Serverless Elasticsearch: the Architecture Transformation from Stateful to Stateless

Iraklis Psaroudakis
Elastic
Greece
iraklis.psaroudakis@elastic.co

Pooya Salehi
Elastic
Germany
pooya.salehi@elastic.co

Jason Bryan
Elastic
USA
jason.bryan@elastic.co

Francisco Fernández Castaño
Elastic
Netherlands
francisco.fernandezcastano@elastic.co

Brendan Cully
Elastic
Canada
brendan.cully@elastic.co

Ankita Kumar
Elastic
USA
ankita.kumar@elastic.co

Henning Andersen
Elastic
Denmark
henning.andersen@elastic.co

Thomas Repantis
Elastic
USA
thomas.repantis@elastic.co

Abstract

Elasticsearch (ES) is a distributed search and analytics engine consisting of a cluster of nodes, each hosting a disjoint subset of data. ES has a shared-nothing architecture that relies on local disks to store cluster data such as index files, transaction logs, and cluster state metadata. This stateful architecture couples compute with storage, and leads to different data tiers (e.g., hot, warm, cold, frozen) of hardware and configurations that the administrator chooses from to balance cost, performance, and high availability. In this paper, we show a new serverless architecture that decouples compute from storage. Serverless ES offloads data to an affordable, highly available cloud object store, while supporting the same APIs and read-after-write semantics. We show why and how this stateless architecture simplifies the tiers to just two: indexing and search, allowing indexing and searching practically limitless data while scaling each tier independently. We describe how we wrap index data in a custom batch commit format to the object store to decrease upload costs by up to 100x, how we batch transaction log uploads to decrease upload costs by up to 30x, and how we delete files from the object store. We experimentally show that Serverless ES can get twice the indexing throughput of (stateful) ES on comparable compute hardware by using object storage for durability instead of replication, and can scale linearly to match ingestion workloads.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; *Dependable and fault-tolerant systems and networks*; • **Information systems** → Cloud based storage.

Keywords

Elasticsearch, Serverless, Stateless, Search Engines, Autoscaling

ACM Reference Format:
Iraklis Psaroudakis, Pooya Salehi, Jason Bryan, Francisco Fernández Castaño, Brendan Cully, Ankita Kumar, Henning Andersen, and Thomas Repantis. 2025. Serverless Elasticsearch: the Architecture Transformation from Stateful to Stateless. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772245

1 Introduction

In this age of big data and distributed computing, software is transitioning from on-premises to public cloud IaaS (Infrastructure as a

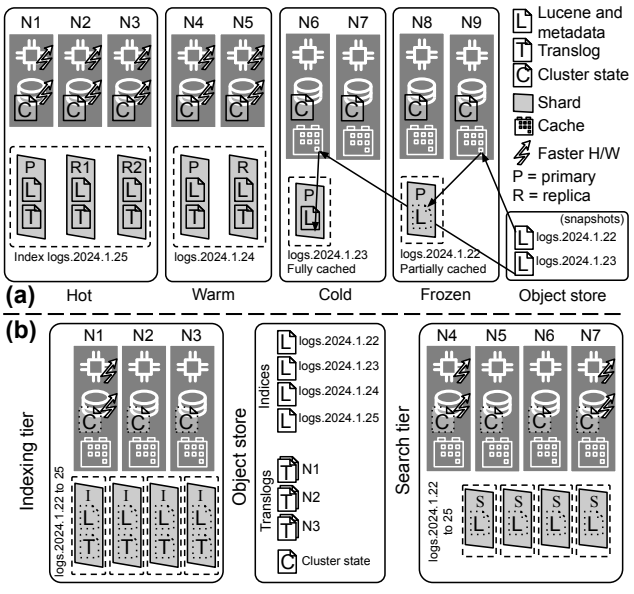

Figure 1: (a) ES architecture with data tiers. (b) New Serverless ES architecture separating storage from compute.

SoCC '25, Online, USA

Service) such as Amazon Web Services (AWS) [2], Google Cloud Platform (GCP) [23], or Microsoft Azure [34], also called Cloud Service Providers (CSPs). Beyond simplifying administration and improving operational flexibility, cloud-based software promises effortless scalability, high availability (H/A), and a competitive pay-as-you-go model based on demand. Elastic Cloud enables running Elasticsearch (ES) [16], a popular distributed search and analytics engine [44], as a service on the major cloud vendors. It automates the orchestration of an ES cluster and removes the operational burden of its maintenance and upgrade.

The existing ES architecture was not able to fully utilize the advantages of modern public cloud services due to its stateful architecture, which relies on storing data on local disks and scaling to a distributed cluster by exchanging data and information across the nodes through the network. This model has several disadvantages:

- Users need to provision a suitable cluster for their desired performance, considering dataset and workload size, and hardware (H/W) characteristics (RAM, CPU, disk, network).
- Provisioning CPU and RAM is coupled to provisioning storage, which can lead to underutilized resources.
- CPUs are shared between ingestion and searching, which can lead to overprovisioning, and hurt low-latency searches.
- Users need to consider using different data tiers (e.g., hot, warm, cold, and frozen) of H/W and configurations to balance cost, performance and high availability (H/A).
- H/A requires maintaining active replica shards of the data across the ES cluster, and/or separate snapshots to an object store for backups, consuming more storage and CPU.
- Upgrades and maintenance can be slow and expensive. Data arriving during a node upgrade will need to be replayed to re-establish a copy of the affected shards. An infrequent upgrade cadence can miss out on features and bug fixes.
- Relocating shards, including when scaling up/down, is slow and expensive as it requires physically moving data around.

To overcome these burdens, we designed the new Serverless Elasticsearch [20] on the premise of stateless, a.k.a. storage-disaggregated databases [32, 39]. This is driven by modern demands for SaaS (Software as a Service) that decouples compute from storage to exploit modern distributed cloud services, and achieve resilience and quick scalability over pools of commodity resources [9, 45]. Serverless ES offers the same API support and read-after-write semantics as ES, but without the hassle of worrying about cluster specifics, such as H/W, upgrading, H/A, or data tiers. Customers can use a pay-as-you-go model for data retention and indexing, searching practically limitless data [43]. This is all enabled by offloading data to an affordable, highly available object store [21].

The new design of Serverless ES is shown in Figure 1. Serverless ES is made stateless by offloading the index data, the transaction log (translog) and the cluster state (see §2.2) from the disk to the object store. We simplify data tiers to just two: indexing and search. This is intuitive, as the tiers correlate to the users' workloads of ingestion and searching and can scale independently to accommodate them.

Contributions. We hope the lessons shown are of value to other serverless engineering projects. We show how Serverless ES:

- Offloads index and translog data to an object store, which is the single source of truth for data (dispensing with replicas), provides durability in the face of node failure, and is the synchronization point between the indexing and search tiers.
- Uses "thin" (i.e., stateless) shards, and can recover/relocate effortlessly across the nodes without copying a lot of data. Disks are not used for data persistence (but, e.g., for caching).
- Reduces upload costs by up to 100x, by batching compound commits uploads. This approach increases inter-node traffic but maintains the same read-after-write semantics as ES.
- Reduces upload costs by up to 30x, by batching translog uploads. This approach slightly increases indexing latency, but does not impact the overall workload indexing throughput.
- Deletes unused offloaded data from the object store to decrease object storage footprint and data retention costs.
- Scales automatically in response to ingestion or search workloads. Clients call APIs without worrying about resources.
- Can experimentally surpass the performance of ES by up to 2x, and is able to scale linearly to match ingestion workloads.

2 Background

Elasticsearch (ES) is a distributed, RESTful, JSON-based, highly available, resilient search and analytics engine. ES is built on a popular open source library, Apache Lucene [7]. Lucene offers many text search components, e.g., inverted indexing, document processing pipelines, query execution algorithms, compression, and scoring functions like BM25 [41]. Lucene, however, does not prescribe how to assemble these components into a distributed search application, or offer a way to index and query the data via convenient interfaces (e.g., a REST API) [11]. ES offers features such as a Query DSL, analytics, timeseries, geospatial queries, scripting, vector search (supporting AI/ML), searchable snapshots, etc. Next, we review the architectural aspects of ES pertinent to this paper.

2.1 Elasticsearch overview

ES is crucially a document database. Every document (doc) is stored in an *index*: an independent searchable doc collection with a schema (either defined by the user or inferred as documents are added). The index is a *logical* collection, which is physically made up of a set of *shards* (see §2.2 for details). Shards are stored on *nodes*, independent processes typically running on separate machines.

The ES service runs on a cluster of at least one node. A node can accept client requests (e.g., search requests or doc indexing operations like insert or delete) which it will forward to the appropriate node or handle itself. To determine how to handle a request, the node consults a cluster state table with metadata common to the cluster, such as node membership, the set of indices defined, and the current mapping from an index shard to the hosting node. Every node maintains a copy of this metadata, but updates to the cluster state are coordinated via an elected cluster leader (the *master node*) responsible for applying the updates and ensuring that every update is committed on a quorum of nodes before it is acknowledged.

To add (or update/delete) a doc to an index, a client sends a REST request describing the operation to an arbitrary node. The request includes the index to which the doc belongs, and the operation payload. E.g., an insert request will contain a JSON-formatted

document to be indexed. The node receiving the request (the *coordinating node*) determines the shard to which the doc belongs by hashing a document id (which it generates if not provided) to produce a shard id, and then it forwards the request to the node that is currently hosting the primary replica of that shard per the cluster state's shard routing table. The receiving node applies the operation to the local shard. If the index is configured with replication, then the primary replica forwards the request to its replicas. When the replicas have applied the request and acknowledged it, or if replication isn't enabled for the index, then the primary replica acknowledges the coordinating node's request with the result of the operation. The coordinating node then responds to the client.

Searches and bulk operations (e.g., index an array of documents) work similarly. But since the requests may span multiple shards, the coordinating node handles splitting the incoming request to subrequests per shard, fanning them out to the nodes handling them, and aggregating the results for the client.

2.2 Index, translog, and cluster state data

Here, we detail the main data that ES keeps on disk. In later sections we show how Serverless ES offloads them to the object store.

Index shard data. Every index in ES is partitioned into one or more disjoint document collections called *shards*. A shard comprises one *primary* read/write shard and potentially multiple read-only *replica* shards. Figure 2 visualizes an example. The shard's documents are stored by Lucene. Lucene writes docs into a sequence of immutable segments, which are fundamentally enriched inverted indices and columnar stores used to serve search requests of various types. Each segment is assigned a monotonically increasing *generation* number, and contains a range of docs. Each doc is assigned a sequence number (*seqno*). Newly ingested data (doc insertions, updates, and deletions) go first to the primary shard's Lucene's in-memory indexing buffers, and are also replicated to any replica shards before being acknowledged to the user. For the data to become searchable, the index needs to be "*refreshed*", producing the next segment in the chain. This is necessary because search queries are done based on the index structures available in segments. For the segments to become persisted (fsync'ed) on disk, the index needs to be "*flushed*", creating a persisted commit point on disk. Periodically, segments may be merged together in the background, into a new segment, to make searches more efficient and reclaim space retained by deleted docs. This merge process is similar to the one used by LSM-tree [38] based databases (e.g., Cassandra [29]).

Shards provide scalability, load balancing and H/A. At ingestion time, ES distributes docs across index shards which are balanced across nodes. At search time, ES splits queries over the necessary shards (and may choose replicas as well for load balancing), aggregates sub-results into the final combined result, and sends that back to the client. If a primary fails, a replica is promoted to become the primary. In such failure cases, a number called the "*primary term*" of the shards is increased. The combination of the primary term of the primary shard that coordinates a doc change, and the sequence number of the operation [18], allows for optimistic concurrency control [16] to ensure that an older version of a doc does not overwrite a newer version.

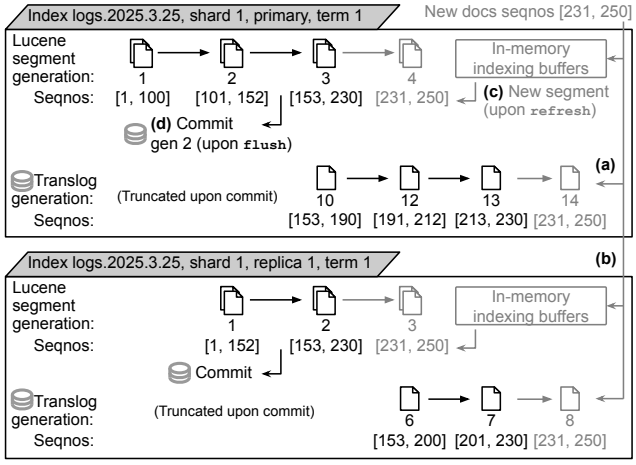

Figure 2: An index with a primary and replica shard. Docs are (a) added to the primary's Lucene and the translog, (b) forwarded to the replica, (c) refreshed to a new searchable segment, and (d) committed to disk upon flush.

The number of shards for an index in ES is determined at index creation time with an index setting. The number is fixed for the index's lifetime. However, in Serverless ES, we intend to start with a small default number of shards and allow their number to grow and shrink dynamically and transparently based on load. We already support "*autosharding*" [10] for data streams, and plan to support it for regular indices as well.

ES may relocate shards across nodes to achieve a better balance. The default balancing algorithm [13] computes the ideal balanced state for the cluster every round, based on node weights and stats like shard sizes, workload resources, etc., and then moves the shards to reach that state. When a shard is moved, all its data is transferred to the target node. This can be an expensive operation. In Serverless ES, we use the same balancing algorithm, but we strive to keep shards as thin as possible (see § 3.2) to make relocations fast.

Translog data. New data is also written to the translog. Its main purpose is to persist uncommitted operations, so they can be replayed in the event of ephemeral failures such as a crash or power loss. It comprises a sequence of files which are always persisted to disk before acknowledging the operations they contain. Once a shard flushes a Lucene commit to disk, which ES may automatically trigger [14], the corresponding applied operations are truncated from the translog. A secondary purpose of the translog is to facilitate real-time GETs of documents without refreshing [16].

Cluster state data. The cluster state is metadata that contains the most important information about the cluster, e.g., the nodes, indices, their mappings, the shards and their primary terms, the allocation of shards on nodes, etc. A *master* node is elected by the nodes to mutate the cluster state and coordinate its publication across nodes. ES uses its own consensus protocol [18], similar to Paxos [30], or Raft [36], to effect cluster state changes.

How Serverless ES offloads cluster state data to the object store is out of the scope of this paper. In summary, cluster state is stored

in term specific folders. A single lease file [6] is used to elect a new leader for a term. This uses CSP atomic compare and swap functionality to avoid scenarios such as split-brain.

2.3 Data streams and data tiers

A significant share of ES workloads is based on time-based data, like logs and metrics, where typically data is appended. ES can optimize for this by creating *data streams* [16] of indices, where read-only indices store older data, and the latest index receives new docs. ES provides the ability to create Index Lifecycle Management (ILM [16]) policies for data streams. E.g., the latest index can be automatically "rolled over" when the day changes. A data stream search may also become more efficient, e.g., if it only searches recent data, it can filter out older indices.

Since older data receive typically fewer queries and less ingestion, indices may become over-provisioned as they age. Data tiering mechanisms aim to help retention costs. Figure 1a shows an example of the "hot", "warm", "cold", and "frozen" ES data tiers, as the indices of a logs data stream automatically roll over and gradually move to a colder and cheaper data tier as they age. Colder tiers may contain less powerful H/W (e.g., cheaper CPU and spinning disks instead of SSDs), and cheaper index configurations (fewer replicas). The coldest tiers eliminate replicas by storing read-only *searchable snapshots* [16] of the data on a cheap cloud object store. The cold data tier fully caches searchable snapshots on disk, while the frozen tier is driven by the search workload to dynamically fetch and cache searchable snapshots parts on disk. The block-based *cache* acts as an indirection layer for the Lucene segments to the blob store. It caches 16 MiB file chunks. It uses a least frequently used (LFU) algorithm to evict less frequently used chunks.

Serverless ES takes away the complex performance and cost tuning of data tiers, by simplifying data tiers to just two: indexing and search (see §3). It also offers data stream lifecycle support [16], in lieu of ILM, to automate data streams management according to user-defined retention requirements, and offers autosharding [10] for data streams, which are left out of the scope of this paper.

3 Serverless architecture

Serverless ES is based on the principle that compute should be separated from storage, using highly durable and available CSP object stores as the primary storage of all data [32]. Figure 1b visualizes the stateless architecture, which persists all data in the object store. Nodes are divided into two main tiers: indexing and search, which respectively serve ingestion and search requests. All requests coming from the clients go through a proxy that routes them to the correct tier according to the request type. This architecture solves many disadvantages of stateful ES (see §1):

- An object store provides H/A when nodes are lost, without replication or snapshots. This avoids unnecessary CPU work and I/O for resilience.
- Ingestion and search workloads, which have very different resource profiles, are handled by independent tiers. We can scale the tiers independently and automatically based on the workloads (see §7). This allows serverless ES to make more efficient use of H/W in each tier.

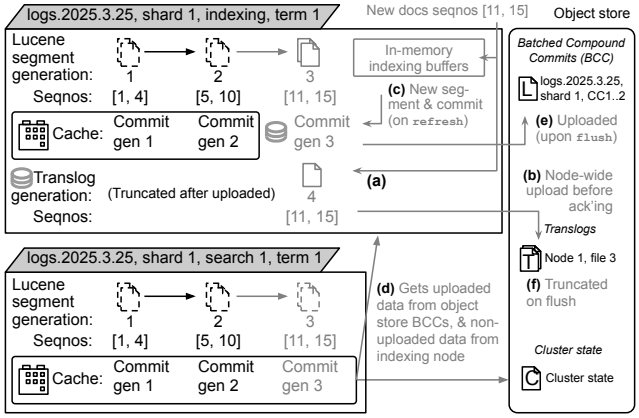

Figure 3: In Serverless ES, docs are (a) added to Lucene and the translog, (b) uploaded to the object store before ack'ing the client, (c) refreshed into a new committed segment, (d) searched via the network whilst not uploaded, (e) uploaded as a BCC upon flush, (f) truncating unnecessary translogs.

- No need for additional data tiers nor related ILM policies that transition data through tiers, with their tuning burden and dependence on accurately predicting future workloads.
- Adding and removing nodes is faster, as shards only need to load metadata until they are indexed or searched (see §3.2).

Figure 3 shows how ingestion and searches work. The indexing tier has indexing shards that handle ingestion, which write commit segment files to the object store regularly in a compound format (see §4). Each node writes a node-wide translog to the object store (see §5), and acknowledges ingestion requests after their corresponding translog operations have been written to the object store. This prevents data loss of acknowledged indexing requests when the corresponding commits were not uploaded prior to node loss, since operations can be replayed from the translog. Finally, the indexing tier is responsible for deleting any older translog operations and commits on the object store that are not used anymore (see §6).

The search tier searches data in the object store (see §4.1). Upon a refresh, the indexing shard creates a new segment and commits it, and, as an optimization, notifies the search shard about the new commit. Upon a flush, the indexing shard collects all not yet uploaded commits into a new blob on the object store, and as an optimization, notifies the search shard about the uploaded commits.

Both tiers use a partially cached model akin to ES's frozen tier, allowing them to index and search much more data than could be contained on the nodes themselves. Specifically, an indexing node services indexing requests (of which some parts may do some lookups), generates segments in local files first, which are then uploaded, and are deleted afterwards when they are unused. A search node services client searches by dynamically fetching commit data into the cache either from the indexing shard for recent commits that have not been uploaded yet, or from the object store. In general, the object store holds the source of truth for commit data.

The absence of data tiers does not inhibit handling data as it ages. As data ages, the percentage of the data cached will drop naturally. Shards do not have to move to benefit from lower cost as data ages.

This architecture enables us to alleviate a lot of administrative burdens for users. First, there is no need to manually tune cluster size, since Serverless ES automatically scales up and down resources as needed for the client workloads. The upgrade cadence is frequent and automatic, without interruption, improving code security. Clients simply interact with API endpoints. Behind these endpoints, elasticity of the clusters is managed automatically.

Serverless ES supports largely the same client APIs [16] as ES, but does not support APIs that are administrative or expose internal details (that are not needed as the cluster is automatically managed). E.g., the "cluster state API" [16] is not supported, as it exposes internal info about nodes, shards, and the system. API endpoints provide an encapsulation away from the ES version underneath.

3.1 The object store

Serverless ES is not tied to a single CSP. In order to shift primary responsibility for data durability and availability to the object store, the object store must provide some common properties, all of which are found on the object stores on which Serverless ES currently runs (Amazon S3, Google Cloud Storage, and Microsoft Azure Storage).

First, the object store should provide high durability and availability. While it is impossible to precisely quantify durability and availability on classic ES because users control the degree of replication of an index and the failure rate depends on the specific H/W the customer uses, all of the object stores offer much higher durability and H/A than would be provided in the typical configuration of two replicas for each shard. All CSP object stores offer at least 99.999999999% durability and 99.9% availability [3, 26, 35]. This degree of durability allows Serverless ES to consider data durable as soon as it is acknowledged by the object store, without additional redundancy. Although nodes both cache index data locally, the cache exists to provide performance and cost benefits, and not specifically to improve availability. If the object store is unavailable, indexing cannot proceed because it does not acknowledge indexing operations until their corresponding entries in the translog have been acknowledged by the object store. Searches may not be able to proceed if the corresponding commits are not present in the local cache. Failed object store operations are retried indefinitely to recover automatically when the object store becomes available again. Because indexing depends on the object store being available, an object store outage produces backpressure on indexing clients and bounds the amount of work that is required to recover.

To ensure consistency during node failure recovery, Serverless ES expects read-after-write consistency from the object store, so that other nodes can discover the latest transaction log entries and index commits as soon as they have been uploaded. The three supported CSPs all provide these properties [4, 8, 25].

3.2 Thin shards

In Serverless ES, shards are "thin" in that they do not require all commit files to be present on local disk. They can recover and relocate across the nodes of the cluster quickly and cheaply because they do not need to copy large amounts of data from other nodes. To reduce object store API requests for commit data, nodes maintain a block-based cache on local storage to cache blob chunks, largely similar to the one used by searchable snapshots (see §2.3). When Lucene needs to read from a file, the cache fetches the corresponding data chunk (usually 16 MiB) from the object store and stores it on disk for future reads.

Search shards only fetch the required parts of commit files either from the indexing node or from the object store into the local cache, and they never produce new Lucene files on disk. Indexing shards manage the lifecycle of data files from their local creation on disk to their upload to the object store and their deletion from disk. This increases disk space only temporarily, since no replication to other nodes (as in stateful ES) is needed. Durability is guaranteed by the object store.

Once uploaded, Lucene commit files can be deleted from the local disk. Only a small portion of the total shard data is necessary to be on disk to serve ingestion. Once ingestion stops, there is no need for any data on disk [31]. Half of the disk space is reserved for servicing ingestion on local Lucene commit files, and the other half is reserved for the cache (for reading uploaded commits).

Indexing documents may require rereading some uploaded files to look up a document id, the value of a field in a document, or to load the existing source to apply an update script [31]. In such cases, the indexing shard will fetch portions of the required files from the object store into the local blob cache.

For better performance, indexing shards also prepopulate the cache when uploading blobs. This way the data most likely to be needed for subsequent indexing is already available when the shard transitions from accessing local files to reading the corresponding uploaded blob.

Fast recoveries and relocations. Shard recovery is the process of bringing a shard into service on a node. It is executed when an empty shard is created for a new index, when a shard needs to be relocated from a node to another, e.g., for balancing shards [13], or when a shard needs to be recovered from the object store. In Serverless ES, recovery does not require fully copying all segment files between nodes as in ES [31]. Only the necessary commit blob chunks are fetched from the object store, using the blob cache, and the translog to replay if relevant, for an indexing shard. This typically requires only a fraction of the total shard size, and allows the shard to start fast. Adding and removing nodes is also fast, since relocating shards is fast. This enables us to do fast, frequent, and transparent rolling (node by node) upgrades automatically across the fleet without the user's participation or notice.

A search shard can be relocated from a source node to a target node almost instantaneously. That is because the target search shard can immediately access object store data; there is no need to move data between nodes or to wait for any outstanding searches on the source node; they can be completed even after the search shard relocates away from the source node. On the other hand, an indexing shard receives ingestion, and relocating it needs to handle the transition of the ingestion workload correctly. The source indexing shard briefly pauses ingestion, flushes the current data to segment files, and uploads them to the object store. The target indexing shard recovers from the last uploaded commit files, and then the source shard hands off ingestion to the target shard.

4 Offloading index data

To record the effects of indexing operations against a shard, Lucene writes segment files onto the local disks of indexing nodes. Periodically the shard generates a commit, representing the entire state of the shard at a point in time. In Serverless ES, the commit files (i.e., all the segment files in the commit) are wrapped into a *Compound Commit (CC)*, along with a header, as shown in Figure 4. The CC header contains metadata information, e.g., the commit's generation and primary term, the compound translog start file and node ephemeral ID for possible recovery (see §5), and a map of the commit's filenames to object store locations (i.e., the object store key, offset, and length). This map is necessary because a commit can refer to files from previous commits that have already been uploaded to the object store, and so the map is used to translate between Lucene's local commit file references and their corresponding locations within previously uploaded objects. After the header, the commit's new files are simply appended to the object, with a potential padding to round up to a CC's page-aligned boundary. Packing multiple files in the object helps reduce the cost incurred by CSPs' PUT requests, which are typically expensive. To amortize PUTs even further, we may concatenate multiple CCs into a single object, the *Batched Compound Commit (BCC)* [22]. Figure 4 shows two BCCs, one containing two CCs (generations 1 and 2), and one containing a single CC (generation 3). A BCC has no additional header. Batching compound commits is particularly valuable when a shard is refreshed frequently (see §4.1).

After a BCC is uploaded, the commit files can be deleted from the disk. They are accessible for reading through the blob cache (see §3.2), allowing the shard to occupy only a negligible footprint on the local disk beyond what is necessary to serve the active workload.

4.1 Read after write semantics

The architecture of Serverless ES, where we separate search and indexing workloads, fundamentally differs from the ES operation-based replication model (see §2), in which operations are applied synchronously across shard replicas. In ES, operations become available for searches either immediately for real-time gets or after a refresh (either through the API or through a recurring refresh interval, typically 1s) for searching across all in-sync shard copies.

In Serverless ES, we implement a segment-based replication model. Indexing nodes process incoming ingestion operations, create Lucene segments, and upload them to the object store. These segments become searchable when the search tier has downloaded the BCCs containing them from the object store into its local cache. However, to reduce latency between a refresh operation and the data becoming available to the search node, we also allow querying index nodes directly for BCCs that have not yet been uploaded.

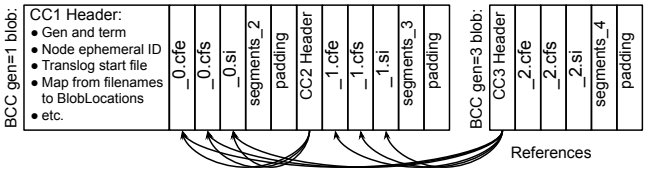

Figure 4: Composition of Batched Compound Commit blobs.

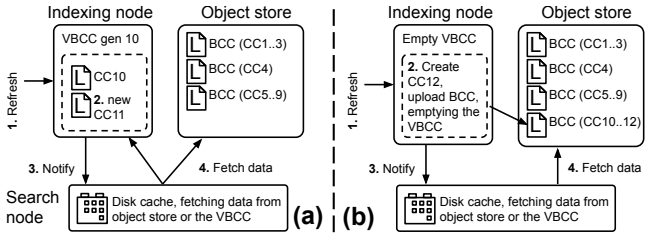

Figure 5: Refreshing a new commit into (a) an existing VBCC, and (b) into a VBCC that uploads a BCC to the object store.

The indexing nodes notify their corresponding search nodes of the latest refresh, and search nodes can pull the related segments from the indexing nodes if they are not yet available in the object store.

Virtual Batched Compound Commit. As illustrated in Figure 5a, following a refresh, a reference to the new CC is appended to a *"Virtual" Batched Compound Commit* (VBCC) which is kept only in the indexing shard's memory, and not yet uploaded to the object store [22]. The indexing node appends CCs to the VBCC at every index shard refresh. Upon a flush (which may also be initiated once the VBCC surpasses a total byte size, or a set number of CCs, or a set age), the indexing shard uploads the VBCC to the object store as a BCC, and the VBCC becomes empty again, for new CCs to be appended to it. This is illustrated in Figure 5b.

At every CC append, or when a BCC is uploaded, a lightweight coordination protocol is initiated between the primary shard's indexing node and the search shards' nodes. This protocol notifies search nodes about newly available segments and indicates whether the search nodes should query the indexing node directly or access segments from their object store locations. Figure 5a shows an example where the search node's cache may fetch data from the indexing node directly for files of CC generations 10 and 11, and from the object store for files of CC generations 1 through 9. Figure 5b shows that once the VBCC is uploaded, the search node will fetch data only from the object store.

Real-time GETs. ES's GET API allows fetching one or more documents by providing their IDs. While searches require a refresh to reflect newly indexed docs in the result, a request to the GET API, specifying the real-time flag, can fetch docs that are not yet refreshed. Similar to searches, the GET API [16] is serviced by search nodes. However, since the real-time version of the GET API can query docs from possibly not-yet-refreshed data, the search node may not have the most recent data to serve that request. To service these real-time GETs, the search shard reaches out to the indexing node to ask for the latest data for the queried docs. The indexing node may return the latest data if it is not yet exposed to the search node, e.g., from the translog, or may return an answer to the search node that it has up-to-date data to serve the GET request itself.

Recurring refresh. While maintaining ES's typical 1 second background refresh interval would provide the best user experience, frequent refreshes in Serverless ES have some disadvantages. First, they result in increased object store costs relating to uploading commits. Second, a refresh in Serverless ES does more than in ES,

as it includes creating a new Lucene commit that needs to be communicated to search nodes and ultimately get uploaded to the object store. For these reasons, the default and minimum recurring refresh interval in Serverless ES is currently set to 5 seconds.

Throttling for manual refreshes. Alongside the recurring refreshes, clients can manually call the refresh API. But a workload could manually refresh so frequently that they incur an inordinately expensive object store request rate for uploading commits. To protect against such pathological workloads, we use batching and throttling for manual refreshes of indices. We allow for a large budget of spontaneous refreshes within the last hour, currently set to 1 manual refresh per 5 seconds per 4GiB of the cluster's RAM on average within the last hour, and throttle manual refresh batches over the budget to a refresh every 5 seconds per index shard.

5 Offloading the translog

To preserve index operations in the case of failure, ES maintains a transaction log (translog) of operations that mutate an index such as document insertions or deletions (see §2.2). The translog is committed to storage before the operations it contains are acknowledged to the user, and its tail is truncated when the corresponding operations are known to be contained in committed Lucene files. In the event of an ephemeral failure such as power loss or a crashed process, the index can be recovered up to at least the last acknowledged operation by replaying the translog onto the last flushed commit.

In ES, the translog is written to local storage and does not provide protection from the loss of the hosting node. For that, the indexing shard transmits operations directly to each of the shard's replicas and waits for the replicas to acknowledge them before it in turn acknowledges the operation to the client (see Figure 6). Using replication couples durability, which is a storage property, to compute, in that the degree of durability is determined by the number of active shard replicas that acknowledge the update.

Serverless ES durability. In Serverless ES, we have decoupled durability from compute by uploading the translog to a CSP object store (e.g., S3), as shown in Figure 7. Uploading the translog to the object store instead of replicating to a set of running shards provides a number of benefits. For one, it simplifies the machinery required to provide durability, because the indexing shard does not need to coordinate with multiple replicas. It also scales easily, because shards can be initiated using the data held on the object store instead of retrieving that data from a node.

Hosting the translog on the object store also has some disadvantages. Foremost is that while the cost of *storage* in an object store is low, particularly for short-lived translog records, the cost of *requests* is not. A naive implementation would perform more than one object store operation for each logical operation in the translog: an initial put, potentially multiple gets, and an eventual delete after the logical operation is known to be in an uploaded BCC. This would be prohibitively expensive, so Serverless ES amortizes these operations by batching and combining the operations of every shard on the node into a node-wide compound translog file, named per the node's ephemeral ID (regenerated whenever a node joins the cluster) and a sequence number incremented with

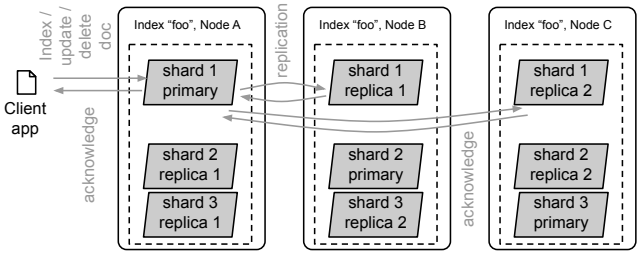

Figure 6: ES replication-based durability.

each upload. Operations are acknowledged to the client after the compound translog file that contains them has been uploaded.

Using only the translog for operation durability means we must take additional steps to ensure that a node cannot continue to acknowledge indexing operations if it has lost ownership of a shard (e.g., due to a network partition), since we cannot rely on a quorum of replicas [15]. We maintain a lease object on the object store, which the cluster leader updates when it has decided to eject a node and reassign its shards [6]. After a node has written a translog object, it reads the lease object to confirm that it is still a member of the cluster before acknowledging the indexing operation to the client. This lease check relies on the object store providing read-after-write consistency, which is guaranteed by the CSPs we use.

Translog recovery. Aggregating operations for all the shards on a node into a blob on the object store creates a discovery problem. During recovery, the new host of the shard has to determine which translog objects may contain operations that should be replayed. Translog objects are identified in the object store according to the node that produced them, but the assignment of shards to nodes is not fixed. Shards can be moved from node to node when nodes are added or removed from the cluster or even when the workload changes. To account for this, CC headers include a pointer to their associated translog, as illustrated in Figure 4. Shard recovery starts by finding the most recent BCC in the object store for the shard, and follows its translog pointer to find the files that may contain operations that must be reapplied on top of that commit to recover all operations that may have been acknowledged on the shard. Note that translog recovery only kicks in after an unexpected shutdown, to bring indexing shards up to date from their most recent uploaded commit. We would see slow translog recovery if there was an extended period of time where the object store was accepting translog uploads but not commit uploads, followed by an unclean shutdown of the index node. For context, our production telemetry shows only 2.5 seconds of translog recovery for P99 in a month, while P95 is 0.5 seconds.

Performance. Aggregating translog operations dramatically reduces object store request cost, but that comes with some tradeoffs: indexing latency goes up because indexing operations have to wait for an entire batch of operations to be written, and recovery of a shard can become more expensive. On a node hosting many busy shards, the latency penalty of batching is minimal, as shown by our experimental evaluation (see §8.2), because there are many simultaneous operations. However, we also bound latency by uploading,

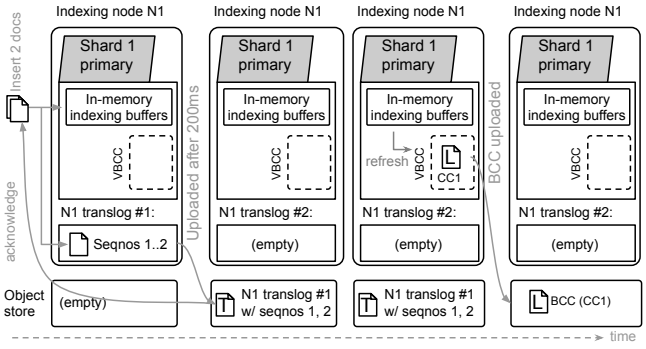

Figure 7: Serverless ES translog-based durability.

or "flushing", the compound translog file whenever any operation has been pending for more than a configurable upper bound. We also set an upper bound on the size of translog files to bound both upload time and recovery work in the case that a node translog object is dominated by operations of other shards. The flush interval is a simple tradeoff between cost and performance: latency goes down as requests per second go up. In our environment, we chose to flush the compound translog file every 200ms (5 CSP PUTs per second per node) but a range of values are reasonable. Maximum object size is a similar, if less straightforward, tradeoff. We set the upper bound on the translog file size to 16 MiB.

Example. Figure 7 shows a simple example of inserting two documents to an empty shard. Initially, the indexing operations go to the in-memory indexing buffers of Lucene, and are appended to the node's compound translog file with ordinal #1. After the flush interval, the translog file is uploaded to the object store in a path prefixed by the node's ephemeral node ID. A new empty translog file #2 on the node will receive any future indexing operations. Once the shard is refreshed, a CC is appended to the VBCC with the ingested documents. Once the BCC is uploaded (see §4.1) to the object store, then the indexing node knows that all the operations included in the first translog file have been uploaded as part of BCCs, and can safely delete translog file #1 from the object store.

6 Object store file deletions

In this section, we detail how we handle the deletions of blobs in the object store once we are sure they are safe to be deleted.

6.1 BCC garbage collection

We store Lucene files in BCC blobs in the object store, and CCs may reference files in other BCCs (see §4). This architecture complicates garbage collection, as relying on Lucene file deletion mechanisms is not enough to determine which blobs to delete. A subset of files in a blob might still be in use even when others are not.

To minimize shard operation coordination, we design the garbage collection to operate independently for each indexing shard. This independent approach needs to address several challenges:

- Search nodes may hold references to older commits to serve ongoing searches.
- Background processes such as backup snapshots may maintain additional references.
- Coordinator node involvement must be minimized to ensure efficient scalability.

Our solution implements reference counting at the indexing shard level, tracking how many active processes are using each blob. Each indexing shard maintains a record of which blobs remain in use and only deletes blobs when their reference count reaches zero. While this requires some coordination between indexing and search nodes, we leverage an existing protocol: As described in §4.1, we already have a mechanism to notify search nodes when new segments become available in the blob store. We enhanced this protocol to allow search nodes to inform indexing nodes about which Lucene commits they still have open, significantly reducing the communication overhead for reference tracking.

To prevent an isolated stale indexing node from deleting blobs that might be used by another node, we employ the same lease mechanism through the blob store described in §5. This lease mechanism requires nodes to verify their cluster membership status before performing destructive operations, ensuring that only nodes with valid leases can delete blob data.

6.2 Translog garbage collection

For translog blobs garbage collection, we employ reference counting similar to our BCC approach. Each indexing node is responsible for tracking and cleaning up its translog blobs once they are no longer needed. Each indexing shard maintains precise records of which translog files contain operations that have not yet been incorporated into a BCC in the blob store as described in §5.

Using reference counting, we track translog operations until they are successfully incorporated into uploaded BCCs. Once all operations from a translog file have been safely persisted in the blob store as part of BCCs, the translog blob's reference count drops to zero, marking it eligible for deletion. This mechanism ensures we maintain only the minimum necessary translog data while preserving operational integrity.

To prevent data loss during node failures or network partitions, we utilize the same lease mechanism described in §5. Before any translog blob deletion, the node verifies its current cluster membership through the lease in the blob store. This verification primarily prevents isolated nodes from performing destructive operations that could compromise data integrity, ensuring that only nodes with up-to-date cluster state can delete translog data.

7 Overview of autoscaling

Decoupling compute from storage facilitates replacement of nodes in the cluster. As the data is persisted on the object store rather than on disk only, relocating the shards to a new node does not require copying all of the data from the source node to the target node. This enables autoscaling in Serverless ES to adjust the cluster size much quicker than (stateful) ES. Serverless ES heavily relies on autoscaling to adjust the cluster size automatically to match the search and ingestion load while ensuring resources not required are scaled down in a timely manner [40, 42].

Autoscaling is driven by a set of metrics that are exposed by Serverless ES itself. The metrics used for the indexing and search

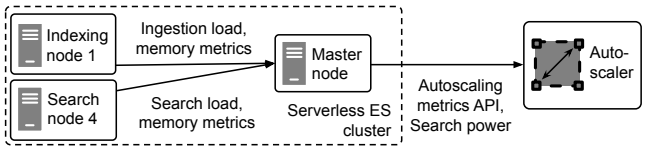

Figure 8: The Autoscaler monitors Serverless ES metrics and periodically adjusts the tiers' sizes if necessary.

tier aim to estimate the required CPU, memory and disk resources in the entire tier, and their minimum value on each node. Serverless ES periodically reports the resources required to handle the ingest and search load on the indexing and search tier, respectively. The Autoscaler, which is an external component to Serverless ES, monitors the reported metrics, which are gathered by the master node of the cluster, and scales each tier independently up and down if necessary, as shown in Figure 8. The Autoscaler scales each tier horizontally and/or vertically to provide both minimum available resources per node and a total amount of available resources per tier. The largest node size has 64GiB memory and 32 CPUs. Any further scale-up adds new 64GiB nodes, allowing a cluster to scale horizontally. All available node sizes follow the same fixed ratio between memory, CPU and disk.

Indexing tier autoscaling. The two metrics used for ingest autoscaling in Elasticsearch are memory usage and ingestion load. The memory metrics report the minimum required memory that each indexing node must have, along with the minimum available memory for the entire indexing tier. Note that the former impacts choosing the minimum node size for the indexing tier. The memory metrics are estimated such that each node must have a minimum available heap memory which is required for Serverless ES's baseline usage. This is the basic requirement to handle indexing/search workloads, and hold metadata for the cluster and the portion of indices and their shards assigned to the node in memory.

Ingestion load characterizes the amount of CPU needed to handle the current indexing load. Each indexing node reports its ingestion load, which accounts for both the currently running, the received, and queued indexing requests. The ingestion load also indirectly includes the disk I/O requirements of the indexing workload, since the I/O work during indexing is handled by the same thread that processes the indexing request. Therefore, if I/O becomes a bottleneck in the current indexing tier size, the higher reported ingestion load (which accounts for queued items) triggers a scale-up event.

Search tier autoscaling. Similarly, search tier autoscaling relies on memory metrics, search load, and the current dataset size. The memory metrics and search load are calculated similarly to their indexing tier counterparts. Unlike the indexing nodes that push indexed data into the object store periodically and free up their local disk, the search nodes rely on caching the data on their disks to serve search requests. Users are able to configure what percentage of the dataset is cached locally, a configuration called "search power" in Serverless ES. This configuration impacts search performance and provides a trade-off between resource usage and speed [12]. This is exposed as a choice to the user because it translates directly to cost, and unlike in indexing, where clients control resource usage through their indexing throughput, it is more difficult for clients to provide a clear signal of their search latency requirements implicitly. Caching a higher portion of the dataset improves search performance while requiring a larger cache which uses the node's local disk. Based on this configuration, the search nodes could also scale up, in order to provide a larger disk to accommodate the requested local cache size.

Frequency. Metrics are updated every few seconds, or upon important changes, e.g., creation of an index or updates to index field types or shard count. The Autoscaler polls for aggregated metrics of both tiers every 5 seconds, and adjusts the tiers' sizes. To avoid flapping, there is a cool down window of 30 minutes for scale downs. This means that after we detect the first time that a scale down is needed, we wait for 30 minutes of continuous scale down signal and only scale down afterwards. Any set of metrics exposed during this window that indicate a scale down is not needed (e.g., a burst in the workload) would reset the window. When calculating the ideal tier size to scale down to, we consider some memory and CPU headroom (currently 20%). These ensure that when we scale down, we have seen a sustained period of low load, and the tier size we scale down to still allows for some burstiness in the workload. For scale ups, the ingestion/search load that is exposed from each node already accounts for some acceptable queued work and exposes a load value higher than the node's capacity only if the queued work is higher than the acceptable threshold. If the node estimates that the queued work will be cleared within this threshold, it will cap its reported load to the maximum that the node can sustain.

8 Experimental evaluation

In this section, we show microbenchmarks relevant to the core techniques in this paper, and then end-to-end experiments on Elastic Cloud comparing (stateful) ES and Serverless ES. Serverless ES is available as a managed cloud service by Elastic (Elastic Cloud Serverless), supporting all major hyperscalers, and has been generally available (GA) since December 2024. Note that the settings and values we present may change in the future, and that our evaluation aims to primarily show the qualitative trends and differences of the techniques of this paper. We focus primarily on ingestion, as searches operate largely similar to searchable snapshots in ES.

8.1 Microbenchmarks

Here, we show how the core parameters relevant to the upload of BCCs, such as the maximum amount of commits inside a VBCC, and of translog compound files, such as the flush interval, can affect performance and object store costs.

Experimental configuration. Each microbenchmark runs in an integration test (IT) on a single GCP e2-standard-16 machine [27] with 16 vCPUs, 64 GiB RAM and a 250 GiB SSD disk, running on Ubuntu 22.04. The IT starts a master, an indexing and a search (virtual) node. The file system is used as the object store. Each node shares the CPU resources of the host, but uses a separate cache on disk that is sized to fully contain the workload's dataset.

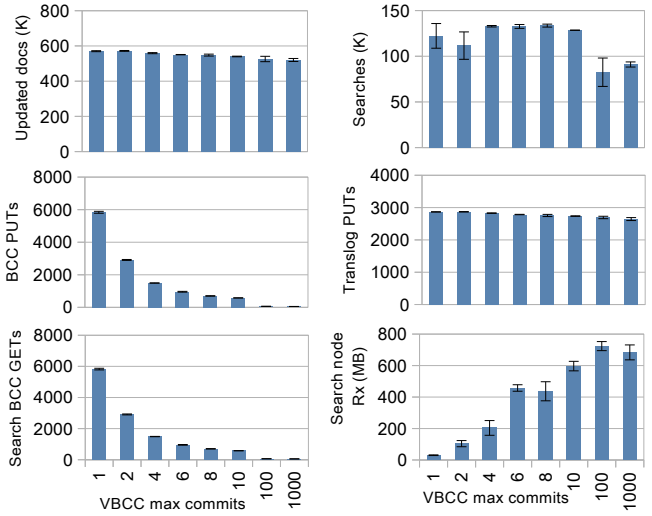

Figure 9: Increasing the max amount of commits in a VBCC lowers object store PUTs at the expense of inter-node traffic.

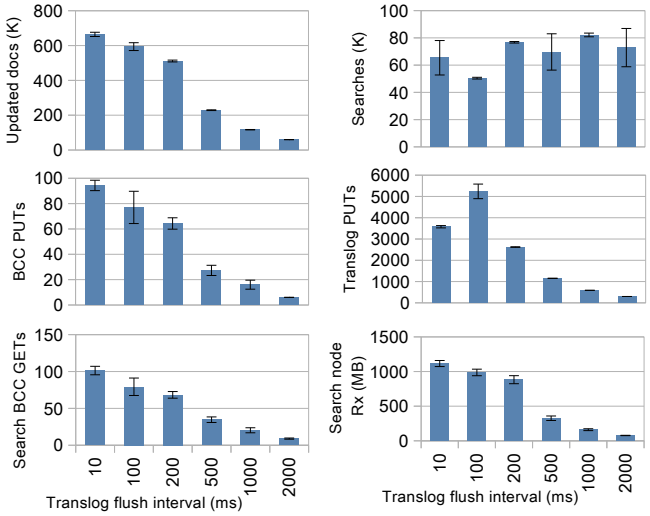

Figure 10: Increasing the translog flush interval lowers object stores PUTs at the expense of indexing latency.

The workload consists of a startup phase that creates an index with 100K documents, each with an ID equal to its document number, and a field with 25 random characters. The evaluation phase starts 4 update and 4 search client threads that run concurrently for 5 minutes. Each update thread is assigned a disjoint slice of the last 10K documents, and continuously updates the field (with 25 random characters) of 100 random documents of the slice, refreshing (unthrottled) the index as well. The refreshing frequency is deliberately aggressive, to magnify the effects we would like to explore. Each search thread continuously searches 100 random documents (based on their doc IDs) of the last 10K documents. Each thread has a think time of 1-5ms. Results shown are averages of 3 iterations, with the standard deviation plotted as error bars.

VBCC max commits. For these experiments, we keep the default flush interval at 200 ms, and toggle the maximum number of commits (CCs) in a VBCC before it is uploaded to the object store. The results are shown in Figure 9. Update performance, and the number of translog uploads, is similar across all experiments. However, as we increase max commits, every new commit, done by refreshes, is less likely to trigger a VBCC upload. While the VBCC is not uploaded, the searches will fetch any non-cached data from the indexing node. This is why we observe a gradual decrease of the BCC uploads and the BCC object store GETs from the search nodes, while the received transport traffic of the search nodes increases, which takes a toll on the number of searches as well. Overall, we observe we can save up to 100x object store PUTs for BCCs, at the expense of a 25x increase in transport traffic between the nodes, which may not incur CSP costs.

Translog flush interval. For these experiments, we keep the default max commits in a VBCC at 100, and toggle the flush interval for translog uploads. The results are shown in Figure 10. The number of updated docs decreases as the flush interval increases, because each client waits for the bulk request's acknowledgement before proceeding further. The acknowledgement's latency increases as the flush interval increases. For the same reason, the number of translog PUTs to the object store also decreases. Since fewer updates are ingested, the number of BCC PUTs also decreases. The number of searches is virtually unaffected, but they tend to search increasingly older data and cached data, and that is why the number of BCC GETs that the search nodes cache decreases, along with the inter-node traffic. Overall, we observe that we can save up to 30x of translog PUTs at the expense of a 10x decrease in update latency.

Batching effects. In summary, commit batching does not affect ingestion and can slightly worsen search performance (with additional network traffic). Translog batching does not affect searches but can severely affect indexing latency, and that is why we bound translog batching to every 200ms. Note that while indexing latency is affected, we optimize for throughput which does not suffer for real-world workloads that typically index many operations concurrently (see §8.2 for an evaluation).

8.2 ES vs Serverless ES

Here, we compare the ingestion performance of Serverless ES against (stateful) ES, without the aid of autoscaling. We show that the ingestion performance of Serverless ES is comparable, and can outperform ES under high load, with a similar H/W configuration.

Experimental configuration. For ES, the cluster consists of 3 nodes, each with 60GiB RAM and 27.9GiB JVM heap. For Serverless ES, the cluster is pinned with 3 64GiB RAM indexing nodes, each with a 28.9GiB JVM heap. Each system resource configuration is represented by instances publicly available in Elastic Cloud Hosted and Elastic Cloud Serverless. Nodes in both environments were equipped with 32 CPU cores. Environments were sized to be as similar as possible. The sizing differences are small enough to have negligible effects on performance results.

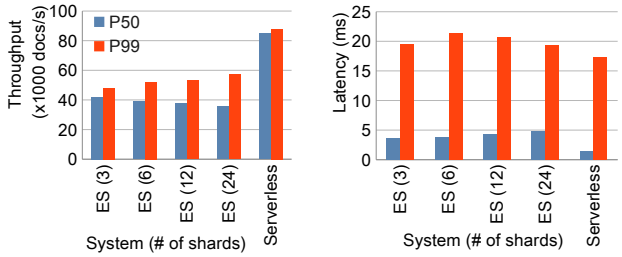

Figure 11: The throughput and latency of ES (with different shards) vs Serverless ES (which uses autosharding).

Data is ingested, unthrottled and the ingest workload is 1560GiB of JSON formatted real-world Github data with an average document size of 4.33KiB run with Rally (an ES benchmarking tool) [19]. To simulate a realistic ES indexing workload, testing was executed against data streams in 3, 6, 12, and 24 primary shard count increments. The indexing test was executed against Serverless ES having a data stream in a single-shard configuration, since Serverless ES has autosharding [10]. Indexing was performed by 96 parallel clients using a 2000 document batch size per request. The ES data stream was configured with one replica for data redundancy. Serverless ES relies on the object store for this. The search tier is at its lowest size due to no search activity.

Evaluation. Each test iteration measures indexing throughput and latency. Figure 11 shows the 50th and 99th percentile throughput and latency. ES results are shown for each primary shard count increment along with the result for serverless ES (with autosharding). The 2x 50th percentile throughput advantage of Serverless ES over ES is mainly attributed to their different data durability mechanisms: ES uses replication in which every indexing operation is performed first on the primary shard and then on each replica, whereas in Serverless ES indexing operations become durable when they are uploaded to the object store without additional indexing.

8.3 Serverless ES ingest autoscaling

In this section, we evaluate the ingest scaling behaviour of Serverless ES. We show that Serverless ES is able to effectively scale relying on its ingest autoscaling approach (see §7) to utilize extra resources depending on the workload, while increasing throughput and maintaining indexing latency.

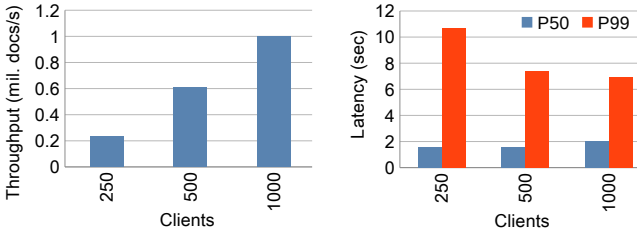

Figure 12: Impact of increasing indexing load and ingest autoscaling on indexing performance.

The setup uses the standard Serverless configuration where the indexing tier starts with a small node with 2GiB RAM, and goes through predefined cluster configurations where the number of nodes and their size can vary. The largest node size is 64GiB RAM. Each predefined cluster configuration that uses nodes smaller than 64GiB, has a maximum of 3 nodes. If further resources are needed, we scale up to the next node size. Once at the largest node size, the cluster can scale only horizontally by adding more 64GiB nodes. The CPU and disk available to each node is a function of its memory, and uses the same ratio on all node sizes.

The workload used for this experiment is 4TiB of existing internal observability data consisting of logs, metrics and application trace data. Ingest targeted multiple data streams with autosharding enabled [10]. In the experiment, we ingest this raw data unthrottled and rely on ingest autoscaling to scale up the indexing tier accordingly to be able to handle this. Each client sends bulk indexing requests which include 2500 documents. Upon sending a request, the client waits for an acknowledgement, before sending a new request. We increase the number of clients from 250 to 1000 and measure the average indexing throughput and indexing latency.

During the experiment, the increased indexing load resulted in scaling up the indexing tier to 8 nodes of 64GiB RAM to handle the 250 clients. Increasing the number of clients further, scaled up the cluster to 19 and 30 nodes (64GiB RAM) to handle the load from 500 and 1000 clients, respectively. Figure 12 shows that as we increase the indexing load, Serverless ES is able to utilize the increased resources and maintain a linear increase in throughput

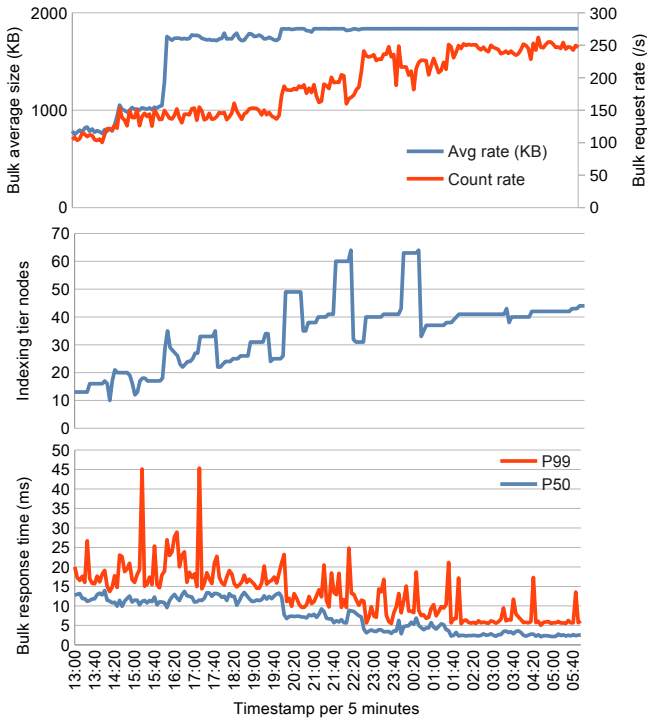

Figure 13: Bulk requests of an increasing real-world workload, and how Serverless ES auto-scales to improve and stabilize bulk response times.

matching the load. As the cluster scales up to handle the load, it is able to maintain the same P50 and P99 for the indexing latency.

8.4 Real-world example of ingest autoscaling

This is an example from our internal dashboards, showing a time window where a demanding Serverless ES real-world client is increasing their ingestion workload gradually, and how Serverless ES is autoscaling to improve the efficiency and performance of the ingestion workload. In Figure 13, the top graph shows the throughput of the client's workload increasing as autoscaling grows the cluster. The middle graph shows the number of "pods" (nodes) that autoscaling has provisioned for the cluster as it reacts to changes in workload. Finally, the bottom graph shows request latency dropping as the client workload is distributed across more nodes.

9 Related work

To fully leverage modern public cloud infrastructure, database systems have moved away from traditional *shared-nothing architectures* in favor of *storage-compute disaggregation* [39]. Several well-known database systems follow this design principle, including Snowflake [9], Amazon Aurora [45], AlloyDB [24], AnalyticDB [46], Polaris [1], Quickwit [33], and OpenSearch [37].

Snowflake [9] is a cloud-native data warehousing solution that supports structured and semi-structured data. It employs a centralized storage layer (e.g., Amazon S3) while dynamically provisioning independent compute clusters, called *Virtual Warehouses*, for scalable query processing. Additionally, Snowflake maintains a separate *Cloud Services layer* to manage metadata and system state.

Amazon Aurora [45] is a relational database that offloads redo log processing to a smart storage layer, addressing the *write amplification* problem and improving performance. Instead of writing full pages, Aurora stores redo logs in a distributed storage service, which then reconstructs data pages on demand as needed. To ensure fault tolerance and high availability, Aurora's distributed storage automatically creates six copies of data across three availability zones (AZs) for high availability.

Google AlloyDB [24] similarly decouples compute and storage using *write-ahead logging (WAL)*, enabling independent scaling. It relies on Google's distributed storage infrastructure for durability. AlloyDB supports two types of instances: A *Primary instance* for transactional processing (reads and writes) and *Read Pool instances* to handle read scaling.

AnalyticDB [46] is an OLAP database that runs on Alibaba cloud. It follows an architecture that decouples reads and writes, served by read nodes and write nodes respectively. These two types of nodes are isolated from each other and can scale independently.

Polaris [1], the distributed SQL engine behind Azure Synapse Analytics, adopts a stateless architecture that goes beyond traditional storage-compute separation. It not only decouples compute from storage but also separates compute from state, closely aligning with Serverless ES's design philosophy.

Quickwit [33] is a distributed search engine decoupling compute and storage, that uses Lucene for inverted indices. It is optimized for append-only workloads and does not lend itself naturally to updates and deletes, because it avoids merges. It also stores data in multiple formats: Inverted index, columnar and row-oriented storage depending on the search use case.

OpenSearch [37] perhaps is architecturally most similar to Elasticsearch since it is built on top of a fork of Elasticsearch. Performance comparisons between Elasticsearch 8.7 and OpenSearch 2.7 suggest that ES outperforms OpenSearch in six major areas including text querying and resource utilization [28]. These performance comparisons were done on a self-managed stateful cluster. However our experiments show that Serverless ES surpasses the performance of (stateful) ES. Architecturally, OpenSearch Serverless is similar to Serverless ES. One limitation we note is that OpenSearch Serverless has a refresh interval of 10s as opposed to 5s for Serverless ES [5]. A full comparison with OpenSearch Serverless is an extensive endeavor that is left out of the scope of this paper.

10 Conclusion

In this paper we presented the new architecture behind Serverless Elasticsearch which decouples compute from storage through the use of a highly durable and available cloud object store, scales automatically in response to client workload changes, and provides automated, frequent upgrades. This approach relies on the object store as the main storage for index, translog, and cluster state data. We showed how Serverless Elasticsearch batches index commits and translog operations to reduce costs incurred by the object store API calls, while maintaining the same read-after-write semantics as Elasticsearch. The results of our experimental evaluations confirm that this new stateless architecture outperforms Elasticsearch with up to 2x better indexing throughput, and better latency. Furthermore, we show how thin indexing shards enable fast scale-ups. This allows Serverless Elasticsearch to scale its throughput linearly, using increased resources, without an increase in latency.

Future work. We aim to improve Serverless ES even further to enhance performance and cost effectiveness. A major area of focus is around transparently supporting multiple client "projects" (the basic unit of deployment of client workloads in Serverless ES) on the same Serverless ES cluster. Coupled with guarantees for avoiding noisy neighbor problems, this can improve cost effectiveness without sacrificing performance. We could use the project identifier (that is part of Serverless API endpoints) to decide which multi-tenant cluster would serve a request. Furthermore, we intend to dynamically unload unnecessary objects from memory for indexing shards that are inactive, and re-load them transparently ad-hoc once indexing recurs, to decrease the memory footprint of client workloads. Moreover, autosharding is already supported for data streams [10], and we plan to transparently support autosharding for regular indices as well, to improve their performance while avoiding oversharding degradation [17]. Finally, we are investigating using CSP storage classes to control performance and cost.

Acknowledgments

We thank the ES Distributed Systems team and in particular David Turner, Tanguy Leroux, Tim Brooks, Yang Wang, Ievgen Degtiarenko, Dianna Hohensee, Artem Prigoda, Nick Tindall, Mikhail Berezovskiy, Albert Zaharovits, David Brimley, Victor Ricart, and Rory MacKenzie for their contributions, and also the entire ES engineering team.

References

[1] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, Lazar Davidovic, Maja Damjanic, Slobodan Djunic, Nemanja Djurkic, Charles Feddersen, Cesar Galindo-Legaria, Alan Halverson, Milana Kovacevic, Nikola Kicovic, Goran Lukic, Djordje Maksimovic, Ana Manic, Nikola Markovic, Bosko Mihic, Ugljesa Milic, Marko Milojevic, Tapas Nayak, Milan Potocnik, Milos Radic, Bozidar Radivojevic, Srikumar Rangarajan, Milan Ruzic, Milan Simic, Marko Sosic, Igor Stanko, Maja Stikic, Sasa Stanojkov, Vukasin Stefanovic, Milos Sukovic, Aleksandar Tomic, Dragan Tomic, Steve Toscano, Djordje Trifunovic, Veljko Vasic, Tomer Verona, Aleksandar Vujic, Nikola Vujic, Marko Vukovic, and Marko Zivanovic. 2020. POLARIS: the distributed SQL engine in azure synapse. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3204–3216. doi:10.14778/3415478.3415545

[2] Amazon. 2002. Amazon Web Services. https://aws.amazon.com/

[3] Amazon. 2025. Amazon S3 Durabilty and Data Protection. https://aws.amazon.com/s3/faqs/#topic-10

[4] Amazon. 2025. Amazon S3 Strong Consistency. https://aws.amazon.com/s3/consistency/

[5] Amazon. 2025. What is Amazon OpenSearch Serverless? https://docs.aws.amazon.com/opensearch-service/latest/developerguide/serverless-overview.html

[6] Henning Andersen. 2024. Stateless: Data safety in a stateless world. https://www.elastic.co/search-labs/blog/data-safety-stateless-elasticsearch

[7] Apache. 2025. Lucene. https://lucene.apache.org

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 143–157. doi:10.1145/2043556.2043571

[9] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. doi:10.1145/2882903.2903741

[10] Andrei Dan. 2024. Autosharding of data streams in Elasticsearch Serverless. https://www.elastic.co/search-labs/blog/datastream-autosharding-serverless

[11] Josh Devins, Julie Tibshirani, and Jimmy Lin. 2022. Aligning the Research and Practice of Building Search Applications: Elasticsearch and Pyserini. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining* (Virtual Event, AZ, USA) *(WSDM '22)*. Association for Computing Machinery, New York, NY, USA, 1573–1576. doi:10.1145/3488560.3502186

[12] Elastic. 2025. Elastic Cloud Serverless Pricing. https://www.elastic.co/pricing/serverless-search

[13] Elastic. 2025. Elasticsearch Cluster-level shard allocation and routing settings. https://www.elastic.co/docs/reference/elasticsearch/configuration-reference/cluster-level-shard-allocation-routing-settings

[14] Elastic. 2025. Elasticsearch Flush API. https://www.elastic.co/docs/api/doc/elasticsearch/operation/operation-indices-flush

[15] Elastic. 2025. Elasticsearch Reading and Writing Documents. https://www.elastic.co/docs/deploy-manage/distributed-architecture/reading-and-writing-documents

[16] Elastic. 2025. Elasticsearch Reference Guide. https://www.elastic.co/docs/reference/elasticsearch/

[17] Elastic. 2025. Elasticsearch Size your shards. https://www.elastic.co/docs/deploy-manage/production-guidance/optimize-performance/size-shards

[18] Elastic. 2025. Formal models of core Elasticsearch algorithms. https://github.com/elastic/elasticsearch-formal-models

[19] Elastic. 2025. GitHub Archive Rally Track. https://github.com/elastic/rally-tracks/tree/master/github_archive

[20] Elastic. 2025. Serve more with Elastic Cloud Serverless and Search AI Lake. https://www.elastic.co/cloud/serverless

[21] Ken Exner. 2022. Serve more with Serverless. https://www.elastic.co/blog/elastic-serverless-architecture

[22] Francisco Fernández Castaño and Henning Andersen. 2024. How we optimized refresh costs in Elasticsearch Serverless. https://www.elastic.co/search-labs/blog/elasticsearch-refresh-costs-serverless

[23] Google. 2008. Google Cloud Platform. https://cloud.google.com/docs/overview/

[24] Google. 2025. AlloyDB. https://cloud.google.com/alloydb/docs/overview

[25] Google. 2025. Cloud Storage Consistency. https://cloud.google.com/storage/docs/consistency

[26] Google. 2025. Cloud Storage Data Availability and Durability. https://cloud.google.com/storage/docs/availability-durability

[27] Google. 2025. General-purpose machine family for Compute Engine. https://cloud.google.com/compute/docs/general-purpose-machines

[28] George Kobar and Ugo Sangiorgi. 2023. Elasticsearch vs. OpenSearch: Unraveling the performance gap. https://www.elastic.co/blog/elasticsearch-opensearch-performance-gap

[29] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. doi:10.1145/1773912.1773922

[30] Leslie Lamport. 1998. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. doi:10.1145/279227.279229

[31] Tanguy Leroux. 2024. Introducing Serverless Thin Indexing Shards. https://www.elastic.co/search-labs/blog/thin-indexing-shards-elasticsearch-serverless

[32] Leaf Lin, Tim Brooks, and Quin Hoxie. 2022. Stateless — your new state of find with Elasticsearch. https://www.elastic.co/search-labs/blog/stateless-your-new-state-of-find-with-elasticsearch

[33] François Massot. 2023. Quickwit 101 - Architecture of a distributed search engine on object storage. https://quickwit.io/blog/quickwit-101

[34] Microsoft. 2008. Microsoft Azure. https://azure.microsoft.com/

[35] Microsoft. 2025. Azure Storage Redundancy. https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy

[36] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, USA, 305–320.

[37] OpenSearch. 2025. Introduction to OpenSearch. https://opensearch.org/docs/latest/getting-started/intro/

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[39] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. *Proc. ACM Manag. Data* 2, 3, Article 180 (May 2024), 26 pages. doi:10.1145/3654983

[40] Matteo Piergiovanni and John Verwolf. 2024. Search tier autoscaling in Elasticsearch Serverless. https://www.elastic.co/search-labs/blog/elasticsearch-serverless-tier-autoscaling

[41] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. 1995. Okapi at TREC-3. *Nist Special Publication Sp* 109 (1995), 109.

[42] Pooya Salehi, Henning Andersen, and Francisco Fernández Castaño. 2024. Ingest autoscaling in Elasticsearch. https://www.elastic.co/search-labs/blog/elasticsearch-ingest-autoscaling

[43] Clint Scott. 2024. Elastic Cloud Serverless pricing and packaging. https://www.elastic.co/blog/elastic-cloud-serverless-pricing-packaging

[44] solid IT. 2025. DB-Engines Ranking - popularity ranking of search engines. https://db-engines.com/en/ranking/search+engine

[45] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. doi:10.1145/3035918.3056101

[46] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2059–2070. doi:10.14778/3352063.3352124