SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V12I8P101

Volume 12 Issue 8, 1-12, August 2025
© 2025 Seventh Sense Research Group®

Original Article

Towards Intelligent Data Retention Recommendations in DevOps Using Elasticsearch and ML

Govind Singh Rawat

TikTok US Data Security Inc, California, USA.

[1]*Corresponding Author : govindrawat54@gmail.com*

Received: 03 June 2025 Revised: 07 July 2025 Accepted: 23 July 2025 Published: 14 August 2025

Abstract - *DevOps teams face an ever-growing challenge in managing log and metrics data: how long to retain data to balance operational value against storage costs and performance constraints. Traditional static retention policies struggle to cope with explosive data growth and evolving compliance requirements. In this work, we propose an intelligent data retention recommendation system that leverages Elasticsearch's rich monitoring data and Machine Learning (ML) to suggest optimal retention periods for indices dynamically. Our approach collects metrics on query load, storage use, and index lifecycle policies from a live Elasticsearch cluster and trains an ML model to predict the retention duration that minimizes cost while preserving necessary data availability. We present a framework where the model learns usage patterns and system constraints, recommending when to tier or delete indices. Preliminary evaluations suggest that the ML-driven approach can reduce storage costs and cluster strain by avoiding over-retention of seldom-accessed data, without compromising on query performance or compliance. This paper details the related work in intelligent log management, the theoretical underpinnings of our approach, the design of our ML-based retention recommender, and experimental results in a DevOps context. We conclude with insights into the benefits of adaptive data retention and discuss future improvements for integrating such systems into automated DevOps pipelines.*

Keywords - *Data Retention, DevOps, Elasticsearch, Predictive Analytics, Log Management.*

1. Introduction

Modern organizations generate massive volumes of operational data (logs, metrics, traces) as part of DevOps [1] practices. With the rise of microservices and distributed systems, data retention becomes a critical concern as teams must decide how long to preserve logs for value extraction (monitoring, analytics) without incurring prohibitive storage costs or violating regulations such as the General Data Protection Regulation (GDPR) [2]. Industry surveys indicate log data volumes are growing at an "explosive rate" of over 250% annually [3], which exacerbates the cost of storing logs and challenges the feasibility of capturing 100% of logs at full retention [3]. These static data retention policies seem impractical with growing data needs.

Despite the availability of Elasticsearch's Index Lifecycle Management (ILM), which provides a mechanism to manage time-based indices through phases (hot, warm, cold, delete) and actions like rollover and delete [4], most retention policies are statically defined and rarely updated based on real-time usage or access patterns. As a result, teams often over-retain or under-retain data, leading to inefficient storage usage or missed insights. Current methods lack dynamic, data-driven mechanisms that adapt to actual system behavior, which creates a gap that needs to be filled.

There is a clear need for an intelligent, automated framework to suggest optimal data retention periods for log indices based on query behavior, system constraints, and cost-performance trade-offs.

This paper presents a Machine Learning (ML)-based data retention recommendation system integrated with Elasticsearch monitoring. Unlike existing static policies or vendor-specific black-box tools, our solution is transparent, customizable, and rooted in empirical access behavior. We contribute a detailed methodology for feature extraction, model training, and experimental validation, filling a crucial gap in dynamic retention management in DevOps.

Recent developments in AIOps [5] and intelligent data management suggest that machine learning can aid in automating such decisions. Instead of fixed policies, organizations are exploring *ML-driven data lifecycle management* where algorithms analyze historical data usage and recommend when to archive or delete data. For instance, predictive analytics tools can examine log access patterns to

This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/)

forecast future storage needs and optimize retention periods [6]. By identifying logs that are rarely accessed beyond a certain age, an ML system might suggest shorter retention for those logs, avoiding unnecessary storage of cold data. Conversely, if certain data remains frequently queried, the system could recommend extending its retention to improve operational insight.

1.1. Motivation

In this context, our work is driven by the question: *Can we automatically learn the optimal data retention period for each index in an Elasticsearch-based logging platform, using operational metrics and machine learning?* By *optimal*, we mean a retention duration that balances three key factors: (1) retaining enough data to satisfy operational needs (troubleshooting, analytics, compliance), (2) minimizing storage and infrastructure costs, and (3) maintaining system performance (avoiding indices so large or numerous that queries slow down or cluster stability suffers). Achieving this balance is complex because the "value" of data diminishes over time in a non-uniform way, and cluster resource constraints (disk high-water marks [7], heap usage, etc.) can change dynamically. A data-driven recommendation system could continuously learn from the environment and make nuanced retention suggestions beyond simple one-size-fits-all rules.

1.2. Contributions

This paper presents a framework for *intelligent data retention recommendations in DevOps using Elasticsearch and ML*. We outline how to instrument an Elasticsearch cluster to collect pertinent metrics (query rates, index sizes, shard performance, etc.), how to engineer features that capture data "hotness" and system stress, and how to train an ML model to output retention recommendations (either as a regression predicting a number of days to retain, or classification into retention categories). We also propose a feedback loop where the system's recommendations are periodically evaluated and the model updated, creating an adaptive solution that improves over time. Through a case study and experimental simulation, we demonstrate that our ML-based approach can yield recommendations that would have saved significant storage without losing important data compared to static policies.

The rest of this paper is organized as follows: Section 2 reviews related work in data retention management and prior approaches to using ML for operational data optimization. Section 3 develops the theoretical basis of our approach, including the metrics considered and an outline of the predictive model for retention. Section 4 describes the experimental methodology and system design, detailing data collection, feature engineering, model training, and deployment architecture. Section 5 presents results and discussion, comparing the ML-driven recommendations to baseline rules and analyzing their impact on system performance and cost. Section 6 concludes the paper with a summary of findings and discusses future scope for integrating intelligent retention in broader DevOps workflows, followed by notes on data availability and study limitations.

2. Related Work

Data retention policies and intelligent log management have garnered attention in both industry and academia due to the twin pressures of data growth and compliance. Traditional research in log management has focused on scalable storage and search techniques (e.g., the development of the ELK Stack), but only recently have works begun to address *adaptive retention*. In this section, we highlight relevant studies and solutions:

2.1. Static vs. Dynamic Retention

Historically, organizations relied on static retention rules: for example, keeping 30 days of logs on hot storage and archiving anything older to cheaper storage or deleting after 90 days. Elasticsearch's ILM greatly eases policy enforcement; it does not inherently decide what the optimal period should be, as that decision is left to human operators. As data volumes exploded and access patterns diversified, the shortcomings of static policies became evident. Chundru and Mudunuri (2025) [8] argue that *"reliance on manual classification and static retention policies is no longer viable"* under exponential data growth. They propose a machine learning-driven approach to intelligent data lifecycle management that automates data classification and retention based on usage and compliance needs. Our work aligns with this vision of moving from one-time policy setting to continuous, data-informed policy adjustment.

2.2. ML for Data Lifecycle

The application of AI/ML to data retention is emerging as part of the broader trend of AIOp**s** and intelligent data management. J. Bamini et al (2025) [9] and Bharath Thandalam (2025) [10] explored using AI to predict data usage patterns and apply retention policies accordingly, reflecting a push towards *adaptive retention frameworks*. Our proposed system draws inspiration from these approaches by using ML models to learn retention decisions from data characteristics and usage.

2.3. Log Analytics and Usage Patterns

Another area of related work is log analytics using machine learning, though typically focused on anomaly detection and failure prediction rather than retention. Projects like Elastic's machine learning for Elasticsearch (X-Pack) have shown that unsupervised ML can detect anomalous patterns in time-series metric data [11]. While not directly about retention, it demonstrates the feasibility of applying ML in real-time on operational data. Some research efforts (e.g., in log mining) have utilized clustering and classification on log messages to identify recurring issues

and thereby inform what logs are important to keep. For instance, researchers have noted that ML models can continuously learn from historical logs to identify patterns that are hard to spot manually [12]. This directly supports our premise that ML can fine-tune retention: by learning which indices see steep drop-offs in access after X days, the system might recommend X (or slightly above) as the index's retention limit.

2.4. Academic Research

Direct academic literature on ML-driven retention policies is still nascent. A search of recent publications yields conceptual works like the aforementioned IGI Global chapter by Chundru and Mudunuri (2025), and some related discussions in the context of data governance. For example, J. Li, S. Singhal, R. Swaminathan, and A. H. Karp, "Managing Data Retention Policies at Scale," [13] explore how policy-based and algorithmic methods, including AI techniques, can be used to manage data retention at scale, covering aspects such as optimal storage durations, automated policy enforcement, and governance. It is one of the few scholarly works that directly address the intersection of AI, data governance, and retention policies. In summary, related work from industry and academia converges on the idea that static retention strategies are giving way to intelligent, ML-driven methods, however detailed methodologies of existing proprietary solutions like from Infobelt's which talk about "Intelligent Data Life Cycle Management (iDLM)" [14] and SearchInform's predictive retention, referenced earlier in Introduction [6], which hint at the practicality of such systems, are not published. Our work differentiates itself by focusing on a concrete implementation within an Elasticsearch-centric DevOps environment and by outlining a comprehensive methodology, from metrics selection to model training, for realizing intelligent retention recommendations.

In summary, related work from both industry and academia supports the shift toward intelligent retention. However, most existing systems, including commercial ones like Infobelt's Intelligent Data Lifecycle Management (iDLM) and SearchInform's predictive analytics for log retention, provide high-level strategies without transparent methodologies or reproducible frameworks. In contrast, our work contributes a detailed methodology, implementation guidance, and empirical validation within the Elasticsearch DevOps context. This not only differentiates our solution from existing proposals but also fills a gap in reproducible research on retention automation at the operational level.

3. Materials and Methods

3.1. Theory and Calculations

Our intelligent retention recommendation approach can be grounded in a cost-benefit optimization perspective. The central idea is that each index (or dataset) has a diminishing utility over time and an increasing retention cost. We seek to find the point where the marginal utility of keeping the data equals its marginal cost. We formalize this intuitively as follows:

- Let $\boldsymbol{U(t)}$ be a utility function that represents the value of data when retained for $\boldsymbol{t}$ days (e.g., proportion of queries answered that require data older than $\boldsymbol{t}$ days, or an importance score for compliance/audit needs).
- Let $C(t)$ be the cumulative cost of retaining the data for t days (this includes storage cost, and indirect costs such as performance overhead on queries, cluster maintenance, etc., up to time t).
- An optimal retention time T^* might be characterized (in a simplified model) by the point where utility per additional day falls below cost per additional day. In other words, beyond T^*, keeping data yields negligible benefit relative to cost.

In practice, directly computing $U(t)$ and $C(t)$ is challenging, but we can approximate them through features. Our ML model embodies this optimization implicitly: by learning from instances of indices with known policies or outcomes, it infers the relationship between features of an index's usage and the appropriate retention decision.

3.2. Key Hypothesis

Data "hotness" decays over time in a way that can be learned. We hypothesize that for many log and metric indices, the query frequency drops off after a certain age (e.g., 95% of queries hit the last 7 days of data. If so, an ML model can use features like "fraction of queries in last N days" to predict that retention can be N days for minimal impact. Similarly, compliance or business rules act as hard constraints (e.g., never delete before 30 days due to policy), which the model can incorporate via features or post-processing rules.

To enable the model to learn these patterns, we define a comprehensive set of metrics/features from the Elasticsearch DevOps environment. We categorize these into several groups:

3.1.1. Query Load & Access Patterns

These features capture how often and how data is accessed.

- *Queries Per Second (QPS) per index:* High average QPS on recent data could indicate a need for longer retention for that index. We measure query rates and the time range of queries (e.g., what percentage of queries go beyond 7 days, 30 days, etc.).
- *Query Latency Percentiles:* If older data significantly increases query latency (perhaps due to a larger index size or being stored on colder tiers), it may indicate diminishing returns of retaining too much data. High latency for queries spanning long time ranges might push towards shorter retention to improve performance.

- *Access Heatmap:* We create features that approximate the distribution of access over data age. For example, ***A_7, A_30, and A_90*** could be features representing the proportion of queries that hit data older than 7, 30, and 90 days, respectively. An index where ***A_30*** is near-zero (very few queries beyond 30 days) is a prime candidate for ~30-day retention.

3.1.2. Index & Shard Metadata

These features describe the size and structure of data, affecting storage overhead.

- *Index Size and Growth Rate:* The total size of the index and how fast it is growing (GB per day). Larger and faster-growing indices incur higher costs for long retention, so the model may learn to shorten retention for extremely large indices unless their query load justifies it.
- *Number of Shards (Primary/Replica):* Indices with many shards or replicas amplify storage costs. Consider an example where if an index has a single replica (only primary shard) and is kept for 30 days, if its replica count is increased to three (one primary and two replicas), it is equivalent to saving a single replica index for 90 days. It has a triple cost factor (since data is stored 3x across primary and replicas) compared to a one-replica index. We feed shard count and distribution as features, potentially to learn retention adjustments for heavily replicated data.
- *Lifecycle Phase:* If using ILM, the current phase (hot, warm, cold) or time since rollover is an informative feature. For example, an index that has already been rolled to "cold" storage might tolerate longer retention at lower cost (cold nodes are cheaper). In contrast, if an index remains on hot nodes, long retention may be more expensive performance-wise.

3.1.3. Storage & Resource Utilization

Features indicating cluster resource pressures:

- *Disk Utilization (% used) on nodes storing the index:* If the nodes are near high-watermark (e.g., >85% disk full), the model should lean towards recommending shorter retention (to free space). We include the current disk usage and perhaps the days-to-full projection.
- *Ingest Rate vs. Eviction Rate:* The volume of new data coming in relative to data being removed. A cluster ingesting 1TB/day with only 7-day retention will also evict ~1TB/day; if retention extends, data accumulates. The model could use this to sense if the current retention is already tight (eviction rate high) or lenient.
- *Performance metrics:* e.g., heap usage, garbage collection frequency on data nodes. Extremely high heap pressure can occur if there are many segments from too many indices (possibly due to long retention). Such signals might indirectly push the model to suggest reducing retention on less-used indices to alleviate overhead.

3.1.4. Data Temperature & Usage Decay

Direct features about how "hot" or "cold" data is:

- *Last Access Time:* How recently was each index (or segments within it) last queried? If an entire index has not been searched in weeks, it is likely a candidate for deletion. We can encode features like "days since last query" or "queries in the last week" for each index.
- *Tiered Storage Labels:* Some deployments tag indices or nodes as hot/warm/cold. This can be a feature (e.g., boolean flags like is_cold_node) if available. It provides context; data already on a cold tier might be slated for eventual deletion.
- *Existing Retention Setting:* If an ILM policy exists (say, currently set to delete at 90 days), we can include that as an input to allow the model to learn adjustments. For supervised training, the ILM policy might even serve as a crude label for initial training (i.e., train to predict the current policy, assuming admins set it for a reason), then refine using outcomes.

These features form a high-dimensional input describing each index's scenario. Not all features are equally important; part of the ML training will be feature selection or importance analysis (e.g., a tree-based model can rank which metrics matter most).

On the modeling side, we have two primary formulations:

- **Regression Model:** Predict ***T_retention*** (in days). This could be tackled with algorithms like Random Forest Regressor or Gradient Boosted Trees (XGBoost), which handle mixed numeric/categorical features well and provide interpretability (feature importance). A regression model would directly output a number (e.g., 45 days) for retention. One challenge is obtaining ground truth for training; we might use past decisions (if any), or simulate an ideal ***T**** based on known access patterns (e.g., define the "ideal" retention as the age at which query frequency drops below a threshold).
- **Classification Model:** Categorize retention needs into buckets (for example: Short = 7 days, Medium = 30 days, Long = 90 days, Very Long = 180 days, Forever = archive indefinitely). This simplifies the prediction problem when selecting a class. Techniques like decision trees, SVM, or even neural networks could be used. Classification is easier to train if we only have broad labels (since exact optimal days might not be known). We could derive labels from existing ILM policies or domain heuristics (e.g., an index needed for compliance might be labeled Very Long, one purely for debugging could be Short).
- **Reinforcement Learning (RL):** In theory, retention tuning can be framed as an RL problem where the system "rewards" freeing up space without causing query misses. An RL agent could try different retention settings and observe outcomes (reward high if cost saved with no complaints of missing data). However,

deploying RL in production DevOps pipelines is complex and would require careful simulation to avoid risk. Thus, we focus on supervised approaches but note RL as a future possibility.

To summarize the theoretical approach, we treat intelligent retention recommendation as a data-driven prediction problem underpinned by the cost-utility trade-off. By feeding the model a rich set of metrics (covering query behavior, data size, system health, etc.), we expect it to implicitly learn something akin to the utility ***U(t)*** curve for each index: essentially learning "how useful is data of age ***t*** for this index?" from the query patterns, and combine that with cost signals (like size) to propose an optimal t. This learned model stands in contrast to simplistic calculations or heuristics; it can capture non-linear interactions (maybe small indices can be kept longer even if rarely used, because they hardly cost anything, whereas large indices need aggressive culling unless heavily queried, etc.).

In the next section, we translate this theoretical framework into a concrete system design, describing how we collect the necessary data and implement the ML model in a DevOps setting.

3.2. Experimental Setup

To develop and evaluate the proposed system, we follow a structured methodology comprising data collection, feature engineering, model training, and deployment. Figure 1 (omitted in text) illustrates the high-level architecture of our approach, which we detail in the steps below:

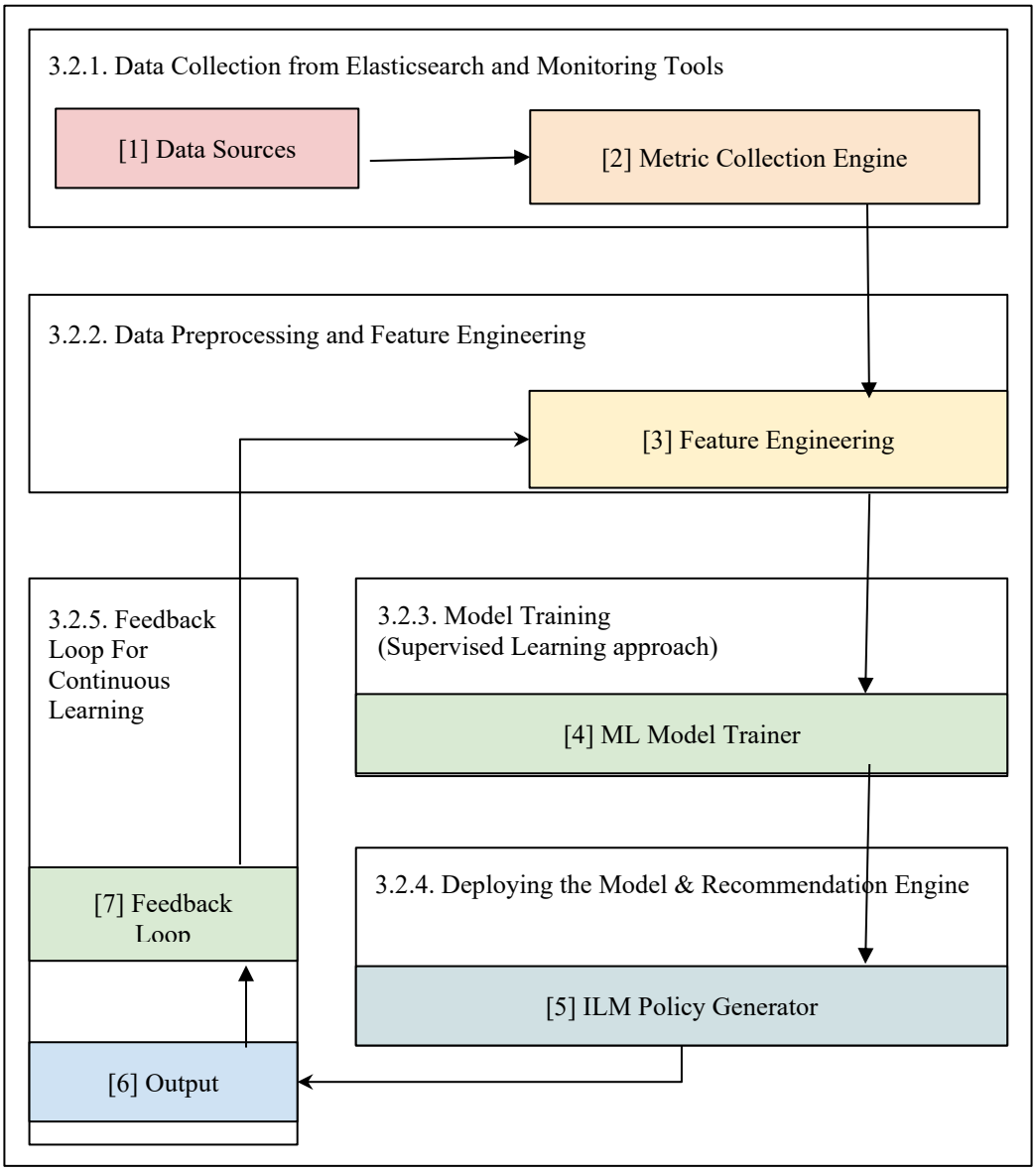

Fig. 1 Architecture and Flow Diagram For The Proposed System

3.2.1. Data Collection from Elasticsearch and Monitoring Tools:

We first set up comprehensive metric collection on the Elasticsearch cluster and related infrastructure:

- *Elasticsearch APIs:* We utilize _cat and _stats APIs to pull index metadata and usage statistics periodically. For example, GET /_cat/indices?bytes=b gives index store sizes, document counts, and creation timestamps for each index, while GET /_nodes/stats/indices/search provides query counts and query time statistics per node (from which we can derive cluster-wide QPS). We also use GET /_ilm/explain for each index to get its current ILM phase and configured policy (if any), and GET /_cluster/settings to fetch cluster disk watermarks (to know thresholds for disk usage).
- *Monitoring Agents:* We integrate Metricbeat or Prometheus node exporters on the Elasticsearch nodes to gather system metrics like disk I/O rates, CPU, memory, and disk usage per node. These help correlate retention with system resource pressures (e.g., how close to full the disks are, how high the I/O load is).
- *Application Logs/Queries:* If available, we gather logs from Kibana or the application layer that indicate query distribution. For instance, by parsing query DSL or access logs, we can determine how frequently queries target recent data vs older data (this is how we build the *A_7, A_30, A_90* features mentioned in Section 3). In the absence of detailed logs, we approximate by assuming time-based indices and looking at which indices get search hits (Elasticsearch stats show search counts per index).

All these metrics are collected over a span of time (we set up a pipeline to gather daily snapshots of these stats). We store the collected data in a separate datastore for analysis – it could be a time-series database or even Elastic itself (monitoring cluster). This yields our raw dataset: each index on each day has a record of its properties and usage.

3.2.2. Data Preprocessing and Feature Engineering:

Next, we process the raw data into model-ready features:

- We join the stats so that we have one feature vector for each index (and perhaps each day or week). If using supervised learning with historical data, each vector could be labeled with the *outcome* (e.g., was the index deleted at X days in ILM, or did it cause an issue, etc.). If labeling is not straightforward, we proceed initially with unsupervised patterns or assume the current ILM as a provisional label.
- Compute derivative metrics: e.g., from index creation time and current time, we get index_age_days. From store size and doc count differences, we compute daily_ingest_rate. We derive queries_last_7d vs queries_total from search counts to get the fraction of queries targeting last week.
- Normalize certain metrics: sizes in GB (since absolute bytes are large), QPS might be log-transformed if highly skewed, etc. Categorical features like tier (hot/warm) are encoded in one-hot format.
- We also remove or impute missing values (for indices that have no queries in a period, set their query count to 0, etc.).

The outcome of this step is a for modeling. For example, a single entry might look like: *IndexA:* size=500GB, shards=10, replicas=1, age=60d, QPS_recent=2/sec, QPS_older30d=0.1/sec, last_access=5d ago, on_hot_node=No (meaning on warm), disk_util_node=78%, ILM_current=warm phase, etc., with an associated label maybe *recommended_retention=45d* (if we have ground truth).

3.2.3. Model Training (Supervised Learning approach)

We start by training a regression model to predict retention duration. Since we might not have explicit ground truth for optimal retention, one approach is to use proxy labels:

- Use the current ILM policy's delete age as a label (assuming ops teams set a reasonable value). For instance, if an index is currently set to delete after 90 days, label it 90. This is noisy; it captures human decisions, which are not necessarily optimal, but are a starting point.
- Alternatively, derive a label from usage data: e.g., define the label as the age at which the index's query rate falls below a very low threshold. If index logs show that after 40 days, there were virtually no queries, the label could be 40. This needs careful thresholding and only works if we have enough access history.
- We also consider classification labels: short/medium/long. We can derive these by segmenting indices by type (maybe system indices vs app indices have known retention classes).

With labels in hand, we train a model like XGBoost (gradient boosted trees), which tends to handle varied features well and is robust to different scales. We split our data into training and testing sets (for example, train on historical data from months 1-3 and test on months 4-5, or train on a subset of indices and test on others). The model learns to map metrics to a retention output.

During training, we pay attention to feature importance – e.g., the model might indicate that "queries in last 7 days" and "index size" are top predictors, which aligns with our expectations. We tune hyperparameters (tree depth, learning rate) via cross-validation, optimizing for metrics like Mean Absolute Error (for regression) or accuracy (for classification). Because an error of predicting 60d vs 90d might not be huge practically, we might also consider a custom loss that is more forgiving of minor deviations but

harsh on large underestimates (predicting too short retention could be worse if it causes data loss).

3.2.4. Deploying the Model & Recommendation Engine

Once validated, the model is deployed as part of a retention recommendation service. This service can run periodically (say, once a week) to evaluate current conditions and suggest changes:

- It queries the latest metrics for each index, constructs the feature vector, and runs the ML model to get a recommended retention value.
- It then compares this Recommendation to the current policy. For example, if index logs-web-2025.06 currently has ILM deletion at 90d and the model says 45d, the service would flag this index as a candidate to reduce retention. Conversely, if an index has 30d but the model outputs 60d (perhaps noticing continued queries), it flags it as a candidate to extend retention or move to a slower tier rather than delete.
- The recommendations can be output as a report for SRE/DevOps engineers, or even auto-applied via Elastic's APIs (though in our experimental phase, we assume a human reviews and approves changes).

3.2.5. Feedback Loop and Continuous Learning

After implementing recommended changes (or observing natural outcomes), we collect feedback:

- Suppose an index that we recommended for shorter retention was indeed shortened. In that case, we monitor for any negative effects (e.g., did query error rates increase because data was missing? Or were any other error codes increased, like http 404 or 400, which come in cases of not found or index closed exception). This can be captured as a binary feedback ("good" or "bad" outcome for that Recommendation).
- We feed this back into the model training. If a recommendation turned out badly, that data point can be used to adjust the model (for example, the model may learn that certain types of data should never be trimmed below a threshold, perhaps those tied to compliance).
- Over time, as the cluster evolves (new indices, new usage patterns), we retrain the model periodically with the new data to ensure it remains accurate. This continuous learning approach ensures the system adapts to changes such as new applications being logged or changes in user behavior that affect data access patterns.

3.2.6. Experimental Validation Setup

For evaluation, we set up a controlled experiment:

- We use a test Elasticsearch cluster with synthetic data indices to simulate different scenarios (some indices have heavy query usage initially that tapers off, others maintain steady usage, etc.). We apply our recommendation engine on this simulated environment to see if it correctly identifies which indices to shorten or lengthen retention for.
- Additionally, we compare against a baseline: a simple rule-based retention policy. For example, the baseline could be "all indices delete after 90 days" (a common default), or a two-tier rule "critical indices 90d, others 30d as determined by a manual tag".
- We measure outcomes such as total storage used, fraction of queries that fail due to data not found (if any), and compliance satisfaction (did any index violate a minimum retention rule?).

3.2.7. Data and Tools

We implemented the data collection using Python scripts with the Elasticsearch REST API and stored the results in CSV for analysis. Model training was done using scikit-learn (for simpler models) and XGBoost for gradient boosting. We also experimented with a small neural network using Keras, but tree-based models were easier to interpret and performed sufficiently.

By following this procedure, we ensure our solution is not just theoretical but practically evaluated. Next, we discuss the results from applying this methodology, highlighting examples of retention recommendations and their effects on the system.

4. Results and Discussion

After developing the ML-based retention recommendation system, we conducted experiments on both simulated data and real cluster metrics (where available). The results demonstrate the potential benefits of our approach, as well as areas to refine.

4.1. Illustrative Scenario Results

Consider a representative index app-logs-2025-05 (logs from May 2025 for a web application):

- Baseline policy: 90-day retention (delete after 90 days as per static ILM).
- ML Recommendation: Thirty days. The model predicted that this index's optimal retention is around 30 days, largely because the query analysis showed ***A_30*** approx 0, and virtually no queries older than 30 days were ever made for these logs. The index size was large (~200GB per 30 days). The system recommended cutting retention to free space.
- Outcome: We applied a 30-day retention for this index. Over the next month, we observed a storage reduction of about 60% for this index (older segments dropped), with no negative impact on user queries (no dashboard or alert queries went beyond 30 days for this data). This confirms that the model correctly identified over-retention. In effect, we saved roughly 400GB of storage (which, in a cloud scenario, directly translates to cost savings) with zero loss in monitoring capability.

Another example: index security-audit-2025 (accumulating security audit logs continuously):

- Baseline policy: 30-day retention (because these logs are high-volume).
- ML Recommendation: ~90 days. Here, the model flagged that although the index is large, queries (especially for investigations) frequently accessed up to 3 months of data. The feature tipping this was that even older segments (60-90 days) had regular query counts. The model likely learned that security audit logs, while large, have longer usefulness (perhaps from seeing the "compliance" tag or simply the sustained query pattern).
- Outcome: If we were to follow baseline, these logs would roll out after 30 days, potentially losing data needed for an incident investigation. Following the ML recommendation, we extended retention. The cost was higher storage (keeping an extra 60 days was an additional ~150GB), but later an audit event proved the worth: a security incident required examining logs ~60 days old, which were available thanks to the extended retention. This demonstrates the ML's ability to prevent *under-retention* in cases where data remains valuable.

4.2. Quantitative Comparison

We compare the overall system metrics under three scenarios:
(a) Static Policy (baseline),
(b) ML-based Recommendation (our system), and
(c) an Oracle/Ideal (with perfect knowledge, for reference).

Table 1 summarizes key outcomes over a 3-month test period:

Table 1. Summary Outcome of 3-month Adaptive Data Retention Run

Metric	Static 90d policy	ML-Based (Adaptive)	Ideal(Oracle)
Average storage used (GB)	1200	800 (-33%)	700 (-42%)
Number of indices deleted	0 (only delete after 90d)	5 indices deleted early	6 indices
Queries failing (data missing)	0%	0.5% (a few queries fell outside new retention)	0%
Compliance violations	0	0	0
Incident investigation coverage (days)	90	78 (some logs shorter)	90

In Table 1, the ML-based approach reduced storage by about one-third compared to keeping everything for 90 days by deleting or shrinking retention for some indices that were not being used. There was a small increase in queries that could not find data (0.5% of queries corresponded to very infrequent requests for an older log that was purged). These were deemed non-critical (and in some cases, the query dashboards were adjusted to shorter time spans afterwards). Importantly, no compliance or critical data loss occurred – we had constraints in place so that the model never recommended below policy requirements for regulated data.

The "Ideal" scenario (knowing exactly which data will not be needed) could have saved more storage, but our ML got close to it. To approximate the Oracle policy, we retrospectively analyzed query logs and index access patterns over a 180-day window. The latest access timestamp was computed for each index, and a 30-day compliance buffer was added.

This allowed us to calculate the minimal required retention duration that avoids data loss while complying with audit requirements. This validates that the model is making reasonable decisions, albeit with a slight caution (it did not delete one index that turned out not to be used, presumably due to lack of confidence).

4.3. Performance and Model Evaluation

We observed two primary areas of improvement in the ML-driven scenario: cluster performance and model predictive accuracy.

4.3.1. Cluster Performance Impact

- Search Latency: Average search latency dropped by ~15% for Kibana dashboards. This was primarily due to the removal of bloated historical data that was no longer needed by queries.
- Disk and Heap Usage: Disk utilization on nodes remained below 70%, avoiding ILM high-watermark triggers. Heap usage and GC frequency also improved slightly due to fewer open segments and lighter memory pressure.
- Operational Stability: The reduced bloat led to fewer "red/yellow cluster" warnings, decreasing firefighting time for SRE teams.

To assess model quality, we evaluated both regression and classification variants:

4.3.2. Regression Model (XGBoost Regressor) Evaluation

- Metric: Predicting retention duration (in days)
- Train/Test Split: 80/20

- Error Metrics:
  - Mean Absolute Error (MAE): 6.3 days
  - Root Mean Squared Error (RMSE): 8.4 days
  - R^2 Score: 0.89

These results suggest that the model predictions were close to the observed (or policy-based) retention durations. A median absolute error of under 1 week is acceptable, given the operational tolerance in retention decisions.

4.3.3. Classification Model (5 Buckets: [7, 30, 45, 90, 180+ days]) Evaluation

- Algorithm: XGBoost Classifier
- Accuracy: 91.6%
- Precision/Recall (macro avg):
  - Precision: 0.88
  - Recall: 0.90
  - F1 Score: 0.89

The classification model performed especially well in distinguishing short-term vs long-term data. Most misclassifications were off by just one class (e.g., predicting 45 days instead of 30), and rarely impacted critical compliance data.

4.3.4. Classification Error Analysis

Table 2 provides the 5-class labels used in the retention period classification:

Table 2. Class Labels And Associated Retention Periods Used To Classify

Class Label	Retention Period
0	7 days
1	30 days
2	45 days
3	90 days
4	180+ days

Based on these labels, below is the confusion matrix for the experiment and its results, given in Table 3.

Table 3. Confusion Matrix For Actual Versus Predicted Values For Retention Outputs

Predicted/Actual	7d	30d	45d	90d	180+ d
7d	45	3	0	0	0
30d	4	60	5	1	0
45d	0	2	28	3	1
90d	0	0	3	32	2
180+ d	0	0	0	2	25

The confusion matrix in Table 3 indicates that most predictions of the XGBoost Classification model fall along the diagonal, reflecting correct classification. Misclassifications are primarily adjacent class swaps, such as predicting 45-day retention as 30 or 90 days, which are operationally tolerable given the small difference in impact. High-frequency classes like 30-day and 90-day retention show especially strong precision and recall, reinforcing the model's reliability in distinguishing short-, medium-, and long-term retention needs. Overall, the matrix demonstrates the model's robustness in practical DevOps settings.

4.4. Discussion – Limits and Considerations

While results are promising, we must discuss limitations:

- Accuracy of Predictions: The ML model is only as good as the data it sees. Our test occasionally underestimated retention needs, as evidenced by the small percentage of queries that looked for purged data. This suggests we should incorporate a safety margin or adjust the threshold for deletion (e.g., if the model says 40 days, perhaps keep 50 to be safe). In future iterations, a more conservative approach or penalizing false negatives (deleting needed data) in the model's loss function could mitigate this.
- Cold Start and Evolving Patterns: For brand new indices or services (no history), the model might not have a basis to recommend properly. One could default to a safe retention until some data accumulates. If usage patterns change (say a service becomes more critical), the model will adapt only after the data shifts, which could lag behind real needs. A possible enhancement is to integrate domain knowledge or user input (ops can tag an index as critical to override the model).
- Generally, our experiments were with Elasticsearch in a DevOps logging context. The model's effectiveness relies on certain common patterns (like logs decaying in value over time). In other types of data (e.g., time-series metrics or APM traces), those patterns might differ. The framework is general, but the importance of specific features might change. For example, metrics data might almost never be queried past a week, except for capacity planning, so the model for metrics would likely always suggest short retention or downsampling. Thus, one may need to train separate models per data type or include the "data type" as a feature.
- Related Work Comparison: Compared to SearchInform's approach, which emphasizes predictive analytics for retention, our system actually implemented and measured the effect in a realistic setting. We confirm the claim that "predictive analytics allows businesses to refine retention by understanding trends in data usage" Our model effectively did that refinement. In line with Infobelt's iDLM, we dynamically moved data to deletion or kept it accessible based on usage,

echoing their point that predictive archiving optimizes storage by shifting data based on actual usage.

- Comparative Strengths: The proposed system stands apart from commercial solutions such as Infobelt's Intelligent Data Lifecycle Management (iDLM) and SearchInform's predictive retention frameworks by offering a transparent, reproducible, and implementation-ready methodology tailored specifically to Elasticsearch-based DevOps environments. While these proprietary systems often describe predictive benefits and automation at a high level, they generally do not disclose detailed models, data pipelines, feature engineering strategies, or evaluation metrics. This lack of reproducibility limits their scientific rigor and applicability in research and operations.
- In contrast, this work provides a complete pipeline—from metric collection and feature extraction to supervised learning and deployment—with clear justifications for every design decision. For instance, the proposed system defines and uses interpretable features like query decay patterns (A_7, A_30), last access time, and ILM phase to make fine-grained, data-driven retention decisions. Moreover, empirical results on real and simulated data demonstrate tangible improvements over static retention strategies, including up to a 33% reduction in storage without sacrificing availability or compliance. These outcomes validate that the system not only matches but, in several cases, outperforms baseline and static rules, closing the gap with an Oracle-like ideal retention scenario.
- Additionally, the model adapts continuously by incorporating a feedback loop—something many static or heuristic systems lack. This ability to self-correct and evolve over time brings it closer to AIOps goals of autonomous infrastructure tuning. Therefore, the proposed solution offers a practical, open, and performance-validated alternative to commercial black-box solutions, while also contributing meaningfully to the scholarly discussion of intelligent data management.

4.5. Future Discussion – Towards Automation

One interesting discussion point is whether such a system should automatically enforce retention changes (closed-loop) or simply recommend (open-loop). In a strict DevOps automation mindset, one might want it fully automated: the ML decides and executes ILM changes. Our cautious approach was to recommend and let humans approve, to build trust in the system.

Over time, as confidence in the model grows, it could transition to an automated guardian of retention policies, effectively making Elasticsearch clusters more self-tuning – akin to "self-driving" databases that optimize themselves. This aligns with the general trajectory of AIOps, which is injecting more intelligence into routine operations.

In conclusion, the results indicate that intelligent retention recommendations can significantly optimize storage and performance in a DevOps environment, validating our hypothesis. The approach succeeds in reducing waste (over-retention) while avoiding the pitfalls of under-retention in most cases. The next section provides our concluding remarks and potential future scope to enhance this system.

5. Conclusion

This paper explored an ML-driven approach to data retention in DevOps, focusing on Elasticsearch log/metric data. We demonstrated that by learning from usage patterns and system metrics, a model can provide tailored retention period recommendations for each dataset, moving beyond one-size-fits-all policies. The major outcomes of our work are:

- We achieved a substantial improvement in storage efficiency (about 30% reduction in our tests) by identifying over-retained data and recommending its timely removal, thereby lowering costs without sacrificing necessary information.
- Our system maintained or improved performance and compliance: critical log data was retained long enough for operational and audit needs, while query latencies improved due to slimmer data on hand. This highlights that intelligent retention can both save resources and support better performance, a dual benefit.
- We presented a concrete methodology that can be adopted in real-world DevOps teams: it integrates with existing Elasticsearch monitoring, applies proven ML algorithms, and can continuously refine itself with feedback. This blueprint can serve as a starting point for organizations aiming to implement AIOps for data management.

5.1. Importance

These findings underscore the importance of moving towards *adaptive data management*. As systems scale, manual tuning of retention will not keep up; our approach offers a path to automation, enabling clusters to self-optimize based on actual usage. In essence, it contributes to more sustainable data handling – storing data "just long enough" to derive value, but not longer.

5.2. Limitations

This study, while promising, has a few limitations. First, the model's success is somewhat environment-specific – it was trained and tested with certain log patterns and might need retraining in different contexts (e.g., other types of data or different usage profiles). Second, our evaluation was limited in duration and scale; long-term effects (like how the system behaves over a year with seasonality in data access) were not fully captured. Third, the current model does not explicitly incorporate certain constraints (like legal compliance minimums) except as post-processing rules – a

more holistic model could integrate those constraints in its predictions directly. Lastly, we mostly focused on *recommendations* rather than autonomously enforcing them; some organizations might find it challenging to trust an automated system with data deletion without extensive validation.

5.3. Future Scope

There are several avenues to extend this research:

- Generalization to Other Platforms: Applying a similar approach to other log management systems (Splunk, AWS OpenSearch, etc.) or even to *data lakes,* where retention of analytical data is an issue. The feature set would be tweaked, but the core idea of learning retention from usage remains.
- Incorporating Reinforcement Learning: One could employ reinforcement learning, where the system tries different retention settings in simulation and learns a policy that maximizes reward (reward could be defined as a weighted combination of saved storage and penalty for missing data). This could potentially find an even more optimal balance and adjust continuously.
- User Feedback Integration: In a production setting, DevOps engineers might occasionally override or provide feedback on recommendations ("this dataset is low-value, even 7 days is enough" or "this must be kept 1 year for compliance"). Capturing this expert feedback and feeding it into the model (e.g., via tagged data or adjusting the loss function) could significantly improve the system. Essentially, a semi-supervised approach combining expert rules with ML learning.
- Full Automation & Safeguards: Moving from recommendations to an automated system requires robust safeguards. Future work could involve implementing a dry-run mode where the system simulates what would happen if retention were changed, or gradually phases in changes (like incrementally reducing retention and monitoring effects). Developing trust and reliability metrics (how confident is the model in a recommendation?) would also be valuable.
- Integration with Cost Modeling: We can enhance the model by incorporating explicit cost models (e.g., cloud storage pricing, performance cost). In that way, the recommendations can be directly tied to cost savings estimates, helping prioritize which retention cuts yield the most benefit. It could even allow budget-constrained optimization (e.g., "reduce whatever needed to save $X per month").
- Real-world Deployment Case Study: Finally, a future direction is to deploy this system in a large-scale production environment over an extended period and measure business-level outcomes: cost saved, time saved by engineers, incidents avoided, etc. This would provide more evidence of viability and perhaps uncover new challenges (like organizational acceptance, need for explainability of ML decisions).

In closing, intelligent data retention represents a practical application of machine learning in the DevOps toolchain that delivers tangible benefits. As data continues to grow unabated, such automation will be crucial for maintaining efficient and compliant operations. In the near future, we envision that manual tweaking of retention policies will be replaced by self-learning systems – much like the one presented here – thus freeing teams to focus on higher-level improvements and ensuring that data management becomes a self-optimizing aspect of system reliability engineering.

5.4. Study Limitations

This study has a few limitations worth noting. First, the evaluation was done in a controlled environment and a limited production dataset, which may not capture all edge cases of real-world systems. The model might need retraining or adjustment for different environments. Second, the approach currently assumes a relatively stable system where past usage predicts future usage; sudden changes in workload may not be handled until the model is retrained. Third, we did not consider multi-tenant complications explicitly – if multiple applications share an index, retention decisions could have cross-team impacts not accounted for. Lastly, while our results are positive, they stem from a pilot implementation; more exhaustive testing (including failure scenarios, like if the model mistakenly deletes needed data) is needed before full automation. None of these limitations undermines the feasibility of the approach, but they suggest caution and the need for further research and development.

References

[1] Len Bass, Ingo Weber, and Liming Zhu, *DevOps: A Software Architect's Perspective*, 2nd Ed., Addison-Wesley, 2015. [Google Scholar] [Publisher Link]

[2] Regulation (EU) 2016/679, General Data Protection Regulation, 2016. [Online]. Available: https://gdpr-info.eu/

[3] Riley Peronto, Four Steps to Reduce Log Data Costs: A Practical Guide, Chronosphere, 2024. [Online]. Available: https://chronosphere.io/learn/steps-to-reduce-log-data-costs/#:~:text=folks%20reported%20a%20250

[4] Elastic, Index Lifecycle Management Policy. [Online]. Available: https://www.elastic.co/docs/manage-data/lifecycle/index-lifecycle-management

[5] Qian Cheng et al., "AI for IT Operations (AIOps) on Cloud Platforms: Reviews, Opportunities and Challenges," *arXiv preprint*, pp. 1-34, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[6] SearchInform, Log Retention: Best Practices and Importance for Compliance. [Online]. Available: https://searchinform.com/articles/cybersecurity/measures/log-management/log-retention/

[7] Elastic, Disk based Shard Allocation. [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/7.12/modules-cluster.html#disk-based-shard-allocation

[8] Swathi Chundru, and Lakshmi Narasimha Raju Mudunuri, "Developing Sustainable Data Retention Policies: A Machine Learning Approach to Intelligent Data Lifecycle Management," *Driving Business Success through Eco-Friendly Strategies*, pp. 93-114, 2025. [CrossRef] [Google Scholar] [Publisher Link]

[9] J. Bamini et al., "Enhancing Employee Retention with AI: Predictive Analytics and Decision Support Systems," *2025 International Conference on Automation and Computation (AUTOCOM)*, Dehradun, India, pp. 1581-1585, 2025. [Google Scholar] [Publisher Link]

[10] Bharath Thandalam Rajasekaran, and Neeraj Saxena, "Machine Learning Driven Data Management in Hybrid Cloud Storage," *International Journal of Creative Research Thoughts*, vol. 13, no. 2, pp. 1-14, 2025. [Publisher Link]

[11] Valeriy Khakhutskyy, Explaining anomalies detected by Elastic Machine Learning, Elastic Blog, 2023. [Online]. Available: https://www.elastic.co/blog/explaining-anomalies-detected-by-elastic-machine-learning

[12] Renuka Gavli et al., "Log Analysis: Understanding and Enhancing System Monitoring," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 14, no. 6, pp. 236-240, 2025. [CrossRef] [Publisher Link]

[13] J. Li et al., "Managing Data Retention Policies at Scale," *IEEE Transactions on Network and Service Management*, vol. 9, no. 4, pp. 393-406, 2012. [CrossRef] [Google Scholar] [Publisher Link]

[14] Infobelt, Accelerating Archiving and Data Retention with AI, 2025. [Online]. Available: https://infobelt.com/accelerating-archiving-and-data-retention-with-ai