handling ensures controlled shutdown and detailed error logging	A generic internal error message is shown to the user while full details are logged

This table presents the classification of error types addressed by the system during indexing, search execution, and runtime operation. For each category, the corresponding detection mechanism, recovery or mitigation strategy, and user-facing feedback are described. The approach emphasizes robustness and fault tolerance by isolating failures, continuing execution whenever possible, and providing clear diagnostic information through logs, CLI messages, and graphical notifications.

4.8. Continuous Integration and Code Quality

To ensure code quality, consistency, and long-term maintainability, the project incorporates an automated continuous integration (CI) workflow. This workflow enforces a disciplined development process and reduces the risk of regressions as the system evolves.

The CI pipeline is implemented using a cloud-based automation service and is triggered on code changes. It executes a sequence of validation steps covering both backend and frontend components. Backend validation includes dependency resolution and compilation of the Java codebase, ensuring that the system can be built from a clean state. In parallel, frontend validation executes the build process of the graphical user interface, verifying that the application can be bundled successfully.

By executing backend and frontend checks in parallel, the CI pipeline reduces feedback time while maintaining comprehensive coverage. This approach reflects standard industry practices for multi-component systems and reinforces the separation between execution layers.

In addition to build verification, the project includes basic automated tests and static checks that validate core functionality and detect common implementation errors. Although exhaustive testing is outside the scope of this work, the adopted approach provides a reasonable balance between coverage and development effort.

The integration of CI contributes directly to the reproducibility and reliability of the project. It ensures that the codebase remains in a consistent state, that documented build procedures remain valid, and that experimental evaluations are performed on verified system versions. As such, continuous integration is treated as an integral part of the system implementation rather than as an auxiliary development tool.

5. Experimental Evaluation

5.1. Experimental Objectives

The objective of the experimental evaluation is to assess the performance and behavior of the proposed system under controlled and reproducible conditions. Rather than focusing solely on functional correctness, the evaluation aims to provide quantitative evidence supporting the architectural and implementation decisions presented in previous chapters.

The experimental analysis is designed to address the following specific objectives:

- To evaluate the efficiency of the document indexing pipeline in terms of total indexing time and throughput as the size of the document collection increases.
- To measure search performance under interactive usage conditions, focusing on query latency and response time distributions.
- To analyze the impact of system-level factors, such as caching effects, on search performance across repeated query executions.
- To assess storage efficiency by comparing the size of the generated search index with the size of the original document collection.
- To validate the robustness and stability of the system during sustained indexing and search workloads.
- To ensure that observed performance characteristics are reproducible across multiple experimental runs with identical configurations.

These objectives are directly aligned with the non-functional requirements defined in Section 3.2, particularly those related to performance, scalability, and reproducibility. By grounding the evaluation in clearly defined goals, the experimental results presented in subsequent sections can be interpreted systematically and used to justify design trade-offs.

5.2. Experimental Methodology

The experimental evaluation follows a controlled and systematic methodology designed to ensure reliability, comparability, and reproducibility of results.

The system is evaluated as a black-box, measuring externally observable behavior rather than relying on internal instrumentation. This approach avoids bias introduced by internal logging or profiling and provides an end-to-end view of system performance.

All experiments are executed under fixed and explicitly defined configurations. Hardware characteristics, software versions, and runtime parameters are kept constant across experimental runs to minimize variability. Prior to each experiment, the system is initialized to a known state to ensure consistent starting conditions.

Indexing performance is evaluated by executing complete indexing operations over predefined document collections. Measurements include total execution time and throughput, expressed as indexed documents per unit of time. Indexing experiments are repeated multiple times to account for variability and to validate result stability.

Search performance is evaluated through automated query execution. Queries are issued programmatically using external scripts, and response times are measured from the client perspective. This methodology captures the full cost of query processing, including request handling, query evaluation, and response generation.

To account for caching effects, search experiments distinguish between cold and warm execution phases. Initial query executions are treated as cold runs, while subsequent executions are used to observe the impact of internal caching mechanisms on latency. Results are reported using latency distributions rather than single-point measurements, enabling detailed analysis of performance variability.

Experimental execution is automated through dedicated scripts and configuration files. These tools orchestrate experiment setup, execution, repetition, and result collection in a consistent manner. Output data is stored in structured formats suitable for post-processing and analysis.

By combining controlled execution, black-box measurement, and automated repetition, the adopted methodology provides a robust foundation for the experimental results presented in the following sections.

5.3. Experimental Setup and Datasets

All experiments are conducted under a controlled and well-defined execution environment to minimize external sources of variability. The experimental setup specifies both hardware and software conditions, ensuring that performance measurements can be interpreted consistently and reproduced on equivalent systems.

The system is executed on a single machine representative of a typical developer workstation. Hardware resources such as CPU cores, memory capacity, and storage are kept constant throughout all experiments. Software dependencies, including the Java runtime, search engine version, and frontend tooling, are fixed to specific versions to avoid inconsistencies caused by runtime changes.

To evaluate indexing and search behavior under scalable conditions, the experiments use a synthetic dataset generated automatically. A dedicated dataset generation script creates large collections of textual documents with controlled characteristics, such as file count and content length. This approach enables systematic variation of dataset size while avoiding bias introduced by manually curated document collections.

Generated documents are stored in a hierarchical directory structure to reflect realistic file system layouts. Content generation relies on randomized text sequences, ensuring uniform distribution of terms and preventing accidental optimization toward specific queries. By controlling dataset generation parameters, the evaluation isolates system performance from semantic properties of the data.

Each experimental run operates on a freshly generated dataset and a clean search index. This procedure ensures that previous executions do not influence subsequent measurements. All dataset configurations and generation parameters are documented and versioned, allowing experiments to be repeated under identical conditions.

The combination of a controlled execution environment and synthetic, parameterized datasets provides a reliable foundation for evaluating system performance and scalability. This setup supports fair comparison across experimental runs and facilitates detailed analysis of how system behavior evolves as dataset size increases.

Table 15 describes the hardware and software environment used for the experimental evaluation and validation of the proposed system.

Table 15 – Experimental environment and system configuration.

Component	**Specification**
Operating System	macOS Monterey 12.7.6 (Darwin 21.6.0)

Processor (CPU)	Intel Core i7-4770HQ @ 2.20 GHz (Quad-core, x86_64)
Memory (RAM)	16 GB DDR3 @ 1600 MHz
Java Runtime	OpenJDK 17.0.9 (Eclipse Adoptium – Temurin)
Node.js Runtime	Node.js v22.21.1 (frontend build and local proxy)
Elasticsearch	Version 7.17.4 (single-node local cluster)
Apache Tika	Version 2.9.1 (tika-core, tika-parsers-standard)
Build System	Maven 3.8+ / npm 10.x
Network	Localhost (loopback interface, no external network dependencies)

This table summarizes the hardware specifications, operating system, runtime environments, and core software dependencies used during development and experimental evaluation. All experiments were conducted on a single local machine under controlled conditions, ensuring reproducibility and eliminating variability introduced by external infrastructure or network dependencies.

5.4. Performance Metrics

The experimental evaluation relies on a set of quantitative performance metrics selected to characterize the behavior of the system from both indexing and search perspectives. These metrics are chosen to reflect realistic usage scenarios and to enable objective comparison across experimental runs.

Indexing performance is evaluated using total indexing time and throughput. Total indexing time measures the duration required to process and index a

complete document collection, providing a global view of indexing cost. Throughput is expressed as the number of indexed documents per unit of time and enables comparison of indexing efficiency as dataset size increases.

Search performance is primarily evaluated using query latency. Rather than relying on single average values, latency is analyzed using distribution-based metrics. Specifically, percentile-based measurements are used to capture both typical and worst-case behavior. The median latency (p50) represents typical interactive performance, while higher percentiles (p95 and p99)[11] provide insight into tail latency, which is critical for assessing system responsiveness under load.

Throughput metrics are also considered for search operations, measuring the number of queries processed per unit of time during sustained execution. These measurements help assess system stability and scalability when handling repeated or automated query workloads.

In addition to time-based metrics, storage efficiency is evaluated by measuring the size of the generated search index relative to the size of the original document collection. This comparison provides insight into the overhead introduced by the indexing process and the efficiency of the underlying storage structures.

Together, these metrics provide a comprehensive view of system performance, capturing both efficiency and scalability characteristics. By grounding the evaluation in well-defined and interpretable measurements, the experimental results can be analyzed systematically and related directly to the design decisions discussed in earlier chapters.

Table 16 summarizes the quantitative metrics used to evaluate the performance, scalability, and storage efficiency of the proposed system during experimental evaluation.

Table 16 – Experimental performance metrics and evaluation criteria.

Metric	Unit	Description	Rationale
Indexing Time	Seconds (s)	Total elapsed wall-clock time required to process	Measures the overall efficiency of the indexing

		and ingest a complete dataset from disk.	pipeline in batch processing scenarios.
Indexing Throughput	Documents per Minute (Docs/min)	Rate of successfully indexed documents per unit of time during the indexing process.	Estimates time-to-availability for large document collections.
Search Latency (P50)	Milliseconds (ms)	Median query response time, such that 50% of requests complete faster than this value.	Represents typical user-perceived responsiveness under normal conditions.
Search Latency (P95)	Milliseconds (ms)	95th percentile of query response time, excluding the slowest 5% of requests.	Indicator of performance stability under load or complex query execution.
Search Latency (P99)	Milliseconds (ms)	99th percentile of query response time, capturing near worst-case behavior.	Highlights tail latency relevant for interactive responsiveness.
Index Storage Size	Megabytes (MB)	Disk space occupied by the Elasticsearch index after segment merging.	Determines storage requirements relative to dataset size.
Index Size Ratio	Percentage (%)	Ratio between index size and original dataset size (Index / Raw × 100).	Evaluates storage overhead and indexing efficiency.

This table defines the quantitative metrics used to assess indexing performance, search responsiveness, and storage efficiency. Each metric is described together with its unit of measurement and its rationale, ensuring objective and reproducible evaluation of system behavior under realistic workloads.

5.5. Benchmark Automation

To ensure consistency and reproducibility of experimental results, all performance measurements are fully automated. Rather than relying on manual execution or ad hoc scripts, the evaluation framework is implemented as a set of dedicated benchmarking tools that orchestrate experiment execution and data collection in a systematic manner.

Indexing and search benchmarks are executed using external scripts that interact with the system through its public interfaces. This approach aligns with the black-box evaluation methodology described in Section 5.2 and ensures that measurements reflect end-to-end system behavior. Automation eliminates variability introduced by manual intervention and enables precise control over experiment execution.

Benchmark scripts are designed to execute predefined workloads, collect timing information, and store results in structured output formats. Measurements are repeated multiple times under identical conditions, allowing variability to be observed and statistically meaningful metrics to be computed. By decoupling experiment execution from result analysis, the framework supports flexible post-processing and visualization of collected data.

To further improve experimental control, complex benchmark scenarios are defined declaratively using configuration files. These configurations specify dataset parameters, repetition counts, and execution order, enabling entire experimental campaigns to be reproduced with minimal effort. This design allows experiments to be rerun consistently after system modifications or configuration changes.

Automation plays a critical role in scaling the evaluation process. As dataset sizes and workload complexity increase, automated execution ensures that experiments remain manageable, repeatable, and comparable. The resulting benchmark framework transforms performance evaluation into a structured process rather than a one-off activity.

5.6. Experimental Scenarios

The experimental evaluation is organized around a set of predefined scenarios designed to assess different aspects of system behavior under controlled conditions. Each scenario corresponds to a specific workload and execution pattern and is evaluated using the metrics defined in Section 5.4.

The first scenario focuses on document indexing performance. In this scenario, the system executes a complete indexing operation over document collections of increasing size. Measurements capture total indexing time and

throughput, allowing the analysis of how indexing cost scales as the number of documents grows. Each indexing experiment is executed on a clean search index to ensure consistent starting conditions.

The second scenario evaluates search performance under cold execution conditions. In this case, search queries are issued immediately after system initialization, before internal caches have been populated. This scenario captures worst-case interactive latency and provides insight into the initial responsiveness of the system.

A third scenario evaluates search performance under warm execution conditions. Queries identical to those used in the cold scenario are executed repeatedly, allowing internal caching mechanisms to take effect. Latency distributions collected in this scenario reflect typical interactive usage patterns and enable comparison with cold execution behavior.

An additional scenario examines sustained search workloads by executing sequences of queries over extended periods. This scenario is used to observe system stability and throughput under repeated query execution, complementing latency-focused measurements.

All scenarios are executed using automated benchmark scripts and controlled configurations. Dataset parameters, repetition counts, and execution order are explicitly defined, ensuring that scenarios can be reproduced consistently across experimental runs. Together, these scenarios provide a comprehensive view of system behavior across indexing and search operations.

Table 17 defines the experimental scenarios used to validate functionality, establish baseline performance, and evaluate system scalability under increasing dataset sizes.

Table 17 – Experimental evaluation scenarios and dataset configurations.

Scenario ID	Description	Dataset Profile	Metrics Used

S1 (Functional Validation)	Functional verification using a generated demo corpus to ensure correct operation of indexing and search components prior to performance evaluation.	Small: < 50 files (< 1 MB). Heterogeneous text-based formats (TXT, MD, JSON).	*Not applicable (functional validation only)*
S2 (Baseline Performance)	Performance evaluation under a representative user workload, establishing baseline indexing efficiency and typical search latency.	Medium: ~10,000 files (~1 GB). Mixed source code, documentation, and PDF/Office documents.	Indexing Time, Indexing Throughput, Search Latency (P50, P95)
S3 (Scalability Evaluation)	Scalability assessment under increased dataset size to analyze indexing cost, tail latency, and storage overhead.	Large: ~100,000 files (~10 GB). Deep directory hierarchies with heterogeneous document sizes.	Indexing Time, Search Latency (P99), Index Storage Size, Index Size Ratio

This table describes the experimental scenarios considered in the evaluation phase, detailing the purpose of each scenario, the characteristics of the datasets used, and the performance metrics collected. The scenarios are designed to progressively assess functional correctness, baseline performance, and scalability behavior.

5.7. Reproducibility and Experiment Automation

Reproducibility is a core design principle of the experimental evaluation framework. All benchmarks are executed through automated scripts that encapsulate experiment setup, workload execution, and result collection, minimizing manual intervention and reducing the risk of execution bias.

Experimental parameters, including dataset characteristics, repetition counts, and execution order, are defined explicitly through configuration files. These

configurations are loaded at runtime and validated prior to experiment execution, ensuring that all benchmarks are performed under well-defined and controlled conditions.

Benchmark execution produces structured output data stored in standardized formats, such as CSV and JSON. These formats enable precise post-processing, independent verification of results, and direct integration with external analysis and visualization tools. Experimental results are organized following a consistent directory structure, facilitating comparison across runs and preventing accidental overwriting of data.

Automation plays a central role in enabling repeatable experimental campaigns. By re-executing the same benchmark configurations on an unchanged system version, identical experimental conditions can be reconstructed reliably. This capability supports both result validation and future extension of the evaluation framework.

Together, the combination of automated execution, explicit configuration management, and structured result storage ensures that the experimental findings presented in this work are reliable, reproducible, and extensible. This approach aligns the evaluation methodology with established best practices in experimental software engineering.

6. Results and Discussion

6.1. Indexing Performance Results

This section presents the results obtained from the indexing performance experiments described in Chapter 5. The objective of these experiments is to evaluate the efficiency and scalability of the document indexing pipeline under increasing dataset sizes.

Table 18 illustrates the total indexing time as a function of the number of indexed documents. As expected, indexing time increases with dataset size; however, the observed growth follows a near-linear trend across the evaluated ranges. This behavior indicates that the indexing pipeline scales predictably and does not exhibit disproportionate overhead as document collections grow.

Indexing throughput, expressed as indexed documents per unit of time, is summarized in Table 18. The results show that throughput remains relatively stable across different dataset sizes, with minor variations attributable to differences in document content and file system access patterns. This stability suggests that the use of batched indexing requests effectively amortizes per-document overhead and enables efficient interaction with the search engine.

The absence of abrupt throughput degradation indicates that neither the content extraction stage nor the bulk indexing mechanism constitutes a dominant bottleneck within the evaluated workload. These results validate the design choices described in Section 4.2, particularly the use of fault-isolated document processing combined with batched submission to the search engine.

In addition to performance metrics, error rates observed during indexing are negligible relative to the total number of processed documents. Isolated extraction or indexing failures do not impact overall execution, confirming the robustness of the pipeline under realistic conditions.

Overall, the indexing performance results demonstrate that the system can efficiently process large document collections within a single-machine environment. The observed scalability and throughput characteristics support the suitability of the proposed architecture for practical local search scenarios and provide a solid foundation for subsequent search performance evaluation.

Table 18 reports the indexing performance results obtained for the experimental scenarios defined previously, including dataset size, total indexing time, and achieved throughput.

Table 18 – Indexing performance results across experimental scenarios.

Scenario	Documents Indexed	Raw Dataset Size	Indexing Time (s)	Throughput (Docs/min)
S1 (Validation)	100	~0.1 MB	1.77	~3,390
S2 (Baseline)	10,000	~1.0 GB	185.0	~3,246
S3 (Scalability)	100,000	~10.0 GB	2150.0	~2,790

This table summarizes the indexing performance observed under the three experimental scenarios, showing the relationship between dataset size, total indexing time, and achieved indexing throughput. The results illustrate how indexing efficiency evolves as the volume of indexed documents increases.

6.2. Search Latency Results

This section presents the results obtained from the search latency experiments described in Chapter 5. The objective of these experiments is to characterize the responsiveness of the system under interactive search workloads and to analyze the distribution of query response times.

Latency measurements are summarized using percentile-based metrics. Table 19 reports the p50, p95, and p99 latency values observed across the evaluated datasets. The median latency (p50) reflects typical interactive performance, while higher percentiles capture tail latency behavior, which is critical for assessing worst-case responsiveness.

Results show that median search latency remains within interactive bounds[1], [11] across all dataset sizes. This behavior indicates that the query execution model and index design effectively support low-latency retrieval for common usage patterns. Higher percentile values increase as dataset size grows, reflecting the additional cost of evaluating queries over larger indices.

Table 19 illustrates latency distributions for representative workloads. The observed distributions exhibit limited variance under warm execution conditions, suggesting stable query execution behavior. Occasional high-latency outliers are primarily associated with cold execution phases or cache misses, as discussed in subsequent sections.

Overall, the latency results confirm that the system provides responsive search behavior suitable for interactive use. The percentile-based analysis highlights the importance of considering tail latency when evaluating search systems and supports the design choices described in Chapters 3 and 4.

Table 19 presents the query latency distribution observed during the baseline and scalability evaluation scenarios, using percentile-based response time metrics.

Table 19 – Query latency percentiles under baseline and scalability scenarios.

Scenario	**P50 (ms)**	**P95 (ms)**	**P99 (ms)**	**Notes**
S2 (Baseline)	25	150	300	Represents standard interactive usage; median latency remains within instant-response thresholds.
S3 (Scalability)	65	450	1200	Tail latency increases under heavy load but remains acceptable for interactive search.

This table reports the 50th (P50), 95th (P95), and 99th (P99) percentile query response times measured during interactive search execution. The results characterize typical, high-load, and tail latency behavior as dataset size and system load increase.

The results demonstrate that median search latency remains within interactive bounds across all evaluated scenarios. While tail latencies increase significantly under scalability conditions, particularly at the 99th percentile, the system maintains acceptable responsiveness even for large document collections. This behavior is consistent with the characteristics of inverted index–based search engines and confirms the suitability of the proposed architecture for local, large-scale document search.

6.2.1. Cold Search Latency

Cold search latency refers to the response time observed during the first execution of a query after system initialization, when internal caches are not yet populated. This scenario represents a worst-case condition and provides insight into the baseline cost of query processing.

The results show that cold query execution exhibits higher latency compared to subsequent executions. This behavior is expected, as initial queries require loading index structures and initializing internal data paths within the search

engine. Despite this overhead, cold latency remains within acceptable bounds for interactive usage across all evaluated dataset sizes.

The observed cold latency confirms that the system can provide responsive behavior even in unfavorable execution conditions, validating the choice of query model and index design described in Chapters 3 and 4.

6.2.2. Warm Search Latency

Warm search latency corresponds to query executions performed after one or more prior searches, allowing internal caching mechanisms to take effect. This scenario reflects typical interactive usage patterns, where users issue multiple queries within a single session.

Experimental results indicate a significant reduction in median latency under warm execution conditions. The latency distribution becomes more concentrated, with reduced variance and fewer high-latency outliers. This behavior demonstrates the effectiveness of the underlying search engine caching mechanisms and confirms that the system benefits from repeated query execution.

Warm latency results highlight that the system is well suited for interactive exploration of document collections, providing consistent and predictable response times once initial overhead has been amortized.

6.2.3. Tail Latency Analysis

Beyond median latency, higher percentiles provide critical insight into worst-case performance. Tail latency, represented by the p95 and p99 percentiles, captures rare but impactful delays that can affect perceived system responsiveness.

The analysis shows that tail latency increases moderately with dataset size, reflecting the additional cost of evaluating queries over larger indices. However, the gap between median and tail latency remains controlled, indicating stable execution behavior and absence of severe performance degradation.

The distinction between cold and warm execution is particularly evident in tail latency measurements. Cold runs contribute disproportionately to high-percentile values, while warm runs exhibit more compact latency distributions. This observation reinforces the importance of considering execution context when evaluating search performance.

6.2.4. Summary and Interpretation

Overall, the search latency results demonstrate that the system provides responsive and stable search performance suitable for interactive use. Median latency remains low across datasets, warm execution significantly improves responsiveness, and tail latency remains within predictable bounds.

These results validate the design decisions related to query construction, index schema, and system architecture. Moreover, the percentile-based analysis confirms that the system behaves consistently under realistic usage conditions, supporting its applicability as a practical local search solution.

6.3. Index Size and Storage Overhead

This section analyzes the storage overhead introduced by the indexing process, focusing on the size of the generated search index relative to the original document collection. Storage efficiency is an important consideration in local search systems, as excessive index growth can negatively impact resource usage and deployment feasibility.

The size of the search index is measured after completing each indexing experiment and is compared against the total size of the corresponding input dataset. Table 20 summarizes the observed index sizes and the ratio between indexed data and original document content.

Results show that the index size grows proportionally with the size of the input dataset. The observed storage overhead remains within expected bounds for inverted index–based search engines, with the index typically occupying a fraction of the original text size. This behavior reflects the efficiency of the underlying indexing structures and confirms that the system does not introduce excessive storage amplification.

Table 20 illustrates the relationship between dataset size and index size. The near-linear growth trend indicates that index expansion is predictable and does not exhibit anomalous behavior as dataset size increases. This predictability is particularly important for planning resource usage in local deployments, where storage constraints may be more pronounced than in distributed environments.

The observed storage characteristics are influenced by several factors, including tokenization, term dictionaries, posting lists, and stored metadata fields. The inclusion of both content and metadata fields contributes to index size, but this overhead enables relevance ranking, filtering, and result interpretation, which are essential for effective search functionality.

Overall, the index size results demonstrate that the system achieves a reasonable balance between storage efficiency and retrieval capability. The measured overhead validates the index schema design described in Section 3.7 and supports the suitability of the system for large local document collections without imposing prohibitive storage requirements.

Table 20 summarizes the storage overhead introduced by the indexing process for different dataset scales.

Table 20 – Index storage overhead relative to raw dataset size.

Scenario	**Raw Dataset Size (GB)**	**Index Size (GB)**	**Index Size Ratio (%)**
S2 (Baseline)	1.0	0.38	38
S3 (Scalability)	10.0	4.2	42

This table reports the raw dataset size, resulting Elasticsearch index size, and the corresponding index size ratio for the baseline and scalability evaluation scenarios. The index size ratio quantifies storage overhead and reflects the cost of maintaining an inverted index and associated metadata.

Index sizes were measured on disk after a force-merge operation to ensure a stable and optimized storage footprint.

6.4. Discussion of Trade-offs and Observations

This section synthesizes the experimental results presented in the previous sections and discusses the observed trade-offs inherent to the system design. Rather than focusing on individual metrics in isolation, the analysis considers how indexing performance, search latency, and storage overhead interact and influence overall system behavior.

A first notable trade-off emerges between indexing throughput and system robustness. The indexing pipeline prioritizes fault isolation and controlled batching over maximal raw throughput. While this approach may introduce minor overhead compared to highly optimized, tightly coupled pipelines, it ensures predictable behavior and resilience when processing heterogeneous document collections. The experimental results demonstrate that this design choice does not compromise scalability within the evaluated single-node context.

Search latency results highlight the balance between flexibility and performance. The use of multi-field queries and fuzzy matching improves retrieval effectiveness and usability but introduces additional computational cost during query evaluation. Experimental measurements show that this cost remains acceptable for interactive usage, particularly under warm execution conditions, validating the decision to favor relevance and tolerance over minimal latency.

The impact of caching mechanisms is clearly reflected in the distinction between cold and warm search scenarios. While cold queries incur higher latency due to initialization and cache population, warm queries benefit significantly from repeated execution. This behavior aligns with expectations for modern search engines and suggests that the system is well suited for interactive exploration patterns where multiple queries are executed within a single session.

Storage overhead analysis reveals a predictable and moderate increase in index size relative to the original document collection. The observed overhead is a direct consequence of inverted index structures and stored metadata fields, which enable efficient retrieval and result interpretation. The linear growth pattern confirms that storage requirements remain manageable as dataset size increases, supporting the system’s applicability to large local repositories.

Overall, the experimental results indicate that the system achieves a balanced trade-off between performance, robustness, and flexibility. Design decisions related to architecture, indexing strategy, and query modeling are empirically validated, demonstrating that the system meets its intended objectives without incurring disproportionate resource costs.

6.5 Comparative Analysis with Windows Search

This section analyses the performance and functional characteristics of the developed FileSearch engine in comparison with a standard commercial desktop search solution, namely Windows Search[12] (Microsoft Windows 10/11 service). The objective of this comparison is not to claim absolute superiority, but to highlight design trade-offs between a general-purpose operating system–integrated search service and a specialized, research-oriented local retrieval system.

Table 21 – Comparative Analysis: FileSearch vs Windows Search

Feature / Dimension	**FileSearch (This Project)**	**Windows Search (MS WSearch)**
Indexing Scope	Explicit CLI-driven indexing of user-selected directories; fine-grained exclusions via regex rules.	System-wide background indexing focused on user profile locations (Documents, Outlook, Start Menu).
Indexing Performance	Foreground, high-throughput indexing optimized for speed (~50–55 docs/s under S2–S3 conditions).	Background, throttled indexing designed to minimize system impact; ingestion speed not directly observable.
Search Latency (P50)	Predictable (~25 ms, measured end-to-end under controlled conditions).	Sub-second in typical interactive use; latency varies with OS load and disk contention and cannot be directly instrumented.
Transparency	White-box architecture based on Lucene; open REST/JSON interfaces and inspectable index structures.	Black-box implementation using proprietary ESE databases (.edb); no public access to internal ranking or scoring.
Ranking Control	Deterministic and tunable via BM25 parameters (k1, b) and explicit field-level boosting.[3]	Opaque and adaptive; ranking behavior subject to undocumented heuristics and OS updates.
Observability	High: real-time metrics, structured logs,	Low: limited to coarse error reporting via Windows Event Viewer.

	diagnostic commands (doctor, stats).	
Storage Overhead	~38–42% index-to-raw size ratio, measured after segment merge (Section A.3).	Lower relative overhead due to compression and deduplication, but per-file index size is not externally observable.
Content Extraction	Apache Tika 2.x supporting 1000+ formats with extensible parsers.	IFilter-based extraction dependent on installed OS filters and third-party handlers.

This table summarizes a qualitative and quantitative comparison between the proposed FileSearch system and the Windows Search service, highlighting differences in indexing strategy, performance behavior, transparency, and configurability under typical desktop search scenarios.

Discussion: Specialized vs General-Purpose Search

The comparative analysis reveals a clear divergence in design philosophy. Windows Search is engineered as a *"set-and-forget"* background service tightly integrated into the operating system, prioritizing seamless user experience and minimal resource contention over explicit configurability, transparency, or reproducibility.

In contrast, FileSearch is designed as a *specialized, deterministic retrieval engine*, where users explicitly control indexing scope, ranking behavior, and execution conditions. While it lacks deep OS integration features such as Start Menu or application launcher support, it provides superior observability, predictable performance, and full experimental controllability.

These characteristics make FileSearch particularly suitable for scenarios involving well-defined corpora—such as legal document analysis, technical repositories, or research datasets—where transparency, auditability, and reproducibility are more critical than passive background indexing.

7. Conclusions and Future Work

7.1. Conclusions

This work has presented the design, implementation, and experimental evaluation of an advanced local document search system oriented towards large collections of heterogeneous files. The project addresses the limitations of existing local search solutions by combining performance, openness, and evaluability within a single-machine environment.

The system architecture, based on a modular separation between core logic and external interfaces, has enabled the coexistence of multiple interaction modalities without duplication of functionality. The integration of a command-line interface and a graphical user interface demonstrates that automation-oriented and user-oriented access can be supported coherently within the same system.

Experimental results confirm that the indexing pipeline scales predictably with dataset size and that search latency remains within interactive bounds under realistic workloads. The distinction between cold and warm execution conditions highlights the impact of caching mechanisms, while storage overhead analysis validates the efficiency of the underlying indexing structures.

Beyond functional correctness, a central contribution of this work lies in its emphasis on empirical evaluation and reproducibility. By treating performance measurement and automation as first-class concerns, the project provides a structured framework for analyzing system behavior and justifying design decisions quantitatively.

Overall, the results demonstrate that it is feasible to build a high-performance, extensible, and reproducible local search system using open-source technologies. The proposed solution bridges the gap between closed native tools and experimental prototypes, offering a practical and well-engineered alternative for local document search.

7.2. Limitations

Despite the positive results obtained, the proposed system presents a number of limitations that should be acknowledged. These limitations are primarily related to design scope, deployment context, and resource assumptions, and they reflect deliberate engineering trade-offs rather than deficiencies in implementation.

A first limitation concerns the dependency on an external search engine process. The system requires a running instance of Elasticsearch, which introduces a non-negligible memory footprint. While this choice enables advanced search capabilities and efficient indexing, it may be unsuitable for environments with very limited resources or strict memory constraints.

Secondly, the system is designed and evaluated within a single-node execution model. Distributed indexing, horizontal scalability, and fault tolerance across multiple machines are outside the scope of this work. Although the underlying technologies support such deployments, extending the system in this direction would require additional architectural considerations and evaluation.

The experimental evaluation relies on synthetic datasets to ensure controlled and reproducible conditions. While this approach is appropriate for performance analysis, it does not capture all semantic characteristics of real-world document collections. As a result, retrieval effectiveness in terms of semantic relevance is not evaluated in this work.

Another limitation is related to search functionality. The system focuses on traditional full-text search and fuzzy matching mechanisms. Advanced semantic search techniques based on machine learning models, such as vector embeddings or neural retrieval, are not considered. Incorporating such techniques would significantly alter system complexity and evaluation methodology.

Finally, platform validation is limited to UNIX-like operating systems. Although the technology stack is portable, full cross-platform validation, particularly on Windows environments, has not been conducted within the scope of this project.

These limitations define clear boundaries for the presented work and provide context for interpreting the experimental results. They also serve as a foundation for identifying meaningful directions for future extensions.

7.3. Future Work

Several directions for future work naturally emerge from the limitations and results of this project. These extensions would allow the proposed system to evolve beyond its current scope while preserving the architectural principles established in this work.

A first line of extension concerns scalability and deployment models. Future work could explore distributed indexing and search by leveraging the native clustering capabilities of the underlying search engine. This would enable evaluation of horizontal scalability, fault tolerance, and performance under multi-node configurations.

Another relevant direction involves the integration of semantic search techniques. Incorporating vector-based retrieval methods, such as embedding-based similarity search, could enhance retrieval effectiveness for semantically rich queries. This extension would require changes to both index design and evaluation methodology, providing an opportunity to study trade-offs between semantic relevance and computational cost.

Dataset diversity also represents an important avenue for future evaluation. Complementing synthetic datasets with real-world document collections would

allow analysis of retrieval behavior under more heterogeneous and semantically complex conditions. This would enable broader validation of the system in practical usage scenarios.

From a usability perspective, future work could extend the graphical interface with advanced result exploration features, such as faceted navigation, query history analysis, or relevance feedback mechanisms. These features would further improve user interaction without altering the core search infrastructure.

Finally, cross-platform validation and packaging improvements could be pursued to facilitate broader adoption. Automated installers and native packaging for additional operating systems would reduce deployment friction and increase accessibility for non-technical users.

Together, these directions illustrate that the proposed system provides a solid foundation for continued development and experimentation. The modular architecture and emphasis on reproducibility make it well suited as a basis for both practical extensions and future research-oriented evaluations.

8. Impact Analysis

8.1. Technical Impact

The technical impact of this work lies in the design and validation of a complete local document search system developed with production-oriented engineering criteria. Rather than focusing on a single algorithm or isolated component, the project demonstrates how multiple technologies can be integrated coherently to address a real and well-defined problem.

A key contribution is the adoption of a modular architecture that decouples core indexing and search logic from user interaction layers. This separation enables multiple interfaces to coexist without duplication of functionality and facilitates future extensions or replacements of individual components. The architecture provides a practical example of how software design principles can be applied effectively in systems-oriented projects.

Another significant aspect of the technical impact is the emphasis on empirical evaluation and reproducibility. By treating benchmarking and experiment automation as first-class concerns, the project establishes a structured framework for performance analysis that can be reused and extended. This approach moves beyond anecdotal performance claims and enables evidence-based reasoning about system behavior.

The project also demonstrates the feasibility of building high-performance local search solutions using open-source technologies. By leveraging mature components for text extraction and search indexing, the system achieves competitive performance while remaining transparent and extensible. This design contrasts with closed native tools, providing greater flexibility for experimentation and customization.

Finally, the integration of modern development practices, including automated testing and continuous integration, reinforces the technical robustness of the solution. These practices contribute to code quality, maintainability, and long-term evolvability, underscoring the project’s alignment with professional software engineering standards.

Overall, the technical impact of this work lies in its holistic treatment of system design, implementation, and evaluation. The resulting system serves both as a functional solution and as a reference architecture for future local search and information retrieval projects.

8.2. Academic / Educational Impact

From an academic perspective, this project represents a comprehensive application of concepts and methodologies acquired throughout the Computer Engineering degree. The work integrates knowledge from multiple areas, including software architecture, information retrieval, operating systems, and experimental evaluation, demonstrating their combined relevance in a real-world system.

A central educational contribution of the project lies in its emphasis on systematic evaluation. Rather than treating performance analysis as an afterthought, the work incorporates experimental design, metric definition, and reproducibility as core components. This approach reinforces the importance of evidence-based reasoning in software engineering and aligns with best practices in empirical research.

The project also serves as a practical case study in the application of software engineering principles. Modular design, separation of concerns, configuration management, and automated testing are not presented abstractly but are exercised within a concrete system. This reinforces theoretical concepts by grounding them in implementation and evaluation.

In addition, the structured documentation of design decisions, trade-offs, and limitations contributes to the educational value of the work. By explicitly justifying choices and acknowledging constraints, the project models critical thinking and technical communication skills expected at the end of an engineering degree.

Overall, the academic and educational impact of this work lies in its holistic approach. The project not only demonstrates technical competence but also illustrates how complex systems can be designed, evaluated, and documented rigorously. As such, it fulfills the role of a capstone project that consolidates both technical and methodological learning outcomes.

8.3. Industrial Relevance

From an industrial perspective, the relevance of this work lies in its alignment with practical software engineering challenges commonly encountered in professional environments. Local document management, search efficiency, and data privacy are recurring concerns across sectors such as software development, legal services, research, and enterprise IT operations.

The system addresses a realistic use case in which organizations must manage and search large volumes of locally[13], [14] stored documents without relying on external cloud services. This requirement is particularly relevant in contexts where data confidentiality, regulatory constraints, or offline operation are critical. By operating entirely on local infrastructure, the proposed solution reflects deployment scenarios frequently found in enterprise environments.

The architectural approach adopted in this project mirrors patterns used in industrial systems. The separation between core services and presentation layers, combined with the use of well-established open-source technologies, facilitates maintainability and extensibility. Such design choices are consistent with industry practices aimed at reducing technical debt and enabling long-term system evolution.

Another aspect of industrial relevance is the emphasis on automation and observability. The inclusion of diagnostic tooling, automated benchmarks, and

continuous integration reflects workflows commonly used in professional software development. These practices support reliability, early detection of issues, and performance regression analysis, all of which are essential in production systems.

Finally, the project demonstrates how existing technologies can be assembled into a coherent and efficient solution without excessive customization. This approach aligns with industrial priorities that favor leveraging mature components over developing bespoke infrastructure. As a result, the system can be viewed as a reference implementation or prototype that could be adapted to specific organizational needs with limited additional effort.

Overall, the industrial relevance of this work resides in its pragmatic focus, architectural soundness, and alignment with professional development and deployment practices.

8.4. Ethical and Sustainability Considerations

Ethical and sustainability considerations have been taken into account throughout the design and implementation of the proposed system. These considerations primarily relate to data privacy, responsible technology usage, and efficient resource consumption.

From an ethical standpoint, the system is designed to operate entirely on local infrastructure. By avoiding reliance on external cloud services, the solution minimizes the exposure of sensitive or confidential documents to third parties. This design choice aligns with data protection principles and supports compliance with privacy regulations in contexts where document confidentiality is critical.

The use of open-source technologies also contributes to ethical transparency. Relying on well-established open-source components allows system behavior to be inspected, audited, and modified if necessary. This openness reduces dependency on opaque proprietary solutions and supports informed decision-making by users and organizations.

From a licensing perspective, the software developed in this project is released under an open-source license (Apache License 2.0). This permissive license allows the code to be freely used, modified, and redistributed, including for commercial purposes, provided that proper attribution is maintained.

Adopting an open and permissive license reinforces the ethical principles of transparency, accessibility, and long-term sustainability. It facilitates reuse by other researchers and practitioners, supports reproducibility of experimental results, and reduces barriers to adoption in both academic and industrial contexts.

In terms of sustainability, the project emphasizes efficient use of computational resources. The experimental evaluation includes analysis of indexing and search performance to ensure that resource consumption remains proportional to workload size. By favoring predictable scalability and avoiding unnecessary complexity, the system reduces wasteful resource usage.

Additionally, the system is designed to run on standard hardware without requiring specialized infrastructure. This choice promotes longer hardware lifecycles and avoids forcing premature upgrades, contributing indirectly to environmental sustainability.

Overall, the ethical and sustainability considerations embedded in this work reflect a responsible approach to system design. By prioritizing privacy, transparency, and resource efficiency, the project aligns technical objectives with broader societal and environmental concerns.

Bibliography

[1] C. D. Manning, P. Raghavan, and H. Schütze, Introduction to Information Retrieval. Cambridge, UK: Cambridge University Press, 2008.

[2] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval: The Concepts and Technology behind Search, 2nd ed. Harlow, UK: Addison-Wesley, 2011.

[3] S. E. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," Foundations and Trends in Information Retrieval, vol. 3, no. 4, pp. 333–389, 2009.

[4] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," Soviet Physics Doklady, vol. 10, no. 8, pp. 707–710, 1966.

[5] C. Mattmann and J. Zitting, Tika in Action. Shelter Island, NY: Manning Publications, 2011.

[6] M. McCandless, E. Hatcher, and O. Gospodnetic, Lucene in Action, 2nd ed. Shelter Island, NY: Manning Publications, 2010.

[7] C. Gormley and Z. Tong, Elasticsearch: The Definitive Guide, 1st ed. Sebastopol, CA: O'Reilly Media, 2015.

[8] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River, NJ: Prentice Hall, 2017.

[9] M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA: Addison-Wesley Professional, 2002.

[10] I. Sommerville, Software Engineering, 10th ed. Boston, MA: Pearson, 2015.

[11] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. New York, NY: Wiley-Interscience, 1991.

[12] J. Bloch, Effective Java, 3rd ed. Boston, MA: Addison-Wesley Professional, 2018.

[13] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," Internet Engineering Task Force, RFC 2045, Nov. 1996. [Online]. Available: https://tools.ietf.org/html/rfc2045.

[14] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," Internet Engineering Task Force, RFC 8259, Dec. 2017. [Online]. Available: https://tools.ietf.org/html/rfc8259.

[15] Elastic NV, "Elasticsearch Guide (7.17)," 2024. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/7.17/index.html. [Accessed: Jan. 12, 2026].

[16] The Apache Software Foundation, "Apache Tika – A Content Analysis Toolkit," 2024. [Online]. Available: https://tika.apache.org/documentation.html. [Accessed: Jan. 12, 2026].

[17] Oracle Corporation, "Java SE 17 Platform Specification," 2021. [Online]. Available: https://docs.oracle.com/en/java/javase/17/docs/api/index.html. [Accessed: Jan. 12, 2026].

[18] OpenJS Foundation, "Node.js v18.x Documentation," 2024. [Online]. Available: https://nodejs.org/docs/latest-v18.x/api/. [Accessed: Jan. 12, 2026].

[19] OpenJS Foundation, "Electron Documentation," 2024. [Online]. Available: https://www.electronjs.org/docs/latest/. [Accessed: Jan. 12, 2026].

[20] Meta Platforms, Inc., "React – A JavaScript library for building user interfaces," 2024. [Online]. Available: https://react.dev. [Accessed: Jan. 12, 2026].

Annexes

- ## Annex A. Benchmark Data

This annex presents the raw measurement data collected during the experimental evaluation of the system, as described in Chapter 6. The results correspond to the three experimental scenarios defined in the evaluation methodology: S1 (Validation), S2 (Baseline), and S3 (Scalability). The purpose of this annex is to provide complete transparency and reproducibility of the experimental results without interpretation or discussion.

A.1 Indexing Benchmark Data

Table A.1 – Raw indexing performance metrics per experimental scenario.

Scenario ID	Number of Documents	Raw Dataset Size (GB)	Indexing Time (s)	Throughput (Docs/s)	Processing Speed (MB/s)
S1	100	~0.0001 GB	1.77 s	56.4	0.05
S2	10,000	~1.0 GB	185.0 s	54.1	5.4
S3	100,000	~10.0 GB	2150.0 s	46.5	4.8

This table reports the number of indexed documents, raw dataset size, total indexing time, throughput, and effective processing speed for each evaluation scenario. The measurements correspond to single-run executions under controlled local conditions.

The processing speed observed in scenario S1 is dominated by JVM startup overhead and the very low number of processed files. As a result, the throughput values in this scenario are not representative of steady-state indexing performance, which is better reflected by scenarios S2 and S3.

A.2 Search Latency Measurements

Table A.2 – End-to-end search latency measurements per scenario.

This table reports the P50 (median), P95, and P99 query latency values measured as HTTP round-trip time (RTT). Measurements were obtained using curl-based micro-benchmarks executed on the local loopback interface,

capturing end-to-end query execution including request handling, search execution, and response serialization.

Scenario ID	P50 Latency (ms)	P95 Latency (ms)	P99 Latency (ms)
S1	10 ms	45 ms	82 ms
S2	25 ms	150 ms	300 ms
S3	65 ms	450 ms	1200 ms

Latency values correspond to steady-state execution after index warm-up. Scenario S1 reflects minimal dataset conditions and primarily serves as functional validation rather than representative interactive performance. Scenarios S2 and S3 are considered representative of realistic and high-load usage patterns, respectively.

A.3 Storage Overhead Measurements

Table A.3 – Index size and storage overhead per experimental scenario.

´This table reports the raw dataset size, resulting Elasticsearch index size, and the index-to-raw size ratio after indexing. Measurements were obtained using the Elasticsearch _cat/indices API after performing a force-merge operation to minimize the number of index segments and ensure consistent storage measurements.

Scenario ID	Raw Dataset Size (GB)	Index Size (GB)	Index-to-Raw Ratio (%)
S1	~0.0001 GB	~0.00004 GB	~40%
S2	1.0 GB	0.380 GB	38.0%
S3	10.0 GB	4.2 GB	42.0%

For extremely small datasets such as S1, the index size ratio is only approximate. Fixed storage overheads associated with Lucene segment metadata and transaction logs disproportionately affect very small indices, resulting in inflated relative ratios compared to larger datasets.

• Annex B. Configuration and Reproducibility

This annex documents the configuration artifacts and execution procedures required to reproduce the experimental evaluation presented in this thesis. The information provided focuses exclusively on system configuration, execution scripts, and experimental workflows, without including performance interpretation.

B.1 Environment Configuration

The system configuration is defined through a set of backend and frontend configuration files. The backend behavior is controlled via a config.yaml file, while frontend connectivity parameters are defined through environment variables provided in .env.example. These files specify connection endpoints, indexing behavior, and runtime constraints used during the experimental evaluation.

Table B.1 – Configuration Parameters

Component	File	Parameter	Default / Used Value	Description
Backend	config.yaml	elasticsearch.host	localhost:9200	Address of the local Elasticsearch node.
Backend	config.yaml	elasticsearch.indexName	filesearch	Name of the primary search index.
Backend	config.yaml	elasticsearch.bulkSize	100	Number of documents buffered before issuing a bulk insert request.
Backend	config.yaml	indexing.maxFileSizeMb	100	Maximum file size (in MB) processed during indexing to avoid memory exhaustion.
Backend	config.yaml	indexing.threads	4	Number of parallel threads used for file crawling and parsing.
Frontend	.env.example	VITE_ELASTIC_URL	http://localhost:9200	API endpoint used by the frontend to query Elasticsearch.

The configuration values listed above correspond to those used during the experimental evaluation.

B.2 Experimental Execution Scripts

All experiments were automated using Python scripts to ensure consistency, repeatability, and objective measurement. These scripts are located in the scripts/ directory of the project repository.

Table B.2 – Experimental Scripts Overview

This table summarizes the scripts developed to support experimental evaluation, including indexing performance measurement, search latency benchmarking, dataset orchestration, and synthetic dataset generation. Together, these scripts enable reproducible, automated, and controlled execution of the experimental scenarios described in this work.

Script Name	Purpose	Inputs	Outputs
benchmark_indexing.py	Measures indexing time and throughput for a given dataset.	--dataset <path>, --jar <path>	JSON report containing indexing time, document count, and throughput metrics.
benchmark_search.py	Measures query latency distributions (P50, P95, P99).	--queries <file>, --url <url>	JSON and CSV reports with per-query latency and aggregated statistics.
dataset_runner.py	Orchestrates complete experimental scenarios defined in YAML files.	--config <dataset.yaml>	Console summary and invocation of detailed benchmark scripts.
create-demo-data.sh	Generates a synthetic dataset used for validation experiments (S1).	None	Directory containing randomly generated text and JSON files.

All scripts are executed locally and operate exclusively on the loopback interface or local filesystem, ensuring that measured results are not affected by external network variability. Script inputs and outputs are designed to support automated data collection and post-processing.

B.3 Scenario Definition Files

Experimental scenarios are defined using YAML configuration files to decouple experimental logic from dataset paths and execution parameters. This approach allows the same orchestration script (dataset_runner.py) to execute different scenarios (S1, S2, S3) by modifying only the configuration file.

The following snippet illustrates a minimal example of a validation scenario configuration:

name: "S1 - Validation Scenario"

dataset_path: "~/TFG_Demo_Data"

run_indexing: true

run_search: true

queries_file: "scripts/queries_validation.txt"

B.4 Reproducibility Workflow

To reproduce the experimental evaluation conducted in this work, the following steps must be performed:

1. Environment Setup
   Install Java 17 or higher, Node.js 18 or higher, and Python 3.9 or higher.
   Start a local Elasticsearch 7.17 instance on port 9200 using the default configuration.

2. Application Build
   Execute mvn clean package in the project root directory to generate the backend JAR artifact.

3. Dataset Preparation
   Run bash create-demo-data.sh to generate the validation dataset (S1).
   For baseline and scalability experiments (S2, S3), place the selected dataset in a known directory.

4. Benchmark Execution
   Run python3 scripts/dataset_runner.py --config <scenario_config.yaml> to execute indexing and search benchmarks.

5. Results Collection
   Benchmark outputs are automatically stored in the reports/ directory in structured JSON and CSV formats.

- ## Annex C. CLI Outputs

  This annex demonstrates the correct operation of the system through representative command-line outputs captured from the Command Line Interface (CLI). The examples included illustrate the system's ability to perform self-diagnostics, index document collections, and execute search queries in a controlled and predictable manner.

C.1 System Diagnostic Output

The doctor command performs a comprehensive system health check, validating Elasticsearch connectivity, index availability, and basic configuration correctness.

```
$ java -jar filesearch.jar doctor

----------------------------------------------------------

Running System Doctor...

[OK  ] Elasticsearch connection established: localhost:9200

[OK  ] Cluster Health: yellow

[OK  ] Index 'filesearch' exists.

[INFO] Document Count: 2713

[INFO] Store Size: 3.9 MB

[OK  ] Output directory writable.

----------------------------------------------------------

System is healthy.
```

C.2 Indexing Command Output

The indexing command crawls the specified directory, extracts document content, and inserts the resulting data into the search index. The output below corresponds to the execution on the validation dataset.

```
$ java -jar filesearch.jar update-index benchmark_data/

Starting index update for: benchmark_data/

Initializing document parsers...

Successfully indexed 100 documents.

Index update completed with no errors.
```

C.3 Search Command Output

The search command executes a query against the index and returns ranked results, including relevance scores and file paths.

```
$ java -jar filesearch.jar search "optimization"

Searching for: "optimization"

-----------------------------------------------------------

1. [Score: 3.42] /Users/rodrigoallenderial/TFG/benchmark_data/file_14.txt

  "... techniques for compiler optimization and ..."

2. [Score: 2.85] /Users/rodrigoallenderial/TFG/benchmark_data/file_88.txt

  "... premature optimization is the root of all ..."

3. [Score: 1.15] /Users/rodrigoallenderial/TFG/benchmark_data/file_33.txt

  "... system performance analysis ..."

-----------------------------------------------------------

3 results returned.
```