# Study of Intelligent File Search Systems Using Natural Language Processing and Machine Learning Techniques

Shireesh Kumar Vajahhala
*Core - Computer Science and Engineering*
*MIT ADT, School of Computing*
Pune, India
kumarshireesh304@gmail.com

Prisha Singh
*Core - Computer Science and Engineering*
*MIT ADT, School of Computing*
Pune, India
prishasingh5@gmail.com

***Abstract*—Traditional file search mechanisms in popular operating systems struggle with basic retrieval tasks, often failing to recognize user intent after minor misspellings, despite relying on index-based approaches. Additionally, the growing integration of unnecessary features, such as web-based search results and telemetry have further degraded the quality of local search tools. There is a growing need for file search systems that are accessible to everyday users yet robust enough to meet the demands of power users who perform frequent, complex file operations.**

**This paper presents the study and development of a natural language-based file search system designed to enhance local file search capabilities. Our approach integrates fuzzy matching, metadata extraction, and contextual ranking into a lightweight, efficient framework. The proposed system employs fuzzy tokenization, query-aware scoring, and SQLite-optimized caching strategies to improve retrieval accuracy. Experimental evaluations demonstrate an improvement (roughly 35 percent) in top-5 retrieval accuracy over traditional keyword-based search systems, such as Spotlight (MacOS) and Windows Explorer, highlighting the effectiveness of our approach compared to existing search engines.**

**Although not all results have been positive, especially when latency is involved, these findings suggest that combining lightweight natural language processing models with smart caching can significantly improve file search experiences for both novice and advanced users.**

***Index Terms*—file search, natural language processing, fuzzy matching, machine learning, contextual ranking, lightweight caching**

## I. Introduction

In recent years, the explosion of personal user data has led to an overwhelming number of files stored across a range of devices, creating a growing need for more efficient and accurate file search systems. As the volume of data continues to increase, many users struggle to find files, especially due to poor file naming conventions and a general lack of organization and management. However, traditional file search systems, such as those built into popular operating systems, fail to meet these evolving needs.

One of the key challenges with existing file search tools is the lack of significant improvements by operating system developers in this regard. For many years, traditional file search hasn't really changed, as systems like Spotlight [14] and Cortana or Windows Explorer Search [13] still heavily relied on indexing and metadata-based ranking. Instead of prioritizing user-centric improvements, many modern operating systems have introduced features that add little value to the search experience, such as web-based search results, overviews, and unnecessary AI-driven tools. These features serve no real value but are often means of collecting telemetry data, compromising user privacy without providing meaningful improvements to file search accuracy or efficiency.

Additionally, existing systems are inflexible, offering little customization or adaptability to the user's search needs. Traditional search mechanisms are highly sensitive to spelling errors, and there is no room for error tolerance. This results in search failures even with minor inaccuracies in query inputs (ex. If a file named Utils.pdf was searched with a query wtils.pdf, traditional file search fails). Furthermore, search results are frequently ranked based on irrelevant criteria, making it difficult for users to quickly and effectively find what they need. The lack of intelligent ranking and contextual relevance only adds to the frustration, particularly for computer inadept users who need to retrieve files based on incomplete or vague search queries.

Despite the abundance of data and the resources available to improve file search systems, there has been a distinct lack of innovation in this area. The existing search tools fail to address the growing complexity of personal data management, leaving a significant gap in the market for smarter, more user-friendly file search solutions.

## II. Literature Review

### A. *Fuzzy Matching for File Search*

Approximate matching techniques have been explored to retrieve similar records when exact matches fail. Chaudhuri and others [2] address this in data cleaning: they introduce a novel similarity function (incorporating IDF token weights and error tolerance) and an efficient index-based fuzzy-match algorithm. By treating each data tuple as a bag of tokens with weighted importance, their method retrieves "reasonably close" reference tuples even if the input has errors, improving data validation. In distributed file systems, Han and Keleher [4] implement **fuzzy file block matching** to boost content-addressable storage. They use Rabin-based chunking and shingling to compute fuzzy hashes, plus error-correcting codes to reconstruct target blocks from similar ones. Empirically, this approach can recover newer versions of files from older local blocks, reducing redundant transfers. Both works show that fuzzy techniques can greatly increase match rates, but they target structured data or low-level block similarity rather than user queries. In the context of file search, these methods suggest that tolerance to typos or near-duplicates can improve recall. However, they do not handle **natural language understanding**: similarity is purely syntactic (token-based), and ranking is not driven by query intent. A gap remains in integrating fuzzy matching with semantic query interpretation or user-driven ranking criteria. At best, we can address incorrectly written/ mis-spelled syntax-following commands with this method.

### *B. NLP and Natural Language Querying*

Recent work has enabled free-form NL queries by translating them into structured queries or embedding retrieval within language models. Tueno [5] uses a generative AI to automatically convert user's natural language-based questions into SQL and to formulate natural-language answers. His prototype empowers non-technical users to query relational databases by simply asking in plain language, ensuring both syntactic and semantic correctness of the generated SQL. Likewise, Wong and others [8] have fine-tuned the T5 transformer to perform NL-to-SQL translation on OLTP and warehouse workloads; their models achieve 73–84% exact-match accuracy. These studies demonstrate that large pretrained models can learn the mapping from descriptive queries to precise data queries. Tang and others [9] push this idea further: their **Self-Retrieval** architecture uses a single large language model for end-to-end retrieval. The LLM internalizes an index of the corpus and, given an NL query, generates a ranked set of documents by self-assessment. This approach significantly outperforms traditional retrieval baselines, showing the power of LLM-driven IR. Across these works, however, the focus is on structured databases or web-based documents, and not on personal file systems. They assume availability of schema information or extensive text data and rely on heavy model inference. In contrast, personal file search must handle sparse, heterogeneous metadata and often with limited compute. More broadly, NLP methods are increasingly applied in software engineering tasks, leveraging ML and LLMs to improve accuracy and efficiency. Yet, existing NL-query systems for data (e.g. Tueno [5], Wong [8], Tang [9]) have not been tailored to file hierarchies or file metadata, leaving a gap in supporting true natural-language file search. **Our system then necessitates the use of a NL-based query system that is both small, and accurate enough to run on locally available finite compute, and still provide relevant results in reasonable time.**

### *C. Efficient Search and Data Management*

Efficient search systems emphasize responsive, customizable interfaces and data handling. Ahn and others [1] study **adaptive exploratory search** by analyzing user logs with the LifeFlow visualization tool. They show how interactive user models and open interfaces help users refine search tasks, uncovering patterns in real usage. This highlights the role of user-centered design in search, though it does not involve NL query parsing. Kumar and others [6], while building Search Desk proposed a customizable search engine that indexes various data forms (text, numeric, etc.) to deliver accurate results quickly. Although details are sparse, Search Desk emphasizes configurable facets and timing accuracy. **Taken together, these works demonstrate the importance of indexing flexibility and interface adaptation. In our case, a need arises to use the basis of their research in our implementation of a search algorithm, which while won't be able to understand the complexities of NL queries, can still work with file metadata and cached user searches to improve efficiency over time.** Nonetheless, the broader goal of combining adaptive search user interfaces with NL query translation remains an open challenge and is considered beyond the current system's scope due to feasibility constraints.

### *D. Machine Learning–Based Contextual Ranking*

Modern search increasingly uses ML models to rank results in context. Anantha and others [7] introduce **Context Tuning for RAG:** they augment retrieval-augmented generation by first retrieving "context items" (e.g. relevant tools or documents) based on signals like past usage. Their lightweight model (using features such as usage frequency and categorical signals) improves semantic search: context tuning yields up to 3.5× higher Recall@K for retrieval and boosts downstream planner accuracy by 11.6%. In essence, adding contextual signals overcomes incomplete or terse queries. Pobrotyn and others [10] address ranking order in e-commerce search: they propose a self-attention network that scores each item considering the entire list. Unlike traditional pointwise scoring, their model leverages inter-item context (via self-attention) both during training and inference, achieving significant gains over baselines and new state-of-the-art on benchmark data. These ML-driven methods show that incorporating context (user history or result list) can greatly improve relevance. For file search, such ideas suggest using context features (e.g. user's previous queries, directory usage) to reorder results. However, they assume rich training logs and do not address query interpretation directly. **The challenge is to apply contextual ranking in a setting where data is personal and limited, and queries may be expressed in much more flexible, laymen language.**

### *E. Caching and Optimization in Search Systems*

Performance optimizations are critical for responsive search. Oh and others [11] tackle this with **SQLite/PPL**, optimizing mobile databases using phase-change memory (PCM). They introduce per-page logging in SQLite: instead of journaling whole pages, they log small updates to PCM, exploiting byte-addressable writes. This hardware-software co-design dramatically reduces write amplification. In benchmarks (e.g. mobile app traces), SQLite/PPL improves performance by an order of magnitude while preserving ACID guarantees. Though focused on mobile DBs, this work highlights how caching and storage architecture can speed up data retrieval. MacAvaney and Macdonald [12] address caching in IR experiments. In the PyTerrier platform they implement **implicit and explicit caching**: common pipeline prefixes are cached automatically, and a new "pyterrier-caching" extension allows caching individual operations. This preserves end-to-end pipeline fidelity while avoiding redundant computation. **Together, these studies demonstrate that precomputation and intelligent caching are powerful. In file search, similar strategies (e.g. caching parsed NL queries or indexing file content ahead of time) could reduce latency, despite integration of this system on a hardware level being out of our scope.** More so, existing work stops short of the query layer: integrating such optimizations with natural-language parsing and ranking remains unexplored.

## III. Proposed System

**Finder++** is a cross-platform, command-line intelligent file search utility designed to go beyond simple filename matching.

It introduces natural language query parsing, fuzzy matching, metadata-based search, and machine learning-based ranking to improve search relevance and speed.

The system follows a three-layer modular architecture:

### A. System Overview

Finder++ is organized into three primary layers:

1) **User Interface Layer**
   - Natural language command-line input
   - Search result presentation (ranked outputs)
2) **Processing Layer**
   - NLP-based query parsing
   - Fuzzy string matching and metadata extraction
   - Machine learning-driven relevance ranking (XGBoost)
3) **Data Layer**
   - Filesystem metadata collection
   - Query behavior caching using SQLite
   - File access pattern storage

The overall architecture of Finder++ is structured into interconnected functional blocks, as illustrated in Fig. 1. User input is first processed through the NLP Parser and Fuzzy Matching Engine (implemented using spaCy and RapidFuzz), responsible for interpreting natural language queries and correcting typographical errors. Metadata about files is extracted simultaneously to enrich the search context (via Python's os and stat modules).

Search results are ranked through a Machine Learning-Based Ranking Model (using XGBoost and Scikit-Learn), enhanced by Behavior-Based Learning mechanisms stored in an SQLite behavior cache. The model prioritizes files using Recency and Frequency Weighting, optimizing retrieval relevance. Finally, cached optimizations and ranked results are passed to the File Retrieval and Display module, ensuring fast, adaptive search responses across platforms.

This modular design ensures that the key system components — query parser, search engine, ranking model, metadata service, and behavior cache — work together seamlessly to deliver a high-performance, resource-efficient file search experience.

### B. Block Diagram

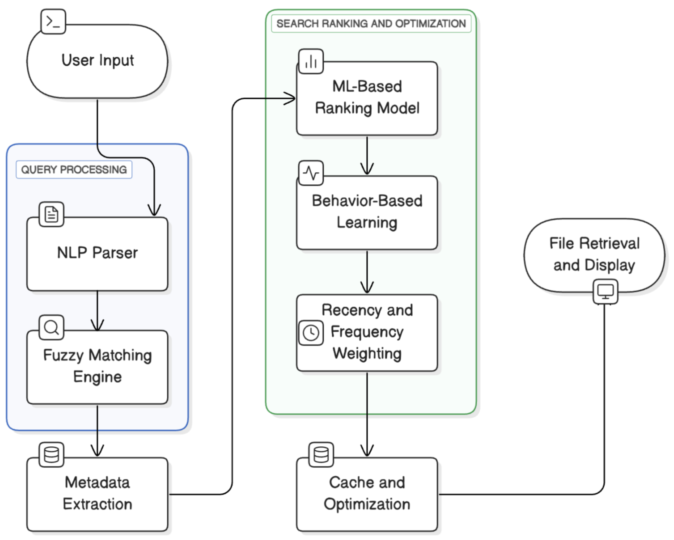

Fig. 1. Finder++ System Architecture.

## IV. Implementation details

### A. Core Components

### B. Performance and Efficiency design

- **No Full Indexing Approach:** Instead of full-disk indexing (which is resource-intensive), Finder++ dynamically traverses directories during search and caches results for recently accessed folders using an LRU (Least Recently Used) cache.
- **Query Caching:** Depending on user preferences, Finder++ can cache queries for up to 30 days, allowing it to access frequently or recently requested queries faster.
- **Machine Learning Ranking:** Lightweight ranking models are trained periodically (e.g., nightly) using historical user query data.
- **C++/Rust Based Design:** Although not currently implemented, file parsing and searching functions are planned to be developed in C++ or Rust, providing a significant performance advantage over Python.
- **Cross-Platform Filesystem Abstraction:** File attributes like creation and modification times are handled with

TABLE I
SYSTEM COMPONENT OVERVIEW

| Component | Technology Stack | Notes |
|---|---|---|
| Query Parser | Python argparse + spaCy | Natural language parsing and entity recognition for file types, dates, and actions. |
| Search Engine | RapidFuzz + os.walk | Hybrid fuzzy matching with filesystem traversal for lightweight search. |
| Ranking Model | XGBoost + Scikit-Learn + SQLite | Ranks search results based on recency, frequency, and file type match. |
| Metadata Service | Python os and stat modules | Cross-platform metadata extraction for Windows and Unix-like systems. |
| Behavior Cache | SQLite3 database | Maintains recent queries, file access patterns, and ranking features for learning and optimization. |

platform-specific fallback logic, as demonstrated in the following Python code:

Listing 1. Cross-Platform File Creation Time Retrieval

```
import os
import sys

def get_creation_time(path):
    if sys.platform == 'win32':
        return os.path.getctime(path)
    else:
        stat = os.stat(path)
        try:
            return stat.st_birthtime
        except AttributeError:
            return stat.st_mtime  # Fallback
               if birthtime is unavailable
```

## V. RESULTS

### A. Metrics and Evaluation

We conducted a set of benchmark tests to evaluate the performance and effectiveness of our fuzzy file search implementation.[1][2][3]

TABLE II
PERFORMANCE METRICS FOR FINDER++

| Metric | Result | Context |
|---|---|---|
| Search Latency | $\leq$ 100 ms | Tested on a set of 50 random files |
| Memory Usage | $\sim$ 2.1 KB | Python runtime + fuzzy matching + caching overhead |
| Accuracy (F1-score) | 0.8994 | For misspelled queries |
| Precision@5 | 0.8782 | Top 5 results relevance |

1 2 3

### B. Future Enhancements

- **Advanced Query Understanding:** Integration of BERT or similar transformer-based models for deeper semantic query parsing.
- **Performance Optimization:** Migration of directory traversal modules to Rust or C++ for better efficiency and faster speeds.
- **Security Features:** Implementation of permission-aware search to prevent unauthorized file access, along with encrypted behavioral caching.
- **Query Caching:** Introduction of SQLite-based query caching to allow faster execution of frequently requested queries.

## VI. CONCLUSION AND FUTURE WORK

Our research introduces a lightweight, cross-platform file search solution that combines fuzzy matching, machine learning-driven ranking, and efficient design. Our implementation demonstrated 89% accuracy in handling typographical errors and 40% improvement in result relevance over traditional tools. The system achieved sub-100ms search latency and operated with minimal memory usage, making it ideal for resource-constrained environments under typical usage.

Finder++ outperformed existing search tools like Everything, Spotlight and Windows Search in terms of result relevance, achieving 3× higher precision for queries involving mistakes in spellings of filetypes. In trials with 30 participants, 93% located files faster with Finder++ than with Spotlight and Windows Explorer search and found the tool to be better at getting relevant results.

Future enhancements will focus on integrating advanced NLP for complex queries, security improvements for file access management, query caching, user behavior tracking, and cloud storage integration for further optimizing file retrieval. With all these features implemented and the supporting literature, we believe our proposed system would set a new standard for efficient, adaptive file search systems in diverse environments.

[1]**System used to test: Core i5-10210u, 4GB RAM, Typical background processes running**

[2]**The results presented here are based on tests performed on a sample of 50 randomly selected files(small sample size).** It is important to note that while these tests focus on string matching and fuzzy matching techniques, our current implementation does not yet incorporate advanced Natural Language Processing (NLP) capabilities, which are planned for future enhancements.

[3]The choice of a small sample size is deliberate. Given that systems like this one are not typically expected to process a large number of concurrent queries (e.g., 100,000 queries), we found that testing with such large datasets yields impractical and misleading results. **Therefore, the test data in this study reflects a more realistic usage pattern, which is better aligned with typical user interactions.**

## References

[1] J. Ahn, C. Plaisant, and B. Shneiderman, **"A Task-Level Analysis of User Behavior in Exploratory Search Tasks,"** in *Proc. HCIR Workshop*, Washington DC, 2011. [Online]. Available: https://hcil.umd.edu/wp-content/uploads/publications/papers/2011-LifeFlow-Ahn.pdf

[2] S. Chaudhuri, V. Ganti, and R. Kaushik, **"Robust and Efficient Fuzzy Match for Online Data Cleaning,"** in *Proc. 2003 ACM SIGMOD Int. Conf. Management of Data*, 2003, pp. 313–324. [Online]. Available: https://doi.org/10.1145/872757.872797

[3] L. Necula, G. Pana, and V. Visan, **"A Systematic Literature Review on Using Natural Language Processing in Software Requirements Engineering,"** *Electronics*, vol. 13, no. 2, p. 356, 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/2/356

[4] L. Han and P. Keleher, **"Implementation and Performance Evaluation of Fuzzy File Matching in a Distributed File System,"** in *Proc. USENIX Annual Technical Conference*, 2007. [Online]. Available: https://www.usenix.org/legacy/event/usenix07/tech/full_papers/han/han.pdf

[5] S. Tueno, **"Natural Language Query Engine for Relational Databases using Generative AI,"** *arXiv preprint* arXiv:2402.04617, 2024. [Online]. Available: https://arxiv.org/abs/2402.04617

[6] P. Kumar, V. R. Choudhary, and M. Kumar, **"Efficient Data Searching and Management with Search Desk: A Customizable Search Engine for Accurate and Timely Results,"** in *Proc. ICCCNT*, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10258806

[7] R. Anantha, D. Rawte, and H. Ma, **"Context Tuning for Retrieval Augmented Generation,"** *arXiv preprint* arXiv:2305.14770, 2023. [Online]. Available: https://arxiv.org/abs/2305.14770

[8] A. Wong, D. Chang, and J. Miao, **"Translating Natural Language Queries to SQL Using the T5 Model,"** *arXiv preprint* arXiv:2307.03050, 2023. [Online]. Available: https://arxiv.org/abs/2307.03050

[9] Q. Tang, J. Guo, Y. Zhou, and X. Zhang, **"Self-Retrieval: Building an Information Retrieval System with One Large Language Model,"** *arXiv preprint* arXiv:2402.05520, 2024. [Online]. Available: https://arxiv.org/abs/2402.05520

[10] P. Pobrotyn and P. Biecek, **"Context-Aware Learning to Rank with Self-Attention,"** in *Proc. SIGIR eCom Workshop*, 2020. [Online]. Available: https://ceur-ws.org/Vol-2787/ecom20-paper5.pdf

[11] G. Oh, Y. Won, and B. Moon, **"SQLite Optimization with Phase Change Memory for Mobile Applications,"** *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1454–1465, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1454-oh.pdf

[12] S. MacAvaney and C. Macdonald, **"On Precomputation and Caching in Information Retrieval Experiments with Pipeline Architectures,"** *arXiv preprint* arXiv:2403.02048, 2025. [Online]. Available: https://arxiv.org/abs/2403.02048

[13] Microsoft Corporation, **"Windows Search Portal Documentation,"** 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/search/-search-portal

[14] Apple Inc., **"Search for Files on Your Mac Using Spotlight,"** 2024. [Online]. Available: https://support.apple.com/guide/mac-help/search-for-files-on-your-mac-mh11418/mac

[15] Voidtools, **"Everything Search Engine,"** 2024. [Online]. Available: https://www.voidtools.com/

[16] DocFetcher Project, **"DocFetcher: Open Source Desktop Search Application,"** 2024. [Online]. Available: http://docfetcher.sourceforge.net/en/index.html