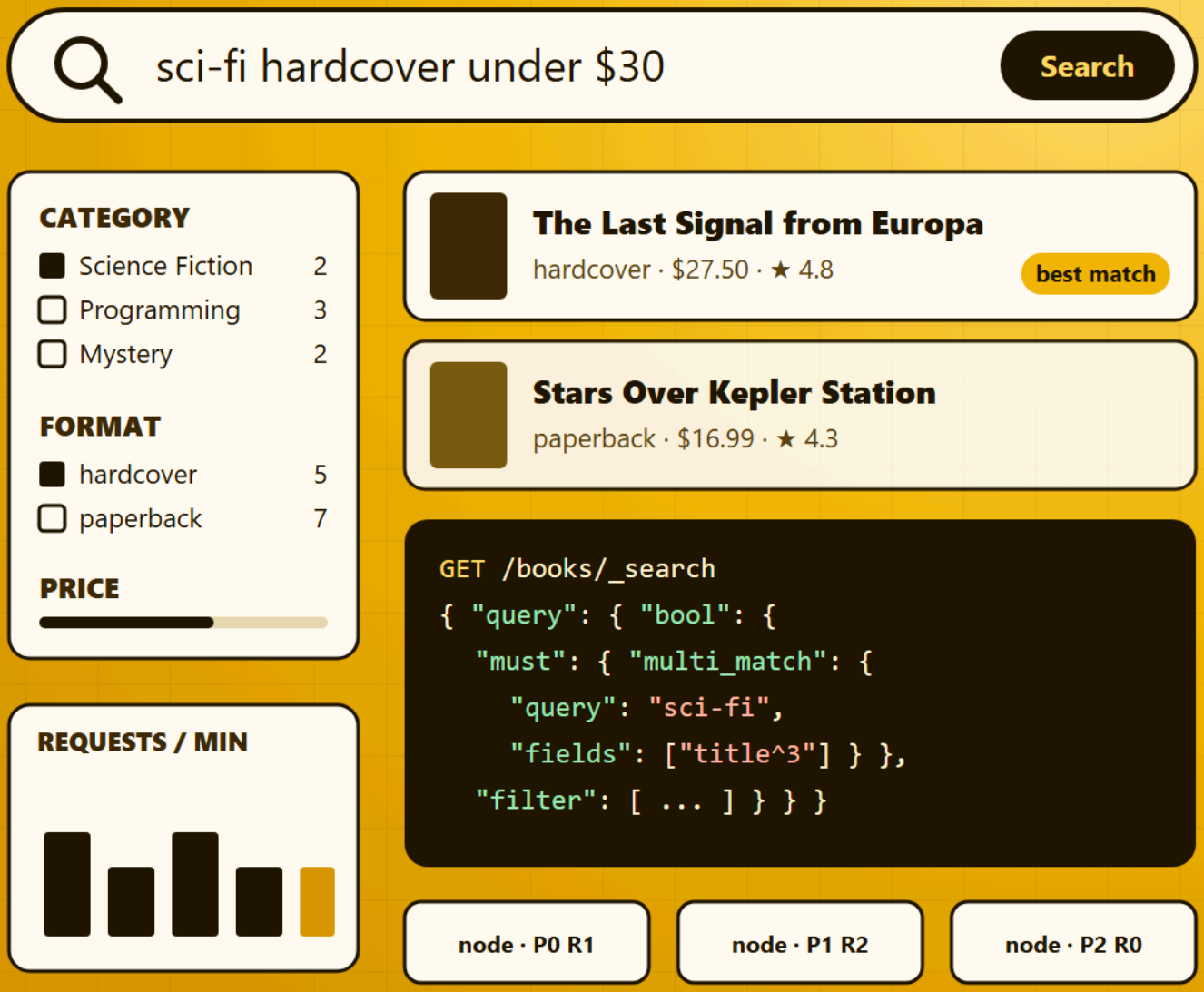


A PRACTICAL GUIDE · FIRST EDITION

Elasticsearch and OpenSearch in Practice

Search, relevance, aggregations, ingestion, and running clusters in production

Query DSL · BM25 · Facets · Data streams · Ingest pipelines · Vectors
ILM & ISM · Security · Monitoring · Snapshots · Upgrades

SEARCH · ANALYTICS · INGESTION · OPERATIONS

THENEXBOOKS

Elasticsearch and OpenSearch in Practice

Search, Relevance, Aggregations, Ingestion, and Running Clusters in Production
From your first query to a secured, monitored, backed-up cluster — built step by step

First Edition

THENEXBOOKS

Elasticsearch and OpenSearch in Practice
Search, Relevance, Aggregations, Ingestion, and Running Clusters in Production

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples, JSON requests, and configuration files may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Elasticsearch, OpenSearch, their consoles, clients, and plugins change frequently; APIs, defaults, licensing terms, and feature availability can differ between versions and subscription tiers. Always consult the official documentation and current license terms for the version you run, and test every change on a non-production cluster first. Nothing in this book is legal advice.

Elasticsearch, Kibana, Logstash, and Beats are trademarks of Elasticsearch B.V., registered in the U.S. and in other countries. OpenSearch is a trademark of LF Projects, LLC. Apache Lucene, Apache, and Lucene are trademarks of The Apache Software Foundation. Docker is a trademark of Docker, Inc. Python is a trademark of the Python Software Foundation. Linux is a registered trademark of Linus Torvalds. Windows is a trademark of the Microsoft group of companies. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Elastic, the OpenSearch Project, the OpenSearch Software Foundation, The Linux Foundation, or The Apache Software Foundation.

Pagewright, the online bookstore and marketplace used as the running example in this book, is fictional, as are its staff, sellers, books, authors, publishers, customers, log events, and all sample data. Any resemblance to real books or people is coincidental. IP addresses use ranges reserved for documentation.

Preface

Every application eventually gets a search box, and every system eventually produces more logs than anyone can read. Both problems look simple at first. A `LIKE '%noodles%'` query finds the cookbook; a `grep` finds the error. Then the catalog grows, customers type "sci-fi" instead of "science fiction", someone asks why the best-selling book appears on page three, the logs arrive from many servers at once, and the simple approach stops working.

Elasticsearch and OpenSearch were built for exactly these problems. They index text so that it can be searched by meaning rather than by exact characters, rank results by relevance, count and group large volumes of documents interactively, and spread the work across a cluster of machines. They share a common heritage — OpenSearch began as a fork of Elasticsearch — and most of what you learn applies to both. Where they have diverged, this book shows you both ways.

We do not stop at the first query. Search engines are easy to start and surprisingly hard to run well: mappings that cannot be changed later, relevance that drifts, shards that multiply, disks that fill up, clusters left open to the internet. This book takes you from the first `GET /` all the way to a secured, monitored, backed-up production cluster, and explains *why* each piece works the way it does, so you can make good decisions on your own systems.

Who This Book Is For

- **Backend developers** who need to add fast, relevant search, filters, facets, and autocomplete to an application.
- **Data and platform engineers** who ingest logs and events and need reliable pipelines, data streams, and retention.
- **DevOps engineers and SREs** who run (or have just inherited) a cluster and need to size, secure, monitor, back up, and upgrade it.
- **Teams choosing between Elasticsearch and OpenSearch**, or moving from one to the other, who want a practical view of what is shared and what differs.

You should be comfortable with JSON, HTTP, and basic Python. No prior search-engine knowledge is assumed.

What You Will Learn

- How search engines work inside: documents, indices, shards, the inverted index, and near-real-time refresh.
- How to design mappings and text analysis — analyzers, tokenizers, token filters, synonyms — so that the right documents match.
- How to write queries with the Query DSL, understand BM25 relevance, and tune ranking with boosts, function scores, and business signals.

- How to build search features people expect: highlighting, autocomplete, "did you mean", sorting, deep pagination, and faceted navigation.
- How to analyze data with aggregations, data streams, dashboards, SQL, ES|QL, and PPL.
- How to ingest data reliably with the bulk API, Python clients, ingest pipelines, and shippers, and how to change mappings with zero downtime.
- How to build a search API in Python, add vector and hybrid search, and measure search quality with real metrics.
- How to run clusters in production: architecture and capacity planning, lifecycle management, security, monitoring, troubleshooting, snapshots, and upgrades.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains what search engines are, sets up a local lab for both products, and covers documents, the REST API, mappings, and text analysis. **Part II — Search and Relevance** (Chapters 6–10) teaches the Query DSL, full-text search, how scoring works, relevance tuning, and the search features users expect. **Part III — Aggregations and Analytics** (Chapters 11–15) covers metrics and bucket aggregations, faceted search, advanced aggregations, log analytics with data streams, and query languages beyond the DSL. **Part IV — Ingestion and Data Modeling** (Chapters 16–19) covers bulk indexing, the Python clients, ingest pipelines, log shippers, data modeling, and zero-downtime reindexing. **Part V — Building Search Applications** (Chapters 20–23) builds a Python search API, adds vector and hybrid search, measures search quality, and covers geo, percolator, and multilingual search. **Part VI — Running Clusters in Production** (Chapters 24–28) covers cluster architecture, index lifecycle management, security, monitoring and performance, snapshots, upgrades, and a complete production case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Pagewright

Throughout the book we build search and analytics for **Pagewright**, a fictional online bookstore and marketplace. Pagewright sells new books itself and lets independent used-book sellers list their own offers next to them. Its catalog is deliberately small — twelve books across programming, science fiction, mystery, cooking, history, and children's titles — so that every hit count, bucket, and total in this book can be checked by hand. On the product side we design the `books` index, fix the "sci-fi returns nothing" problem with synonyms, add facets and autocomplete, expose everything through a small Python API, and experiment with vector and hybrid search. On the operations side we collect Pagewright's web logs into the `pw-weblogs` data stream, parse them with ingest pipelines, chart errors and latency, and finally run the whole system on a secured, monitored, backed-up production cluster.

Versions and Conventions

Examples target **Elasticsearch 8.x and 9.x** and **OpenSearch 2.x and 3.x**, with Kibana and OpenSearch Dashboards as consoles and the official Python clients. Most requests work unchanged on both products. Where the APIs differ — security, lifecycle management, vector search, point in time, query languages, snapshot policies, alerting — we show both versions, Elasticsearch first, and a Note explains the difference. Both products move quickly, so when a feature is new or varies between releases we say "recent versions"; check the documentation for the exact release you run. Licensing and subscription tiers also change over time, and some features are available only in certain tiers or distributions; we point these out and recommend that you read the current terms yourself.

No search engine output in this book should be read as a benchmark. Response excerpts are introduced as "something like this": the hits, counts, and buckets match the sample data, but values such as `took`, generated IDs, node names, and relevance scores will differ on your machine.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, a difference between Elasticsearch and OpenSearch, or a useful aside.

> **WARNING**
> A common mistake or something that can cause data loss, outages, or security problems.

Requests are shown in the Dev Tools console format used by Kibana and OpenSearch Dashboards: an HTTP method and path on the first line, followed by a JSON body. Field names, index names, and code appear in `monospace`; long lines are wrapped to fit the page. Names follow a consistent convention — the `books` alias, the `books-v1` and `books-v2` indices, the `pw-weblogs` data stream, the `pwsearch` Python package — explained in Chapter 2 and summarized in Appendix B.

Contents

Preface

PART I — FOUNDATIONS

1 Search Engines, Elasticsearch, and OpenSearch: What They Are and When to Use Them

2 Setting Up Your Lab: Docker, Security, Dev Tools, and Python Clients

3 Documents, Indices, and the REST API

4 Mappings and Field Types: Designing the Books Index

5 Text Analysis: Analyzers, Tokenizers, and Token Filters

PART II — SEARCH AND RELEVANCE

6 Query DSL Fundamentals: Match, Term, Bool, and Filters

7 Full-Text Search in Depth: Multi-Match, Phrases, Fuzziness, and Highlighting

8 How Relevance Works: BM25, Explain, and Boosting

9 Tuning Relevance: Function Score, Synonyms, and Business Signals

10 Search UX: Autocomplete, Suggestions, Sorting, and Pagination

PART III — AGGREGATIONS AND ANALYTICS

11 Aggregation Fundamentals: Metrics and Buckets

12 Faceted Search: Combining Queries, Filters, and Aggregations

13 Advanced Aggregations: Nested, Pipeline, Composite, and Approximate Results

14 Log Analytics: Data Streams, Time Series Queries, and Dashboards

15 Beyond the Query DSL: SQL, ES|QL, and PPL

PART IV — INGESTION AND DATA MODELING

16 Bulk Indexing and the Python Clients

17 Ingest Pipelines: Parsing, Enriching, and Transforming Documents

18 Shipping Data: Agents, Beats, Logstash, Data Prepper, and Fluent Bit

19 Data Modeling and Reindexing: Nested, Join, Denormalization, and Zero-Downtime Changes

PART V — BUILDING SEARCH APPLICATIONS

20 Building a Search API: Search Templates, Query Builders, and a Python Service

21 Vector and Hybrid Search: Embeddings, kNN, and Rank Fusion

22 Measuring Search Quality: Judgments, Metrics, and Search Analytics

23 Geo, Percolator, and Multilingual Search

PART VI — RUNNING CLUSTERS IN PRODUCTION

24 Cluster Architecture: Nodes, Shards, Replicas, and Capacity Planning

25 Index Lifecycle: ILM, ISM, Rollover, Data Tiers, and Retention

26 Security: Authentication, Roles, TLS, and API Keys

27 Monitoring, Troubleshooting, and Performance Tuning

28 Snapshots, Upgrades, Disaster Recovery, and the Pagewright Production Case Study

APPENDICES

A Elasticsearch and OpenSearch Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What search engines do that databases do not, how Elasticsearch and OpenSearch relate, a local lab for both products, documents and the REST API, the mapping that defines Pagewright's books index, and the text analysis that decides what matches.

1 · Search Engines, Elasticsearch, and OpenSearch

2 · Setting Up Your Lab

3 · Documents, Indices, and the REST API

4 · Mappings and Field Types

5 · Text Analysis

CHAPTER 1 · PART I — FOUNDATIONS

Search Engines, Elasticsearch, and OpenSearch: What They Are and When to Use Them

Before we send a single request, we need a clear picture of what a search engine does that a database does not, where Elasticsearch and OpenSearch came from, how they relate today, and what we will build with them in this book. This chapter gives you that map, introduces Pagewright, and explains the conventions every later chapter uses.

The Problem a Search Engine Solves

Imagine a small online bookstore that keeps its catalog in a relational database. A customer types *kepler station* into the search box. The first version of the search feature, written in an afternoon, looks like this:

SQL — the naive approach

```
SELECT book_id, title, price
FROM books
WHERE title LIKE '%kepler station%'
   OR description LIKE '%kepler station%'
ORDER BY title;
```

It works for exactly one kind of input: the exact substring, in the exact order, with the exact spacing the customer happened to type. It fails in ways customers notice immediately:

- **Word order and partial matches.** *station kepler* returns nothing, even though the customer clearly means *Stars Over Kepler Station*. A pattern match compares characters, not words.
- **Word forms.** A customer who types *star* wants the book with *Stars* in the title. A customer who types *cafe* wants *Murder at the Café Lumière*. Plain substring matching knows nothing about plurals, accents, or letter case beyond what the collation happens to provide.
- **Ranking.** If twenty books mention *mystery*, which one should be first? The query above sorts by title, which is alphabetical, not useful. Customers expect the most relevant result on top: a book with the word in its title should usually beat a book that mentions it once in a long description.
- **Speed at scale.** A leading wildcard (`'%kepler%'`) cannot use an ordinary B-tree index, so the database scans every row. That is fine for a dozen books and painful for millions of rows or log lines.
- **Facets.** Search pages show counts next to filters: *Mystery (2)*, *Paperback (7)*, *Under $15 (3)*. Computing them for every query, quickly, is rarely cheap in a database.

A **search engine** is a system designed around those requirements. It analyzes text into **terms** when documents are stored, keeps a data structure that maps each term to the documents that contain it, scores every match for relevance, and computes counts and statistics over the matching documents in the same request. Elasticsearch and OpenSearch are two search engines that share that design and also go well beyond it: they are **distributed** (one logical index can be spread across many machines), **near real time** (a document becomes searchable about a second after you write it), and they expose everything through a **JSON REST API**.

The four ideas that make it work

Almost every feature in this book is a consequence of four ideas. We introduce them here and return to each one in depth later.

1. **Text analysis.** When a document is indexed, each text field runs through an **analyzer** that splits it into tokens and normalizes them: lowercasing, removing accents, dropping very common words, reducing words to a base form. The same analysis runs on the user's query, so *Stars* and *star*, or *Café* and *cafe*, can meet in the middle. Chapter 5 is entirely about analysis.
2. **The inverted index.** Instead of storing "document → words", the engine stores "word → documents", like the index at the back of a printed book. Finding every document that contains *kepler* is a lookup, not a scan. Chapter 3 draws one by hand.
3. **Relevance scoring.** Every matching document gets a **score** that estimates how well it matches. The default scoring function, **BM25**, rewards rare terms, rewards repeated terms with diminishing returns, and favors shorter fields. Chapters 8 and 9 explain and tune it.
4. **Columnar values for analytics.** Alongside the inverted index, the engine stores field values in a column-oriented structure called **doc values**, which makes sorting, counting, and aggregating fast. That is why the same engine serves both a search box and a log dashboard. Part III is built on it.

What People Use These Engines For

Because of that combination, the same software shows up in projects that look very different from the outside. The most common families of use cases are these:

- **Product and site search.** Online catalogs, documentation sites, job boards, help centers, internal knowledge bases. The priorities are relevance, typo tolerance, autocomplete, facets, and fast responses. Our books index is this kind of project.
- **Log and event analytics.** Applications and servers produce a stream of timestamped events. Engineers want to search them ("show me every 500 error from the search service in the last hour"), count them per minute, and chart latency. The priorities shift toward high ingest rates, time-based indices, retention, and aggregations. Our web log data stream is this kind of project.
- **Observability.** A broader version of log analytics that also covers metrics and traces, usually with agents collecting data from many hosts and dashboards on top.

- **Security analytics.** Searching and correlating authentication events, network flows, and endpoint data to investigate suspicious activity.
- **Vector and hybrid search.** Text, images, or other content turned into numeric vectors (embeddings) by a machine learning model, searched by similarity rather than by shared words, and often combined with classic keyword search. Chapter 21 builds this for Pagewright.

Many teams start with one use case and add another later, which is exactly the path Pagewright follows in this book.

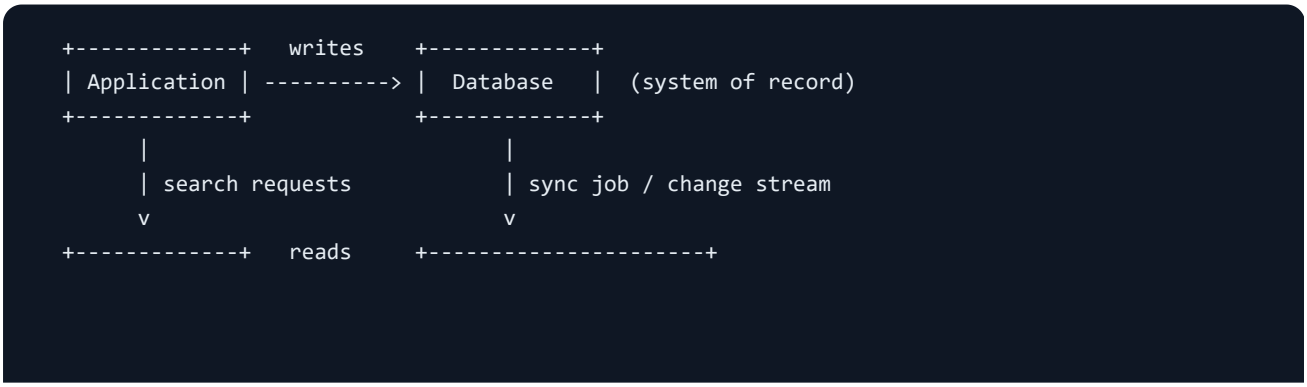
When Not to Use a Search Engine

A search engine is a powerful secondary system and a poor primary one. Knowing where the boundary sits will save you from the most expensive architectural mistakes.

- **Not your system of record.** The authoritative copy of an order, a payment, or a customer account belongs in a database designed for durability and transactions. The search index holds a *copy* of the data, shaped for searching, that you can rebuild from the source if needed. We design Pagewright so that the books index can be deleted and reloaded at any time.
- **No multi-document transactions.** Each single-document write is atomic, and there is optimistic concurrency control per document (Chapter 3), but there is no way to say "update these three documents together or not at all". If your workflow needs that, the transaction happens in the database, and the index follows.
- **Relational joins that change often.** The engine prefers **denormalized** documents, where everything you need to search and display lives inside one document. There are tools for relationships (nested objects, parent/child joins, Chapter 19), but data that is heavily normalized and constantly re-joined in new ways is a better fit for a relational database.
- **Exact counts on huge data with strict guarantees.** Some aggregations are approximate by design, such as distinct counts and percentiles (Chapter 13). That is the right trade-off for dashboards and the wrong one for an invoice.

The typical architecture therefore looks like this: the application writes to its database, a sync process (a job, a change-data-capture stream, or the application itself) sends documents to the search engine, and the search box reads from the search engine.

Typical architecture



```
  | Search API  | ----------> | Elasticsearch or     |
  +-------------+             | OpenSearch cluster   |
                              +----------------------+
```

> **TIP**
>
> Design every index so that you could delete it and rebuild it from the source of truth. That single rule makes mapping changes, upgrades, and disaster recovery much less frightening, because the worst case becomes "reindex" instead of "data loss".

Apache Lucene: The Engine Inside the Engine

Both Elasticsearch and OpenSearch are built on **Apache Lucene**, an open source Java library for full-text indexing and search. Lucene provides the inverted index, the analyzers, the scoring functions, doc values, and the on-disk file formats. It runs inside a single process and knows nothing about networks, clusters, or JSON.

Elasticsearch and OpenSearch wrap Lucene in a distributed system. Each **shard** of an index is a complete Lucene index. The engine decides which machine holds which shard, keeps copies (replicas) in sync, routes each document to the right shard, fans each search out to all relevant shards, and merges the results. It adds the REST API, the JSON Query DSL, aggregations across shards, security, snapshots, and much more. Recent major versions of both products are built on Lucene 10.

You rarely touch Lucene directly, but its vocabulary leaks through everywhere: segments, merges, doc values, norms, analyzers, BM25.

A Short History, and a Note on Licensing

Elasticsearch is built on Apache Lucene and developed by Elastic. In 2021 Elastic changed the license of Elasticsearch and Kibana from Apache 2.0 to a choice of the Server Side Public License (SSPL) or the Elastic License 2.0; in 2024 Elastic added the GNU AGPLv3 as a further option. Some features require a paid subscription tier.

OpenSearch began in 2021 as a fork of Elasticsearch 7.10.2 and Kibana 7.10.2 (the latter as OpenSearch Dashboards), is licensed under Apache 2.0, and in 2024 moved to the OpenSearch Software Foundation under the Linux Foundation.

> **WARNING**
>
> License terms and which features belong to which tier change over time — read the current terms yourself; this book is not legal advice. Before you deploy either product, check the vendor's current license and subscription pages, and involve whoever handles legal questions in your organization if you plan to offer the software as part of a service.

We do not compare the two products on performance or popularity, and we do not recommend one over the other. Both are actively developed. Your choice depends on factors this book cannot weigh for you, such as your team's skills, available managed services, needed features, and license preferences. What this book can do is make you productive with either one.

How the Two Products Relate Today

Because OpenSearch started from the same code base, the two products share a large common core. If you learn that core well, most of your knowledge transfers directly:

- The document and index model, the REST endpoints for documents (`/<index>/_doc/<id>`, `_bulk`, `_mget`), and the `_cat` and `_cluster` APIs.
- The Query DSL: `match`, `term`, `bool`, `range`, `multi_match`, `function_score`, `nested`, highlighting, and the rest.
- Mappings and field types, the analysis chain, and the `_analyze` API.
- Aggregations: metrics, buckets, pipelines, and composite paging.
- Index templates, component templates, aliases, ingest pipelines, data streams, and snapshots.

Since the fork, each project has added features independently, sometimes solving the same problem with different APIs. Those differences cluster in a few areas, and this book shows both syntaxes every time we touch one of them. The table below is a map of where the paths diverge; each row is covered in detail in the chapter noted.

Area	Elasticsearch	OpenSearch	Chapter
Security APIs	Built in, on by default since 8.0; `_security/...`	Security plugin; `_plugins/_security/api/...`	2, 26
Index lifecycle	ILM (`_ilm/policy`), data stream lifecycle	ISM (`_plugins/_ism/policies`)	25
Vectors	`dense_vector` field, `knn` search option	`knn_vector` field, `index.knn`, `knn` query	21
Query languages	SQL and ES|QL	SQL and PPL	15
Point in time	`POST /books/_pit`	`POST /books/_search/point_in_time`	10
Snapshot policies	SLM (`_slm/policy`)	Snapshot Management (`_plugins/_sm`)	28
Alerting	Kibana rules (Watcher in some tiers)	Alerting plugin monitors	27
Console	Kibana	OpenSearch Dashboards	2, 14
Shippers	Elastic Agent, Filebeat, Logstash	Data Prepper, Fluent Bit, Logstash with a plugin	18

Coordination role	`master`	`cluster_manager` (since 2.0)	24
Python client	`elasticsearch`	`opensearch-py`	2, 16

A few of these rows deserve a sentence more right away, because they affect how you work from day one.

Security. Elasticsearch has had security switched on by default since 8.0: a fresh install generates a password for the built-in `elastic` user and sets up TLS. OpenSearch ships a Security plugin that the default Docker image enables with demo certificates, and since 2.12 its demo setup refuses to start without an initial admin password that you provide. Either way, the days of an unauthenticated cluster on port 9200 are over, and Chapter 2 sets up both labs with passwords from the start.

Clients. The official Python clients are *not* interchangeable. The `elasticsearch` client verifies that it is talking to Elasticsearch and refuses to work with OpenSearch, so OpenSearch users install `opensearch-py`. Match the client's major version to the server's major version. Our Python package hides this choice behind a single setting, `PW_ENGINE`, so the rest of the code does not care.

Plugin paths. Many OpenSearch features live under the `_plugins/` path prefix, so a path that contains `_plugins` is almost certainly OpenSearch syntax.

> **NOTE**
>
> **Elasticsearch vs OpenSearch.** The two products share a common ancestor, not a common roadmap. A feature with the same name may behave slightly differently, and a feature that exists in one may arrive later, differently, or never in the other. Throughout this book we say "recent versions" whenever a feature is not available in every minor release, and we recommend you keep the reference documentation for *your* version open while you work.

Running It Yourself or Using a Managed Service

You can run either engine yourself, on servers, virtual machines, or Kubernetes. That gives you full control and makes you responsible for upgrades, backups, monitoring, and security patches.

Alternatively, several cloud providers offer managed services. Elastic Cloud runs Elasticsearch and Kibana; Amazon OpenSearch Service runs OpenSearch and OpenSearch Dashboards; other providers offer hosted versions of one or both. A managed service takes over much of the operational work, at the price of some control and with its own feature set and pricing, which change too often for a book to describe. Compare current offerings yourself.

Everything in Parts I to V works the same against a self-managed or a managed cluster, as long as you can reach the REST API. Part VI takes the self-managed point of view, because that is where you need to understand the machinery.

Meet Pagewright

Throughout this book we work on one realistic project instead of a collection of unrelated snippets. **Pagewright** is a fictional online bookstore and marketplace. It sells new books itself and also lets independent used-book sellers list offers for the same titles, so a single book page can show "new from Pagewright" next to "used, good condition, from a small shop".

Three people at Pagewright will show up now and then:

- **Nadia Brooks** is the search engineer. She owns the product search: the books index, relevance tuning, autocomplete, facets, and the Python search API.
- **Marcus Hale** is the platform and SRE engineer. He runs the clusters and owns log analytics: the web log data stream, ingest pipelines, dashboards, lifecycle, alerting, and backups.
- **Elena Ruiz** is the merchandising lead. She asks the business questions that drive the work, such as "which in-stock books can we feature under $20?" or "why does *sci-fi* return nothing?"

You work alongside them.

Project one: product search

The catalog is deliberately small, twelve books, so that you can check every result by eye and every count we print is easy to verify. Small does not mean simple: the data has multi-valued categories, multiple authors, accented characters, books that are out of stock, and nested seller offers with different conditions and prices. Those details are exactly what makes real catalogs hard. Here are the books, split into two narrow tables:

ID	Title	Categories	Price (USD)
PW-1001	Python for Busy Analysts	Programming, Data	34.99
PW-1002	Running Distributed Systems	Programming	54.00
PW-1003	The Search Engine Cookbook	Programming, Data	29.00
PW-1004	Stars Over Kepler Station	Science Fiction	16.99
PW-1005	The Last Signal from Europa	Science Fiction	27.50
PW-1006	Murder at the Café Lumière	Mystery	14.50
PW-1007	The Harbor Street Mystery	Mystery	9.99
PW-1008	Weeknight Noodles	Cooking	32.00
PW-1009	Baking Bread at Home	Cooking	19.95
PW-1010	A Short History of Maps	History	38.00
PW-1011	Rivers of the Ancient World	History	22.00
PW-1012	The Little Robot Who Could Not Sleep	Children	12.99

ID	Authors	Format	In stock
PW-1001	Mara Linwood	paperback	yes
PW-1002	Tobias Renner, Lena Fischer	hardcover	yes
PW-1003	Asha Varghese	paperback	yes
PW-1004	Jonas Vale	paperback	yes
PW-1005	Ines Moreau	hardcover	no
PW-1006	Clara Whitcombe	paperback	yes
PW-1007	Rafael Duarte	paperback	yes
PW-1008	Kenji Arakawa	hardcover	yes
PW-1009	Hanna Lindqvist	paperback	yes
PW-1010	Theo Marchetti	hardcover	yes
PW-1011	Amara Obi	paperback	no
PW-1012	Poppy Ashford	hardcover	yes

Each book also has a description, a publisher, a publication date, a page count, a rating with a rating count, sales over the last 30 days, tags, and a list of **offers**. There are three sellers: `S01` Pagewright Direct (the store itself, selling new copies), `S02` Lantern Lane Books, and `S03` Second Chapter Used Books (both selling used copies). Offers carry a condition (`new`, `like_new`, `good`, or `acceptable`), a price, and a quantity. A book is in stock exactly when at least one of its offers has a quantity above zero, which is why *The Last Signal from Europa* and *Rivers of the Ancient World* show "no" above.

Over Parts I to V, Nadia's product search grows from a single `match` query into a complete search service: a carefully designed `books` index, relevance tuned with business signals, synonyms so that *sci-fi* finds science fiction, facets with correct counts, autocomplete, a Python API with tests, vector and hybrid search, and an evaluation harness that tells us whether a change actually made search better.

Project two: log analytics

Pagewright's storefront, search API, and checkout services all write access logs. Marcus wants to answer questions like "how many requests failed in the last hour, and from which service?" and "is search getting slower?" We use a sample of twelve web log events from a single morning, small enough to verify by hand, with a few successes, one not-found page, and two server errors. In Part III we load them into a **data stream** called `pw-weblogs` and chart them; in Part IV we parse the same events from raw text lines with an ingest pipeline and look at the shippers that move logs in production; in Part VI we give the stream a lifecycle policy, alerting, and backups.

Finally, Part VI turns both projects into a production deployment called `pagewright-prod`: dedicated cluster-manager nodes, hot and warm data nodes, TLS, roles and users, monitoring, snapshots, and an upgrade plan.

The Plan of the Book

The book has six parts, and each builds on the previous ones.

- **Part I — Foundations** (Chapters 1–5): the lab, documents and the REST API, mappings, and text analysis. By the end you have the `books` index loaded and you understand why each field is mapped the way it is.
- **Part II — Search and Relevance** (Chapters 6–10): the Query DSL, full-text search, how BM25 scoring works, relevance tuning, and search UX features such as autocomplete and pagination.
- **Part III — Aggregations and Analytics** (Chapters 11–15): metrics and buckets, faceted search, advanced aggregations, log analytics with data streams, and the SQL-like query languages.
- **Part IV — Ingestion and Data Modeling** (Chapters 16–19): bulk indexing from Python, ingest pipelines, log shippers, data modeling, and reindexing with zero downtime.
- **Part V — Building Search Applications** (Chapters 20–23): a search API in Python, vector and hybrid search, measuring search quality, and geo, percolator, and multilingual search.
- **Part VI — Running Clusters in Production** (Chapters 24–28): architecture and capacity, lifecycle, security, monitoring and troubleshooting, and snapshots, upgrades, and the full case study.

If you are new to search, read Parts I and II in order. If you have inherited a running cluster, read Chapters 1 to 4, then jump to Part VI.

Conventions Used in This Book

Requests in Dev Tools format

Most examples are written in the **console format** used by Dev Tools in both Kibana and OpenSearch Dashboards: an HTTP method and a path on the first line, followed by an optional JSON body. You can paste these straight into the console, which handles the host, the authentication, and the content type for you. A caption above each block tells you which product it applies to.

Both — Dev Tools

```
GET /books/_search
{
  "query": { "match": { "title": "kepler" } }
}
```

When the two products differ, we show two blocks, Elasticsearch first, each with its own caption, or we point out the difference in a note titled "Elasticsearch vs OpenSearch". We never silently show only one product's syntax for a feature that differs.

curl and Python

Some tasks are easier from a terminal, such as loading a data file. We show those as **curl** commands in bash/zsh style. Against the Elasticsearch lab they use `http://localhost:9200` and the `elastic` user; against the OpenSearch lab they use `https://localhost:9201` and the `admin` user, with `-k` because the lab uses demo certificates. Chapter 2 explains both, including how to run them from Windows PowerShell. Application code is Python, using the official clients, organized in a small package called `pwsearch` that grows chapter by chapter.

Responses are "something like"

When we print a response, we introduce it with "you should see something like this" and trim it with `...` where it gets long. Values that depend on the data, such as hit counts, which documents match, and bucket counts, sums, and averages, match the sample data exactly, and you can rely on them to check your work. Values that depend on your machine, such as `took` (milliseconds), node names, IDs, and internal sequence numbers, will differ.

Relevance **scores** deserve a special mention. They depend on index statistics that can vary with the number of shards, deleted documents, and version details, so we never claim exact scores. When a response shows a `_score`, treat it as a plausible illustration; the *order* of results is what we discuss.

Versions

This book targets Elasticsearch 8.x and 9.x and OpenSearch 2.x and 3.x. We avoid tying features to specific minor releases. When a feature is new or still changing, we say "recent versions", and when a feature is marked as a preview or technical preview by its vendor, we say so. The lab in Chapter 2 asks you to pick the current release from each vendor's download page rather than hard-coding a version number that would be out of date by the time you read this.

> **NOTE**
>
> Everything in the lab is **lab only**: single-node clusters, zero replicas, a plain HTTP endpoint for Elasticsearch, demo certificates for OpenSearch, and example passwords printed in the book. We call these shortcuts out each time we take one, and Part VI shows what changes in production.

A First Look at the API

You will set up the lab in the next chapter, but it helps to see the shape of a complete exchange now. A search engine answers JSON with JSON. When we later ask the `books` alias for in-stock books that mention mysteries, the request and a trimmed response look like this:

Both — Dev Tools

```
GET /books/_search
{
  "query": {
    "bool": {
      "must": [ { "match": { "title": "mystery" } } ],
      "filter": [ { "term": { "in_stock": true } } ]
    }
  },
  "_source": ["title", "price"]
}
```

You should see something like this:

Response (something like)

```
{
  "took": 4,
  "timed_out": false,
  "_shards": { "total": 1, "successful": 1, "skipped": 0, "failed": 0 },
  "hits": {
    "total": { "value": 1, "relation": "eq" },
    "max_score": 1.2,
    "hits": [
      {
        "_index": "books-v1",
        "_id": "PW-1007",
        "_score": 1.2,
        "_source": { "title": "The Harbor Street Mystery", "price": 9.99 }
      }
    ]
  }
}
```

The `bool` query combines a scored full-text clause (`must`, "the title should match *mystery*") with an unscored yes/no clause (`filter`, "only in-stock books"). The response tells us how long the search took inside the engine, how many shards answered, how many documents matched in total, and then lists the hits, each with its index, its ID, its relevance score, and the fields we asked for. Only *The Harbor Street Mystery* has the word in its title; the other mystery, *Murder at the Café Lumière*, mentions it only in its description, which this query does not search. Whether that is what Elena wants is a relevance question, and answering such questions well is what Part II is about. The score shown here is illustrative; yours will differ.

Notice also that the request used `books` but the hit reports `books-v1`. The name we search is an **alias**, a stable name that points at a real index. Chapter 4 creates it, and Chapter 19 uses it to swap in a redesigned index without the application noticing.