Distributed Search Engine Presentation

Mudit Gupta , Anubhav Mishra , Shubham Dewangan

April 24, 2025

1 Introduction

Modern large-scale search systems must handle millions of pages efficiently while remaining extensible, fault-tolerant, and observable. Our pipeline comprises three primary stages: crawler, indexer, and search engine, designed to process, store, index, and serve web content at scale.

2 Architectural Patterns & ADRs

2.1 Pipeline Architecture

Stages: Crawler → Storage → Spark Indexer → Search Engine. Each stage is isolated, with well-defined inputs and outputs.

2.2 Microkernel / Plugin-Based

Core engine orchestrates plugin modules: `fetcher`, `parser`, `storage`, `text_cleaner`, enabling easy extension.

2.3 Master–Worker

Master: Redis frontier coordinates URL distribution. Workers: concurrent threads/processes perform fetch–parse–store loops.

2.4 Shared-Nothing (Spark)

Spark executors operate independently on RDD partitions, ensuring no shared memory and improving fault isolation.

2.5 Architectural Decision Records (ADRs)

ADR	Decision	Rationale
1	Redis for frontier	Lightweight, atomic operations for queue and set management.

2	PySpark for indexing	Scalable map-reduce, built-in partitioning and shuffle.
3	JSON index on disk	Human-readable, no external DB dependency for MVP.
4	FastAPI + Uvicorn	Asynchronous web framework, easy templating.
5	MD5-hashed filenames	Deterministic, filesystem-safe identifiers.

3 High-Level System Design

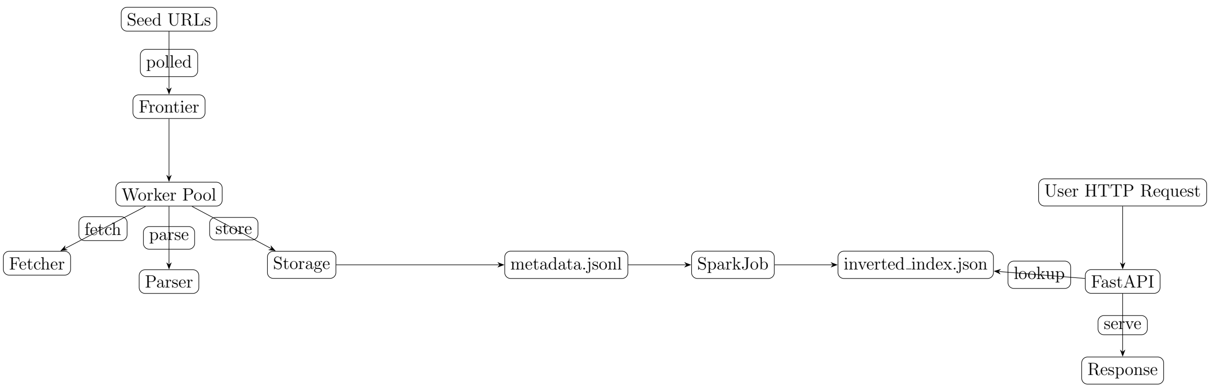

Figure 1: High-Level System Design

3.1 Crawler Layer

The crawler is responsible for discovering and fetching web pages, and it comprises three main components:

- **Seed URLs** initialize the system with a curated list of starting points (e.g. major sites in sports, movies, music).
- **Frontier** maintains a distributed queue (via Redis) and a deduplicated visited set. This ensures workload balancing and avoids repeated fetches across multiple crawler instances.
- **Worker Pool** consists of concurrent threads or processes:
  - `Fetcher` retrieves page HTML over HTTP(S) with retry and back-off logic for reliability.
  - `Parser` uses BeautifulSoup to extract clean text, title, and out-links, normalizes relative URLs, and filters by domain or robots.txt policies.
  - `Storage` persists raw HTML (hashed filenames) and metadata (URL, title, timestamp) into MongoDB for durable, queryable storage.

This layer is horizontally scalable: you can spin up n instances of `main_distributed.py` on separate machines, all pointing to the same Redis frontier and MongoDB backend, yielding near-linear speedup in pages/sec.

3.2 Indexer Layer

Once pages are stored, the indexer transforms the raw data into a searchable inverted index:

- **Input:** The metadata collection in MongoDB is exported to a JSON-lines file (`metadata.jsonl`), containing {`url, title, filename`}.
- **SparkJob:** A PySpark job reads the file into an RDD, applies:

  map(json.loads) → extract_text → tokenize → (term, doc) → reduceByKey(count).

  Custom partitioning (‘hash(term) $\bmod N_{\text{shards}}$‘)$shardstheindexintoN$ JSON files.
- **Output:** Sharded inverted indices (`shard_0.json`, ..., `shard_3.json`), each mapping terms to posting lists (docID, tf).

The shared-nothing nature of Spark executors allows large-scale parallelism without memory contention. Executors can run across a cluster for even greater throughput.

3.3 Search Layer

The search engine serves real-time queries via an HTTP API:

- **FastAPI + Uvicorn** hosts endpoints for both JSON and HTML responses.
- On startup, each node *preloads* all index shards into memory, building in-process dictionaries for O(1) term lookup.
- For a query:
  1. Tokenize and optionally expand synonyms.
  2. Retrieve posting lists for each term from the in-memory index.
  3. Merge and rank documents using TF-IDF or BM25 heuristics.
  4. Extract snippets by reading the stored HTML (or cached snippet field), highlighting query terms.
  5. Render results via Jinja2 templates (or return raw JSON).
- **Scalability:** Multiple FastAPI instances behind an HTTP load balancer (Nginx or cloud LB) provide elasticity. Sticky sessions are not required, as the service is stateless.

3.4 Asynchronous Decoupled Workflow

- The pipeline is fully asynchronous: the crawler continuously writes into MongoDB/Redis without blocking the indexer or search layer.
- Each stage can be upgraded, scaled, or replaced independently—ensuring high availability and fault isolation.
- For example, index updates can be scheduled hourly via Spark, while the search layer reloads fresh indices without downtime.

This high-level design balances simplicity (clear separation of concerns) with scalability (distributed queues, sharded indices, horizontally scalable services), making it well-suited for production deployment.

4 Low-Level System Design

4.1 Crawler Module

- **Frontier:** In-memory queue (`Frontier`) or Redis-based (`DistributedFrontier`).
- **Fetcher:** HTTP GET with retry and timeout (`requests`).
- **Parser:** Unbiased link extraction, relative URL resolution.
- **Storage:** MD5-hashed HTML files, JSONL metadata.

4.2 Indexer Module

PySpark RDD pipeline:

$$\begin{aligned}&\text{Read metadata.jsonl} \rightarrow \text{RDD} \rightarrow \text{map(json.loads)}\\&\rightarrow \text{map(extract text)} \rightarrow \text{flatMap(tokenize)}\\&\rightarrow \text{reduceByKey(count)} \rightarrow \text{groupByKey}\\&\rightarrow \text{collectAsMap} \rightarrow \text{write index.json}\end{aligned}$$

4.3 Search Module

FastAPI endpoints load index and metadata on startup, compute TF-IDF, extract snippet, highlight terms, and return JSON or HTML.

4.4 Utilities

Text cleaning, tokenization, snippet extraction, highlighting, centralized configuration parameters.

5 Performance Metrics & Testing

5.1 Overview of Metrics

We collected five key performance metrics to evaluate the end-to-end system:

1. **Indexing Performance** (PySpark inverted index build time)
2. **MongoDB Latency** (update, find, drop operations)
3. **Crawl Throughput** (pages stored per second)
4. **Search API Latency** (average, median, p95, max)
5. **System Resource Utilization** (CPU, I/O, network during crawl)

5.2 1. Indexing Performance

- **Command:** `time PYTHONPATH=. python3 indexer/spark_indexer.py`
- **Results:**

Metric	Value
Wall-clock (real) time	18.94 s
User CPU time	2.55 s
System CPU time	0.46 s

- **Analysis:** Most of the 19 s elapsed is spent in Spark startup and I/O (reading from MongoDB, network shuffle), not pure Python work ($\approx$ 3 s CPU).
- **Recommendation:** Persist RDDs, reuse Spark contexts, coalesce partitions, and pre-warm the cluster to reduce overhead.

5.3 2. MongoDB Read/Write/Drop Latency

- **Script:** micro-benchmark via PyMongo command listener
- **Results:**

Operation	Latency (ms)
Upsert (replaceOne)	84.11
Read (findOne)	0.91
Drop (collection)	33.47

- **Analysis:** Reads are sub-ms thanks to in-RAM cache, writes are dominated by network/journaling, and collection drops take tens of ms to deallocate metadata.
- **Recommendation:** Buffer writes in `bulk_write()`, tune write concern, ensure proper indexes on `filename`, and shard for scale.

5.4 3. Crawl Throughput

- **Script:** custom crawler throughput test (20 URLs)
- **Results:**

$$\text{Crawled \& stored 20 pages in 10.20 s} \implies 1.96\ \frac{\text{pages}}{\text{s}}$$

- **System Metrics** (sample `dstat` snapshot):

```
usr sys idl wai stl | read   writ | recv   send
  3   2  95   0   0 |    0  1251k | 258k  9612B
  2   0  96   1   0 |    0    32k | 275k  9817B
  ... (CPU mostly idle, I/O wait 0, network ~200{300KB/s)
```

- **Analysis:** Crawl is *network-bound*, CPU idle 95%, I/O wait 0%.
- **Recommendation:** Increase concurrency (threads or `aiohttp`), reduce or remove `sleep(1)`, and scale horizontally across machines.

5.5 4. Search API Latency

- **Script:** repeated queries against `/api/search`
- **Results:**

	avg (ms)	p50 (ms)	p95 (ms)	max (ms)
Search latency	185.51	3.54	1742.31	2516.46

- **Analysis:** Median 3.5 ms (in-memory index lookup), but cold-start tail 1.7–2.5 s due to shard file I/O, snippet extraction, and template rendering.
- **Recommendation:** Preload index shards into memory, cache snippet results, enable Jinja2 bytecode cache, increase Uvicorn workers (`--workers 4`), and offload HTML rendering for API clients.

5.6 5. System Resource Utilization

We ran `dstat`, `iostat`, and `htop` alongside the crawl test:

- **CPU:** idle 95–98%, minimal user/system usage
- **I/O Wait:** ¡1%, disk not a bottleneck
- **Network:** 200–300KB/s receive, matching crawl rate (2 pages/s)

Recommendations:

- **Network Concurrency:** Increase parallel connections (more threads or async) to better saturate available bandwidth.
- **Distributed Instances:** Deploy multiple crawler instances across machines (sharing Redis) to multiply aggregate pages/sec.
- **Monitoring:** Continue using `htop`, `iotop`, `iftop`, and `mongostat` to observe how resource usage shifts as you optimize.

5.7 Overall Conclusions

- **Crawling** is network-bound at $\approx$ 2 pages/sec; boosting concurrency or adding more nodes is the fastest win.
- **Indexing** takes $\approx$ 19s for the current corpus; optimizing Spark I/O and reusing contexts can reduce overhead.
- **Storage** in MongoDB is read-fast (< 1ms) but write-slow ($\approx$ 84ms); adopting bulk writes and tuning write concern will improve throughput.
- **Search** delivers an excellent median response (< 5ms) but suffers a cold-start tail (1.7–2.5s); preloading index data and caching snippets/templates will flatten the latency curve.
- **System Resources** are under-utilized (CPU and disk), confirming that network and framework overhead dominate; focus optimizations on concurrency and distributed scaling.

6 Scalability & Cost Analysis

6.1 1. Data Volume & Storage Cost

- Assumed text size per page: 100 KB
- For 1 M pages: $1{,}000{,}000 \times 100$ KB ≈ 100 GB
- MongoDB overhead (indexes & replicas): $\sim 2\times$ raw $\rightarrow$ 200 GB
- Example cloud cost (AWS gp3 EBS at \$0.08/GB-month):

$$200 \text{ GB} \times \$0.08/\text{GB-month} = \$16/\text{month}$$

6.2 2. Compute Cost for Crawling

- Observed throughput: ≈ 2 pages/sec per 5-thread instance
- Hourly rate per instance: $2 \times 3600 = 7{,}200$ pages/hr
- Time to crawl 1 M pages on one instance:

$$\frac{1{,}000{,}000}{7{,}200} \approx 139 \text{ hours}$$

- With 10 machines: ≈ 14 hours total
- Spot instance cost (m5.large at \$0.096/hr):

$$10 \times 14 \text{ hr} \times \$0.096/\text{hr} \approx \$13$$

6.3 3. Indexing Cost (Spark)

- Baseline: 19 s to index 50 K pages
- Extrapolate to 1 M pages ($20\times$):

$$19 \text{ s} \times 20 = 380 \text{ s} \approx 6.3 \text{ minutes}$$

- Example cluster: m5.xlarge (4 vCPU) at \$0.192/hr $\rightarrow$ indexing \$0.02 per full run

6.4 4. Search Serving Cost

- Sustained 200 QPS (5 ms p50) $\rightarrow$ 518 M requests/month
- FaaS example cost (\$0.10 per M invocations):

$$518\,\text{M}/\text{M} \times \$0.10 \approx \$52/\text{month}$$

6.5 5. Network Egress

- Crawl bandwidth: 200 KB/s $\rightarrow$ 17 GB/day per node
- AWS inter-AZ egress at \$0.01/GB $\rightarrow$

$$17\ \text{GB/day} \times 30\ \text{days} \times \$0.01/\text{GB} \approx \$5/\text{month}$$

Total Estimated Cost Example: 10 crawler nodes + 1 indexer + 1 search cluster \$300–500/month.

Putting it all together, a minimal 10-node crawler + 4-core indexer + 4-core search cluster prototype might cost on the order of \$300–500/month in cloud resources—well within a small team's budget.

7 Front-End Software Architecture

7.1 Request Flow

1. **Client:** Browser issues GET `/search?q=...`
2. **Uvicorn:** Receives request, passes to FastAPI
3. **FastAPI:** Calls `SearchEngine.search()`, returns Python results
4. **Jinja2:** Renders `search.html` with the results context
5. **Response:** Fully-rendered HTML sent back to browser

7.2 Template Rationale

- **Jinja2 Advantages:**
  - No-JS dependency for bare-bones UX
  - Easy term highlighting with `<mark>`
  - Bytecode cache for faster render on reloads
- **Static Assets:** Served via FastAPI's `StaticFiles` or CDN, with `Cache-Control` headers.
- **Progressive Enhancement:** Base functionality in HTML $\rightarrow$ optional client-side enhancements (e.g. autocomplete) via small React/Alpine.js widget.

8 Assumptions

- **Seed URLs Provided:** Initial seed set in `data/seed_urls.txt`. No dynamic seeding.
- **Centralized MongoDB:** Single primary instance for metadata; no passive/replica backups.

- **Environment Stability:** Network latency within expected bounds; DNS resolves reliably.
- **Resource Availability:** Sufficient CPU/RAM per container; Spark cluster configured for parallelism.
- **Data Format Consistency:** HTML pages parseable by BeautifulSoup; metadata JSONL schema stable.
- **Security Posture:** No crawler credentials required for seed domains.

9 Failures & Challenges

During development and testing we observed several limitations and failure modes:

- **Network Failures:** HTTP timeouts, transient DNS errors, and dropped connections leading to missed or delayed crawls.
- **Data Skew:** Highly link-dense domains (e.g. large aggregators) can overwhelm individual workers and cluster partitions.
- **Resource Exhaustion:** Spark executors occasionally ran out of memory on very large posting lists, causing task retries and slowdowns.
- **Cold–Start Latency:** First-time search queries incur shard file I/O and template compilation overhead, leading to high p95 tail latencies.
- **Index Staleness:** Pages updated on the web after initial crawl remain outdated until the next full re-index.
- **Concurrency Races:** Rare race conditions in Redis frontier when multiple workers push/pop at the same moment, leading to spurious duplicate visits.
- **Cache Invalidation:** Ensuring fresh snippet and ranking caches when content changes is non-trivial without a dedicated invalidation mechanism.

10 Future Work

To enhance functionality, performance, and user experience, we plan the following improvements:

- **Pagination & Infinite Scroll:** Break long result lists into pages or dynamically fetch more results on scroll.
- **Incremental & Real-Time Indexing:** Apply MapReduce or streaming updates to only changed pages, reducing full re-index cost.
- **Autocomplete & Spell Correction:** Provide query suggestions and fuzzy match to improve recall and UX.
- **Learning-to-Rank Models:** Incorporate machine-learned ranking signals (click logs, dwell time) for better relevance.

- **Domain-Aware Politeness:** Respect per-site crawl-delay and robots.txt rules, adaptively throttling by domain.
- **Distributed Cache Layer:** Use Redis or CDN edge caches to serve hot queries and static assets with minimal latency.
- **Analytics & Monitoring Dashboard:** Real-time visualization of crawl progress, index health, QPS, latency, and error rates.
- **Multilingual Support:** Extend tokenization, stemming, and ranking to additional languages (e.g. Korean, Hindi).
- **Mobile-First UI & PWA:** Optimize the front end for mobile devices and consider a progressive web app for offline search.
- **A/B Testing & Experimentation:** Framework to test ranking, UI changes, and new features safely in production.