Automated Realistic Benchmarks for OpenSearch Clusters

Jean Perbet

2025 CERN Summer Student

Project Report

July 25, 2025

Team
IT-DA-DS

Project Advisors
Sokratis Papadopoulos
Luis Pigueiras

CERN
Geneva, Switzerland

Abstract

The OpenSearch service at CERN has been operating since 2016, storing and analysing 900 TBs of data to support various use cases such as log analytics and full-text search. It is currently hosted on puppet-managed servers, but since 2024 efforts are underway to migrate the deployment to a more standardized Kubernetes-based setup. As part of this transition, it's essential to evaluate the performance of both small and large OpenSearch clusters within Kubernetes and understand how this shift impacts performance. More broadly, having a straightforward method to benchmark OpenSearch under various configurations is valuable for identifying bottlenecks and weighing the advantages and limitations of different deployment approaches. This report introduces an easy-to-use pipeline for benchmarking OpenSearch clusters in Kubernetes with customizable parameters, using real-world scenarios inspired by CERN's primary customers use cases.

Contents

Introduction

Efficient data storage and processing have always been a main concern at CERN, and the Data Storage section is no exception to the rule. In the OpenSearch team, previous summer students had already laid a strong foundation, separately focusing on benchmarking the performance of the existing setup, and exploring OpenSearch deployment within Kubernetes environments. When I joined, I was able to pick up where they left off, benefitting from a good timing with the release of an official benchmarking tool [1] for OpenSearch. With this new tool, I was able to combine and extend the previous work, validating performance metrics in a Kubernetes setup and pushing the project forward in a more integrated way.

This project consisted of three main parts. After getting familiar with the benchmarking environment, the first step was to study the primary use cases of OpenSearch's main CERN users. Based on this, I designed specific workloads, i.e. scenarios that reflect real-world usage, to serve as the foundation for benchmarking. Next, I assessed the performance of the current setup by comparing the standard Puppet-based deployment with various Kubernetes-based OpenSearch clusters leveraging different kinds of underlying storage, using the newly defined workloads. Finally, I automated the entire process by building an all-in-one pipeline, making it easy to trigger benchmark runs for different configurations on the best-performing Kubernetes setup identified in the previous step.

All the workloads, as well as the CI pipeline, are available on a GitLab repository [2]. Current setup performance results, as well as CI results, are/will be stored on `os-perfmon`, the cluster used by OpenSearch team for general performance monitoring.

[1] OpenSearch Benchmark - `https://docs.opensearch.org/docs/latest/benchmark/`

[2] CERN OpenSearch benchmarking suite - `https://gitlab.cern.ch/opensearch/others/opensearch-benchmark-suite`

Chapter 1 Workloads

A workload is defined by several components: the data that will be indexed in OpenSearch, the way the data will be indexed and the queries that will be run against the indexed data. The indexed data, referred to as the corpora, is provided in one or more JSON documents. Likewise, the index settings are specified across one or several JSON files, depending on the number of indices needed, while queries and the overall structure are grouped into a final JSON file. I chose to obtain real-world data directly from existing OpenSearch clusters of key customers, utilizing pre-configured index settings. I developed relevant queries closely aligned with these customers, either by examining their production code or by discussing their OpenSearch usage at a more conceptual level.

1.1 Customers & Datasets

The customers I focused on are the Monitoring team [1], the Security team and the team running the Inspire service [2]. They represent a representative sample of OpenSearch usage at CERN, and they provided me with valuable information on their main use cases. I decided to define one workload per *cluster*, rather than per *customer*. Since the Monitoring team operates three clusters, they have three workloads, whereas both the Security and Inspire teams have one workload each.

Figure 1.1: Customers and their corresponding OpenSearch clusters.

[1] CERN IT Monitoring Team - `https://monit.docs.cern.ch`
[2] Community hub for scholarly information in high-energy physics - `https://inspirehep.net`

Below is a breakdown of the most representative indices of each customer (i.e. tabular data in OpenSearch) that I used for benchmarking. I sub-sampled them to keep the benchmarking efficient, and I imported their corresponding indices settings using OpenSearch API.

Cluster	Index(es)	Sample size (GB)
monit-opensearch	xrootdng_agg_transfer monitoring records of XrootD data transfers	3.3
monit-opensearch-lt	fts_raw_start records of File Transfer Service (FTS) transfers	4.9
monit-timberprivate	cernbox_logs_raw monitoring logs for CERNBox actions	3.3
os-csl	csl_pa_threat threat logs from Palo Alto firewall & csl_audit_exec audit logs (biggest index !!)	2.4 + 4.1
os-inspire-prod	inspire-v1.3.102-records-hep informations about all documents from inspire service	5.8

Figure 1.2: Clusters and their corresponding indices of interest.

Additionaly, I used a default dataset [3] provided by the OpenSearch Benchmarking Tool to stress clusters with higher quantities of data (approx. 100 GBs). This workload was only used during the preliminary results phase, it is not included in the CI due to its long running time.

1.2 Queries

As previously noted, input from the customers was essential in shaping realistic queries, allowing the benchmark to accurately simulate real-world scenarios. The available tasks for benchmarking are similar to operations on the standard OpenSearch API [4]. All workloads start with the same set of tasks, used to safely ingest the data on the benchmarked OpenSearch cluster.

- Delete workload index(es) if already existing and re-create them
- Ingest all workload data
- Wait for cluster to reach steady state (`green`)

[3] `big5` workload - `https://github.com/opensearch-project/opensearch-benchmark-workloads/tree/main/big5`

[4] OpenSearch Benchmarking Tool operations - `https://docs.opensearch.org/docs/latest/benchmark/reference/workloads/operations/`

Then, below is a breakdown of all workload-specific queries.

Customer	Cluster/Workload	Queries
Monitoring	`monit-opensearch` > `monit_fts`	• search by transfer ID & aggregation by endpoint • search by date range • timestamp sorting of all logs • nested aggregation on destination
Monitoring	`monit-opensearch-lt` > `monit_xroot`	For this one ingestion phase included a 25% conflict probability to mimic document overwrite. • nested aggregation on file size and transferred volume • daily volume histogram • timestamp sorting of all logs
Monitoring	`monit-timberprivate` > `monit_cernbox`	• aggregation on data servers • top file extensions per request count • timestamp sorting of all logs
Inspire	`os-inspire-prod` > `inspire`	• citation summary & per year • full-text search • heavy async sort
Security	`os-csl` > `csl`	For this one two indexes were ingested at once. • unique audit hosts & users • top audit hosts table • palo alto firewall threat count & top threats • palo alto firewall vulnerability, spyware & URL overview

Table 1.1: Workloads and associated representative queries.

Chapter 2 Current performance

So as to get an idea of current performance, I ran all previously defined workloads on several deployments. Below is a breakdown of the characteristics of each of these flavours.

Flavour	Storage	Deployment
`os-betatest`	local NVMe	**Puppet**
`os-io3`	Disk max: 300 MB/s, with rate of 5 IOPS per GB, guaranteed min of 500 IOPS, max 2000 IOPS.	**Kubernetes**
`os-io2`	Disk max: 300 MB/s, 500 IOPS.	**Kubernetes**
`os-io1`	Disk max: 120 MB/s, 500 IOPS.	**Kubernetes**
`os-standard`	Disk max: 80 MB/s, 100 IOPS.	**Kubernetes**

Table 2.1: Benchmarked OpenSearch deployments

I created an OpenSearch Dashboard with visually appealing results on `os-perfmon` cluster, called *Benchmarking (2025)*. It displays mean latency & throughput for each task of each workload, with filters on cluster and/or dataset. As expected , the results show performance proportional to the underlying storage throughput, with dominance of NVMe over others.

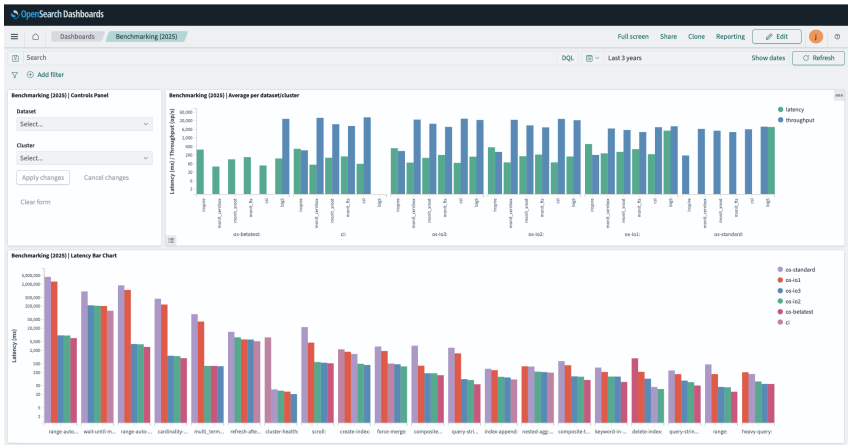

Figure 2.1: Results Dashboard

Interestingly, the gap in search performance increases with the dataset size, since results are the most contrasted on `big5` (i.e. 100 GBs) workload.

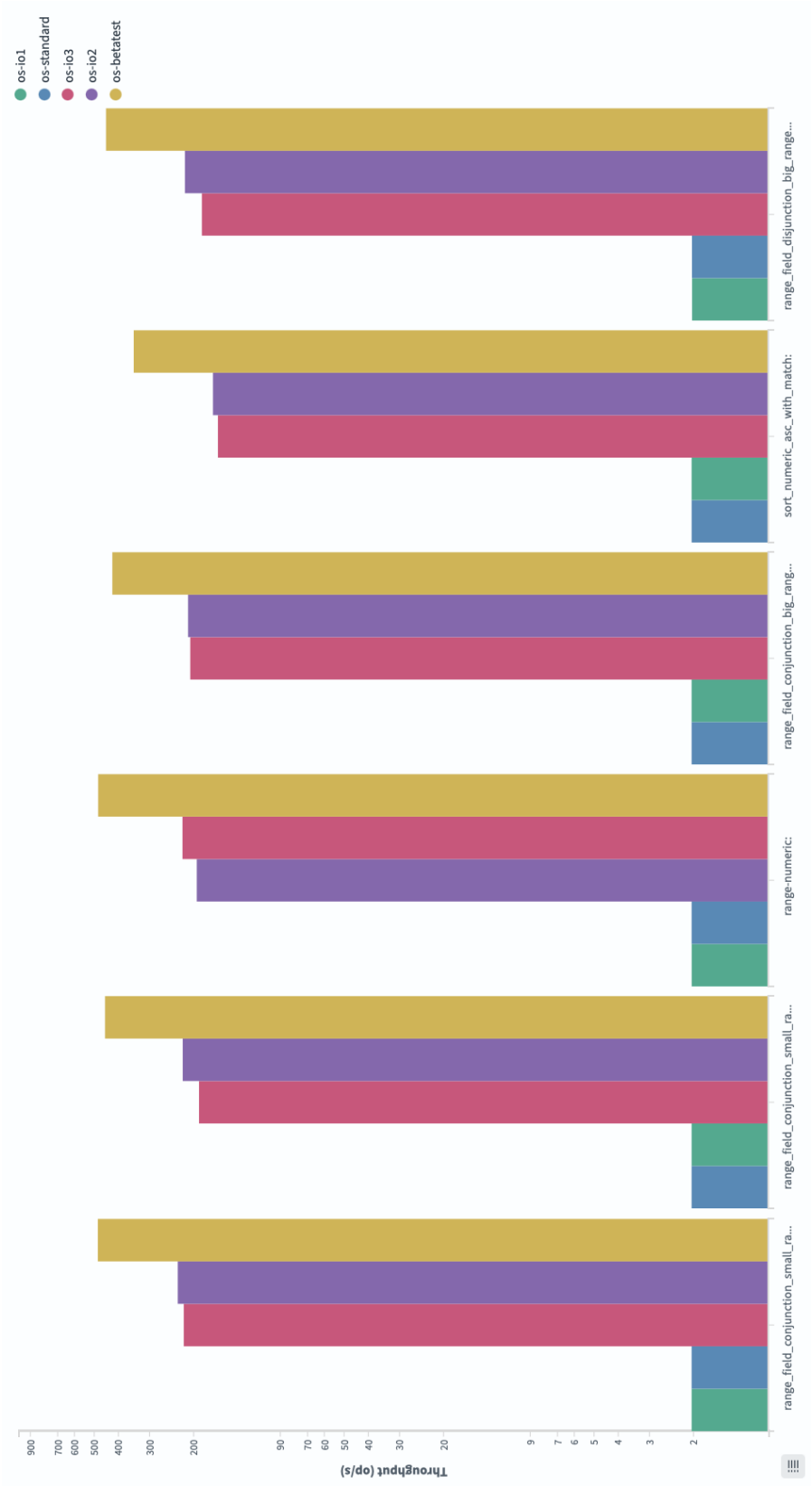

Figure 2.2: `big5` (100 GBs) workload throughput comparison (log. scale)

Chapter 3 Continuous Integration

With the initial benchmark results obtained, I set up a Continuous Integration (CI) pipeline to automatically trigger benchmark runs under predefined parameters. As stated in the introduction, all configurations and logic are maintained in a dedicated GitLab repository, which holds both the workload specifications and the CI orchestration logic. To avoid inflating the repository size, all workload data (corpora of documents) is stored externally in an S3 bucket.

3.1 Overview

The CI pipeline is triggered automatically upon each push to the GitLab repository. It performs a full benchmark test sequence designed to validate and monitor OpenSearch performance under real-world workloads. Specifically, the pipeline executes the following steps:

- **Cluster Bootstrap:** A new OpenSearch cluster is deployed on the `dev-io3-1` Kubernetes cluster. This deployment reads its configuration from the `values.yml` file in the repository, where parameters are easily adjustable.
- **Workload Preparation:** Benchmarking workloads are retrieved from an S3 bucket.
- **Benchmark Execution:** Each workload is run sequentially against the freshly deployed OpenSearch cluster.
- **Performance Comparison:** The results of the benchmark are compared against the latest available results on the `os-perfmon` monitoring cluster. This comparison generates a detailed performance evolution report for each metric, enabling effective tracking of regressions or improvements caused by code or configuration changes.

All benchmark results and logs are made available in real-time through the GitLab UI, under the `run_benchmark` job. The entire CI pipeline typically completes in approximately one hour.

3.2 Technical Details

The benchmarks are run on a Kubernetes deployment that uses the `io3` storage flavour, identified in the previous evaluation phase as the highest-performing option for Kubernetes. This ensures that CI results reflect optimal conditions in Kubernetes, which remains the main objective of my whole project.

3.2.1 OpenSearch Cluster Access

Once the OpenSearch cluster is bootstrapped using `kubectl` from within the GitLab CI runner, it becomes accessible at the internal service URL:

```
https://benchmark-ci-v1-svc:9200
```

This URL is resolvable from within the `dev-io3-1` Kubernetes cluster. The CI job communicates with the cluster by launching a Kubernetes Job, which uses a Docker image built from the repository's `Dockerfile`. This image is responsible for running the benchmark script located at `ci/benchmark.sh`.

3.2.2 Configuration Files and Secrets

The pipeline relies on several configuration files and secrets, which are handled securely:

- `KUBECONFIG` – Required to access the `dev-io3-1` Kubernetes cluster. This is provided as a GitLab file secret and does not require manual intervention.
- `.s3cfg` and `benchmark.ini` – Used respectively to retrieve data from the S3 bucket and to configure the OpenSearch Benchmarking tool to send results to `os-perfmon`. Templates for these files are included in the `ci/` directory and are populated at runtime using GitLab CI secrets. They are then mounted into the job container via a Kubernetes ConfigMap at the path `/config`.

3.2.3 Kubernetes Namespace Handling

To avoid resource conflicts and maintain cleanliness between runs, the pipeline dynamically creates a dedicated Kubernetes namespace called `benchmark-ci` at the beginning of each run. This namespace is deleted upon completion, ensuring no residual resources are left in the cluster.

Future work

There are many possible improvements over this benchmarking suite, which certainly made the tests more realistic and easy to run but lack important metrics. First of all, OpenSearch Benchmarking Tool recently released a new functionality called *redline testing*. This consists in flooding an OpenSearch cluster with many parallel requests to detect its breaking point, i.e. the max. number of clients it can handle in parallel. This may be intersting to integrate such a test in the CI pipeline, comparing whether parameter changes affect the availability of an OpenSearch cluster under high client loads. Also, except from the `big5` workload, all created workloads tend to be relatively small compared to real-world indices, due to benchmark execution time reasons. While it is more efficient, it lacks realism. A potential improvement would be to incorporate fast yet resource-intensive workloads to provide more comprehensive benchmarking coverage.

Acknowledgment

I would like to sincerely thank my supervisors, Luis and Sokratis, for their warm welcome and invaluable support throughout this project. Their expertise has been instrumental, and the completion of this work would not have been possible without their patient answers to my countless questions.