Elasticsearch vs. OpenSearch vs. Loki vs. Quickwit vs. ClickHouse: Long-Term Log Archiving for 7+ Years

2026-05-14

Five-way log archiving comparison: Elasticsearch, OpenSearch, Loki, Quickwit, ClickHouse — tiering, compression, query languages, SaaS options. Part 1 of 3.

Long-term log archiving is a deep topic. Each of the five systems covered here has its own tiering model, object storage integration, query language, operational quirks, and security posture. To do it justice, this comparison is split across **three posts** — take your time reading them, and don't hesitate to come back to individual sections more than once.

- **Part 1 — this post:** Storage tiering, compression, resource consumption, query languages, SaaS options
- Part 2 — Operations: Setup, ingest pipeline, ECS vs. OTel, backup & DR, observability, alerting
- Part 3 — Security & Compliance: Encryption, RBAC, WORM / S3 Object Lock

---

Storing logs for compliance, auditing, or forensic purposes often means retaining data for **seven years or more**. That is a very different problem than storing the last 30 days of operational logs. Hot-tier query latency matters less; cost per GB, the ability to still search old data without restoring it manually, and the operational burden of keeping the system alive for years become the dominant concerns.

This post compares five systems — **Elasticsearch**, **OpenSearch**, **Grafana Loki**, **Quickwit**, and **ClickHouse** — along the axes that matter for long-term archiving: tiering, object storage integration, compression, resource consumption during ingest/compact/search, and available SaaS options. A separate section covers systems that were evaluated but are **not recommended** for this use case.

TL;DR

- **Elasticsearch** is the only system with native automated tiering (ILM + Frozen Tier): old indices move to S3 automatically and remain searchable. LogsDB mode (GA since 8.15/9.x) reduces storage by up to 77 % compared to classic ES.
- **Loki** has the lowest ingest cost and cheapest long-term storage, but fulltext search is a distributed grep — expensive without tight label filters.

- **Quickwit** has the best ingest-to-search trade-off: Rust efficiency, inverted index, object storage as primary storage with no cluster replication overhead. Still pre-1.0; roadmap and enterprise features are uncertain (see SaaS Options).
- **OpenSearch** is the Apache 2.0 fork of Elasticsearch 7.10. Its Frozen Tier (Searchable Snapshots) is **free** — the most important practical difference from Elasticsearch for budget-conscious teams.
- **ClickHouse** offers the most flexible TTL-based automated S3 tiering of all five systems, with exceptional compression and C++ efficiency. The trade-offs: stable fields must be declared as columns (though `ALTER TABLE ADD COLUMN` is zero-downtime, and the `JSON` column type in 24.x handles dynamic/unknown fields); and there is no Lucene-style fulltext inverted index.

Overview

	Elasticsearch	OpenSearch	Loki	Quickwit	ClickHouse
Language	Java (JVM)	Java (JVM)	Go	Rust	C++
First release	Feb 2010	Feb 2021	Apr 2018	Apr 2021	Jun 2016
Current version	9.x (stable)	2.x (stable)	3.7.x (stable)	0.9.x (pre-1.0)	24.x (stable)
GitHub stars	~70,000	~10,000	~28,000	~10,000	~37,000
Long-term tiering	ILM: hot → warm → cold → frozen → delete	ISM + Searchable Snapshots (free)	Retention policy only, no tiering	Retention policy only, no tiering	TTL MOVE TO DISK (SQL, automatic)
Object storage	Add-on (Searchable Snapshots)	Add-on (Searchable Snapshots, free)	First-class (primary store)	First-class (required)	First-class (MergeTree on S3)
Compression	LZ4 default	LZ4 default	Snappy/LZ4/gzip (chunks)	zstd level 8 (docstore)	LZ4/zstd (columnar, 5–10×)
SaaS	Elastic Cloud (Hosted + Serverless)	Amazon OpenSearch Service	Grafana Cloud Logs (~$0.50/GB)	None (Datadog acquisition)	ClickHouse Cloud, Altinity.Cloud

Elasticsearch also supports zstd via the `best_compression` codec and reduces storage by 46–77 % with LogsDB mode (GA since 8.15). OpenSearch supports the same `best_compression` codec but has no LogsDB equivalent.

Long-Term Archiving (7+ Years)

This is where the five systems diverge most sharply.

Elasticsearch: Index Lifecycle Management (ILM)

ILM is the only fully automated tiering mechanism of the five. You define age thresholds and ES moves indices through five phases without manual intervention:

Phase	Storage	Typical age	Notes
Hot	Fast NVMe	0–7 days	Active writes and reads
Warm	HDD / slower SSD	7–30 days	Read-only, force-merged
Cold	Reduced resources	30–180 days	Searchable, slower
Frozen	S3 only (Searchable Snapshots)	180 days – 7 years	Only metadata local, search on demand
Delete	—	7+ years	Configurable

The **Frozen Tier** is the key feature for multi-year retention. Indices are stored entirely in object storage; the cluster only caches metadata locally. Queries still work but take longer (cold read from S3). For compliance use cases where data must be retained but is rarely accessed, this is close to ideal: automated, searchable, and cheap.

Loki: Retention Policy Only

Loki supports configurable retention globally or **per-tenant and per-stream** via the Compactor. This granularity is useful in multi-tenant setups (e.g. retain audit logs for 7 years, debug logs for 30 days). However, there is **no tiering**. All data lives in the same object storage bucket regardless of age. Cost optimisation for older data requires manual S3 Lifecycle Policies at the bucket level — Loki has no concept of moving data to a cheaper storage class automatically.

Schema migrations across years are supported (BoltDB → TSDB is one example), which is a practical requirement for any system operating for 7+ years.

Quickwit: Retention via Janitor Service

Quickwit's janitor service deletes expired splits based on a configured `period` (e.g. `2555 days` for 7 years) on a schedule. Since all data already lives in object storage from day one, there is no "move to cold tier" step — but equally there is no automated differentiation between a 1-day-old split and a 7-year-old split. As with Loki, S3 Intelligent-Tiering can be applied at the bucket level. The Quickwit documentation explicitly warns that S3 Intelligent-Tiering can be counterproductive for search workloads: every read of an old file resets its 30-day hot-tier timer, potentially increasing cost.

ClickHouse: TTL with Automatic Storage Tiers

ClickHouse uses SQL TTL expressions on MergeTree tables to move data parts between named storage disks automatically:

```
TTL timestamp + INTERVAL 30 DAY TO DISK 's3warm',
    timestamp + INTERVAL 180 DAY TO DISK 's3cold'
```

Storage disks are defined in `storage_configuration` (local SSD → S3 warm → S3 cold, each pointing to a different S3 prefix or bucket). ClickHouse moves data parts in the background when the TTL threshold is crossed. Unlike ES/OpenSearch, there is no concept of a "frozen" state: cold-tier parts are still fully queryable via the same SQL interface — the only difference is I/O latency on first access.

Schema evolution is a real concern for any system operating for 7+ years, and ClickHouse handles it differently from ES/Loki/Quickwit:

- **`ALTER TABLE ADD COLUMN`** is a metadata-only operation on MergeTree — no data rewrite, no downtime. Old parts simply return the column default for the new field. This is the primary path for promoting a log field from "occasionally present" to "first-class column".
- **`JSON` column type** (GA in ClickHouse 24.x): a single `attributes JSON` column can store arbitrary nested JSON, with sub-paths readable as typed virtual columns. This is the escape hatch for truly unknown future fields. Query performance on JSON sub-paths is lower than on dedicated columns, but the data is still stored and queryable.
- **`Map(String, String)`** is a simpler alternative for flat key-value log attributes (e.g. OpenTelemetry resource attributes).

The practical pattern for long-term log tables is a **hybrid schema**: known stable fields as dedicated columns (`timestamp`, `level`, `service`, `trace_id`, `message`), and an `attributes JSON` column for everything else. When a dynamic field proves stable, it can be promoted with `ALTER TABLE ADD COLUMN` and optionally backfilled.

The real constraint is not "I cannot add fields" but "queries on `attributes` JSON sub-paths are slower than queries on proper columns." Schema design choices made at deployment time affect query performance for the full retention period.

OpenSearch: Searchable Snapshots (Free)

OpenSearch's Index State Management (ISM) is the functional equivalent of Elasticsearch's ILM. It moves indices through phases and can trigger a Searchable Snapshot for long-term retention. The key difference from ES:

The Frozen Tier is free in OpenSearch. In Elasticsearch it is a paid Enterprise feature. For teams that want the ES data model (Lucene, Kibana-style dashboards) without the licensing cost, OpenSearch is the direct alternative.

OpenSearch does **not** have a LogsDB equivalent — there is no Synthetic Source, no automatic index sorting, and no 46–77 % storage reduction.

Verdict on long-term archiving: Elasticsearch ILM + Frozen Tier and its OpenSearch equivalent (ISM + Searchable Snapshots, free) are the most production-ready automated solutions for 7-year retention with Lucene-style searchability. ClickHouse TTL tiering is the most flexible SQL alternative, with additional storage class granularity. Loki and Quickwit require manual S3 lifecycle policy configuration and offer no in-cluster tiering.

Object Storage Integration

	Elasticsearch	OpenSearch	Loki	Quickwit	ClickHouse
Integration type	Add-on (Snapshot Repositories + Searchable Snapshots)	Add-on (Snapshot + Searchable Snapshots, free)	First-class primary store	Required primary store	First-class (MergeTree S3 disk / SharedMergeTree)
Supported backends	S3, Azure Blob, GCS	S3, Azure, GCS	S3, GCS, Azure, IBM COS, Alibaba OSS, Baidu BOS	S3, Azure Blob, GCS, MinIO, Garage	S3, GCS, Azure, MinIO, HDFS
Local disk required	Yes (primary for hot/warm)	Yes (primary for hot/warm)	Yes (Compactor WAL, Ingester WAL)	No (split cache optional)	SSD for hot parts; none for cold parts

Quickwit and ClickHouse (cold tier) are architecturally the most consistent for archived data: once parts are moved to S3 via TTL, the node holds no local copy. ClickHouse hot-tier parts still live on local SSD during active ingestion.

Elasticsearch and OpenSearch are functionally equivalent here: both treat object storage as a snapshot target and use Searchable Snapshots for the frozen tier. The Compactor in Loki must run as a singleton with persistent local storage for marker files — an operational constraint for teams running fully ephemeral nodes.

Compression

Elasticsearch

- **Default codec:** LZ4 (fast, moderate compression)
- **`best_compression` codec:** zstd since ES 8.17
- **LogsDB mode** (GA since ES 8.15 / 9.x): combines zstd + Synthetic Source (raw JSON not stored separately) + index sorting by `host.name` + `@timestamp`. Result: **46–77 % storage reduction** compared to classic ES. A 162 GB dataset shrinks to 37–40 GB. LogsDB is the recommended mode for new log deployments.

OpenSearch

- **Default codec:** LZ4 (same as ES)
- **`best_compression` codec:** zstd since OpenSearch 2.x
- **No LogsDB equivalent.** OpenSearch does not have Synthetic Source or automatic index sorting. Storage overhead is similar to classic Elasticsearch (pre-LogsDB). For the same dataset, expect 20–30 % more storage than ES 9.x with LogsDB enabled.

Loki

Compression applies to **chunks** (not individual lines):

Codec	Speed	Compression
Snappy	Very fast	Moderate (~5–10× on raw logs)
LZ4	Fast	Good (recommended balance)
gzip	Slow	Best

gzip is CPU-intensive on decompression; prefer LZ4 for latency-sensitive queries.

Loki's minimal index (labels only, no fulltext) keeps the index itself tiny — typically 1–5 % of raw log volume. Total storage is dominated by chunk size.

Quickwit

- **Docstore (raw document bytes):** zstd level 8 (configurable via `docstore_compression_level`)
- **Tantivy index structures:** columnar encoding + dictionary compression within splits
- **Typical compression ratio:** ~2.75x on real-world structured log datasets (index including inverted index, columnar fast fields, and row store)

The tradeoff: Quickwit stores an inverted index in addition to the raw data, so the uncompressed size before compression is larger than Loki's. After zstd the effective storage overhead is moderate.

ClickHouse

Columnar storage is the foundation of ClickHouse's compression advantage: all values of a single column are stored together, so the compressor sees long runs of similar values (repeated `level`, `service`, `host` strings; monotonically increasing timestamps).

- **Default codec:** LZ4 per column
- **Configurable per column:** zstd, or composable codecs such as `CODEC(Delta, ZSTD(3))` for timestamp columns — Delta encoding reduces inter-value differences to near-zero, then ZSTD compresses those tiny deltas to almost nothing. Typical timestamp compression: 10–50×.
- **Effective ratio on structured logs:** 5–10× on raw JSON, often better for columns with high repetition (host, service, log level).
- No large inverted index overhead: the only index overhead is a small sparse primary key index per data part.

Resource Consumption

Ingest

	Elasticsearch	OpenSearch	Loki	Quickwit	ClickHouse
CPU	Very high (~75 % @ 20k lines/s)	Very high (same JVM profile as ES)	Very low (~15 % @ 21k lines/s)	Medium (~7.5 MB/s per core)	Low–medium (C++, columnar)
Minimum RAM	32–64 GB per node (JVM heap)	32–64 GB per node (JVM heap)	4–8 GB per ingester	8 GB per node	4–8 GB per node
Network amplification	2–4× (replica shards)	2–4× (replica shards)	3× (replication factor)	~1× (object storage)	~1× (replication optional)
Durability mechanism	Translog (fsync per bulk request)	Translog	WAL (recommended, persistent volume)	Ingest Queue V2 (disk buffer, 4 GiB default)	Async insert + WAL (configurable)

Elasticsearch and OpenSearch share the same fundamental JVM-based ingest path: every log document is written into four on-disk structures (inverted index, doc values, BKD tree, `_source`), generating high CPU and requiring large JVM heaps. The 31 GB Compressed OOPs ceiling forces 64 GB nodes as a practical minimum.

ClickHouse ingests in batches (recommended: `async_insert` or explicit client-side batching). The C++ columnar write path is significantly cheaper. RAM can be as low as 4–8 GB for modest ingest rates.

Compacting

	Elasticsearch	OpenSearch	Loki	Quickwit	ClickHouse
Mechanism	Lucene segment merge (continuous)	Lucene segment merge (continuous)	TSDB index compaction (periodic)	Split merge (continuous)	MergeTree part merge (background)
CPU	High, unpredictable	High, unpredictable (JVM GC spikes)	Low–medium	Medium, predictable	Medium, predictable (C++)
RAM	High (JVM heap pressure)	High (JVM heap pressure)	10–40 GB (singleton Compactor)	Proportional to 4 GB/core	Low–medium (no JVM heap)
I/O profile	Random local disk I/O	Random local disk I/O	Object storage bandwidth	Local SSD → S3 upload	Local SSD → S3 upload (TTL parts)
Predictability	Poor (JVM GC spikes)	Poor (JVM GC spikes)	Good (isolated, periodic)	Medium	Good

Elasticsearch and OpenSearch segment merges compete with live queries for I/O and CPU; known incidents of 100 % CPU from post-upgrade merge storms are documented in both projects' issue trackers. ClickHouse MergeTree merges are also continuous background processes, but the C++ runtime avoids JVM GC pauses, making resource usage more predictable under load.

Searching Labels / Metadata

All five systems handle label or term queries efficiently — these hit compact index structures and return in milliseconds when data is warm.

	Elasticsearch	OpenSearch	Loki	Quickwit	ClickHouse
Latency (warm)	< 10 ms	< 10 ms	< 100 ms	< 100 ms (with hotcache)	< 10 ms (primary key / skip index)
CPU	Low	Low	Low	Low (with tag pruning)	Low
Anti-pattern	Too many shards	Too many shards	High label cardinality	No `tag_fields` configured	High-cardinality ORDER BY or missing skip index

ClickHouse primary key lookups are extremely fast due to the sparse primary index and block-level skip indexes. The anti-pattern is a query on a high-cardinality column that is not part of the primary key and has no skip index — this degrades to a full columnar scan.

Quickwit's `tag_fields` feature skips splits that do not contain the searched label value entirely, without reading them from S3. For multi-tenant setups (e.g. `tenant_id` as a tag field) this can reduce S3 GET requests by orders of magnitude.

Fulltext Search in Log Messages

This is where the five systems differ most dramatically.

	Elasticsearch	OpenSearch	Loki	Quickwit	ClickHouse
Index type	Inverted index (Lucene)	Inverted index (Lucene)	**None — distributed grep**	Inverted index (Tantivy)	**None** — column scan + bloom filters
CPU per query	Low (index hit)	Low (index hit)	**Extremely high**	Low (index hit)	Medium–high (columnar scan, SIMD)
RAM per query	Medium–high (JVM field data)	Medium–high (JVM field data)	High (chunk decompression)	Medium (configurable caches)	Low–medium
Latency (warm)	Low	Low	High–very high	Low–medium	Medium (selectivity-dependent)

Loki has no inverted index for log content. `|= "error 500"` reads, decompresses, and scans every chunk of every matching stream. Benchmark numbers (Quickwit, 2024, 212 GB / 243 million logs):

- Fulltext over entire dataset: Loki requires **+5,270 % more CPU time** than Quickwit
- Label-filtered fulltext: Loki still requires **+435 % more CPU time** than Quickwit

ClickHouse also has no inverted index for free-text columns. SIMD-accelerated columnar scans are substantially faster than Loki's chunk decompression path. For exact-token queries, `tokenbf_v1` skip indexes can prune entire 8 KB data blocks that do not contain the token without reading them.

For regex or substring searches without a matching bloom filter, ClickHouse falls back to a full columnar scan — faster than Loki in practice, slower than a Lucene or Tantivy inverted index.

Elasticsearch, OpenSearch, and Quickwit all provide fast fulltext via inverted index. Elasticsearch has BM25 relevance scoring and the richest aggregation support; Quickwit has the lowest per-query CPU cost (Rust vs. JVM) and transparent object storage as primary store.

Storage Sizing

Worked example: 100 GB/day raw log volume

100 GB/day × 365 × 7 = **256 TB raw** over the full retention period.

After compression, this is what each system stores in object storage:

	Compression ratio	7-year archive in S3
Elasticsearch (LogsDB)	~4.5×	~57 TB
OpenSearch (no LogsDB)	~2.5×	~102 TB
Loki	~10–20×	~13–26 TB
Quickwit	~2.75×	~93 TB
ClickHouse	~5–10×	~26–51 TB

Scale linearly: 10 GB/day ⟶ divide by 10; 1 TB/day ⟶ multiply by 10.

> **Measure before you size**
>
> Compression ratios depend heavily on log structure. JSON logs with repeated field names, enum-like values (log level, service name), and monotonically increasing timestamps compress the best. Free-form syslog compresses less well. Always measure on a sample of your own logs before committing to a capacity plan.

Hot tier and local disk

Every system needs local storage before data reaches object storage, or keeps a warm tier on local disks throughout:

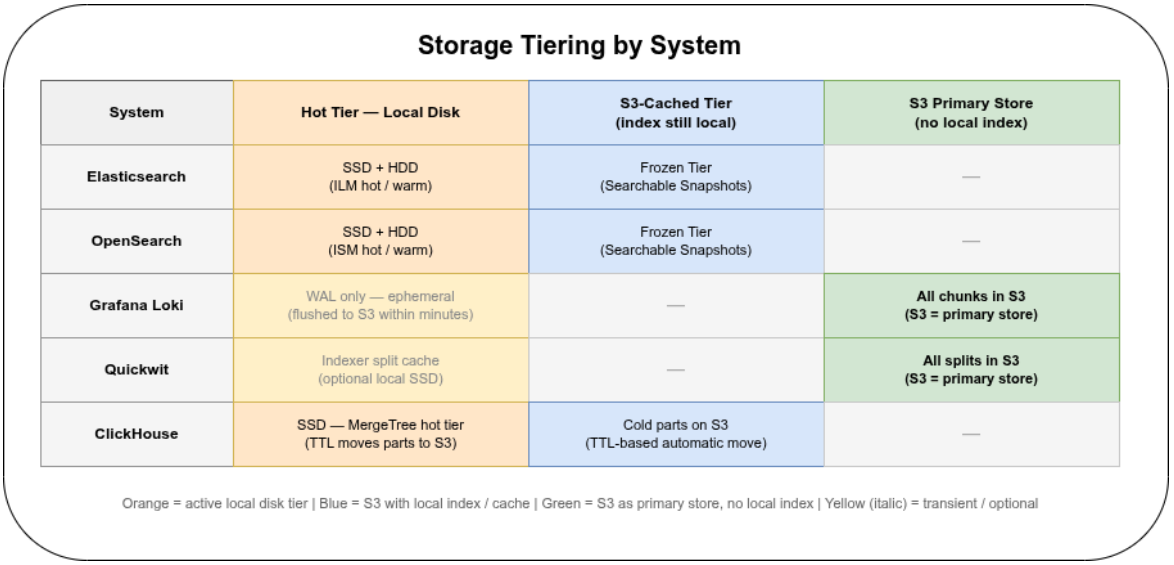

Figure 1: Storage tiering by system: hot local disk, S3-cached tier, and S3 primary store for Elasticsearch, OpenSearch, Loki, Quickwit, and ClickHouse

	What lives on local disk	Local disk at 100 GB/day raw
Elasticsearch	Hot phase (0–7 d) + warm phase (7–30 d); each node holds full shard copies	Hot SSD: ~155 GB; Warm HDD: ~670 GB (LogsDB); **×2 for replicas**
OpenSearch	Same hot/warm structure, no LogsDB compression	Hot SSD: ~280 GB; Warm HDD: ~1.2 TB; **×2 for replicas**
Loki	Ingester WAL + unflushed chunks (flush within minutes to hours)	WAL: a few GB; Compactor needs ~100 MB for state files only
Quickwit	Optional read cache for hot splits; S3 is primary from day one	0 required; 10–100 GB cache improves query latency
ClickHouse	Hot-tier parts until the TTL threshold moves them to S3 (e.g. 30 days)	~430 GB SSD at 30-day hot TTL; set 7-day TTL to cut to ~100 GB

For **Elasticsearch and OpenSearch**, the dominant cost before the frozen tier is *running nodes*: warm-tier data nodes hold full shard copies in a live cluster. Replication doubles the local storage figure. The frozen tier eliminates node compute — only S3 storage charges remain for data older than the cold-to-frozen transition.

For **ClickHouse**, the hot TTL threshold is a direct tuning knob for SSD budget: a 7-day hot tier needs 4× less SSD than a 30-day hot tier, at the cost of more S3 reads when querying recent data.

Object storage cost estimate (OVH)

Based on OVH Object Storage — all traffic (ingress and egress) is free on OVH. Storage only.

For a compliance archive, **Infrequent Access** is the right tier (0.00000652 €/GiB/h ≈ 0.00476 €/GiB/month, no retrieval fee, 30-day minimum retention). Standard (0.00001156 €/GiB/h ≈ 0.0084 €/GiB/month) makes sense only for data accessed daily. Aktives Archiv costs more per GiB than Infrequent Access *and* charges 0.0214 €/GiB on retrieval — avoid it for log archives where you occasionally need to search old data.

The archive builds up linearly over 7 years: the first month holds ~1/84 of the final volume, the last month holds the full archive. The total accumulated cost is therefore approximately the year-7 monthly cost × 42 (half of 84 months):

	Archive at year 7	€/month at year 7	Total over 7 years
ES (LogsDB)	~57 TB	~€271	~€11,400
OpenSearch	~102 TB	~€486	~€20,400
Loki	~18 TB	~€86	~€3,600
Quickwit	~93 TB	~€443	~€18,600
ClickHouse	~37 TB	~€176	~€7,400

Infrequent Access tier. Scale linearly for different ingest rates.

Loki and ClickHouse are the cheapest to store: Loki because it carries no fulltext index overhead, ClickHouse because columnar compression is the most effective on structured log data.

One caveat specific to Quickwit: its documentation warns that S3 Intelligent-Tiering can be counterproductive — every search read of an old split resets its 30-day hot-tier timer, potentially increasing cost. Stick with a fixed storage class for Quickwit archives.

Query Languages and APIs

The choice of log store is also a choice of query interface — for day-to-day operations and for the people who will investigate incidents years later.

	Elasticsearch	OpenSearch	Loki	Quickwit	ClickHouse
Query language	Query DSL (JSON) + KQL / EQL	Same as ES	LogQL	JSON API (Lucene-like syntax)	SQL
Aggregations	Very rich (bucket, metric, pipeline)	Same as ES	Limited (LogQL metric queries)	Basic	Very rich (GROUP BY, window functions)
Primary UI	Kibana Discover	OpenSearch Dashboards	Grafana Explore	Quickwit Web UI	Grafana, clickhouse-client
Grafana datasource	✓	✓	✓ native	✓ community	✓ official plugin
Skill requirement	ES Query DSL (proprietary JSON)	Same as ES	LogQL (new language)	Lucene-like syntax	SQL (widely known)

Elasticsearch / OpenSearch use a JSON Query DSL that is powerful but verbose and proprietary. Kibana Query Language (KQL) provides a simpler filter syntax in the UI. EQL (Event Query Language) in ES adds sequence detection for security scenarios (SIEM, threat hunting) — not needed for general log archiving but relevant for compliance-oriented SOC deployments.

Loki's LogQL is a concise pipeline syntax (`{app="api"} |= "error 500" | json`) that integrates naturally with Grafana. Its weakness is aggregation expressiveness: complex metric queries over log data are more limited than SQL or ES aggregations.

ClickHouse is the only system of the five that uses standard SQL — `GROUP BY`, window functions, `WITH ROLLUP`, and time-series aggregations (`toStartOfHour(timestamp)`) work out of the box. The trade-off: no native inverted index means fulltext queries use `LIKE`, `match()`, or `hasToken()` rather than the richer query semantics of ES Query DSL.

For a system operated by rotating teams over 7+ years, SQL's ubiquity is an underrated operational advantage: onboarding a new engineer to query ClickHouse log data requires no training beyond standard SQL.

Part 2: Operations covers ingest pipeline setup, backup & DR, observability, and alerting. Part 3: Security & Compliance covers encryption, access control, and WORM compliance.

SaaS Options

	Elastic Cloud	Amazon OpenSearch Service	Grafana Cloud Logs	Quickwit	ClickHouse Cloud
Operator	Elastic	AWS	Grafana Labs	None	ClickHouse Inc.
Model	Hosted (dedicated) or Serverless	Hosted (managed cluster)	Usage-based	—	Usage-based (serverless or dedicated)
Price (rough)	Hosted: from ~$190/month; Serverless: ~$0.09/VCU-h + ~$0.047/GB/month	~$0.10–0.24/GB/month (instance-dependent)	~$0.50/GB ingested	—	~$0.023/GB stored/month + compute
Free tier	Trial	—	50 GB/month, 14-day retention	—	Trial
Marketplace	AWS, Azure, GCP	AWS only	AWS, Azure, GCP	—	AWS, Azure, GCP
Long-term retention	Frozen Tier on Cloud	Searchable Snapshots (free)	Configurable (paid add-on)	—	Tiered S3 storage included

As of this writing (May 2026), Quickwit has **no known managed offering**. The company was acquired by Datadog in January 2025. The open-source project (Apache 2.0) continues, but there is no standalone "Quickwit Cloud" and no publicly announced roadmap for one. This may change — verify the current state before making a long-term deployment decision. The acquisition uncertainty remains a strategic risk for teams planning a 7+ year commitment.

When to Use Which

Use Elasticsearch when

- You need **fully automated multi-year tiering** without manual S3 lifecycle management.
- Queries require **rich aggregations, BM25 scoring**, or complex query DSL.
- Your team already operates Elastic stack (Kibana, APM, Fleet).
- You can afford the operational overhead and hardware costs of a JVM-based cluster.

- LogsDB mode is acceptable (no random field access, no custom `_source` access).

Use OpenSearch when

- You want Elasticsearch's data model and Lucene fulltext search, but the **Frozen Tier license cost is a constraint** — OpenSearch's equivalent is free.
- You are on AWS and Amazon OpenSearch Service reduces operational overhead.
- Your team prefers **Apache 2.0** over Elastic License 2.0 / SSPL.
- You do not need LogsDB mode (OpenSearch has no equivalent).

Use Loki when

- Log volume is high, retention is long, but **fulltext queries are rare or always label-filtered** (e.g. `{app="payments"} |= "timeout"`).
- You live in the **Grafana ecosystem** (already using Prometheus, Grafana dashboards).
- You need per-tenant/per-stream retention policies in a multi-tenant setup.
- Cost is the primary constraint and query latency on old data is acceptable.
- You are on Grafana Cloud and want a managed solution without operating infrastructure.

Use Quickwit when

- You need **efficient fulltext search on object-stored logs** without JVM overhead.
- Ingest scale is high and cluster replication cost matters.
- You are building a **new** log pipeline and want cloud-native architecture from day one.
- You accept pre-1.0 software maturity and the associated roadmap uncertainty.

Use ClickHouse when

- Your logs have a **stable core schema** — even if not all fields are known upfront, you can define stable fields as columns and use a `JSON` or `Map` column for dynamic attributes. `ALTER TABLE ADD COLUMN` is zero-downtime for future promotions.
- You need the **best combination of compression and analytical query performance** — SQL aggregations, time-series rollups, dashboards over log data.
- Fulltext grep is rare or always scoped to narrow time windows and specific fields.
- You are already operating ClickHouse for metrics or events — converging log storage avoids a second operational stack.

Not Recommended for This Use Case

The following systems were evaluated but are **not suitable** for 7+ year, cost-effective log archiving with search capability.

VictoriaLogs

VictoriaMetrics's purpose-built log engine is impressive for operational (short-term) use cases: low resource consumption, fast ingest, simple single-binary deployment. However, as of May 2026 it **does not support object storage as a primary store** — data lives on local disk only. For a 7-year archive this would require enormous local storage costs. S3 backend support is on the roadmap but not yet delivered. Revisit once that ships.

Graylog

Graylog is a **log management platform** (UI, pipelines, alerting, access control), not a storage engine. It requires Elasticsearch or OpenSearch as its storage backend. Choosing Graylog does not replace any of the systems evaluated here — it adds a management and routing layer on top of one of them. Evaluate Graylog for its operational interface, not as a storage alternative.

FAQ

Can Loki search 7-year-old logs?

Yes, as long as the data is not expired by retention policy. Loki will load the chunks from object storage and grep through them. For rare, pinpoint queries (`trace_id` with Bloom Filters) this is acceptable. For broad fulltext queries over large time windows on old data, expect high latency and cost.

Does Quickwit really need no local disk?

For the Searcher role: yes, object storage is the primary store and the split cache is optional (improves latency). For the Indexer role: a local SSD is recommended for the split cache and ingest queue, but not strictly required.

Is Elasticsearch LogsDB a breaking change?

LogsDB uses Synthetic `_source` — the stored `_source` field is reconstructed at query time rather than stored as raw JSON. Applications that rely on exact `_source` byte-for-byte fidelity (e.g. re-indexing pipelines) need to be tested. For pure log ingestion and search this is transparent.

What about ClickHouse or Apache Doris for log archiving?

Both are strong alternatives, especially ClickHouse which has excellent compression and fast analytical queries. They are column stores rather than document search engines, which means different trade-offs on fulltext query expressiveness vs. aggregation performance. Worth a dedicated comparison for high-volume analytical log workloads.

Is OpenSearch a drop-in replacement for Elasticsearch?

For most log archiving purposes, yes. OpenSearch started as a fork of ES 7.10 and has diverged since, but the Lucene storage model, ISM (ILM equivalent), Searchable Snapshots (Frozen Tier), and OpenSearch Dashboards (Kibana equivalent) are all present. The most important operational difference: the Frozen Tier is free. Notable gaps: no LogsDB mode, no BM25 improvements merged after ES 7.10 (relevant for search-heavy workloads), and the API has diverged for some advanced features. Migration from ES 7.10 is straightforward; from ES 8.x/9.x requires compatibility testing.

What about Splunk?

Splunk (now a Cisco product following its $28 billion acquisition in 2024) is the enterprise standard for SIEM and security log analytics. It supports S3-based tiered storage via SmartStore and has mature long-term retention features. It is not covered here because it is proprietary and its pricing model — historically charged per GB/day ingested — puts it at roughly 10–100x the cost of the open-source options for the same volume. If budget is not a constraint and your primary requirement is SIEM/compliance with an established vendor, Splunk is a valid choice. For infrastructure log archiving at scale, the open-source alternatives covered in this post are far more cost-effective.

Continue reading: Part 2: Operations — ingest pipelines, Kubernetes and bare-metal operations, backup & DR, observability, alerting. Part 3: Security & Compliance — encryption, RBAC, and WORM compliance.