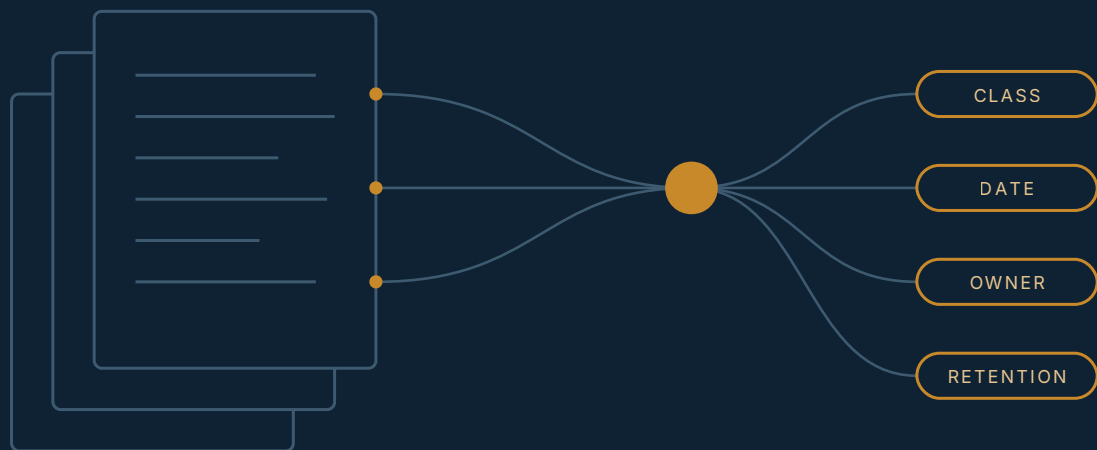


REAL-WORLD ARCHITECTURES

VOLUME 2

Document Management

Metadata-Driven Search over Documents and Records



MARIO NOIOSO

FIELD GUIDES ON DATA, SYSTEMS AND INSTITUTIONS

REAL-WORLD ARCHITECTURES · VOLUME 2

Document Management

Metadata-Driven Search over Documents and Records

Mario Noioso

Software architect and author · Rome

FREE EDITION · NOT FOR SALE · CC BY-NC-ND 4.0

REAL-WORLD ARCHITECTURES, VOLUME 2

Document Management: Metadata-Driven Search over Documents and Records

First edition, 2026. Version 1.0.

Distributed free of charge. Not for sale.

AUTHOR

Mario Noioso · ORCID 0009-0000-9052-944X · ISNI 0000 0004 2128 1497

<https://marionoioso.com>

LICENCE

© 2026 Mario Noioso. This work is licensed under the Creative Commons Attribution, NonCommercial, NoDerivatives 4.0 International licence (CC BY-NC-ND 4.0). You may share it freely, in unmodified form and with attribution, for non-commercial purposes.

DISCLAIMER

The views expressed in this guide are the author's own and do not represent those of his employer or of any organization he works or has worked with. This guide is an independent, non-commercial publication, distributed free of charge, and is not sponsored, reviewed or endorsed by MongoDB, Inc. MongoDB is a trademark of MongoDB, Inc.; other product names are trademarks of their respective owners and are used for identification only. Industry scenarios and the worked example are composites built from common architectural patterns and do not describe any specific organization. This guide is provided for information and does not constitute legal, regulatory or professional advice for any specific system.

ALSO IN THIS SERIES

Volume 1 · Single View: One Governed View of Customers and Citizens (2026). DOI 10.5281/zenodo.23205299.

CONTENTS

	About this guide	5
1	Executive summary	6
2	The document sprawl problem	8
3	What a metadata layer is, and what it is not	10
4	Why the document model fits	13
5	Reference architecture	16
6	The data model	20
7	Search	26
8	Lifecycle, retention and records	31
9	Governance, security and sovereignty	36
10	Industry patterns	39
11	A worked example	42
12	Anti-patterns	44
13	Adoption roadmap and checklist	46
14	References	49
15	About the author	51

PREFACE

About this guide

Audience

Enterprise and solution architects, platform and integration leads, records managers and technical decision makers in public administrations, banks, insurers, energy and engineering companies, telecom operators, publishers and software vendors who build or sell document management platforms, and the delivery teams who modernize them.

Prerequisites

Familiarity with enterprise content management concepts (repositories, versions, renditions, access control), with basic data modelling and with the general principles of records management and data protection regulation. Prior experience with document databases helps but is not required. Readers of Volume 1 will recognize several patterns, in particular the ingestion contract and the provenance model; this volume is self-contained.

Scope and limits

The guide describes an architectural pattern, a governed metadata layer with search over documents and records, and the decisions that make it work in production. It is not a product manual, a sizing guide, a digital preservation standard or a legal opinion. Regulatory and standards references are illustrative: obligations depend on jurisdiction, sector and the specific processing, and must be assessed with the organization's own legal, records and data protection functions.

Environment and verification status

The reference implementation uses MongoDB 8.0 syntax. Search examples use MongoDB Search (Atlas Search) and MongoDB Vector Search syntax. Unless stated otherwise, capabilities are described as available in MongoDB Atlas; availability in self-managed deployments depends on edition and version, summarized in Chapter 4. Code listings in this edition have been reviewed statically for syntax and consistency; they have not yet been executed against a declared reference environment. Readers should test them in their own environment before relying on them, and corrections are welcome.

CHAPTER 1

Executive summary

Organizations rarely lose documents. They lose the ability to find the right one, to prove which version counts, and to delete it when the law says so.

A large organization holds millions of documents: contracts, invoices, technical drawings, correspondence, reports, rulings, medical records, permits. They sit in an enterprise content management platform bought fifteen years ago, in a second one inherited through an acquisition, in a groupware application that nobody dares to switch off, in file shares, in e-mail, in the attachment tables of a dozen line-of-business systems. Each repository has its own notion of a folder, its own metadata fields, its own access model and its own search box. None of them can answer the question a case worker, an underwriter, an engineer or an auditor actually asks: which documents, across everything we hold, are about this matter, in force today, and visible to me.

A metadata layer is the architectural answer to that gap: one governed record for every document and every version, carrying its classification, its declared and extracted metadata, its access rules, its lifecycle state and a pointer to the content wherever that content lives, continuously synchronized with the repositories and the capture channels, and served through governed search and retrieval interfaces. It does not necessarily replace the repositories, which may keep custody of the files. It takes ownership of the description of the files, and of the search over them.

This guide explains how to build one that survives production. It covers the problem, the reference architecture, the data model, the part that decides adoption (search), the obligations that come with keeping records (registration, retention, legal hold, disposition), a worked example, and the mistakes that sink many programmes.

THE ARGUMENT IN FIVE POINTS

1. Document sprawl is the natural result of decades of departmental investment. The files are seldom the problem; the metadata describing them is, because it is inconsistent, incomplete and scattered.
2. The metadata layer works when it is built as an operational registry: one record per document and per version, classified, validated per class, permission-aware, continuously synchronized with where the content lives, and owned by a clear team.
3. The document model fits because document metadata is polymorphic by class, hierarchical (document, versions, renditions, extracted entities) and keeps evolving as new classes and regulations appear.
4. Search is the product. Metadata filters, full text, facets, autocomplete and semantic retrieval must work in one query, trimmed by permissions, with a stated consistency contract. A search engine bolted on as a workaround becomes a second system to keep in sync.
5. Keeping records concentrates obligations. Registration, retention schedules, legal hold, disposition, audit and erasure must be designed into the data model and into every access path from the first release, not added after go-live.

Organizations that get this right gain more than a faster search box. They gain a governed inventory of what they hold: consistent answers to auditors and regulators, proof of which version was in force on a given date, defensible deletion, and a trusted foundation on which document intelligence and AI retrieval can operate without inventing documents or leaking them.

CHAPTER 2

The document sprawl problem

Every department bought the right document system for its own process. Together, they bought a problem.

Sprawl accumulates through reasonable decisions. A regulatory deadline is met with a capture and archiving solution for one document class. A new business line arrives with its own content platform. An acquisition brings a second repository with a different folder philosophy. A collaboration tool is adopted for engineering drawings because the official platform cannot handle their size. A workflow application stores attachments in its own database because integrating with the repository was on the critical path. Twenty years later, the organization runs several repositories, dozens of metadata schemas and a thousand folder conventions, and the same contract exists as a scanned PDF in three places, each with a different date.

How sprawl shows up

- Knowledge workers search in several systems, then in e-mail, then ask a colleague. Finding the right document is measured in minutes or hours, not seconds.
- Nobody can say with confidence which version of a document is in force, because versions live in different repositories and the "final" copy was e-mailed.
- Scanned archives were digitized years ago, but the text produced by optical character recognition was never indexed, so millions of pages are stored and invisible.
- A separate search engine was introduced because the repository's own search was too slow at volume. It now needs its own synchronization jobs, its own permission model and its own operations team, and it drifts from the repository in ways that surface only when a user finds a document that was deleted.
- Requests from auditors, regulators, courts or citizens exercising access rights take weeks, and the answer depends on which department compiles it.
- Retention schedules exist on paper. In practice nothing is ever deleted, because nobody can prove that a document is not under hold somewhere else.
- AI initiatives stall: a model cannot be grounded on documents that cannot be located, deduplicated, dated or filtered by who is allowed to see them.

Why replacing the repositories is rarely the answer

The instinctive response is to migrate everything into one new content platform. In practice this is a multi-year programme with high execution risk: content migration at scale is slow, mappings between metadata schemas are lossy, legacy integrations are undocumented, and the business cannot wait for it to finish before finding its documents. Many migrations stop halfway and leave the organization with one more repository.

The metadata layer takes a different path. It leaves content where it is, at least initially, and builds a layer that describes, classifies, indexes and serves it. It can deliver value in months, make the repositories' eventual migration or decommissioning a gradual choice rather than a prerequisite, and, where a repository is being replaced, take over both metadata and content in a controlled sequence.

The cost of doing nothing

The cost of sprawl rarely appears as a single budget line, which is why it persists. It is spread across time spent searching, duplicated storage and licences, failed audits, penalties for late responses to access requests, legal exposure from documents that should have been deleted, and the inability to use the organization's own documents as a knowledge base. The quality of any document intelligence or retrieval-augmented system is bounded by the quality of the inventory beneath it.

A USEFUL DIAGNOSTIC

Count the repositories that hold documents, the distinct metadata schemas across them, and the number of times in a week that someone asks "is this the latest version?". In the author's experience, once there are several independently maintained repositories with their own search, the organization is already paying for a metadata layer. It is simply paying for several inconsistent ones instead of one governed one.

CHAPTER 3

What a metadata layer is, and what it is not

Many programmes that struggle did not fail at building a document platform. They failed at agreeing on what owned what.

A working definition

A metadata layer for document management is an operational data layer that maintains one governed record per document and per version, assembled from capture channels and existing repositories, continuously synchronized with them, carrying classification, declared and extracted metadata, access rules, lifecycle state and a verified pointer to the content, and served through governed search and retrieval interfaces to the applications, processes and people that need it.

Four properties distinguish it from adjacent concepts:

- **Descriptive:** metadata is a first-class, validated structure, not a bag of free-text properties. Each document belongs to a class, and each class defines what must be known about its documents.
- **Searchable:** filters, full text, facets and semantic retrieval are served from the layer itself, with interactive latency, trimmed by permissions.
- **Authoritative for description:** the layer owns the metadata, the classification and the lifecycle state of every document. It may or may not own the content.
- **Lifecycle-aware:** every record knows whether it is a draft, a registered record, under hold, due for disposition or disposed, and the transitions are controlled and audited.

Neighbouring patterns and their role in this architecture

The table describes the prevalent purpose of each pattern and the role it plays in the architecture proposed here. It does not describe the universal limits of any technology: many products in each category have broadened their scope over time.

PATTERN	PREVALENT PURPOSE	ROLE IN THE PROPOSED ARCHITECTURE
Content repository (ECM, DMS)	Store content and versions, folders, check-in and check-out, workflows.	May remain the custodian of content. The metadata layer indexes it and, where agreed, routes metadata changes back to it. In shape C below, the layer replaces it.
Records management and archive	Declare records, apply file plans and retention, prove authenticity, dispose.	Complementary. The layer implements registration, retention and hold operationally; long-term preservation remains a specialized function (Chapter 8).
Object storage	Durable, cheap, scalable storage of binary content.	The preferred content store for shapes B and C: files live in buckets, the layer holds the pointer, the hash and the metadata.

PATTERN	PREVALENT PURPOSE	ROLE IN THE PROPOSED ARCHITECTURE
Enterprise search engine	Full-text indexing across sources.	Its function is absorbed by the layer's search capability, so that index, metadata and permissions stay in one system. A separate engine is a workaround, not a design.
Data lake or lakehouse	Large-scale storage and processing for analytics and ML.	Complementary. May receive extracted text and metadata for analytics; not the place for interactive, permission-aware retrieval.
Document intelligence (OCR, extraction, LLMs)	Turn content into structured information.	A producer of extracted metadata, with confidence and provenance recorded in the layer. Not the system of record for what was extracted.

RULE OF THUMB

Never let a field of a document's description have two owners. For every field it must be possible to answer one question: which system is allowed to change it? Classification, dossier membership, lifecycle state, access rules and the search projection are always owned by the metadata layer. In shape A the repository keeps ownership of the fields it manages operationally, such as check-out state, its own version numbering and its folder; the ownership register lists them, and the synchronization contract carries each field in one direction only.

Three deployment shapes

Decide explicitly which shape each programme adopts, because ownership and failure modes change at every step. Shapes are not exclusive: a platform often starts as A for legacy repositories and runs as B for new content.

- **Shape A, index over existing repositories.** Content stays in the repositories, which remain custodians and keep their check-in and check-out. The layer holds metadata, search and lifecycle state and a pointer to the content. Safest starting point; delivers unified search early; depends on a reliable synchronization contract with each repository.
- **Shape B, metadata layer plus object storage.** The platform is the document management system. Content lives in object storage, immutable per version, protected by bucket policies and object lock where required. The layer owns metadata, versions, renditions, permissions and lifecycle. This is the reference shape of this guide.
- **Shape C, content inside the database.** Content is stored in the database itself, for example through the GridFS convention. Appropriate for moderate volumes, for isolated or air-gapped environments without object storage, and when a single consistent backup of content and metadata matters more than storage cost. The consistency is in the backup, not in the write: a GridFS upload spans many chunk documents and is not covered by a multi-document transaction, so the registry record is still written only after the upload completes, exactly as with object storage. It is not the default at scale.

OWNERSHIP IN SHAPE A

When a repository stays in operation, its users keep changing documents inside it: they check files out, move them between folders, rename them, add versions. Those changes reach the layer through the synchronization contract and are applied to the fields the repository owns.

The layer's own fields (class, dossier, lifecycle state, access rules, extracted metadata) are never overwritten by a repository event; where a repository needs to see one of them, for example a hold that must block check-out, the layer writes it back through an explicit interface, and the ownership register records that this field flows from the layer to the repository. A field that must be editable in both places is a design error to be removed, not a conflict to be resolved at run time.

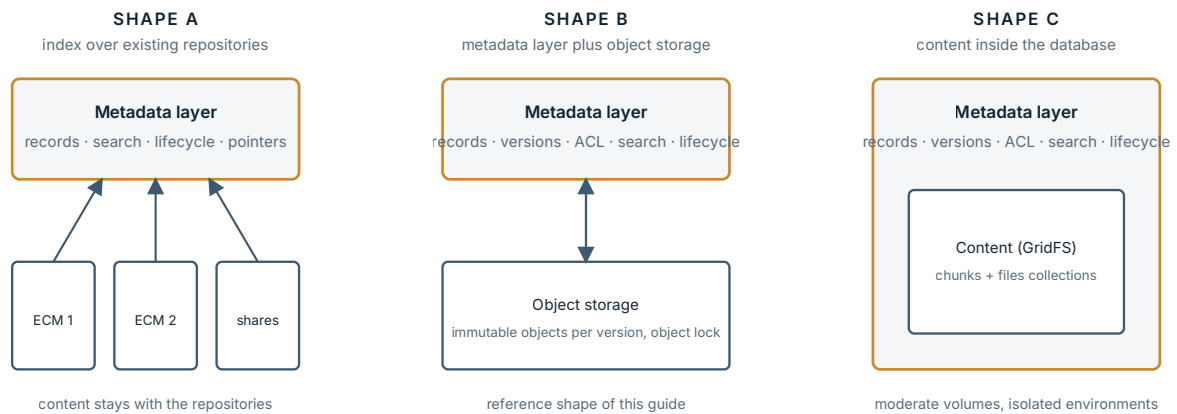


Figure 1. Three deployment shapes. Ownership of content changes; ownership of metadata, search and lifecycle does not.

CHAPTER 4

Why the document model fits

A document's description is itself a document: nested, class-specific, and never quite finished.

A metadata layer can be built on many technologies. This guide uses a document database, MongoDB, as its reference implementation, because the shape of the problem maps closely onto the shape of the model, and because search, vectors, events and encryption are properties of the same record rather than separate systems. The patterns remain valid on other platforms; the effort required to implement them changes.

Four properties of the problem

THE METADATA IS POLYMORPHIC

An invoice carries a supplier, an amount and a due date. A contract carries parties, an effective date, a termination clause and a governing law. A technical drawing carries a plant, a revision and a discipline. A court ruling carries a court, a case number and a decision date. A registered letter carries a protocol number, a sender and a file-plan code. In a relational model these become either one sparse table with hundreds of nullable columns, or one table per class plus a generic property table that nothing can validate. In a document model, each record carries a common core plus a class-specific sub-document, validated per class, in one collection and one index set.

THE METADATA IS HIERARCHICAL

A document has versions; a version has renditions (original, PDF/A, thumbnail, OCR text); a page has extracted entities; a document belongs to a dossier, which belongs to a file-plan node. Reading the operational record of one document (its current version, its renditions, its classification, its permissions) is a single indexed read of one document, not a join across six tables.

THE METADATA KEEPS CHANGING

New document classes arrive with every regulation, product and acquisition. Extraction models improve and produce fields that did not exist a year ago. Additive changes can be absorbed without a blocking schema migration. Not all change is free: validators, indexes, application contracts and backfills must still be versioned and deployed; the model lowers the cost of evolution, it does not remove the discipline.

THE DATA IS READ FAR MORE THAN IT IS WRITTEN, AND WRITTEN IN BURSTS

A platform ingesting two hundred thousand documents a day serves many times that in searches and retrievals. The model is optimized for the dominant read paths: the operational record is assembled once at ingestion, and search indexes are maintained from it. Bursts from

scanning backlogs and migrations are absorbed by horizontal write scaling, with the ingestion contract of Chapter 5 making every write idempotent.

Capabilities used in this guide and where they are available

CAPABILITY	ATLAS	SELF-MANAGED	NOTES AND LIMITS
Documents, indexes, JSON Schema validation, multi-document transactions	Yes	All editions	Transactions require a replica set or sharded cluster. Used for registration, versioning and disposition.
MongoDB Search: full text, facets, autocomplete, highlighting, synonyms	Yes	Community and Enterprise, with the separate mongot search process; generally available since June 2026 for supported version combinations	Indexes are eventually consistent with the collection. Facets are computed by the search layer on fields indexed as token, number or date. Fields encrypted client-side cannot be searched.
MongoDB Vector Search with pre-filtering	Yes	As above	Filters apply only to fields declared as filter fields in the vector index. Embeddings are produced outside the database.
Hybrid search: \$rankFusion, \$scoreFusion	Yes	8.0 and later (\$rankFusion); 8.3 and later (\$scoreFusion)	All sub-pipelines must run against the same collection; cross-collection fusion is done in the application or with \$unionWith (Chapter 7).
Change streams, with pre- and post-images	Yes	All editions	Pre- and post-images must be enabled per collection. Resumability is bounded by the oplog window.
GridFS	Yes	All editions	A driver convention over two collections, not a storage engine. Content is chunked; streaming and range reads are supported; an upload is not a transaction, so metadata is written after it completes. Shape C only.
TTL indexes	Yes	All editions	Delete documents silently after a time. Suitable for caches and ephemeral renditions; not a records disposition mechanism (Chapter 8).
Zone sharding for residency	Yes (Global Clusters)	All editions	Requires a shard key that includes the zone field. Content store, search nodes and backups must be located consistently (Chapter 9).
Client-side field-level encryption and Queryable Encryption	Yes	Automatic encryption: Enterprise. Explicit encryption: all editions	Queryable Encryption supports equality and, from 8.0, range queries on encrypted fields. A collection using it cannot be indexed for Search.
Database auditing	Yes, dedicated tiers	Enterprise	Auditing successful reads requires explicit configuration and has a performance cost.
Online Archive and Data Federation	Yes	Not available	Can move cold metadata and event logs to object storage while keeping them queryable; subject to their own latency.

Availability changes over time. Verify current documentation for the edition and version you deploy.

Consistency of reads

Reads from the primary see the latest acknowledged writes. Reads from secondaries scale capacity but may be slightly behind. Search and vector indexes are maintained asynchronously: a document registered a moment ago is visible by identifier immediately and in full-text search a few seconds later. This distinction must be exposed, not hidden: the API that confirms a registration returns the identifier and an indexing status, and the user interface shows "indexing" rather than "not found". Where a consumer must read its own writes through search, it either waits for the status to clear or looks the document up by identifier.

None of these capabilities is unique in isolation. What matters in a document management programme is having them in one platform and one data model, so that classification, search, permissions, lifecycle and events are properties of the same record rather than four systems to keep in sync. Where they conflict, notably search and client-side encryption, the trade-off must be designed explicitly.

CHAPTER 5

Reference architecture

A document platform is not a repository. It is a pipeline with a registry at its centre and a content store beside it.

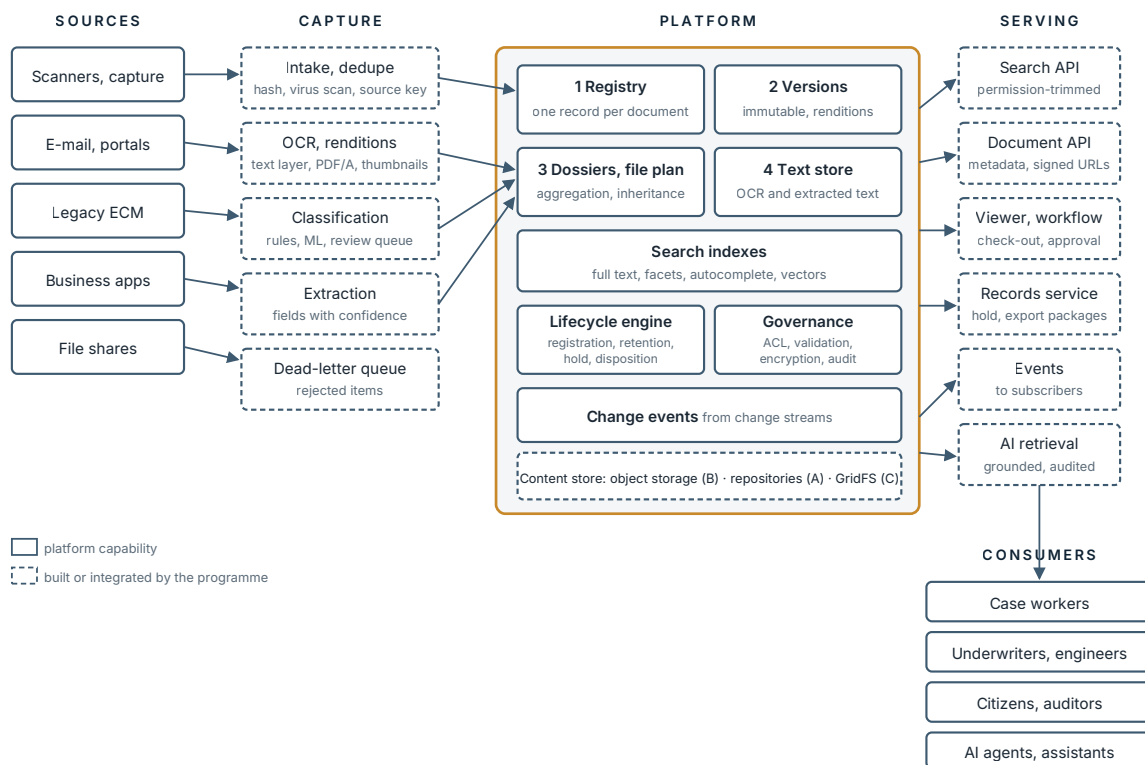


Figure 2. Reference architecture. Solid outlines are platform capabilities; dashed outlines are components the programme must build or integrate (intake, OCR, classification, extraction, APIs, viewer). Rejected items flow to the dead-letter queue.

The flow, step by step

Content enters through capture channels and is written first to the content store, under a key derived from its hash, and then described in the registry (1) with its source key, source position, hash, size and media type. Renditions (2) are produced asynchronously: a text layer from OCR, a PDF/A for preservation, thumbnails for the viewer. Classification assigns the document to a class, by rule where the channel determines it (a scanner line dedicated to invoices), by model where it does not, with a confidence and a review queue below threshold. Extraction populates class-specific fields, each with confidence and provenance. The document is attached to a dossier (3) and inherits its file-plan node and access rules. Extracted text goes to the text store (4) and the search indexes are updated. Where the document is a record, registration assigns an immutable number and timestamp. Every change to the registry record is published as an event.

The ingestion contract

Bursts, retries, migrations and replays are the normal condition of a document platform, not the exception. The contract with each channel and repository must cover six situations explicitly:

- **Content before metadata.** The object is written to the content store first; the registry record is created only after the store acknowledges, with the object key and hash. A record that points to missing content must be impossible; an object without a record is an orphan that a sweeper reclaims after a grace period.
- **Duplicates and shared objects.** The content hash identifies the bytes; the pair of source system and source key identifies the item. The same bytes arriving twice under the same source key are one document; the same bytes under different source keys are two documents sharing one object. An `objects` collection, keyed by hash within the tenant, records size, media type, encryption key reference and a reference count of the versions that point to it. The count is incremented when a version is linked and decremented when a version is disposed; the object is deleted only when the count reaches zero (Chapter 8). Sharing never crosses tenants or residency zones: the key is derived from tenant and hash, not from the hash alone.
- **Ordering.** For repositories that emit versions, the source version or position orders changes. Every write to the registry is conditional: it applies only if the incoming position is newer than the stored one, so that a replayed or late event never overwrites a newer state.
- **Deletes.** A delete in a source repository becomes a lifecycle event in the registry, not a physical removal: the record moves to a "withdrawn at source" state, the hold and retention rules decide what happens to the content.
- **Rejected items.** Items that fail validation, virus scanning or classification go to a dead-letter collection with the reason, are monitored and are reprocessed after correction.
- **Reconciliation.** Periodic comparison of counts, hashes and sizes between each repository or bucket and the registry detects drift in both directions (records without objects, objects without records) and triggers targeted repair.

```

// Idempotent capture of a repository event into the registry, in two steps.
// The object was written to the content store before this call and its
// reference count incremented; evt.object carries key, hash, size, media type.
const objectRef = { store: "s3", bucket: "dms-prod-eu", key: evt.object.key,
                    sha256: evt.object.sha256, size: evt.object.size,
                    mediaType: evt.object.mediaType };

// Step 1: create the record if this item was never seen. A complete skeleton
// is inserted, so that the validator of Chapter 6 is satisfied from the start.
try {
  await db.collection("documents").insertOne({
    _id: new DocumentId(), schemaVersion: 3, recordVersion: 1,
    docClass: evt.docClass || "unclassified",
    source: { system: "ECM-A", key: evt.key, position: evt.position },
    core: { title: evt.title },
    current: { version: evt.version, object: objectRef, capturedAt: new Date() },
    acl: { read: ["g:intake"], write: ["g:intake"], manage: ["g:records"], aclVersion: 0 },
    lifecycle: { state: "captured", holds: [] },
    index: { status: "pending" }
  });
  return "created";
} catch (e) {
  if (e.code !== 11000) throw e;          // 11000: duplicate key on source.system+key
}

// Step 2: the item exists. Apply the event only if it is newer than what is stored.
const r = await db.collection("documents").updateOne(
  { "source.system": "ECM-A", "source.key": evt.key,
    "source.position": { $lt: evt.position } },
  { $set: { "source.position": evt.position,
            "current.version": evt.version, "current.object": objectRef,
            "current.capturedAt": new Date(), "index.status": "pending" },
    $inc: { recordVersion: 1 } });
return r.matchedCount === 1 ? "updated" : "stale, ignored";
// Unique index: { "source.system": 1, "source.key": 1 }. Positions must compare
// correctly: numbers, or zero-padded strings. Replays and out-of-order events
// end in "stale, ignored" and never overwrite a newer state.

```

Listing 1. Positional, idempotent capture into the registry: insert once, then conditional update. The initial access rules grant the intake group only; the dossier's rules replace them when the document is attached.

Serving

Consumers do not query the collections directly and never receive raw storage credentials. They use the search API, which applies the caller's permissions before any result leaves the platform; the document API, which returns metadata and short-lived signed URLs for content; the viewer and workflow services, which implement check-out, annotation and approval; the records service, which handles holds and export packages; and change events, filtered per subscriber. AI retrieval is a consumer like any other: it searches through the same permission-trimmed interface, receives chunks with their document identifier and version, and never sees a document the asking user could not open.

Offloading and replacing legacy repositories

In shape A the first measurable benefit is that search and metadata reads stop hitting the legacy repository. When the registry holds the description and the text, users find documents without the old platform's search, and only the final retrieval of content reaches it. Migration

then proceeds in the opposite order to the usual: description first, content later. The legacy identifier is kept as a source key forever, so that references in other systems keep resolving. Content is copied to object storage class by class, hash-verified, and the record's pointer is switched in a single update once the copy is confirmed. The repository is decommissioned when no record points to it, which the registry can prove with one query.

Freshness and indexing status

State the contract explicitly, for example "a captured document is retrievable by identifier immediately and appears in full-text search within thirty seconds at the 99th percentile under normal operation", monitor it, and expose it. Each record carries an indexing status and the time of the last completed index update, so that a user interface can show "indexing" instead of "no results", and so that a migration dashboard can report how far behind the indexes are.

WHEN THINGS GO WRONG

- **Content store unavailable.** Metadata, search and viewing of cached renditions keep working; retrieval of originals fails explicitly. Captures are queued, never acknowledged without a stored object.
- **OCR or extraction backlog.** Documents are registered and searchable by declared metadata; full-text and extracted fields arrive later, with the record's status showing which steps are pending.
- **Index lag.** Lag beyond the target raises an alert; the search API can add a secondary lookup by identifier for the most recent captures of the asking user.
- **Lost resumability.** A change-stream consumer down longer than the oplog window cannot resume; recovery is a full index rebuild for the affected scope, followed by reconciliation.
- **Replay.** Because registry writes are positional and idempotent and objects are keyed by hash, any interval of events can be replayed safely.
- **Targets.** Define recovery point and recovery time objectives for the registry, the content store and the indexes separately, and rehearse them together.

CHAPTER 6

The data model

Model the record that is read on every search result, every permission check and every retention run. Everything else is a reference.

Design principles

- One registry document per logical document, containing what consumers commonly read together: identity, class, current version summary, declared and extracted metadata, access rules, lifecycle state, dossier membership and the content pointer.
- Versions are immutable documents in their own collection, referenced from the registry; the registry embeds only the current version and a bounded list of recent ones.
- Metadata is split in three parts: a `core` common to all classes; a `byClass` sub-document whose shape is defined and validated per class; and an `extracted` block where every value carries confidence, the extractor that produced it and whether a human confirmed it.
- Properties that arrive from source repositories without a known schema use the attribute pattern: an array of typed key-value elements that one index covers.
- Access rules are denormalized into the record as arrays of principals, so that search and retrieval can filter by permission without walking a folder hierarchy at query time. Inheritance from the dossier is resolved at write time and re-resolved when the dossier changes.
- Content is never stored in the registry record. The record holds a pointer, a hash, a size and a media type; the bytes live in the content store. An `objects` collection, keyed by tenant and hash, holds the reference count of the versions that share each object, so that deduplication never leads to deleting content another document still needs.
- Extracted text is not stored in the registry record either. It lives in a search projection collection (Chapter 7), so that the record stays small enough to read on every operation.
- The document carries a schema version and a record version, to support evolution and optimistic concurrency.

The listings below show the principles in full for the attributes displayed. Some repetitive metadata is omitted for brevity where marked.

A registry record for a contract

```
{
  "_id": "DOC-01J8XK4M2Q7R9TBV3N6P5A8C1E",
  "schemaVersion": 3,
  "recordVersion": 9,
  "docClass": "contract",
  "classVersion": 2,
  "source": { "system": "INTAKE-EMAIL", "key": "msg-7f31a9/att-2", "position": 1,
    "capturedAt": "2026-09-14T08:12:03Z" },
  "core": {
    "title": "Framework agreement for facility services, 2026 renewal",
```

```

    "language": "it",
    "documentDate": "2026-09-10",
    "author": { "type": "organization", "name": "Example Facility S.p.A." },
    "tags": ["procurement", "facility", "renewal"],
    "pages": 42
  },
  "byClass": {
    "parties": [
      { "role": "customer", "name": "Composite Insurance S.p.A.", "partyRef": "P-0004821733" },
      { "role": "supplier", "name": "Example Facility S.p.A.", "partyRef": "P-0009112004" } ],
    "effectiveFrom": "2026-10-01",
    "expiresOn": "2029-09-30",
    "value": { "amount": 1250000, "currency": "EUR" },
    "governingLaw": "IT",
    "renewal": "automatic-unless-notice"
  },
  "extracted": {
    "byClass.expiresOn": { "value": "2029-09-30", "confidence": 0.97,
      "extractor": "contract-fields-v4", "page": 3, "confirmedBy": "u.rossi" },
    "byClass.value.amount": { "value": 1250000, "confidence": 0.91,
      "extractor": "contract-fields-v4", "page": 12, "confirmedBy": null }
  },
  "properties": [ { "k": "legacy.folderPath", "v": "/Contracts/2026/Facility" },
    { "k": "legacy.docType", "v": "AGR" } ],
  "current": {
    "versionId": "VER-01J8XK4M3A1QZ5", "version": 2, "label": "Signed copy",
    "object": { "store": "s3", "bucket": "dms-prod-eu", "key": "sha256/3b/3bd1...e9f2",
      "sha256": "3bd1...e9f2", "size": 2184733, "mediaType": "application/pdf" },
    "renditions": [ { "kind": "pdfa", "key": "ren/VER-01J8XK4M3A1QZ5/pdfa.pdf" },
      { "kind": "text", "status": "done", "chars": 118204 },
      { "kind": "thumbnail", "key": "ren/VER-01J8XK4M3A1QZ5/thumb.png" } ]
  },
  "recentVersions": [
    { "versionId": "VER-01J8XK4M3A1QZ5", "version": 2, "at": "2026-09-14T08:12:03Z" },
    { "versionId": "VER-01J8XJ9C0D2PY7", "version": 1, "at": "2026-08-30T15:02:11Z" } ],
  "dossier": { "id": "DSR-PROC-2026-00412", "filePlan": "03.02.01", "inheritedAcl": true },
  "acl": { "read": ["g:procurement", "g:legal", "u.rossi", "u.bianchi"],
    "write": ["g:procurement"], "manage": ["g:records"],
    "resolvedAt": "2026-09-14T08:12:05Z", "aclVersion": 4 },
  "lifecycle": { "state": "registered",
    "registration": { "registry": "GEN", "number": "2026/0048213",
      "at": "2026-09-14T08:12:06Z" },
    "retention": { "scheduleId": "RS-CONTRACT-10Y", "trigger": "expiresOn",
      "dispositionDue": "2039-09-30",
      "action": "destroy-content-keep-stub" },
    "holds": [] },
  "index": { "status": "indexed", "at": "2026-09-14T08:12:31Z" },
  "sensitivity": "internal",
  "residency": { "zone": "EU-IT" },
  "updatedAt": "2026-09-14T08:12:31Z"
}

```

Listing 2. A registry record for a contract held by a composite insurer. Values are fictitious. The extracted map is keyed by the path of the field it populated; byClass holds the agreed value, whether extracted, declared or corrected. Provenance of core fields is omitted for brevity.

Two choices deserve comment. The content key is derived from the hash, so identical bytes are stored once whatever their source, and integrity can be verified on every retrieval. The access rules are resolved into principal arrays: a search for documents the caller can read is an indexed filter on `acl.read`, not a traversal. The cost is that a change to the dossier's permissions must be propagated to every member document, which the platform does through a

change stream on the dossier collection, in batches, with `aclVersion` showing which documents still carry the old rules.

```
db.documents.createIndex({ docClass: 1, "lifecycle.state": 1, "core.documentDate": -1 })
db.documents.createIndex({ "dossier.id": 1, "current.version": -1 })
db.documents.createIndex({ "acl.read": 1, "core.documentDate": -1 })
db.documents.createIndex({ "properties.k": 1, "properties.v": 1 })
db.documents.createIndex({ "source.system": 1, "source.key": 1 }, { unique: true })
db.documents.createIndex({ "current.object.sha256": 1 })
db.documents.createIndex({ "lifecycle.retention.dispositionDue": 1 },
  { partialFilterExpression: { "lifecycle.state": { $in: ["registered", "closed"] } } })

// Lookup by a legacy property, served by the attribute-pattern index:
db.documents.find({ properties: { $elemMatch: { k: "legacy.docType", v: "AGR" } } })
```

Listing 3. Indexes for the dominant read paths. The partial index on disposition due date keeps the retention run cheap. The attribute pattern uses `$elemMatch` so that key and value are matched on the same array element.

A registered record in a public administration

In the public sector the structure is the same; the content stresses registration, file plans and legal basis. The registration number is assigned once, is never reused, and is the identifier citizens and courts refer to.

```

{
  "_id": "DOC-01J8Y2Q0F7H3KD9MZS1R4T6W8B",
  "schemaVersion": 3,
  "docClass": "registeredCorrespondence",
  "classVersion": 1,
  "source": { "system": "PEC-GATEWAY", "key": "pec-2026-09-15-00128", "position": 1 },
  "core": { "title": "Request for building permit, lot 12, via Esempio",
    "language": "it", "documentDate": "2026-09-15", "pages": 6,
    "author": { "type": "person", "name": "Giovanni Esposito",
      "partyRef": "CIT-IT-58210993" } },
  "byClass": {
    "direction": "inbound",
    "channel": "certifiedEmail",
    "sender": { "name": "Giovanni Esposito", "address": "g.esposito@pec.example" },
    "recipientUnit": "UO-URBANISTICA",
    "subject": "Richiesta permesso di costruire",
    "attachmentsCount": 4
  },
  "dossier": { "id": "DSR-URB-2026-00988", "filePlan": "06.03.02", "inheritedAcl": true },
  "acl": { "read": ["g:urbanistica", "g:protocollo", "g:segreteria"],
    "write": ["g:urbanistica"], "manage": ["g:records"], "aclVersion": 2 },
  "lifecycle": {
    "state": "registered",
    "registration": { "registry": "PROT-GEN", "year": 2026, "seq": 191204, "number": "2026/0191204",
      "entryId": "PROT-GEN/2026/0191204",
      "at": "2026-09-15T09:41:07Z", "by": "svc-pec", "hashChain": "a91c...7d0e" },
    "retention": { "scheduleId": "RS-URB-PERMIT", "trigger": "dossierClosed",
      "dispositionDue": null, "action": "transfer-to-archive" },
    "holds": [],
    "legalBasis": { "type": "publicTask", "reference": "to be specified per processing" }
  },
  "current": { "versionId": "VER-01J8Y2Q1C2", "version": 1,
    "object": { "store": "s3", "bucket": "prot-prod-it", "key": "sha256/9e/9e0a...41c7",
      "sha256": "9e0a...41c7", "size": 418221,
      "mediaType": "application/pdf" } },
  "index": { "status": "indexed", "at": "2026-09-15T09:41:40Z" },
  "sensitivity": "personal",
  "residency": { "zone": "IT-SOVEREIGN" },
  "updatedAt": "2026-09-15T09:41:40Z"
}

```

Listing 4. A registered inbound document in a municipal registry. Values are fictitious. The retention trigger is the closure of the dossier, so the disposition date is computed later; the hash chain links this registration to the previous one in the same registry (Chapter 8).

Versions

A version is written once and never updated except for its lifecycle flags. It records who created it, from which source, the object it points to, the renditions produced from it and the registration that covers it, if any. Keeping versions in their own collection avoids the unbounded array in the registry record and lets a version be registered, held or disposed independently where regulations require it.

```
{
  "_id": "VER-01J8XK4M3A1QZ5",
  "documentId": "DOC-01J8XK4M2Q7R9TBV3N6P5A8C1E",
  "version": 2,
  "label": "Signed copy",
  "createdAt": "2026-09-14T08:12:03Z",
  "createdBy": "u.rossi",
  "reason": "signed version received from supplier",
  "object": { "store": "s3", "bucket": "dms-prod-eu", "key": "sha256/3b/3bd1...e9f2",
    "sha256": "3bd1...e9f2", "size": 2184733, "mediaType": "application/pdf" },
  "renditions": [ { "kind": "pdfa", "key": "ren/VER-01J8XK4M3A1QZ5/pdfa.pdf",
    "sha256": "c01d...88aa" },
    { "kind": "text", "chars": 118204,
      "ocr": { "engine": "ocr-v7", "avgConfidence": 0.96 } } ],
  "supersedes": "VER-01J8XJ9C0D2PY7",
  "registration": { "registry": "GEN", "number": "2026/0048213" },
  "immutable": true
}
// db.versions.createIndex({ documentId: 1, version: -1 }, { unique: true })
```

Listing 5. A version document. The pair of document identifier and version number is unique; a new version is an insert, never an update of the previous one.

Dossiers and the file plan

A dossier (a case file, a project folder, a claim file) aggregates documents about one matter and is the natural unit for permissions, retention triggers and legal holds. It references the file-plan node that defines its retention schedule and carries the access rules that member documents inherit. The file plan itself is a small, versioned hierarchy maintained by the records function; its node codes appear on every record so that reports by series and by retention class are indexed queries.

```
{
  "_id": "DSR-PROC-2026-00412",
  "kind": "procurementFile",
  "title": "Facility services framework agreement 2026",
  "filePlan": { "code": "03.02.01", "version": 7, "series": "Contracts and agreements" },
  "state": "open",
  "openedAt": "2026-08-30", "closedAt": null,
  "owner": "g:procurement",
  "acl": { "read": ["g:procurement", "g:legal"], "write": ["g:procurement"],
    "manage": ["g:records"], "aclVersion": 4 },
  "holds": [],
  "retention": { "scheduleId": "RS-CONTRACT-10Y", "trigger": "expiresOn" },
  "counts": { "documents": 17, "openTasks": 2 },
  "relatedParties": ["P-0004821733", "P-0009112004"]
}
```

Listing 6. A dossier. Member documents are found through the index on dossier.id in the registry, not through an embedded list that would grow without bound.

Validation per class

A JSON Schema validator on the registry collection enforces what every consumer depends on: identifier, class, source, current object, access rules, lifecycle state and schema version. The class-specific part is validated conditionally: the validator lists, for each class, the required fields of `byClass` and their types. Validation can start in warning mode during migra-

tion and move to error mode once data quality is confirmed. New classes are added by extending the validator and incrementing `classVersion`, without touching existing records.

```
db.runCommand({ collMod: "documents", validationLevel: "moderate", validator: { $jsonSchema: {
  bsonType: "object",
  required: ["_id", "schemaVersion", "docClass", "source", "current", "acl", "lifecycle"],
  properties: {
    docClass: { enum: ["contract", "invoice", "registeredCorrespondence", "technicalDrawing",
      "ruling", "medicalReport", "unclassified"] },
    acl: { bsonType: "object", required: ["read", "write", "manage"],
      properties: { read: { bsonType: "array", items: { bsonType: "string" } } } },
    lifecycle: { bsonType: "object", required: ["state"],
      properties: { state: { enum: ["captured", "classified", "registered", "closed",
        "disposing", "disposed", "withdrawnAtSource"] } } }
  },
  oneOf: [
    { required: ["byClass"],
      properties: { docClass: { enum: ["contract"] },
        byClass: { bsonType: "object", required: ["parties", "effectiveFrom"],
          properties: {
            parties: { bsonType: "array", minItems: 1, items: { bsonType: "object",
              required: ["role", "name"] } },
            effectiveFrom: { bsonType: "string",
              pattern: "^[0-9]{4}-[0-9]{2}-[0-9]{2}$" },
            value: { bsonType: "object", required: ["amount", "currency"] } } } } },
    { required: ["byClass"],
      properties: { docClass: { enum: ["invoice"] },
        byClass: { bsonType: "object", required: ["supplier", "amount", "dueDate"],
          properties: {
            amount: { bsonType: ["double", "int", "long", "decimal"] } } } } },
    { required: ["byClass"],
      properties: { docClass: { enum: ["registeredCorrespondence"] },
        byClass: { bsonType: "object",
          required: ["direction", "channel", "recipientUnit"],
          properties: {
            direction: { enum: ["inbound", "outbound", "internal"] } } } } },
    { properties: { docClass: { not: { enum: ["contract", "invoice",
      "registeredCorrespondence"] } } } }
  ]
} } })
```

Listing 7. Conditional validation per class. Each class branch requires `byClass` itself, not only its fields, otherwise a contract without any class-specific block would pass. The last branch lets classes without a declared shape pass, which is the right default during onboarding and the wrong one at steady state: remove it once every class has a schema.

CHAPTER 7

Search

Search is where a document platform is judged by its users, and where a bolted-on engine quietly takes over the architecture.

Users do not ask for a document by identifier. They ask for "the signed framework agreement with the facility supplier, the one that expires in 2029", or "every permit request for via Esem- pio since 2024", or "the ruling where the court said something like this". A search must there- fore combine, in one request, exact filters on metadata, full text on titles and extracted text, facets to narrow the result, autocomplete on titles and reference numbers, and increasingly semantic retrieval on meaning. It must return only documents the caller may open, with a latency that keeps the user in flow, and with a stated contract on how fresh the index is.

The search projection

The registry record is designed to be small and read constantly; extracted text is large and read only by the index. The reference design therefore maintains a search projection collec- tion, one document per registry record, carrying the text and a denormalized copy of every field that search filters or facets on: class, dates, dossier, file-plan code, access principals, life- cycle state and the class-specific fields worth filtering. A change-stream consumer keeps the projection aligned with the registry; the text is added when OCR completes. The registry re- mains the authority: a permission change on a dossier reaches the projection through the same propagation that updates the registry's `acl` arrays, and `aclVersion` lets the platform re- port any lag.

```
{
  "_id": "DOC-01J8XK4M2Q7R9TBV3N6P5A8C1E",
  "versionId": "VER-01J8XK4M3A1QZ5",
  "docClass": "contract",
  "title": "Framework agreement for facility services, 2026 renewal",
  "documentDate": ISODate("2026-09-10T00:00:00Z"),
  "dossierId": "DSR-PROC-2026-00412",
  "filePlan": "03.02.01",
  "state": "registered",
  "aclRead": ["g:procurement", "g:legal", "u.rossi", "u.bianchi"],
  "aclVersion": 4,
  "fields": { "parties": ["Composite Insurance S.p.A.", "Example Facility S.p.A."],
    "expiresOn": ISODate("2029-09-30T00:00:00Z"), "amount": 1250000, "currency": "EUR" },
  "body": "FRAMEWORK AGREEMENT ... (118,204 characters of extracted text)",
  "bodyVersion": "VER-01J8XK4M3A1QZ5",
  "projectedAt": ISODate("2026-09-14T08:12:31Z")
}
```

Listing 8. A search projection document. Dates are stored as BSON dates, which the search index needs for range filters and date facets.

The index

```
// Search index "docs_main" on collection "search_docs"
{ "mappings": { "dynamic": false, "fields": {
  "title": [ { "type": "string", "analyzer": "lucene.standard" },
    { "type": "autocomplete", "tokenization": "edgeGram", "minGrams": 2, "maxGrams": 15,
      "foldDiacritics": true } ],
  "body": { "type": "string", "analyzer": "lucene.italian" },
  "docClass": { "type": "token" },
  "filePlan": { "type": "token" },
  "dossierId": { "type": "token" },
  "state": { "type": "token" },
  "aclRead": { "type": "token" },
  "documentDate": { "type": "date" },
  "fields": { "type": "document", "fields": {
    "parties": [ { "type": "string" }, { "type": "token" } ],
    "expiresOn": { "type": "date" },
    "amount": { "type": "number" } } } }
} } }
```

Listing 9. Index definition. Fields indexed as token and date serve both filters and facets; the older `stringFacet` and `dateFacet` types are outdated and take precedence over the new ones if both are declared, so they are not used. The body uses a language analyzer, which should follow the document's language where the corpus is multilingual.

One query: filters, text, permissions, highlights

```
// Caller principals resolved at authentication time.
const principals = ["u.rossi", "g:procurement", "g:all-staff"];

db.search_docs.aggregate([
  { $search: { index: "docs_main",
    compound: {
      filter: [
        { in: { path: "aclRead", value: principals } },
        { in: { path: "state", value: ["registered", "closed"] } },
        { in: { path: "docClass", value: ["contract"] } },
        { range: { path: "fields.expiresOn", gte: ISODate("2029-01-01T00:00:00Z"),
          lt: ISODate("2030-01-01T00:00:00Z") } } ],
      must: [
        { text: { path: ["title", "body"], query: "framework agreement facility services",
          fuzzy: { maxEdits: 1, prefixLength: 2 } } } ],
      should: [
        { text: { path: "title", query: "framework agreement",
          score: { boost: { value: 3 } } } } ]
    },
    highlight: { path: "body", maxCharsToExamine: 500000, maxNumPassages: 3 },
    sort: { score: { $meta: "searchScore" }, documentDate: -1 }
  } },
  { $limit: 20 },
  { $project: { title: 1, docClass: 1, documentDate: 1, dossierId: 1,
    score: { $meta: "searchScore" }, highlights: { $meta: "searchHighlights" } } }
])
// Expected result with Listing 8 projected: one hit, _id "DOC-01J8XK4M2Q7R9TBV3N6P5A8C1E",
// with up to three highlighted passages from the body.
```

Listing 10. Permission-trimmed search. The permission filter is the first clause and is never optional: the API constructs it from the authenticated caller, never from request parameters.

The `in` filter on principals is the permission model of this architecture. It works because the record carries the resolved principals of every reader. Three practical limits follow. First, doc-

uments readable by thousands of individually named users should use groups; the projection should not carry arrays of tens of thousands of elements. Second, a document visible to everyone carries a sentinel principal such as `g:all-staff` rather than every user. Third, where a deny rule can override a grant, the deny principals are indexed in a separate array and excluded with a `mustNot` clause.

The filter is a performance measure, not the enforcement point. The projection and the index follow the registry with a lag, so for a short time after a revocation a document can still match. The search API therefore verifies every hit of the returned page against the registry's `acl` before any title, highlight or passage leaves the platform: a read of twenty records by identifier, with majority read concern, which drops the ones the caller may no longer see. The same check runs on the passages returned to an assistant. Facet counts cannot be re-checked per document; during propagation lag they may include revoked documents, so the API marks them approximate while `aclVersion` on any affected dossier is behind, and records above the internal sensitivity level are excluded from facet computation altogether.

Facets

```
db.search_docs.aggregate([
  { $searchMeta: { index: "docs_main",
    facet: {
      operator: { compound: {
        filter: [ { in: { path: "aclRead", value: principals } } ],
        must: [ { text: { path: ["title", "body"], query: "facility services" } } ] } },
      facets: {
        byClass: { type: "string", path: "docClass", numBuckets: 10 },
        byFilePlan: { type: "string", path: "filePlan", numBuckets: 20 },
        byYear: { type: "date", path: "documentDate",
          boundaries: [ ISODate("2023-01-01"), ISODate("2024-01-01"),
            ISODate("2025-01-01"), ISODate("2026-01-01"),
            ISODate("2027-01-01") ],
          default: "other" } } } } }
])
// Returns count and bucket counts, computed on the same permission-trimmed set as the results.
```

Listing 11. Facet counts on the permission-trimmed result set. Counts that include documents the caller cannot see are an information leak as well as a usability bug.

Autocomplete

```
db.search_docs.aggregate([
  { $search: { index: "docs_main",
    compound: { filter: [ { in: { path: "aclRead", value: principals } } ],
      must: [ { autocomplete: { path: "title", query: "frame agr" } } ] } },
  { $limit: 8 },
  { $project: { title: 1, docClass: 1 } }
])
```

Listing 12. Autocomplete on titles, trimmed by permission like every other query.

Semantic retrieval and hybrid search

Semantic retrieval answers questions that keywords cannot: "clauses about early termination for convenience", "reports describing a similar failure mode". It operates on chunks, not on

whole documents: the text of each version is split into passages, each passage is embedded by a model outside the database, and the vectors are stored in a chunk collection with the document identifier, the version, the page range and a denormalized copy of the permission principals and the fields used to pre-filter.

```
// Vector index "chunks_vec" on collection "chunks"
{ "fields": [
  { "type": "vector", "path": "embedding", "numDimensions": 1024, "similarity": "cosine" },
  { "type": "filter", "path": "aclRead" },
  { "type": "filter", "path": "docClass" },
  { "type": "filter", "path": "state" } ] }

db.chunks.aggregate([
  { $vectorSearch: { index: "chunks_vec", path: "embedding",
    queryVector: qv, // embedding of the question, same model
    numCandidates: 300, limit: 30,
    filter: { $and: [ { aclRead: { $in: principals } },
      { docClass: { $in: ["contract"] } },
      { state: { $in: ["registered", "closed"] } } ] } } },
  { $project: { documentId: 1, versionId: 1, page: 1, text: 1,
    score: { $meta: "vectorSearchScore" } } },
  { $group: { _id: "$documentId", best: { $max: "$score" },
    passages: { $push: { page: "$page", text: "$text", score: "$score" } } } },
  { $sort: { best: -1 } }, { $limit: 10 }
])
```

Listing 13. Permission-filtered semantic retrieval over chunks, grouped by document. Filters apply only to fields declared in the vector index.

Hybrid search combines a keyword ranking with a semantic ranking, typically with reciprocal rank fusion. MongoDB provides `$rankFusion` from version 8.0 and `$scoreFusion` from 8.3, but every sub-pipeline of either stage must run against the same collection, and Listings 10 and 13 run against two: the search projection and the chunks. The reference design resolves this in one of two ways. The native way indexes the chunk text for full-text search as well, so that both legs run on `chunks` and the fused result is grouped by document identifier afterwards; the chunk collection then carries a second search index on its text and the same denormalized filter fields. The application way runs Listing 10 and Listing 13 separately and fuses the two ranked lists by document identifier with reciprocal rank fusion in the API. Either way the permission filter is identical in both legs, and the re-check against the registry runs on the fused page. For AI assistants, the retrieval result is a set of passages with document identifier, version and page: the assistant cites them, the user interface links to the viewer, and the audit log records which passages were shown to whom.

```
// Native fusion, both legs on "chunks" (text index "chunks_text" on the chunk text).
db.chunks.aggregate([
  { $rankFusion: { input: { pipelines: {
    semantic: [ { $vectorSearch: { index: "chunks_vec", path: "embedding", queryVector: qv,
      numCandidates: 300, limit: 30,
      filter: { $and: [ { aclRead: { $in: principals } },
        { state: { $in: ["registered", "closed"] } } ] } } },
    keyword: [ { $search: { index: "chunks_text", compound: {
      filter: [ { in: { path: "aclRead", value: principals } },
        { in: { path: "state", value: ["registered", "closed"] } } ],
      must: [ { text: { path: "text",
        query: "early termination for convenience" } } ] } } },
      { $limit: 30 } ] } },
    combination: { weights: { semantic: 1, keyword: 1 } } } },
  { $group: { _id: "$documentId", passages: { $push: { page: "$page", text: "$text" } } } },
  { $limit: 10 }
])
```

Listing 14. Native hybrid search on one collection. Sub-pipelines run serially and each should be limited; the fused ranking is then grouped by document.

Consistency, relevance and measurement

Search indexes are eventually consistent. The platform exposes this honestly: a just-registered document carries `index.status: "pending"` and the interface says so. Relevance is tuned with boosts on title and reference numbers, synonyms for domain vocabulary, and language-specific analyzers; it is measured on a labelled set of real queries with expected results, refreshed quarterly. Operational indicators: p95 latency by query type, zero-result rate, index lag, projection lag. Quality indicators: precision at ten on the labelled set, and the share of searches followed by a document open within the first results.

THE BOLTED-ON ENGINE

Many organizations added a separate search engine because the repository's own search could not keep up. The symptom is familiar: a second schema, a second permission model, a synchronization job that silently fails on Friday, and results that show documents deleted weeks ago. Absorbing search into the metadata layer removes the second system: the index is maintained from the same records, trimmed by the same principals, and rebuilt from the same source of truth when needed.

CHAPTER 8

Lifecycle, retention and records

A document system is remembered not for how it stores files but for whether it can prove, keep and delete on command.

A document becomes a record when the organization decides that it must be kept as evidence of an activity: a contract signed, a permit requested, a ruling issued, a claim filed. From that moment it has a lifecycle that regulation, not convenience, defines. The metadata layer is where that lifecycle is executed; the data model makes it enforceable, and the audit trail makes it provable.

Registration

Registration (protocol, in many European administrations) assigns a document a unique, sequential, never-reused number within a registry and a year, a timestamp, and a fingerprint of the content, and binds them together immutably. The reference design implements it with a counter document per registry and year, a registration entry that chains the hash of the previous entry, and a transaction that increments the counter, inserts the entry and transitions the document, or does none of these.

```

const session = client.startSession();
try {
  await session.withTransaction(async () => {
    const year = new Date().getUTCFullYear();
    const counter = await db.collection("counters").findOneAndUpdate(
      { _id: { registry: "PROT-GEN", year } }, { $inc: { seq: 1 } },
      { upsert: true, returnDocument: "after", session });
    const seq = counter.seq, at = new Date();
    const number = `${year}/${String(seq).padStart(7, "0")}`;
    const entryId = `PROT-GEN/${year}/${String(seq).padStart(7, "0")}`; // unique across registries
    const prev = await db.collection("registrations").findOne(
      { registry: "PROT-GEN", year }, { sort: { seq: -1 }, projection: { hashChain: 1 }, session });
    // The chain covers the whole entry, serialized canonically (sorted keys, fixed encoding).
    const entry = { registry: "PROT-GEN", year, seq, number, documentId: doc._id,
      versionId: version._id, sha256: version.object.sha256,
      at: at.toISOString(), by: actor, prevHash: prev ? prev.hashChain : null };
    const hashChain = sha256Hex(canonicalJson(entry));
    await db.collection("registrations").insertOne({ _id: entryId, ...entry, hashChain },
      { session });

    const r = await db.collection("documents").updateOne(
      { _id: doc._id, "lifecycle.state": "classified" },
      { $set: { "lifecycle.state": "registered",
        "lifecycle.registration": { registry: "PROT-GEN", year, seq, number, entryId,
          at, by: actor, hashChain } },
        $inc: { recordVersion: 1 } }, { session });
    if (r.matchedCount !== 1) throw new Error("document not in a registrable state");
  });
} finally { await session.endSession(); }
// Indexes: registrations { registry: 1, year: 1, seq: 1 } unique; counters by _id.
// The counter serializes registrations within one registry, which is the required behaviour:
// sequence numbers must have no gaps and no duplicates. Throughput is per registry.

```

Listing 15. Registration as one transaction. The entry identifier includes the registry, so two registries can both issue 2026/0000001 without colliding. The hash chain covers every field of the entry, including actor and timestamp, and lets an auditor verify that no entry was altered or removed after the fact.

A chain stored only in the database proves internal consistency, not that the database itself was not rewritten. The reference design anchors it externally: at a fixed cadence, daily or hourly according to the registry's legal weight, the head hash of each registry's chain is written to an append-only location outside the platform's administrative control, a locked object in a separate storage account, a qualified timestamp from a trust service provider where the jurisdiction recognizes one, or a signed entry in the organization's audit system. A periodic job re-walks the chain, compares its head with the anchors and stores the verification result.

Immutability of content is enforced where the bytes live: in shape B, versions are written to object storage with object lock in compliance mode for the retention period, so that not even an administrator can delete them early. The registry records the lock expiry alongside the retention schedule.

Retention schedules

A retention schedule, defined by the records function per file-plan series or document class, states a trigger event (document date, contract expiry, dossier closure, end of employment), a period, and an action (destroy content and keep a metadata stub, destroy everything, transfer to an archive, review). The platform resolves the trigger for each record as soon as it is known, stores the computed `dispositionDue`, and recomputes it when the trigger changes, for

example when a dossier closes. The partial index of Listing 3 makes the daily disposition run an indexed scan of what is due.

WHY TTL INDEXES ARE NOT RECORDS MANAGEMENT

A TTL index deletes documents silently when a date passes. It does not check for legal holds, does not record what was destroyed or why, does not delete the content object, and cannot be audited. It is a fine mechanism for expiring cached renditions or upload sessions. Disposition of records needs a job that selects what is due, verifies holds at document and dossier level, writes a disposition record, deletes the content with proof, and leaves a stub that explains what happened.

Legal hold

A hold suspends disposition for a set of documents, usually defined at dossier level, sometimes by query (all documents of a supplier, all correspondence of a unit in a period). Holds are first-class documents with an owner, a reason, a scope and a release date; applying one stamps the hold identifier on every affected registry record's `lifecycle.holds` array through the same propagation mechanism as access rules, so that the disposition run needs no join and the records service can report, for any document, why it cannot be destroyed. Releasing a hold removes the stamp; the disposition run picks the record up on its next pass if it is still due.

Disposition

```
// Daily run, in two phases. Phase 1 (transaction): re-check holds at document and dossier
// level inside the transaction, move the record to "disposing", inventory every object of
// every version, decrement reference counts, and write the disposition record.
const due = db.documents.find({
  "lifecycle.retention.dispositionDue": { $lte: new Date() },
  "lifecycle.state": { $in: ["registered", "closed"] },
  "lifecycle.holds": { $size: 0 }
}, { projection: { _id: 1, dossier: 1, "lifecycle.retention": 1 } }).limit(1000);

for await (const d of due) {
  await session.withTransaction(async () => {
    const dossier = await db.collection("dossiers").findOne(
      { _id: d.dossier.id }, { projection: { holds: 1 }, session });
    if (dossier && dossier.holds.length > 0) throw new SkipError("dossier under hold");
    const r = await db.collection("documents").updateOne(
      { _id: d._id, "lifecycle.state": { $in: ["registered", "closed"] },
        "lifecycle.holds": { $size: 0 } },
      { $set: { "lifecycle.state": "disposing", "current.object": null,
        "current.renditions": [] },
        $inc: { recordVersion: 1 } }, { session });
    if (r.matchedCount !== 1) throw new SkipError("state changed concurrently");
    const versions = await db.collection("versions")
      .find({ documentId: d._id }, { projection: { object: 1, renditions: 1 }, session }).toArray();
    const keys = versions.flatMap(v => [v.object.key,
      ...v.renditions.map(x => x.key).filter(Boolean)]);

    for (const v of versions) {
      await db.collection("objects").updateOne(
        { _id: { tenant: tenantId, sha256: v.object.sha256 } },
        { $inc: { refs: -1 } }, { session });
    }
  })
}
```

```

    await db.collection("dispositions").insertOne({ _id: d._id, at: new Date(), runId,
      scheduleId: d.lifecycle.retention.scheduleId, action: d.lifecycle.retention.action,
      objects: keys.map(k => ({ key: k, deletedAt: null })) }, { session });
  }).catch(e => { if (!(e instanceof SkipError)) throw e; });
}

// Phase 2 (outside any transaction, retryable, run once the longest signed-URL lifetime
// has elapsed): delete objects whose reference count is zero, then finalize the record.
const pending = db.collection("dispositions").find({ "objects.deletedAt": null,
  at: { $lte: urlCutoff } });
for await (const disp of pending) {
  for (const o of disp.objects.filter(o => !o.deletedAt)) {
    const obj = await db.collection("objects").findOne({ key: o.key }, { projection: { refs: 1 } });
    const shared = !(obj && obj.refs > 0);
    if (!shared) await contentStore.deleteObject(o.key); // idempotent, logged
    await db.collection("dispositions").updateOne({ _id: disp._id, "objects.key": o.key },
      { $set: { "objects.$.deletedAt": new Date(), "objects.$.shared": shared } });
  }
  await db.collection("documents").updateOne({ _id: disp._id, "lifecycle.state": "disposing" },
    { $set: { "lifecycle.state": "disposed", "lifecycle.disposedAt": new Date() } });
}

```

Listing 16. A disposition run as a state machine. Holds are re-checked inside the transaction, every version's objects are inventoried, shared objects survive while another document references them, and the record reaches disposed only when every deletion is confirmed. The metadata stub survives with the class, the registration number and the disposition record, so that a later request can be answered with "destroyed on this date under this schedule" rather than with silence.

Content deletion happens after the transaction commits, because an object store is not part of the database transaction. From the moment the record enters `disposing` the document API refuses to issue new signed URLs, and the second phase waits for the longest URL lifetime the platform issues (minutes) before deleting, so that no URL handed out just before the run can still fetch destroyed content. If a deletion fails, the disposition record shows the object as not yet deleted and the next run retries it; the record stays in `disposing` until every object is confirmed. Objects shared with other documents are not deleted and are recorded as shared, with their reference count showing why. Object lock expiry must precede the disposition date, which the platform guarantees by setting the lock to the retention period at write time and extending it when a hold is placed.

Audit

Every lifecycle transition, every permission change and every registration is attributable: the actor, the time, the reason and the record version before and after. Reads are audited at the API, where the operator, the purpose and the fields or passages returned are known; database-level auditing of reads is possible but shows only that an application identity read a document. The two logs share a correlation identifier.

Preservation and export

The metadata layer is an operational records system, not a long-term digital preservation system. Where law or policy requires preservation with specific guarantees (format migration, qualified preservation services, archival description), the platform produces export packages: the content in a preservation format such as PDF/A, the metadata in a documented schema, a manifest with hashes, and the registration and audit entries, following the information pack-

age concepts of the OAIS reference model. The same export serves transfers to a state archive, responses to courts and the organization's exit strategy from the platform itself.

A PRACTICAL DRILL

Before going live, run four exercises end to end: register a document and verify the hash chain; place and release a legal hold on a dossier and show that the disposition run skips and then disposes; dispose a batch and verify that content is gone from the store and the stub answers the question; export a dossier as a package and re-import it into an empty environment. As an internal exercise target, the author suggests completing all four within one working day. It is not a regulatory deadline, and meeting it is a sign of maturity, not proof that governance is complete.

CHAPTER 9

Governance, security and sovereignty

A document platform concentrates the organization's evidence by design. It must concentrate protection with the same determination.

Bringing every document under one registry creates value and an attractive target. It also creates a single place where access rules, encryption, residency and audit can be applied consistently, instead of unevenly across repositories that each had their own model. Treated properly, the metadata layer is a security asset.

Access control: resolved at write time, enforced at the edge

The access model has three levels. The dossier carries the rules for the matter; the document inherits them unless an explicit exception is recorded; the version inherits from the document. Rules are expressed as principals (users, groups, roles, organizational units) with read, write and manage rights. The platform resolves inheritance when a record is written or when a dossier's rules change, and stores the result in the record, which is what makes permission-trimmed search an indexed filter (Chapter 7). Propagation of a change is asynchronous, in batches, and observable: `aclVersion` on each record and a lag metric per dossier tell the security team when every member reflects the new rules.

Enforcement happens at defined control points, each of which evaluates the stored rules:

- **Search API:** trims results by the caller's principals, then re-verifies every hit of the returned page against the registry before returning titles, highlights or passages, so that a revocation not yet propagated to the index is still enforced; facet counts are computed on the trimmed set and marked approximate while propagation is behind.
- **Document API:** re-evaluates the full rule, including deny entries and sensitivity, before issuing a short-lived signed URL for content; the URL is bound to the version and expires in minutes.
- **Events:** change events are filtered or projected per subscriber, so that a downstream system does not learn about documents it may not see.
- **AI retrieval and agent memory:** retrieval is a consumer with purposes, projections and audit; passages retrieved for one user and purpose are not retained for another.
- **Exports and packages:** produced by the records service under its own authorization, recorded with purpose and recipient.

Revocations must propagate. When a person leaves a unit, the group change reaches every record through the same mechanism, and the lag metric says when it is complete. Until then two backstops apply: the search API's re-verification of each returned hit against the registry, and the document API, which evaluates the full rule at request time before issuing any content URL. Neither depends on the index being current, so a revoked reader sees neither

metadata nor content during the lag; what they can still influence is a facet count, which is why counts are labelled approximate until propagation completes.

Sensitivity and field-level protection

Not all metadata is equal. A ruling's title may name a minor; a medical report's extracted fields describe a diagnosis; a personnel file's properties include a national identifier. The record carries a sensitivity level, and the platform applies three measures in combination. Projections per consumer purpose remove fields the purpose does not need. Client-side field-level encryption protects the most sensitive fields so that they are unreadable to database administrators, backups and anyone without the keys. Where an encrypted field must still be looked up by exact value, Queryable Encryption provides equality and range queries on it.

These protections cannot be combined freely with search. Fields encrypted client-side cannot be indexed by MongoDB Search, and a collection using Queryable Encryption cannot be indexed for Search at all. The design therefore decides, field by field, which metadata participates in search and how it is protected: searchable fields are protected by encryption at rest, network isolation, roles, projections and audit; identifiers that need only equality lookup live in a separate vault collection under Queryable Encryption, keyed by document; special-category values that never need searching are encrypted client-side and excluded from the search projection. The extracted text of sensitive classes is indexed only where the organization has decided that the benefit justifies the exposure, with the permission filter as the control.

Content protection

Content is encrypted at rest in the object store with keys managed by the organization, and in shape B each version is an immutable object. Two further measures matter. Per-document or per-dossier content keys enable crypto-shredding: destroying the key renders every copy of the content unreadable, including copies in backups that cannot be selectively deleted. Object lock in compliance mode enforces the retention period at the storage level. The two combine: a document under retention cannot be deleted early, and a document whose retention has expired can be made unreadable even where bytes linger.

The right to erasure and the duty to keep

Erasure requests collide with retention obligations more often in a document platform than anywhere else, because a single file may contain personal data of several people and be a record the organization is required to keep. The platform does not resolve the legal question; it makes the operational answer possible. The registry identifies every document, version, rendition, chunk and export that references a person, through party references on records and dossiers. The records service can then apply the decision the legal function takes: destroy, redact into a new version while retaining the original under lock, or refuse with a documented basis. Search projections, vector chunks, caches and downstream copies are part of the inventory that an erasure must cover; the subscriber registry of Chapter 5 is how the platform knows who received what.

Residency and sovereignty

Residency applies to four things that must be located consistently: the registry and its replicas, the search and vector indexes, the content store with its replicas and versions, and backups of all three. Zone sharding can pin registry records to regions through a field such as `residency.zone` included in the shard key; the content store uses regional buckets with the same zone discipline; search nodes and backup locations follow. Residency of the registry alone is not residency of the documents.

Sovereignty goes further. It concerns who operates the platform, who holds the keys, which jurisdiction's law applies to the operator, and whether the system can run disconnected. B2B archiving services that hold documents on behalf of banks, public registries that hold citizens' correspondence, and defence or justice institutions frequently require on-premises or sovereign-cloud deployment, customer-held keys and the ability to export everything in documented formats. In isolated or air-gapped environments the platform runs with no dependency on external services, which also decides between shape B with a local object store and shape C with content inside the database. These choices should be made explicitly and documented, not inherited from defaults.

Multi-tenancy

Platforms sold as a service to many organizations, a SaaS registry for municipalities or a document management offer for small businesses, add tenancy to every rule above. The reference design keeps one registry schema and isolates tenants by a tenant identifier that is part of every index, every search filter, every ACL principal namespace and every bucket prefix; tenants with stronger requirements receive their own database or cluster and their own keys. Metadata rules per tenant (which classes exist, which fields are required, which retention schedules apply) are data, versioned per tenant, not code.

A PRACTICAL DRILL

Before going live, take a restricted user, run the ten most common searches, and verify that no title, facet count, highlight, autocomplete suggestion or AI passage reveals a document that user cannot open. Then change a dossier's permissions, measure how long the search results take to reflect it, and confirm that the document API refused content during the lag. Finally, produce the inventory of every copy of one person's data, including chunks and exports, within one working day.

CHAPTER 10

Industry patterns

The pattern is the same everywhere. What changes is who keeps the bytes, what counts as a record, and which failure costs the most.

The scenarios below are composites. They combine architectural situations that recur across organizations in each sector and do not describe any specific institution. Volumes are indicative orders of magnitude observed in the field; benefits must be measured on the workloads actually selected.

Public administration: registration and the shared archive

Administrations receive and produce correspondence that must be registered, classified under a file plan, attached to a case file and kept for periods set by law. Volumes range from a few hundred thousand documents a year for a city to tens of millions in a regional or national registry that consolidates many bodies. Three situations recur. A registry offered as a service to many administrations, where the metadata rules differ per tenant and the file plan is data. A regional archive that digitized decades of paper, holds tens of millions of records, and bolted on a separate search engine because metadata queries on the extracted text were too slow; the modernization consists of absorbing that engine into the metadata layer so that one system holds metadata, text and permissions. And a central finance or accounting administration that keeps content in a legacy repository under a long-standing contract and moves only the description, the dossiers and the search into the new layer, shape A, with content migration as a later, separate decision. In all three, the hardest problems are the immutability of registration, the once-only principle (a citizen's document should not be requested twice), and the exit strategy from the platform.

Insurance: the claim file and the underwriting dossier

An insurer ingests documents at a rate that a single case worker never sees: policy documents, claims correspondence, medical reports, surveys, photographs, invoices, hundreds of thousands a day at group scale. Two programmes recur. A document management modernization in which the existing content platform keeps the bytes and the new metadata layer owns classification, extracted fields, full-text and semantic search, and the integration with claims and policy systems, shape A at first and shape B for new content. And an underwriting dossier where the question is retrieval: an underwriter asks for every document about a risk, a broker or a clause, across years and formats, with facets on class and date and with semantic retrieval for "similar cases". Extraction with confidence and human confirmation is the difference between a search that is trusted and one that is bypassed. Health data in medical reports is special-category data, handled as described in Chapter 9.

Banking: the document index renovation

A bank's documents are spread across a core archive, product platforms and branch scanning. The recurring programme is the renovation of the document index: a bounded context dedicated to indexes and metadata, with filters to find and retrieve documents whatever repository holds them. It is shape A by definition, with a long tail of repositories, and its first measurable benefit is that branch and contact-centre staff search once. Digital libraries of regulatory and product documentation, and metadata search over them, are a second, smaller pattern; the lesson from programmes that did not go ahead is that the business case must be built on a measurable operational problem, not on the elegance of the index.

Energy, engineering and construction

Technical documents are heterogeneous: geoscience reports, drawings, specifications, test results, in formats that range from scanned paper to large binary files. An energy company builds a "document lake" to make internal reports searchable and to extract information from them with AI; the metadata layer is the operational registry, the extraction pipeline is the producer, and the lake receives text for analytics. A construction group replaces a groupware-era document and approval application used by thousands of people, separating metadata, search and workflow state in the registry from files in object storage, shape B, with approval workflows as a consumer. The hardest problems are file sizes, revisions and disciplines in drawing management, and the long life of records that outlast the systems that produced them.

Telecom and B2B archiving services

Operators and service providers offer archiving to business customers, starting with banking documents such as contracts, statements and invoices, where the customer's regulator sets the retention and integrity rules. Metadata lives in the registry, content in a store the customer may require to be on premises, and tenancy, keys and residency are contractual terms. The hardest problem is being able to prove, years later, exactly what was stored and when.

Media and publishing

Editorial archives need full-text search across decades of articles, specialist titles and metadata (authors, sections, dates, rights). The metadata layer absorbs the search function across titles while the editorial content system remains the producer. The hardest problems are rights metadata and relevance tuning for professional readers.

Justice

Courts produce rulings that must be shared across jurisdictions, searched by legal professionals and published with personal data redacted. The record carries the ruling's metadata, a published rendition distinct from the internal one, and permissions that differ between them. The hardest problems are anonymization as a controlled version rather than an overwrite, and the authenticity of what is published.

Software vendors

Vendors offering a document management service to small businesses and professionals run the same architecture as a multi-tenant product: metadata rules per customer are configuration, the registry is shared with tenant isolation in every index and filter, and generative features (summaries, answers over a customer's documents) are consumers of permission-trimmed retrieval. The hardest problem is keeping per-tenant flexibility without losing validation.

SECTOR	TYPICAL CONTENT	MAIN DRIVER	HARDEST PROBLEM
Public administration	Registered correspondence, case files, permits	Registration, once-only, shared archives	Immutable registration and exit strategy
Insurance	Claims, medical reports, policies, surveys	Throughput and underwriting retrieval	Trusted extraction; special-category data
Banking	Contracts, statements, branch scans	One index over many repositories	Long tail of sources; measurable business case
Energy and engineering	Reports, drawings, specifications	Searchable technical knowledge, AI extraction	Large files, revisions, very long retention
Telecom, B2B archiving	Customers' regulated documents	Archiving as a service	Proof of integrity; residency and keys
Media and publishing	Articles, editions, rights	Specialist full-text search	Rights metadata; relevance for experts
Justice	Rulings, case documents	Sharing and publication	Anonymization as versioning; authenticity

CHAPTER 11

A worked example

One document, one claim file, two versions, a hold and a disposition, end to end.

A composite insurer runs the platform in shape B. The example follows a medical report in a motor claim file, from an e-mail attachment to its disposition eleven years later.

1. Capture. A report arrives as a PDF attachment to a message in the claims mailbox. The intake service computes its hash, finds no object with that hash in the tenant, writes the object to the content store under the hash key, creates its entry in the `objects` collection with a reference count of one, and creates the registry record with source `INTAKE-EMAIL`, the message and attachment identifiers as source key, class `unclassified` and state `captured`. The same message is delivered twice by the mail gateway; the second intake fails on the unique source key and stops. The same report had also been uploaded by the broker through the portal under a different source key: that is a second document pointing to the same object, whose reference count becomes two, and the platform records the duplicate relation between the two.

2. Renditions and classification. OCR produces a text layer with an average confidence of 0.94. The classifier assigns `medicalReport` with confidence 0.98, above the automatic threshold. Extraction finds a claim number, a patient name, a date of examination and a diagnosis; the claim number matches an open dossier, `DSR-CLM-2026-18422`, so the document is attached to it and inherits its file-plan node and its access rules: the claims team, the medical adviser group and the dossier's handler. The sensitivity is set to `special-category`; the diagnosis field is encrypted client-side and excluded from the search projection; the text is projected for search because the organization decided that claims handlers need to search reports, with permission trimming as the control.

3. Registration and indexing. The document is registered in the claims registry as `2026/0310877`, with the hash chain extended. The search projection is written; twenty seconds later `index.status` becomes `indexed`. A claims handler searching for "knee ligament report" with a facet on `medicalReport` and the dossier filter finds it as the first result, with a highlighted passage.

4. A second version. The clinic sends a corrected report: same claim, a corrected examination date. Intake computes a different hash, finds a document with the same claim number and a title similarity above threshold in the same dossier, and proposes a new version rather than a new document; the handler confirms. Version 2 is inserted, the registry's `current` pointer switches in the same transaction that writes the version, and `recentVersions` shows both. The projection is rebuilt from version 2; version 1 stays retrievable by identifier for anyone with manage rights, and the registration entry for version 1 is not altered.

5. Legal hold. The claim goes to litigation. Legal places a hold on the dossier with a reason and no release date; the propagation stamps the hold on the dossier's forty-one documents within seconds, and the object locks of their versions are extended. The retention trigger of the report's schedule is "dossier closed plus ten years"; when the dossier closes the following year,

`dispositionDue` is computed, but the daily disposition run skips the record because `lifecycle.holds` is not empty. Four years later the litigation ends and the hold is released; the run picks the record up the next morning, finds it not yet due, and leaves it.

6. Disposition. Ten years after closure the record is due. Inside the transaction the run re-reads the dossier and finds no hold, moves the document to `disposing`, removes the content pointer and the renditions, inventories the objects of both versions and their renditions, decrements their reference counts and writes the disposition record. From that moment the document API issues no URL for it. The second phase runs after the longest URL lifetime has elapsed: the object of version 1 reaches zero references and is deleted; the object of version 2 is still referenced by the broker's upload, whose own retention runs on a different schedule, so it is recorded as shared and kept; the renditions are deleted; the record reaches `disposed`. The object locks had expired three days earlier, because they were set to the retention period at write time. The chunk collection and the search projection receive the state change through the change stream and drop their entries. A year later a court asks for the report. The answer is produced in minutes from the stub: registered on this date under this number, two versions, disposed on this date under this schedule, content of this document destroyed, one object retained under another document's retention, disposition record attached.

WHAT MADE THE WHOLE SEQUENCE CHEAP

Content was keyed by hash and written before the record, with a reference count; source keys were unique; versions were immutable inserts; permissions and holds were resolved into the record and re-checked inside the disposition transaction; the retention trigger was computed, not inferred at run time; disposition was a state machine that left a stub. Without any one of these, the same sequence would have required manual repair across the content store, the index and the audit log, or would have destroyed content another document still needed.

CHAPTER 12

Anti-patterns

Most document platform failures are not technical. They are decisions that looked reasonable at the start.

The database as file server

Storing every PDF and scan inside database records because it is convenient at the start makes backups, replication and working sets grow with content rather than with metadata. Content belongs in a content store; shape C is a deliberate choice for specific environments, not a default.

The flat metadata bag

One collection of documents with a free-form `properties` map and no classes means nothing can be validated, faceted or retained by rule. Classes with validated shapes are the backbone; the attribute pattern is for the long tail of legacy properties, not for the core.

The bolted-on search engine

A separate engine introduced to fix a latency problem becomes a second system of record for what is findable, with its own permissions and its own drift. Keep index, metadata and permissions in one platform; rebuild the index from the registry when needed.

The folder as the only structure

Migrating a folder hierarchy as the organizing principle reproduces the old repository's limits: one place per document, permissions by location, no classification. Folders become a legacy property; dossiers and classes carry the meaning.

Permissions resolved at read time

Walking dossier, folder and group hierarchies on every search makes permission-trimmed search slow and facet counts wrong. Resolve inheritance at write time into principal arrays, propagate changes, and keep the API as the enforcement point.

The unbounded version array

Embedding every version, rendition and event in the registry record leads to documents that grow without limit and slow every read. Embed the current version and a bounded recent list; keep versions in their own collection.

TTL as records management

Letting a time-to-live index delete records is deletion without holds, without proof and without the content. Disposition is a job with a transaction, a record and a stub.

Content before the index

Migrating terabytes of content before the registry describes them produces a new repository nobody can search. Describe first, migrate content later, switch pointers per class when copies are verified.

Extraction without confidence

Writing extracted fields as if they were declared facts makes search return wrong documents with full confidence. Every extracted value carries confidence, extractor and confirmation status; thresholds route the rest to review.

The ownerless file plan

A file plan and retention schedules defined once by a project and never maintained drift from the organization and from the law. The records function owns them as versioned data, with a change process.

Governance as a later phase

Adding registration, holds, sensitivity handling and audit after go-live means retrofitting them into every consumer and every migrated record. They belong in the first data model and the first API contract.

CHAPTER 13

Adoption roadmap and checklist

Describe first, search early, migrate content when the index proves it is worth it.

Durations are indicative and depend on the number and quality of repositories, the regulatory context and the team.

PHASE 0 · DISCOVERY (INDICATIVELY 4 TO 6 WEEKS)

Inventory the repositories, their volumes, metadata schemas, access models and change mechanisms. Identify the first consumers and their most frequent searches. Agree the first document classes and their required fields with the business, and the file plan and retention schedules with the records function. Define the freshness contract, the recovery objectives and the success metrics.

PHASE 1 · REGISTRY AND SEARCH FOR ONE DOMAIN, WITH THE RECORDS CONTROLS NEEDED TO GO LIVE (INDICATIVELY 3 TO 4 MONTHS)

Shape A over one or two repositories, or shape B for one capture channel: intake with hash keys, unique source keys and object reference counts; positional, idempotent registry writes; class validation; dossiers with inherited permissions resolved at write time; the search projection and the first index with filters, full text and facets; permission-trimmed search with re-verification, and document APIs; audit, roles and field encryption from the start. The records controls that Chapter 1 calls non-negotiable are in this phase too, in their minimal form: registration with hash chain where the first classes are records, the hold flag with propagation, deletion disabled by default (nothing is destroyed until Phase 3 exists), and audit of every lifecycle transition. A platform goes live with these or it does not go live.

PHASE 2 · CAPTURE PIPELINE AND CONTENT OWNERSHIP

OCR, renditions, classification with review queue, extraction with confidence; object storage with immutable versions and object lock; version collection; migration of content for the first classes with pointer switching; reconciliation jobs in both directions.

PHASE 3 · RECORDS AUTOMATION

Retention schedules as data and computed disposition dates, the disposition run as a state machine with stubs, external anchoring of the hash chains, export packages, the records service. Phase 1 made destruction impossible; this phase makes it controlled.

PHASE 4 · EXPANSION AND DECOMMISSIONING

Further repositories and channels; absorption of the separate search engine; residency zones if the scope crosses jurisdictions; multi-tenancy where the platform is offered as a service; decommissioning of repositories that no record points to.

PHASE 5 · INTELLIGENCE

Chunking and embeddings, semantic and hybrid search, document intelligence as a producer of extracted fields, retrieval for assistants and memory for agents, each registered as a consumer with purposes, projections, audit and revocation handling.

A MINIMAL TEST SET FOR THE LISTINGS

Before the first release, the following scenarios run as automated tests against a reference environment, each with its expected outcome recorded: a duplicate capture under the same source key (one record, one object, reference count one); a replay of an older event after a newer one (stale, ignored, record unchanged); the same bytes under two source keys followed by disposition of one of them (object kept, marked shared, count one); a revocation on a dossier followed immediately by a search as the revoked user (no hit returned, document API refuses, facet marked approximate until propagation completes); a hold placed on the dossier while the disposition run is in progress (run skips or rolls back, record never reaches disposing while held); disposition of a document with two versions and three renditions (every key inventoried, every deletion confirmed, stub answers the question). The author has not yet executed them against a declared environment, as the preface states.

Readiness checklist

Each criterion has an expected evidence and an accountable owner; the result column is filled during review.

CRITERION	EXPECTED EVIDENCE	OWNER	RESULT
First consumers and searches defined	Documented use cases with measurable targets	Product owner	
Classes and required fields agreed	Validator per class; class register with versions	Data architect	
One authoritative description per document	Ownership register: layer owns metadata; custody of content per shape	Data architect	
Ingestion contract	Tests for content-before-metadata, duplicates, shared objects, ordering, deletes, dead letters	Integration lead	
Reconciliation	Scheduled comparison registry vs content store, both directions	Integration lead	
Replay	Successful replay of a past interval with identical result	Integration lead	
Freshness and index status	Target, dashboard, status exposed in API responses	Platform lead	
Permission-trimmed search	Restricted-user drill across results, facets, highlights, autocomplete, passages	Security lead	
Permission propagation	Dossier change test with lag measured; API backstop verified	Security lead	
Extraction quality	Confidence, confirmation status and review queue metrics reported	Capture lead	

CRITERION	EXPECTED EVIDENCE	OWNER	RESULT
Search quality	Precision at ten on labelled queries; zero-result rate; p95 latency	Search lead	
Registration	Transactional numbering per registry; hash chain over full entries, anchored externally, verified by a scheduled job	Records manager	
Retention and disposition	Schedules as data; disposition drill with stub, reference counts, URL lifetime wait and deletion proof	Records manager	
Legal hold	Hold and release drill; disposition skipped then executed	Legal	
Sensitivity and encryption	Field classification; keys documented; search exclusions listed	Security lead	
Subject access and erasure	Inventory of copies incl. chunks and exports; drill within one day	Data protection officer	
Export packages	Dossier exported and re-imported in an empty environment	Records manager	
Residency	Zones, buckets, search nodes and backups mapped	Platform lead	
Backup and restore	Restore test of registry, index rebuild and content store together	Platform lead	
Ownership	Permanent team with product owner, records function and roadmap	Sponsor	

CHAPTER 14

References

Standards and regulation

ISO 15489-1:2016, Information and documentation, Records management, Part 1: Concepts and principles.

ISO 23081-1:2017, Information and documentation, Records management processes, Metadata for records, Part 1: Principles.

ISO 14721:2025, Space data system practices, Reference model for an open archival information system (OAIS), third edition, March 2025.

ISO 19005-1:2005 and later parts, Document management, Electronic document file format for long-term preservation (PDF/A).

ISO 15836-1:2017, The Dublin Core metadata element set, Part 1: Core elements.

OASIS, Content Management Interoperability Services (CMIS) Version 1.1, 2013. docs.oasis-open.org

Regulation (EU) 2016/679 (General Data Protection Regulation), in particular Articles 5, 9, 17, 25 and 32. eur-lex.europa.eu

Regulation (EU) No 910/2014 (eIDAS), as amended, on electronic identification and trust services, including provisions on electronic documents and preservation services.

Regulation (EU) 2018/1724 establishing a single digital gateway, including the once-only technical system.

Agenzia per l'Italia Digitale (AgID), Linee guida sulla formazione, gestione e conservazione dei documenti informatici, in force since 1 January 2022, with their annexes on metadata and preservation. agid.gov.it. Cited as the national framework closest to the registration, classification and preservation examples of this guide; other jurisdictions have equivalents.

Information retrieval

Manning, C. D., Raghavan, P. and Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.

Cormack, G. V., Clarke, C. L. A. and Buettcher, S. (2009). Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. Proceedings of SIGIR 2009.

Reference implementation documentation

MongoDB Search, index definition and field types. mongodb.com/docs/atlas/atlas-search/define-field-mappings

MongoDB Search, compound operator and facets. mongodb.com/docs/atlas/atlas-search/compound · mongodb.com/docs/atlas/atlas-search/facet

MongoDB Search, autocomplete and highlighting. mongodb.com/docs/atlas/atlas-search/autocomplete · mongodb.com/docs/atlas/atlas-search/highlighting

MongoDB Vector Search, index and pre-filtering. mongodb.com/docs/atlas/atlas-vector-search/vector-search-stage

MongoDB hybrid search, \$rankFusion and \$scoreFusion. mongodb.com/docs/vector-search/hybrid-search/hybrid-search

MongoDB Search and Vector Search for self-managed deployments (Community Edition), general availability, June 2026. mongodb.com/products/updates/now-ga-mongodb-search-and-vector-search-for-community-edition

Change Streams. mongodb.com/docs/manual/changestreams

GridFS. mongodb.com/docs/manual/core/gridfs

TTL indexes. mongodb.com/docs/manual/core/index-ttl

Schema validation. mongodb.com/docs/manual/core/schema-validation

Zone sharding. mongodb.com/docs/manual/core/zone-sharding

Client-side field-level encryption and Queryable Encryption. mongodb.com/docs/manual/core/queryable-encryption

Atlas database auditing. mongodb.com/docs/atlas/database-auditing

Web references were checked at the time of writing; documentation changes over time.

CHAPTER 15

About the author

Mario Noioso is a software architect and author based in Rome, with more than thirty years of experience designing and delivering information systems for large enterprises and public institutions. His career spans IBM development laboratories, Microsoft and ServiceNow; he currently works as an Advisory Solutions Architect for the EMEA public sector at MongoDB, focusing on data architecture, artificial intelligence and digital sovereignty.

He is named as an inventor on United States patents in software distribution and real-time analytics, and is the author of *Who Owns Memory Owns the Future: Power, Knowledge, and Imagination from Fire to Mars* (2026), an essay on the relationship between memory, power and technology. He writes on data, AI and technology culture at marionoioso.com.

ORCID 0009-0000-9052-944X · ISNI 0000 0004 2128 1497 · marionoioso.com

About this series

Real-World Architectures is a series of short field guides on data and systems architecture, written from the perspective of practice: what works in production, in regulated industries and in public institutions, and what does not. Each volume addresses one architectural pattern, with its problem, reference design, data model, governance and pitfalls.

Volume 1, *Single View: One Governed View of Customers and Citizens*, is available free of charge at marionoioso.com/real-world-architectures/.

Feedback and corrections are welcome through marionoioso.com.

