Comparative Performance Analysis of Open-Source Vector Databases: Milvus vs. Weaviate

Andreas Fotakis
(Ανδρέας Φωτάκης)
ECE, NTUA
AM: [03121100]
Team ID: [35]

Nikolaos Katsaidonis
(Νικόλαος Κατσαϊδώνης)
ECE, NTUA
AM: [03121868]
Team ID: [35]

George Tzamouranis
(Γεώργιος Τζαμουράνης)
ECE, NTUA
AM: [03121141]
Team ID: [35]

Project's Github Repository Link: [1]

***Abstract*—As AI moves from prototyping to production, selecting the appropriate Vector Database becomes a critical infrastructure decision. This paper presents a comparative performance analysis of two leading open-source systems: Milvus and Weaviate. Through a custom benchmarking suite utilizing the GloVe-100, Arxiv-titles, and H&M datasets, we evaluate ingestion throughput, parallel query performance, the EF-driven recall–latency trade-off, hybrid search with metadata filtering, storage footprint, and distributed deployment behavior. Results show that Milvus excels in large-scale ingestion and high-dimensional search due to offline indexing and SIMD-optimized execution, while Weaviate provides immediate data availability for real-time applications and better storage efficiency. Distributed experiments further reveal that Milvus benefits more from horizontal scaling, whereas Weaviate incurs higher coordination overhead when sharded. Last but not least, we create a mini dataset consisting of vectors derived from images and experiment on how well can each database perform on a real-world problem such as image-based Industrial Anomaly Detection. Overall, the study highlights that architectural choices—not only ANN algorithms—are decisive for vector database performance under real-world workloads.**

I. Introduction

In the rapidly evolving landscape of Generative AI, unstructured data—comprising text, images, and audio—has emerged as the new frontier of information processing. Vector Databases, specialized systems designed to store and retrieve high-dimensional embeddings, act as the long-term memory for Large Language Models (LLMs), enabling critical applications such as semantic search, recommendation systems, and Retrieval-Augmented Generation (RAG).

As these applications transition from experimental prototypes to mission-critical production systems, the performance characteristics of the underlying database become a decisive factor. However, selecting the optimal infrastructure is non-trivial. While many systems rely on similar algorithmic foundations (such as HNSW for indexing), they diverge significantly in their implementation strategies, consistency models, and architectural design patterns.

To navigate this complex landscape, it is essential to understand the trade-offs between different system designs. Broadly, the ecosystem is divided between lightweight libraries, extensions to existing relational databases, and purpose-built distributed systems.

This report focuses on the latter category, presenting a comparative analysis of two leading open-source vector databases that exemplify contrasting architectural philosophies:

- **Milvus** represents the *disaggregated, cloud-native* paradigm. It strictly decouples storage from computation and separates the ingestion path from the query path. This design aims for massive horizontal scalability, allowing heavy background indexing tasks to run independently of search operations.
- **Weaviate** represents the *modular, unified* paradigm. It adopts a unified approach where vector storage, object storage, and inverted indexing are integrated into a cohesive execution model. This architecture prioritizes developer experience and real-time data availability through a Log-Structured Merge-tree (LSM) engine.

This report provides an empirical evaluation of Milvus and Weaviate, driven by a custom-engineered benchmarking suite.

II. Background and System Architecture

To understand the performance characteristics and latency trade-offs observed in our experiments, it is crucial to analyze the underlying architectural design of the two systems. While both enable high-speed vector retrieval, they adopt fundamentally different design philosophies regarding storage, indexing, and cluster management.

A. *Milvus: Cloud-Native & Disaggregated Architecture*

Milvus represents a paradigm shift in vector database design, adopting a "Log as Data" philosophy tailored for cloud-native environments. As illustrated in Figure 1, its architecture is fundamentally disaggregated, meaning that storage, computation, and stream processing are separated into distinct layers. This decoupling allows each component to scale independently (e.g., scaling QueryNodes to handle higher search traffic without provisioning unnecessary storage).

The system is organized into four distinct planes:

*1) **Access Layer (Stateless Entry):*** The entry point is a group of stateless **Proxies**. Proxies do not store data. They validate client requests, encapsulate them into messages, and forward them to the underlying message storage. Similar to Weaviate, Milvus exposes two interfaces:

- *RESTful API:* Available for system observability (health checks, metrics) and lightweight client operations using standard JSON.
- *gRPC (Primary):* The core high-performance channel used by all official SDKs (Python, Java, Go). Unlike Weaviate's legacy architecture where REST was mandatory, Milvus was built as a "gRPC-native" system, treating REST only as an auxiliary convenience layer.

Being stateless, if a Proxy fails, the load balancer simply redirects traffic to another instance, ensuring high availability.

*2) **Coordinator Service (Cluster Management)**:* Acting as the system's "Brain," this service assigns tasks to worker nodes. It is split into specialized roles:

- **RootCoord:** Handles DDL requests (create/drop collection) and issues global timestamps (TSO) for causal consistency.
- **DataCoord:** Manages the topology of data chunks (*Segments*) and orchestrates the flushing of data from memory to object storage.
- **QueryCoord:** Balances the load of query nodes and manages the handoff between historical and streaming data segments.
- **IndexCoord:** Supervises index construction. It assigns CPU-intensive indexing tasks to *IndexNodes*, ensuring that heavy graph-building jobs do not impact search latency.

*3) **Message Storage (Log Broker)**:* The backbone of Milvus is a persistent message queue (e.g., Apache Pulsar or Kafka). Following the "Log-as-Data" principle, every insertion is an event in a continuous stream. Proxies publish write requests to the log, and Worker Nodes subscribe to consume them asynchronously. Data is durable immediately upon ingestion into the log, even before it is indexed or flushed to object storage.

*4) **Worker Nodes & The Segment Lifecycle**:* The execution layer reveals the core reason behind Milvus's high batch throughput. Data is organized into units called **Segments**.

1) **DataNodes (Ingestion):** They subscribe to the log and buffer data into *Growing Segments* in RAM. Once a segment reaches a size threshold, it is "Sealed" and flushed to Object Storage (MinIO). This process is purely I/O bound and extremely fast.
2) **IndexNodes (Offline Acceleration):** Milvus IndexNodes watch for "Sealed Segments." They pull the raw data from storage, build the vector index (e.g., HNSW) in the background, and write the index file back to storage. This **Resource Isolation** prevents indexing CPU spikes from slowing down ingestion.
3) **QueryNodes (Search):** For retrieval, QueryNodes maintain a "Unified View." They load historical indexes (from IndexNodes) and combine them with real-time streaming data (from the Log), ensuring freshness without blocking writes.

*5) **Data Persistence: Columnar Binlogs**:* Milvus organizes data within Object Storage using a **Columnar Storage** format, specifically implemented through *Binlogs* (binary logs). Unlike traditional row-based databases, Milvus stores each field (vector, scalar attributes, and system IDs) in separate file sequences. This architecture is pivotal for analytical performance; during a filtered search, the engine only streams the specific columns required by the query's metadata predicates, significantly reducing disk I/O and memory pressure.

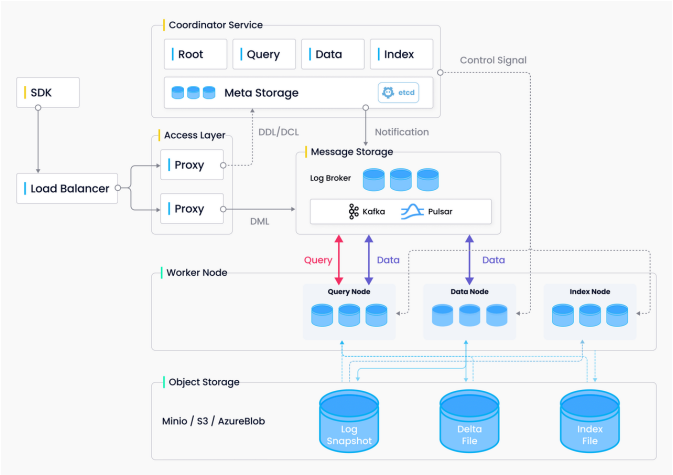


Figure 1: Overview of the Milvus architecture. The disaggregation of DataNodes and IndexNodes allows for high-speed "Offline" ingestion, separating the I/O path from the CPU-intensive indexing path. Source: Milvus Documentation [2].

The persistence layer categorizes these logs into three functional types:

- **Insert Binlogs:** These contain the actual entity data. By organizing them column-wise, Milvus can leverage SIMD instructions to process multiple scalar values simultaneously during a scan.
- **Delta Logs:** Since sealed segments are immutable, deletions are not performed in-place. Instead, Milvus records the primary keys of deleted entities in Delta Logs. During a search, these logs are used to generate a *Deletional Bitset*, which masks out removed entities from the final results.
- **Stats Logs:** These store metadata about the segments themselves, such as minimum and maximum values for scalar fields, enabling "data skipping" optimizations during query execution.

This column-oriented persistence ensures that as the dataset grows, the engine maintains high throughput by only touching the data fragments strictly necessary for the computation of the search result.

*6) **The Search Path: Scatter-Gather & Unified View**:* Retrieved data in Milvus is not stored in a single monolithic structure but is distributed across multiple segments. To ensure "Real-time Informational Freshness," Milvus employs a **Unified View** strategy during retrieval. When a query is issued, it is executed simultaneously across two different types of data:

1) **Sealed Segments:** Historical data that has been indexed

(e.g., via HNSW). Search here is extremely fast as it leverages the pre-built graph structures.

2) **Growing Segments:** Recent data still residing in the DataNodes' buffers. Since these are not yet indexed, Milvus performs a *Brute-Force CPU scan*.

The search process follows a **Scatter-Gather** pattern. The Proxy "scatters" the query to all relevant QueryNodes. Each QueryNode independently searches its assigned segments and applies metadata filters using **Bitsets**. These bitsets act as binary masks (0 or 1) that the vector engine uses to instantly include or exclude candidates during the graph traversal, often accelerated by SIMD instructions.

Finally, the Proxy "gathers" the partial results from all nodes, performs a global merge, reduces the set to the requested *top-k* results, and applies deduplication based on timestamps (TSO). This distributed execution is the reason Milvus excels in high-throughput scenarios, as it can parallelize a single query across dozens of CPU cores simultaneously.

B. Weaviate: Unified Architecture

In contrast to the disaggregated microservices approach of Milvus, Weaviate adopts a **monolithic architecture**. Its design philosophy is rooted in a "Shared Nothing" model, where each node is self-sufficient, managing compute, vector indexing, and object storage locally.

*1) **Interface and Protocol Layer**:* To optimize communication efficiency, Weaviate employs a hybrid protocol strategy. First of all, control Plane (REST/GraphQL), utilized for schema management, metadata configuration, and administrative tasks.Performance-critical operations, such as high-volume ingestion and search, utilize gRPC with Protobuf serialization. This binary protocol bypasses the heavy overhead of JSON parsing associated with legacy REST interfaces.

*2) **The LSM-Tree Persistence Engine**:* The core of Weaviate's storage layer is a custom implementation of the *Log-Structured Merge-tree* (LSM tree). Unlike B-Tree based systems, the LSM engine converts random write operations into sequential disk I/O, making it highly efficient for write-intensive vector workloads.

*a) **The Write Path and Durability**:* Every ingestion operation follows a strict two-stage process to ensure durability: 1) The **Write-Ahead Log (WAL)**, an append-only file on disk that serves as a safety net for crash recovery, and 2) The **MemTable**, an in-memory sorted structure where data is immediately searchable. This dual-path ensures **Real-Time Consistency** without sacrificing ingestion speed.

*b) **SSTables and Compaction**:* Once a MemTable reaches its capacity, it is flushed to disk as a **Sorted String Table (SSTable)**. These files are **immutable**, which simplifies concurrency by eliminating the need for read-locks. To prevent performance degradation from file fragmentation, a background **Compaction** process periodically merges small SSTables into larger ones while purging obsolete data (tombstones). However, this introduces *Write Amplification*, which can impact I/O resources during high-concurrency ingestion.

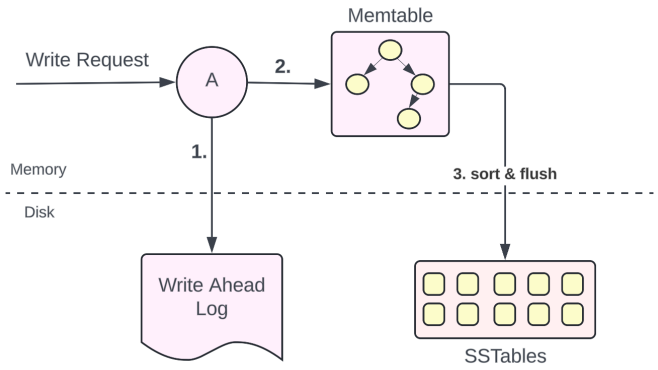


Figure 2: Overview of LSM Tree Persistance Engine. Source: [10]

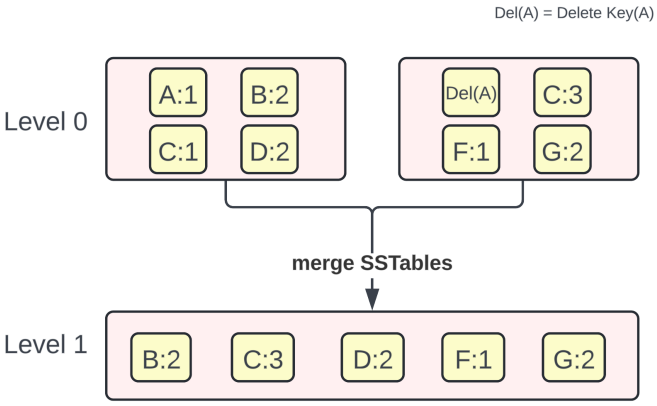


Figure 3: SSTables and Compaction. Source: [10]

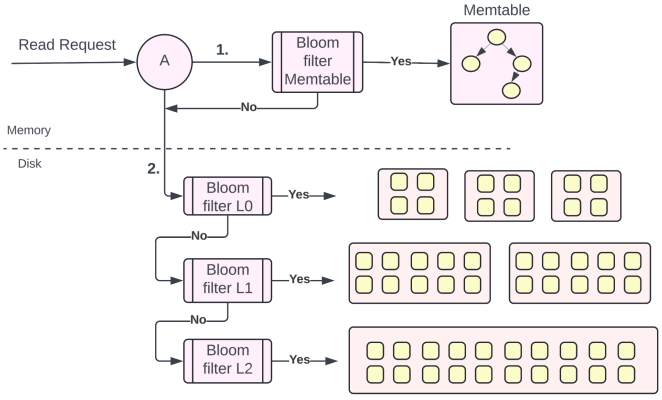


Figure 4: The Read Path. Source: [10]

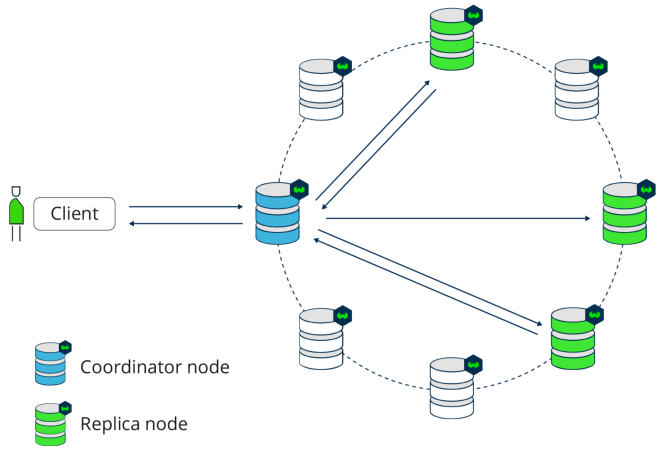


Figure 5: Weaviate's distributed architecture. The diagram illustrates the request routing mechanism where an entry node acts as a **Coordinator**, distributing tasks to relevant shards or **Replica nodes** and aggregating the results. Source: [3]

*c) **The Read Path: Hierarchical Retrieval**:* To maintain low-latency retrieval despite the fragmented nature of LSM trees, the engine follows a hierarchical search sequence:

- **MemTable Search:** The engine first checks the RAM for the most recent data.
- **Bloom Filters:** Probabilistic structures in RAM allow the engine to instantly skip SSTables that do not contain the target key, drastically reducing unnecessary Disk I/O.
- **Sparse Index & mmap:** Once a candidate SSTable is identified, a sparse index facilitates a targeted "jump" to the specific data block. This is further accelerated by **Memory-Mapped Files (mmap)**, which allow the engine to treat on-disk segments as virtual RAM.

You can see a visualization of LSM tree structure and functionality in Figures 2,3,4.

*3) **Distributed Scaling & Modules**:* While our benchmarks focus on a single node, Weaviate is designed to scale horizontally to handle massive datasets using a **Sharding** strategy. Unlike Milvus, which relies on dedicated coordinator services, Weaviate adopts a leaderless architecture. Any node can receive a request and dynamically assume the role of the **Coordinator**, routing queries to the appropriate shards and aggregating results.To ensure durability, Weaviate supports configurable replication. Each shard is replicated across multiple nodes (Replica Nodes), with consistency managed via the Raft consensus algorithm. You can see this visually in Figure 5.

III. Vector Indexing: The HNSW Algorithm

Both Milvus and Weaviate utilize the **Hierarchical Navigable Small World (HNSW)** algorithm as their default indexing mechanism. HNSW is widely regarded as the state-of-the-art for Approximate Nearest Neighbor (ANN) search due to its superior balance between recall accuracy and query latency.

A. Algorithmic Foundation

HNSW borrows concepts from two data structures: *Skip Lists* and *Small World Graphs*. The fundamental idea is to construct a multi-layered graph structure (as shown in Fig. 6):

- **Layer 0 (Base Layer):** Contains all data points (vectors). It is a highly connected graph where neighbors are linked based on proximity (e.g., Euclidean distance or Cosine Similarity).
- **Upper Layers (Hierarchy):** These layers act as "expressways." They contain only a subset of the data points. The higher the layer, the sparser the graph and the longer the links between nodes.

B. The Search Process

Navigation resembles "zooming in" on a map:

1) **Entry Point:** The search begins at the top-most layer with a global entry point.
2) **Greedy Search:** The algorithm greedily moves to the neighbor closest to the query vector.
3) **Descent:** When a local minimum is reached in the current layer (no neighbor is closer), the algorithm drops down to the next layer.
4) **Refinement:** This process repeats until Layer 0 is reached, where a fine-grained local search identifies the final top-k nearest neighbors.

C. Key Hyperparameters

The performance of HNSW is dictated by two critical parameters, which were central to our experimental tuning:

- **M (Max Connections):** Defines the maximum number of outgoing links (edges) per node in the graph. Higher M leads to higher recall in high-dimensional spaces (like our 2048d dataset) but increases memory consumption.
- **ef (Examination Factor):** Controls the size of the dynamic candidate list (beam width) during search.
  - *efConstruction:* Defines neighbor exploration during index building. Higher values improve graph quality but slow down ingestion.
  - *efSearch:* Defines nodes visited during a query. As observed in our experiments, increasing 'ef' (e.g., from 64 to 512) linearly increases latency but significantly boosts recall.

D. Implementation Divergence: Offline vs. Online Indexing

While both systems rely on the HNSW algorithm, their architectural approaches to building and accessing the graph differ fundamentally, impacting how they handle resource contention and data freshness.

Milvus treats index construction as an **offline, compute-intensive task**. The *IndexNodes* are responsible for pulling sealed segments from object storage and building the HNSW graph in isolation.

- **Build Path:** This separation ensures that the heavy CPU and memory load required for graph construction does not interfere with search or ingestion performance.

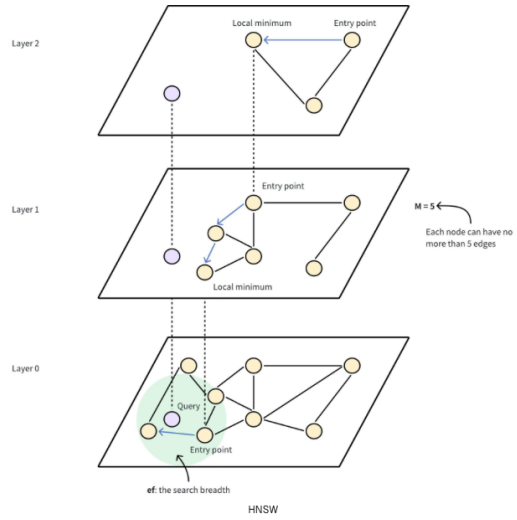


Figure 6: The hierarchical structure of the HNSW graph. Search starts at the top layer (sparse) for fast traversal and descends to the bottom layer (dense) for precision. [4]

- **Search Path:** For retrieval, *QueryNodes* load the completed HNSW index into memory. Milvus optimizes this by using *mmap* to map the index files, but because it is a C++ implementation, it can leverage low-level hardware optimizations like AVX-512 and SIMD instructions to accelerate the distance calculations during the graph traversal.

In contrast, **Weaviate** follows a **tightly coupled, online approach**. The HNSW index is built incrementally as data is ingested into the *MemTable*.

- **Build Path:** As objects are added, Weaviate immediately updates the HNSW graph. While this ensures that data is searchable within milliseconds (Real-time Search), it means that ingestion and indexing share the same CPU and RAM resources, which can lead to performance spikes during high-volume writes.
- **Search Path:** Weaviate's HNSW implementation is written in Go. To overcome the potential overhead of a garbage-collected language, Weaviate utilizes *custom memory pools* and direct pointer arithmetic. Furthermore, the graph is directly linked to the LSM-tree's object store, allowing the search process to seamlessly verify metadata filters (via the Allow List) at each hop of the navigation.

IV. Hybrid Search and Metadata Filtering

Real-world vector search is rarely isolated; it often involves structured constraints (e.g., "Find similar shirts where color is Blue"). To execute these queries efficiently without scanning the entire dataset, both systems implement secondary scalar indexes to facilitate **Hybrid Search**.

A. Architectural Implementation Strategies

*a) **Milvus - Bitset Masking and SIMD Acceleration**:* Milvus separates its scalar indexing from its vector indexing. It supports standard Inverted Indexes and Bitmap Indexes for categorical data.

- **The Masking Process:** Before the vector search begins, the scalar engine generates a **Bitset** (a binary array where 1 represents a valid candidate).
- **Search Integration:** During HNSW traversal, the query engine performs a bitwise AND operation between the candidate nodes and the filter bitset. By leveraging **SIMD instructions**, Milvus can mask thousands of invalid IDs in a single clock cycle, ensuring that the HNSW explorer never visits a node that fails the metadata criteria.

*b) **Weaviate - Roaring Bitmaps and Pointer Locality**:* Weaviate utilizes compressed **Roaring Bitmaps** by default. These are highly efficient data structures that store sets of integers (Object IDs) with high compression and fast intersection speeds.

- **Filtered Traversal:** Weaviate integrates the filter directly into the HNSW greedy search. At each hop of the graph, it consults an "Allow List" derived from the Roaring Bitmap.
- **The Efficiency Gap:** Because Weaviate's data is monolithic (LSM-tree), the pointer that identifies a vector often resides near the metadata. This proximity allows Weaviate to resolve complex filters (e.g., $Price < 50$ AND $Color = Red$) with minimal I/O overhead.

V. Experimental Setup

To ensure the reproducibility of our results, we detail the hardware specifications, the virtualization environment, and the precise software architecture deployed for each vector database.

A. Non-distributed mode setup

*1) **Hardware & Virtualization Environment**:* All experiments were conducted on a local workstation. To simulate a cloud-native deployment while maintaining hardware isolation, the vector databases were deployed as Docker containers running on the Windows Subsystem for Linux 2 (WSL2).

It is crucial to note the resource constraints imposed by the virtualization layer. While the host machine possesses 16 GB of physical RAM, the WSL2 backend was strictly configured to allocate a maximum of **7.6 GB** to the Docker engine.

*2) **Deployment Architecture & Service Components**:* To facilitate a controlled environment, we utilized **Docker Compose** for orchestration. This approach defines the infrastructure as code, ensuring that the interdependent processes (containers) are initialized with consistent networking and storage configurations.

Milvus: For the Milvus deployment, we utilized the "Standalone" configuration. It is important to clarify that "Standalone" in Milvus does not imply a single monolithic binary. Instead, it creates a miniature distributed cluster within a single machine, composed of three distinct **independent applications** (containers) that consume separate resources:

- **Milvus: *The Vector Engine***. This is the core application responsible for vector computations (HNSW traversal).

Table I: System and Environment Specifications

Component	Specification
	Host Machine (HP Pavilion Gaming 15)
CPU	AMD Ryzen 7 4800H (8 Cores, 16 Threads @ 2.90 GHz) *Note: Supports AVX2 Instruction Set (Crucial for SIMD operations)*
Physical RAM	16 GB
Storage	512 GB NVMe SSD (Intel SSDPEKNW512G8H)
OS	Windows 11 Home (Build 26200)
	Virtualization & Docker Environment
Engine Version	Docker Engine v28.5.1
Kernel	WSL2 Linux Kernel (6.6.87.2-microsoft-standard)
Allocated vCPUs	16 Logical Cores
Allocated Memory	7.6 GB

Being stateless, it retains no permanent data. It consumes the majority of the allocated CPU during search operations.

- **Etcd:** ***The Distributed Key-Value Store.*** A separate database process responsible for cluster coordination. It stores metadata (e.g., collection schemas, node states) and configuration while maintaining internal consistency.
- **MinIO:** ***The Object Storage Server - Storage Layer.*** A high-performance file server API compatible with Amazon S3. MinIO acts as the persistence layer. When Milvus flushes data from RAM, it writes binary logs (Binlogs) to MinIO. Crucially, MinIO manages the actual I/O operations on the host's NVMe drive.

Weaviate: In direct contrast, Weaviate was deployed as a single, self-contained container. Weaviate embeds its dependencies directly into the core application. It does not require an external key-value store (like Etcd) or an external object store (like MinIO). Metadata management and disk persistence (LSM Trees) are handled by internal modules and Go routines within the main process. This "Shared Nothing" architecture eliminates the overhead of inter-container communication and allows nearly 100% of the allocated container memory to be utilized for the vector cache and MemTables.

This configuration bypasses the performance penalties associated with mounting Windows host directories (via the Plan 9 protocol), ensuring that disk I/O metrics directly reflect the NVMe speed and are not bottlenecked by the cross-OS file system translation.

B. Distributed mode setup

Milvus: For the distributed-version experiments, Milvus was deployed in cluster mode on a local Kubernetes environment running on a single physical machine. Although the hardware consisted of only one node, Milvus was executed using its full distributed architecture, with separate components (such as proxy, coordinators, query nodes, and data nodes) running as independent services. Object storage was provided via MinIO in standalone mode. Because the experimental environment contained only one Kafka broker, all Kafka topic replication factors were set to 1; higher replication factors would be invalid in a single-broker setup and led to initialization failures. Collections were created with two shards in order to activate Milvus's distributed execution paths and shard-level routing, even though all shards ultimately resided on the same physical machine. This configuration allowed the evaluation of Milvus's distributed pipeline and coordination behavior under controlled, single-node conditions without introducing multi-machine scaling effects.

Weaviate: Weaviate was deployed in a cluster-capable configuration on the same local Kubernetes environment, again using a single physical machine. For consistency with the Milvus experiments, collections were configured with two shards, enabling logical data partitioning and distributed query routing within the Weaviate cluster. Since only one physical node was available, the replication factor was effectively set to 1. This setup enabled the study of Weaviate's distributed behavior and coordination overhead under single-node constraints, while keeping the configuration comparable to Milvus and suitable for fair experimental evaluation.

C. Software - Code

For our experimental evaluation, we built upon the official codebase provided by the Vector-DB-Benchmark repository [7]. On top of this baseline, we developed additional scripts and configuration files tailored to the specific requirements of our experiments. All custom implementations can be found in the directory `/custom` in our github repository [1]. This folder contains all the scripts used in our experimental pipeline, while extensively leveraging existing functionalities from the original repository, such as dataset parsers and utility functions [7]. You can find more information about our code setup and how you can reproduce the expirements in our github repository's README file [1].

D. Datasets

To evaluate performance across varying dimensionalities and search scenarios, we utilized three distinct datasets.

Table II: Dataset Characteristics

Dataset	Dim (d)	Vectors (N)	Filterable Metadata
GloVe-100	100	1,183,514	–
Arxiv-Titles	384	2,200,000	–
H&M-2048	2,048	105,542	✓

Dataset Details & Pre-processing:

- **Data infromation:** Usefull information about each dataset's features are provided in Table II.
- **Data Type:** All vectors were have size of standard 32-bit floating-point numbers (`float32`) to ensure consistent precision.
- **Distance Metric:** The datasets have been built using *Angular* distance (Cosine Similarity). As a result any other type of similarity would result in incosistent recall evaluation during testing.
- **Hybrid Search Scenario:** The H&M dataset introduces the complexity of **Attribute Filtering**. The dataset is derived from real-world clothing product data, making the filtering workload representative of practical e-commerce search scenarios rather than synthetic benchmarks. Queries on this dataset combine vector similarity with categorical constraints (e.g., filtering by "Index Group Name" or "Color"), testing the efficiency of the Bitmaps and Inverted Indexes described in Section IV.

VI. Experiment 1: Ingestion Performance

In this section, we present a comprehensive evaluation of the write capabilities of Weaviate and Milvus. The primary objective was not merely to measure total ingestion time, but to analyze the **scalability** profile of each engine as the volume of stored data increases.

To achieve this, we employed an incremental **loading methodology**. For each dataset (GloVe, Arxiv, H&M), instead of a single bulk load, we captured throughput metrics (Vectors per Second) at distinct checkpoints (e.g., 10,000, 100,000, and Full Dataset). Specifically, we partitioned each dataset so that its size increased linearly, allowing us to examine how the databases respond to a linear growth in data volume. The analysis below dissects these scalability trends, focusing on the trade-offs between raw throughput, data availability (Offline vs. Online indexing), and the impact of metadata overhead.

A. *Throughput vs. Scalability*

In Glove-100 and arxiv-titles datasets, Milvus is overall faster at ingestion across all non-filtered datasets (**2.04× faster** on the large arxiv-titles dataset), making it more suitable for high-volume batch jobs. In contrast, Weaviate exhibits a significant drop in speed (throughput degradation) as the HNSW graph grows, because each new batch must immediately update the neighborhood structure of vectors (***online indexing***). Additionally, the rate of throughput degradation is considerably higher in Weaviate than in Milvus, where it

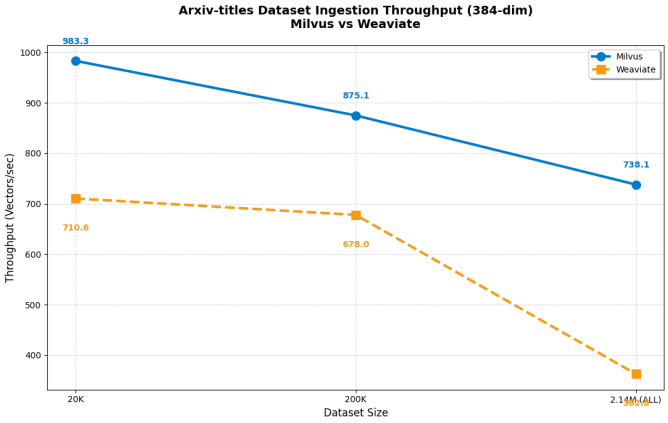


Figure 7: Ingestion Throughput for Arxiv-titles dataset: Milvus VS Weaviate

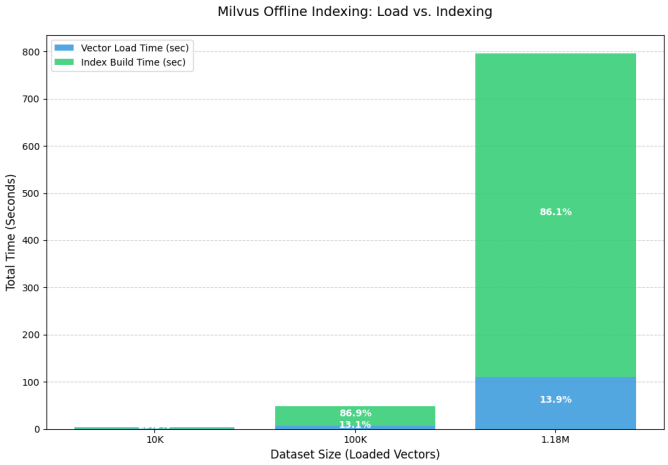


Figure 8: Vectors load VS Indexing time in Milvus

remains almost linear, highlighting the former system's disadvantage when scaling. You can see throughput comparison for *arxiv-titles dataset* at Figure 7. It is also worth noting that the indexing phase in Milvus accounts for the vast majority of the total processing time (**86.1%**), far exceeding the time spent on simple vectors loading(**13.9%**) (See figure 8). This highlights the cost of building the HNSW index, as well as the need for optimization or even the development of new vector indexing algorithms.

B. *The Availability Trade-off: Real-Time vs. Batch*

Despite Milvus's overall speed advantage, Weaviate offers a critical benefit: data becomes searchable immediately upon ingestion. In Milvus (when utilizing offline mode), the user must wait for the index build process to complete. This characteristic makes Weaviate superior for production environments requiring real-time user-facing availability, such as e-commerce platforms where product additions or price updates must be instantly queryable. As observed in Figure 9, Weaviate provides searchable vectors after each batch update.

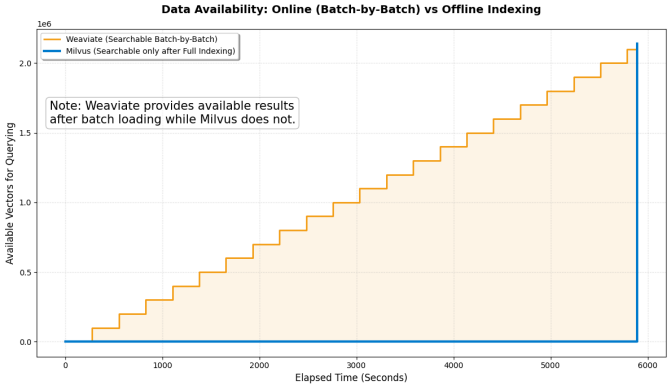


Figure 9: Data availability: Online VS offline Indexing

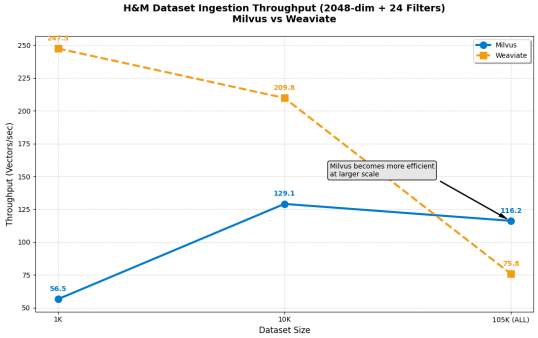


Figure 10: Ingestion Throughput for H&M dataset: Milvus VS Weaviate

C. *Filters Dataset*

Regarding the H&M dataset (filters and 2048 dimensions), we observed an interesting inversion.Weaviate achieves significantly higher throughput on smaller subsets (1K and 10K vectors). This is primarily driven by its LSM-tree (Log-Structured Merge-tree) architecture. During initial ingestion, data is written directly to an in-memory MemTable, allowing for nearly instantaneous writes for small volumes. Furthermore, as a monolithic system, Weaviate avoids the distributed setup overhead (coordinating MinIO, Etcd, and Proxy nodes) that characterizes Milvus's cloud-native design. At this scale, the HNSW graph is small enough that the computational cost of indexing does not yet saturate the CPU.As the volume scales toward the full dataset (105K), Weaviate's performance experiences a sharp decline, dropping to **75.8 vectors/sec**. This is a direct consequence of its Online Indexing approach. Weaviate attempts to perform simultaneous vector and scalar indexing during loading. For every single insertion, the system must update the HNSW graph while concurrently updating 24 different Inverted Indexes (Roaring Bitmaps). With 2048-dimensional vectors, this creates a massive "resource contention" bottleneck, as the CPU struggles to maintain real-time searchability across both high-dimensional vectors and complex metadata filters.

In contrast, Milvus exhibits an upward efficiency trend, with throughput increasing from the 1K to the 10K mark. Milvus overtakes Weaviate at the full scale (105K) for two main reasons:

- **Columnar Storage & Buffering**: During loading, Milvus treats data as "columns" and buffers them as binlogs without building indexes immediately.
- **Batch Indexing:** The heavy computational work is performed in a separate process.By decoupling the ingestion from the indexing, Milvus ensures that the 2048-dimensional complexity does not saturate system resources.

You can see the comparison of the throughputs between the two databases in Figure 10.

D. *Scalar Indexing Overhead*

Additionally, we calculated the percentage time overhead introduced by scalar indexing in both systems by performing ingestion with and without creating scalar indexes on full dataset loading. As shown in Figure 11, Weaviate exhibits a significantly higher overhead from scalar indexing (+109.8%), with ingestion time nearly doubling. One contributing factor is that Weaviate builds inverted indexes for each property concurrently with the HNSW graph; in online build mode, this leads to substantial resource contention in case of loading large amounts of data.

In contrast, Milvus shows a considerably smaller overhead (+25.9%) for the same task. Its architecture allows scalar indexing structures to be built more efficiently after data loading, reducing ingestion-time overhead.

At the implementation level, the two systems follow fundamentally different indexing strategies. Weaviate adopts an indexing approach based on Log-Structured Merge Trees (LSM-Trees) [5]. During ingestion, each inserted object triggers simultaneous updates to both the HNSW vector index [4] and the scalar inverted indexes. When objects contain many filterable properties—such as the 24 metadata fields used in our experiment—the system must update the inverted index separately for each field.

Milvus, on the other hand, does not update scalar inverted indexes per object during ingestion. Instead, scalar values are stored in a columnar format within segments, and the inverted index is constructed later through a batch process, resulting in lower ingestion overhead.

E. *Normalized Efficiency Analysis (MB/s)*

Finally, in order to compare heterogeneous datasets—with different dimensionalities and record counts—we converted throughput from vectors per second to **megabytes per second (MB/s).** This normalizes the fact that a vector in the H&M dataset (2048 dimensions) is about 20 times larger than a vector in GloVe (100 dimensions) and as a result the term "vectors per second" is not fair for comparison. Milvus demonstrates strong stability and an increasing efficiency trend (in MB/s) as data volume grows. Its offline indexing strategy allows the system to make better use of hardware resources

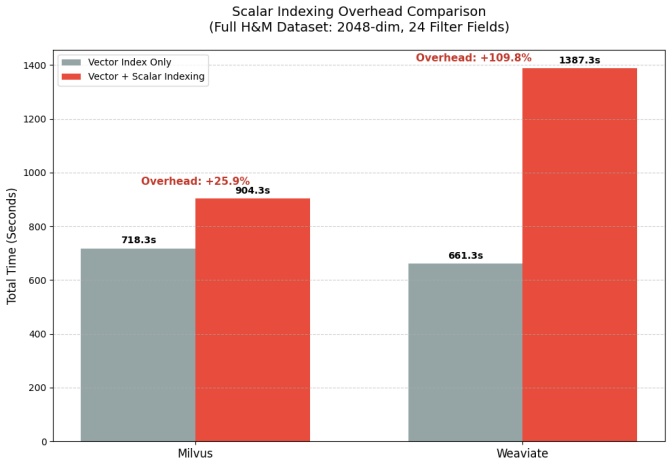


Figure 11: Scalar Index overhead.

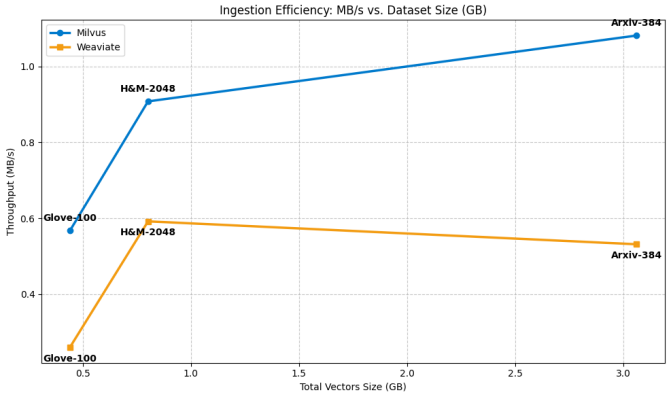


Figure 12: Ingestion efficiency in MBs/second.

(CPU/RAM) at larger scales. Weaviate starts with competitive throughput in MB/s, but shows a clear decline as volume and dimensionality increase. The online indexing approach creates a computational bottleneck, especially in the Arxiv dataset, where MB/s throughput drops to nearly half of its peak value. The gap between the two curves represents the time savings enabled by Milvus' architecture. On large datasets, Milvus consistently processes more than 0.5 MB/s, while Weaviate struggles to maintain rates above 0.5–0.6 MB/s, making Milvus more suitable for high-volume batch loading. You can see the results in Figure 12.

F. Storage Efficiency and Disk Footprint Analysis

Beyond raw performance, storage efficiency is a critical factor for evaluating the two vector databases. Physical disk usage was monitored across three scaling stages (Small, Medium, Full) for each dataset. For Weaviate, usage was measured by printing the overall size of each data collection inside the database, while for Milvus, consumption was aggregated across its three persistence components: the write-ahead logs (Etcd/WAL) and the sealed segments stored in MinIO.

As you can see in Figure 13, the data reveals a consistent trend where Milvus consumes significantly more storage than Weaviate, with expansion ratios ranging from **2.1x to 5.8x** at full scale. This disparity is a direct result of their differing architectural philosophies. Milvus's footprint is split between raw data in Binlogs and the separate HNSW index files, leading to costly data duplication, especially in high-dimensional datasets like H&M. Furthermore, for small-dimension vectors like those in GloVe, the metadata overhead of the MinIO object storage system significantly outweighs the actual data payload.

In contrast, Weaviate achieves superior compactness through its unified LSM-tree architecture, which avoids strict separation between logs and indexes. Its HNSW graph utilizes lightweight internal pointers to reference data stored in SSTables, acting as a navigation map rather than a redundant storage layer. Additionally, background compaction threads continuously merge SSTables to remove obsolete data, ensuring a minimal footprint. While Milvus trades disk space for architectural decoupling and independent scalability of compute and storage, Weaviate's storage efficiency makes it the more viable choice for environments with constrained or expensive SSD resources.

VII. Experiment 2: Parallel Query Evaluation

This section evaluates the search performance of Milvus and Weaviate under varying workloads and configurations. The analysis focuses on two primary dimensions: client-side concurrency (Parallelism) and search depth (EF).

A. Parallel Query Evaluation: Methodology

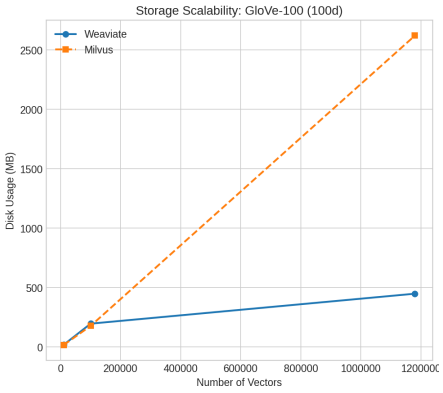
To simulate real-world production environments where a database serves multiple users simultaneously, we introduce the parameter **Parallelism** (P). This parameter represents the number of independent client-side processes—implemented via the Python `multiprocessing` library—that send queries to the database at the same time.

Each process operates as a "client": it submits a query, waits for the complete response, and immediately submits the next. This methodology creates continuous "**stress testing**" on the system resources, allowing us to identify the saturation point of the hardware, specifically the Ryzen 7 4800H processor (8 Physical Cores / 16 Logical Threads).

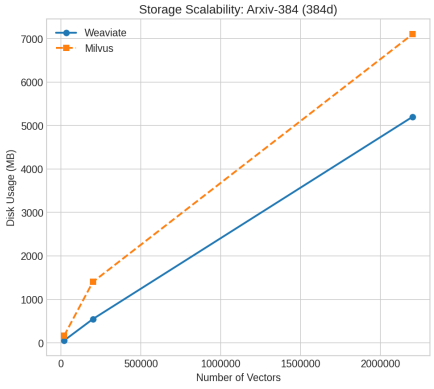
B. Architectural Serving Mechanisms

The handling of concurrent requests reflects the fundamental architectural differences between the two systems:

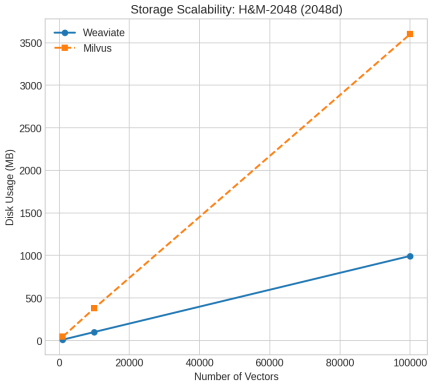
- **Weaviate (Go Runtime):** Incoming gRPC requests are handled by **goroutines**. The Go scheduler dynamically distributes these lightweight threads across CPU cores. The use of immutable SSTables in the LSM-tree allows for lock-free reads, facilitating high concurrency without traditional database locking overhead.
- **Milvus (C++):** The core engine utilizes C++ thread pools and **SIMD (AVX2)** instructions to execute vector distance calculations directly on the hardware, providing high computational density for mathematical operations.



(a)



(b)



(c)

Figure 13: Milvus vs Weaviate: Disk storage over different datasets.

C. *Analysis of Throughput (Queries Per Second) Results*

The results across both datasets indicate a clear performance scaling up to the physical core limit, followed by a regression under extreme concurrency.

a) ***Glove-100 (100 Dimensions)****:* In the Glove-100 experiment (Figure 14), both systems peak at $P = 8$. Weaviate achieves a slightly higher throughput of **436.6 QPS** compared to Milvus's **425.2 QPS**. At this level, each client process is effectively mapped to a physical CPU core. Weaviate's monolithic design provides a slight edge here by avoiding the network and coordination overhead due to its architecture which was analyzed above. However, after $P = 16$, both databases reach a saturation point which is explainable as the machine we conducted the experiments posseses 8 cores.

b) ***Arxiv-384 (384 Dimensions)****:* The trend shifts significantly with the Arxiv-384 dataset (Figure 15). Due to the higher dimensionality, Milvus takes the lead across all levels of concurrency:

- At $P = 8$, Milvus reaches **163.1 QPS** vs. Weaviate's **158.1 QPS**.
- At $P = 16$, Milvus maintains **164.0 QPS**, whereas Weaviate drops to **142.0 QPS**.

This lead is attributed to the increased computational load of 384 dimensions. Milvus's C++ core and SIMD optimizations handle the intensive floating-point math more efficiently than the Go runtime, which begins to struggle with increased memory management and Garbage Collection (GC) overhead.

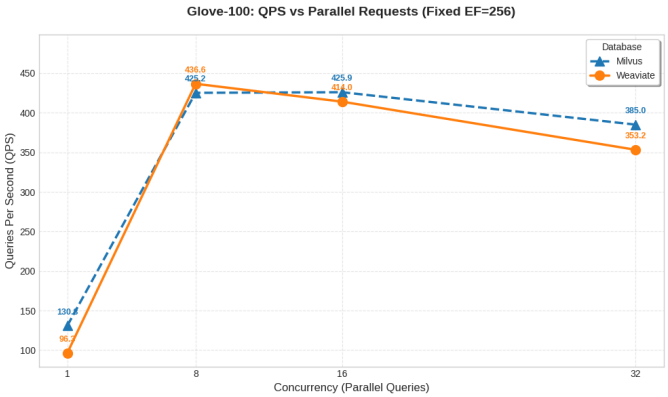


Figure 14: Glove-100 Dataset: QPS VS Parallel Processes

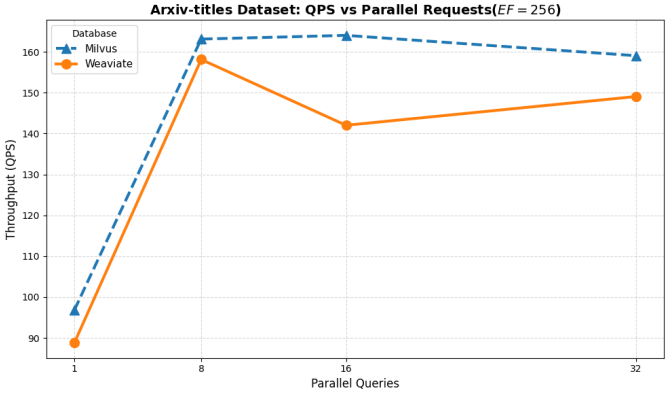


Figure 15: Arxiv Dataset: QPS VS Parallel Processes

VIII. Experiment 3: The Effect of Search Depth (EF) on Throughput and Recall

A. *EF vs Throughput*

The **EF (Examination Factor)** parameter is the primary mechanism controlling the trade-off between search speed and

accuracy in the HNSW algorithm. EF defines the size of the dynamic candidate list maintained during graph traversal at query time. A larger EF allows the algorithm to explore more nodes before selecting the final nearest neighbors, thereby increasing the probability of finding the true nearest vectors (higher recall).

However, this improved accuracy comes at a computational cost. As EF increases, the search process performs more distance calculations and maintains a larger priority queue of candidate nodes. Consequently, the workload becomes increasingly **compute-bound**, since the dominant cost shifts to high-dimensional vector distance computations rather than memory access or graph traversal overhead. Figure 16 illustrates how throughput changes as EF increases (with parallelism $P = 1$).

- **Glove-100:** Milvus maintains a significant performance advantage, with throughput decreasing from **137.05 QPS** at $EF = 128$ to **100.50 QPS** at $EF = 512$. In comparison, Weaviate drops from **98.22 QPS** to **68.40 QPS** (Figure 16). The smaller vector dimensionality (100-D) limits the per-distance computation cost, but the increase in EF still multiplies the number of candidate evaluations, directly reducing QPS.
- **Arxiv-384:** The dimensionality penalty becomes more pronounced, as each distance computation now involves 384 floating-point operations. Even so, Milvus remains faster, achieving **68.6 QPS** at $EF = 512$ compared to Weaviate's **59.6 QPS** (Figure 17). The higher dimensionality amplifies the effect of EF, since each additional visited node requires significantly more arithmetic work.

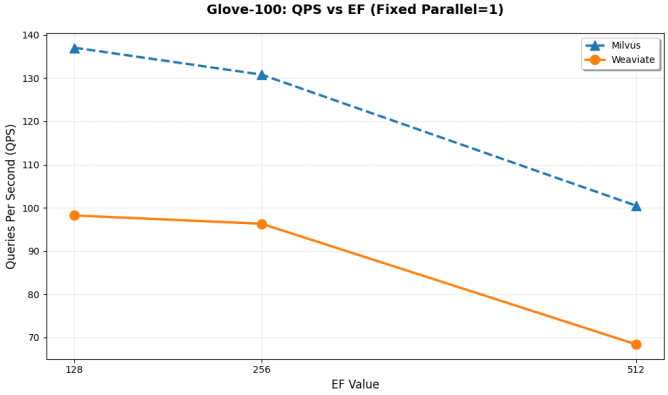


Figure 16: Glove-100 Dataset: QPS VS EF

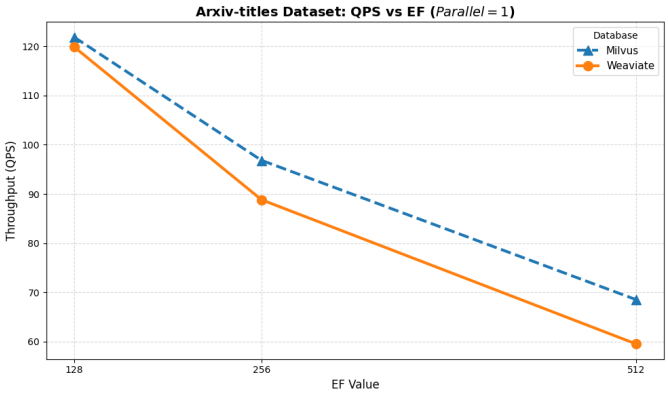


Figure 17: Arxiv Dataset: QPS VS EF

B. *EF vs Recall*

To evaluate the effect of the HNSW search depth parameter (EF) on search quality, we measured recall across increasing EF values for both Milvus and Weaviate. We highlight that the construction parameters such as *M* and *efConstruction* has set to the same values for both databases in order to ensure fair comparison (M=30, efConstruction=360).

*a) **Glove-100 Dataset**:* For the Glove-100 dataset, recall improves steadily as EF increases for both systems (Figure 18):

- **Weaviate:** Recall rises from **0.7951** at $EF = 128$ to **0.9256** at $EF = 512$.
- **Milvus:** Recall increases from **0.8066** to **0.9358** over the same EF range.

This behavior aligns with the theoretical operation of HNSW. A higher EF allows the search algorithm to maintain a larger candidate pool during graph traversal, increasing the probability of discovering true nearest neighbors that may lie outside the immediate greedy search path. As a result, recall improves at the cost of additional distance computations.

*b) **Arxiv-384 Dataset**:* For the higher-dimensional Arxiv-384 dataset, both systems achieve near-perfect recall even at lower EF values (Figure 19):

- **Milvus:** Recall improves marginally from **0.9982** at $EF = 128$ to **0.9993** at $EF = 512$.
- **Weaviate:** Recall increases from **0.9975** to **0.9989**.

The extremely high recall indicates that the dataset structure and index configuration allow the HNSW graph to preserve neighborhood relationships very effectively. In such cases, increasing EF yields diminishing returns, as most true nearest neighbors are already reachable through short graph paths. Moreover, the **latency increases significantly** (See Figure 20). As a result, while higher EF values improve recall by allowing the search to explore more candidate nodes, this comes at the cost of increased latency, as the number of distance computations and graph traversal steps grows proportionally with EF.

Milvus consistently achieves slightly higher recall than Weaviate at all EF values. This suggests that Milvus's graph traversal and candidate management may benefit from implementation-level optimizations.

At the end of this paper, you can find summarizing plots for all the combinations of EF and parallel we experimented.

IX. Experiment 4: Architectural Impact on Metadata Filtering

The H&M-2048 dataset provides critical insights into how storage architecture influences hybrid search performance.

Experimental Methodology The objective of this test is to evaluate how each system handles metadata filters (e.g.,

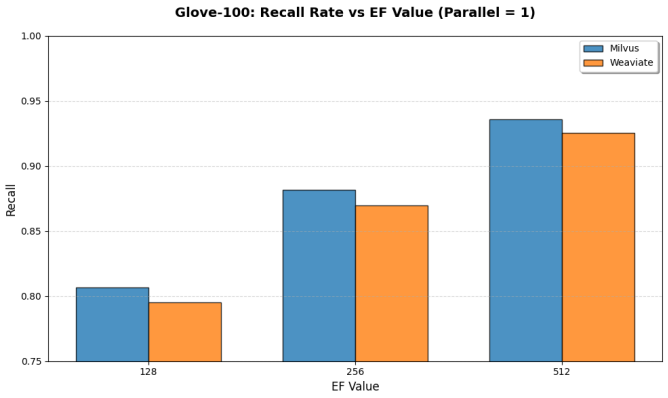


Figure 18: Glove Dataset: Recall VS EF for Milvus & Weaviate

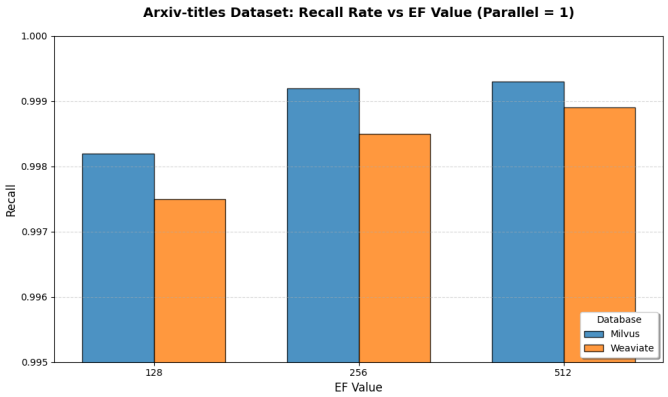


Figure 19: Arxiv Dataset: Recall VS EF for Milvus & Weaviate

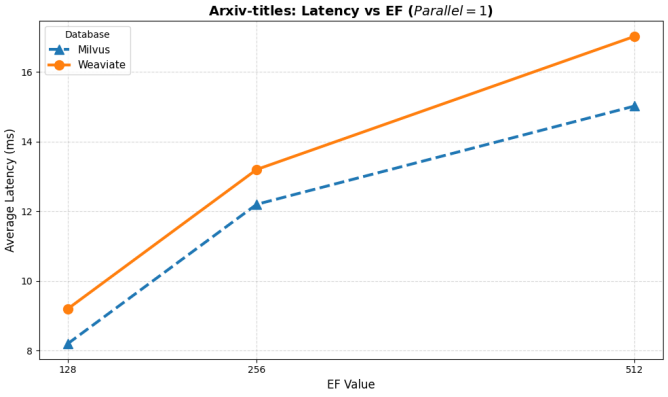


Figure 20: Arxiv Dataset: Latency vs EF

”color=blue”) during a vector search. We analyze two distinct strategies:

*a) **Indexed Filtered Search:** The database utilizes a predefined Scalar Index to quickly identify candidates that meet the metadata criteria.:*

*b) **Unindexed Filtering:** The database performs a raw scan of the metadata to apply filters without the aid of an index.:*

- **Scenario A: Indexed Search (The Milvus Advantage)** When a Scalar Index is present, Milvus achieves **97.4 QPS**, significantly outperforming Weaviate's **69.2 QPS**. This lead is a direct result of Milvus's C++ core and **SIMD-accelerated bitset masking**. Milvus generates in-memory bitsets that function as high-speed masks, allowing the engine to skip irrelevant vectors with minimal CPU cycles. In contrast, Weaviate faces a data retrieval overhead even with Roaring Bitmaps, as it must frequently decompress data from its LSM-tree based storage (SSTables).
- **Scenario B: Unindexed Filtering (The Weaviate Resilience)** The trend inverts in the unindexed scenario, where Weaviate leads with **61.1 QPS** against Milvus's **54.7 QPS**. Weaviate leverages **mmap** and pointers to treat on-disk metadata as an extension of its virtual address space. This allows for direct memory-efficient access, where the engine sweeps through metadata at near-memory speeds by offloading retrieval to the OS Page Cache. Even in a **Standalone** deployment, Milvus maintains its disaggregated microservices logic. The performance bottleneck here arises from the internal ”coordination penalty”: the system must still manage communication channels between the Proxy and the Query Engine, handle data serialization, and aggregate results from different storage segments (segments). This internal message-passing architecture introduces overhead that is absent in Weaviate's single-process, monolithic model.

The findings (Figure 21) underscore that while Milvus is superior for heavily indexed production workloads, Weaviate offers better performance for ad-hoc queries where indexes have not been predefined. This is due to its efficient handling of on-disk metadata through pointer-based access and monolithic locality.

X. Experiment 5: Distributed Version of the Vector Databases

To evaluate the architectural scalability of Milvus and Weaviate, we conducted a distributed benchmark simulating a cluster environment. This section analyzes the overhead introduced by coordination, sharding, and network serialization when transitioning from a standalone binary to a distributed topology. The following expirements that conducted with the distributed modes of the databases were applied only to the medium-size glove-100 dataset because we wanted to avoid crashes due to RAM usage.

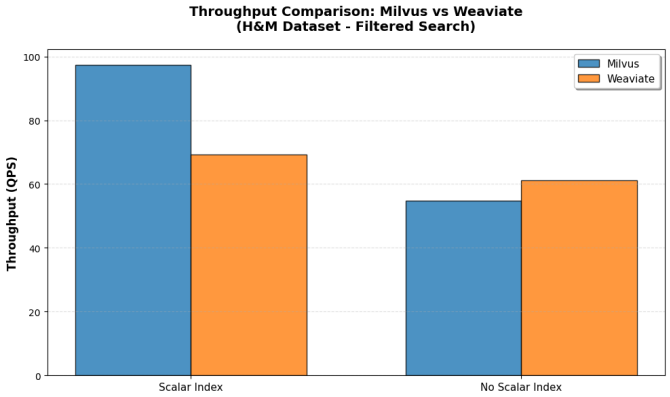


Figure 21: H&M Dataset:Scalar vs No scalar Index QPS

A. *Distributed Ingestion Scalability*

The ingestion tests, shown in Figure 22, illustrate how the throughput of each system changes as the dataset size grows. The results highlight a clear difference in how Milvus and Weaviate handle increasing workloads.

a) Small-Scale Performance (10k Vectors): At a scale of 10,000 vectors, Weaviate is faster, reaching **2,284 vec/sec** compared to Milvus's **1,900 vec/sec**. This is because Weaviate's single-process (monolithic) design has less internal communication to manage at low volumes. Milvus, on the other hand, spends more time coordinating data between its various components (Proxy, Log Broker, and DataNodes), which creates a performance "floor" for small datasets.

b) The Crossover Point (100k Vectors): As the dataset grows to 100,000 vectors, Milvus takes the lead with **2,941 vec/sec**. Milvus becomes more efficient at this scale by grouping data into large blocks (Sealed Segments), which reduces the relative cost of its internal communication.

c) Large-Scale Bottlenecks (1.18M Vectors): At 1.18 million vectors, both systems slow down, but Weaviate's performance drops more sharply to **1,009 vec/sec**. Weaviate's instant (online) indexing becomes increasingly difficult as the graph grows larger. Furthermore, its storage engine must work harder in the background to merge files, leading to "Write Amplification" where the system spends too much time writing redundant data to the disk.

The general decline at this scale confirms that the workload has reached the limit of the available **7.6 GB of RAM**. When memory is full, the systems must wait for the NVMe storage to provide data. This shift from using fast memory to slower disk storage creates "Page Fault" delays, significantly slowing down the ingestion process for both databases.

B. *Standalone vs. Distributed mode:*

The primary observation from the performance diagrams of both **Milvus** and **Weaviate** is that the *Standalone (Non-Distributed)* deployment consistently achieves higher throughput (Queries Per Second – QPS) compared to the *Distributed* deployment (Figure 23, 24). It is important to emphasize that this outcome is a direct consequence of the experimental setup

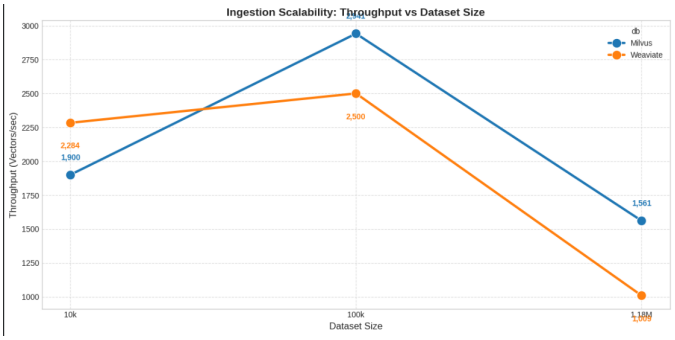


Figure 22: Distributed Version: Ingestion Throughput.

being restricted to a **single physical node**. In an environment where all computational resources are shared within the same machine, the additional architectural complexity of a distributed system becomes an overhead rather than a benefit.

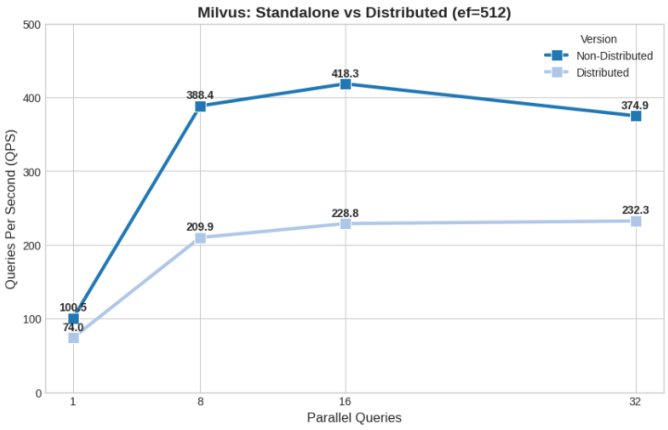


Figure 23: Milvus Distributed vs Non-Distributed version.

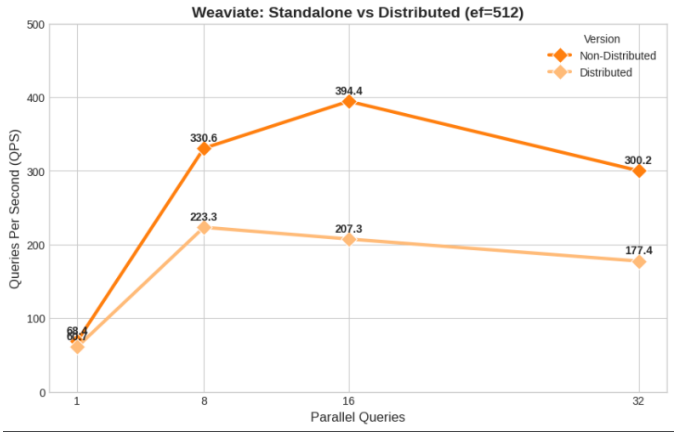


Figure 24: Weaviate Distributed vs Non-Distributed version.

*a) **Architectural Overhead**:* In the Distributed deployment, each query must pass through additional software layers before execution. For instance, in Milvus, the request is processed through components such as the *Proxy* and the *Query*

Node, while in Weaviate, the orchestration layer manages shard coordination and result aggregation. These components are designed to support horizontal scalability across multiple machines, but when deployed on a single node, they introduce extra processing stages without providing any hardware-level parallelism benefits.

As a result, part of the available CPU time is spent on coordination, routing, and merging of partial results rather than on the core vector search computation. This reduces the effective throughput of the system.

*b) **Context Switching**:* Running the Distributed version on a single machine also forces the operating system to manage multiple processes or containers simultaneously for what is logically a single search task. Each shard and coordinator runs as a separate service, leading to increased *context switching*. The CPU therefore spends a non-trivial amount of time switching between processes instead of executing ANN search operations, which further degrades QPS.

*c) **Sharding Without Hardware Parallelism**:* The division of data into multiple shards (two in our case) in the Distributed setup is intended to distribute load across different physical machines. However, when all shards reside on the same processor, there is no true hardware-level parallelism. Instead, the system simply fragments the already limited CPU, memory bandwidth, and cache resources. Under these conditions, the Standalone deployment remains more direct, lightweight, and computationally efficient.

XI. Experiment 6: Image Anomaly Detection Using Vector Databases

A. Dataset

To simulate an industrial visual anomaly detection pipeline, we constructed a mini vector benchmark using the MVTec AD (MVTec Anomaly Detection) dataset [8]. This dataset contains high-resolution images of industrial objects, with separate normal (defect-free) and anomalous samples. We built a feature-space anomaly detection system by extracting semantic embeddings from images using the DINO self-supervised vision transformer as a feature extractor. Each image was converted into a fixed-length vector representation with dimension 384. We then organized these embeddings into JSON datasets containing:

A) the vector representation

B) a categorical filter (e.g., wood, carpet, capsule) corresponding to the MVTec class

Only embeddings from defect-free (“normal”) images were inserted into the vector databases, simulating the common industrial assumption that historical data mostly consists of normal production samples.

B. Queries

For each embedding belonging to an anomalous image was performed k-nearest neighbor (k-NN) search against the database containing only normal vectors. Search was optionally constrained using the corresponding class filter (e.g., querying only within wood vectors). The database returned the k most similar stored vectors based on vector distance and then cosine similarity was computed between the query vector and each of the retrieved neighbors. Only neighbors exceeding a predefined cosine similarity threshold were retained. The intuition is that anomalous samples should be less similar to the distribution of stored normal embeddings, resulting in fewer high-similarity matches. We keep parallel=1 and ef=256 for this experiment. You can see the pipeline for this experiment in Figure 25.

Table III: Image Anomaly Detection Experiment: Query performance comparison

Database	Queries	Throughput (queries/s)	Avg. Latency (ms)
Milvus	1,000	180	5.6
Weaviate	1,000	120	8.3

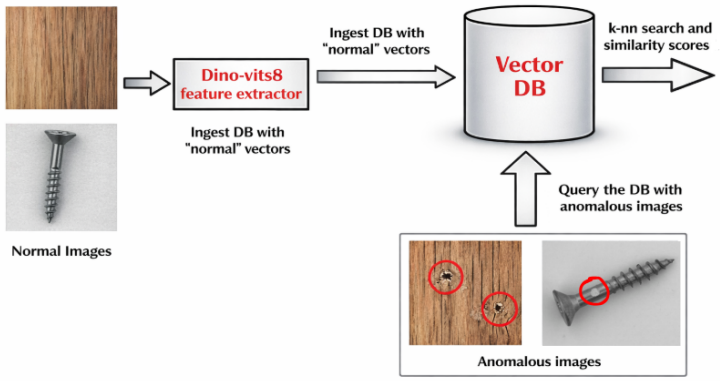


Figure 25: Image Anomaly Detection with vector databases pipeline.

C. Results

This specific experiment was conducted with **the goal of exploring image-based workflows and evaluating the application of vector search in production lines that require high live-time throughput. It was not designed as a stress test, since the dataset we constructed is relatively small in scale, consisting of approximately 1,500 vectors stored in the database (derived from normal images) and around 1,000 anomaly vectors used as queries**. Moreover, feature extractor dino-vits8 returns vectors that are near each other in the vector space because the model was not trained specifically in this dataset and its functionality is just to project similar images to a vector space. As a result the similarities we got between ”good-stored vectors” and ”anomalous” ones were not realistic. The corresponding experimental results can be found in Tables III , IV. Milvus continues to outperform in terms of throughput during the database loading phase, as its offline indexing strategy allows data to be ingested first and indexed later in larger, more efficient batches. In contrast, Weaviate follows an online indexing approach, where vector and metadata indexes are updated incrementally during ingestion. While this enables immediate queryability, it intro-

Table IV: Image Anomaly Detection Experiment: Ingestion performance comparison

Database	Vectors Inserted	Throughput (vectors/sec)	Total Ingest Time (sec)
Milvus	1,500	1,200	1.25
Weaviate	1,500	550	2.73

duces additional per-insert overhead, resulting in lower loading throughput compared to Milvus under the same experimental conditions. It is worth noting that Weaviate also provides the option to perform image vectorization through built-in modules, enabling automatic embedding generation during data ingestion.

XII. Conclusion

This work presented an experimental comparison between **Milvus** and **Weaviate** open source vector databases, demonstrating that architectural design choices influence performance.

Milvus proved superior in large-scale ingestion and high-dimensional search due to its disaggregated architecture and offline indexing strategy. By separating data loading from index construction, it minimizes resource contention and scales more efficiently as dataset size and vector dimensionality increase. Its SIMD-optimized C++ execution further enhances performance in compute-intensive similarity calculations.

Weaviate, on the other hand, excels in real-time data availability. Its online indexing ensures that newly inserted vectors become immediately searchable, which is especially valuable in user-facing and dynamic applications. However, this tight coupling between ingestion and indexing leads to throughput degradation at larger scales.

Under concurrent query workloads, both systems scale effectively up to the hardware core limit. Milvus gains an advantage in higher-dimensional settings due to lower-level compute optimizations, while Weaviate benefits from lower internal overhead in lighter workloads. Experiments on the *EF* parameter confirmed the classic recall–latency trade-off: higher *EF* improves recall but reduces throughput and increases latency. Milvus consistently maintained slightly higher recall across configurations, indicating that implementation details influence not only performance but also accuracy.

Storage analysis revealed another important trade-off. Milvus consumes significantly more disk space because it stores raw data and indexes separately, whereas Weaviate's LSM-based design provides better storage efficiency.

Finally, distributed-mode experiments executed on a single physical node showed lower performance than standalone deployments for both systems. This confirms that cloud-native, distributed architectures provide tangible benefits only when true multi-node scaling is available.

Future research could extend this study to true multi-node distributed deployments, larger billion-scale datasets, and hybrid search workloads combining dense vectors with complex filtering, in order to better evaluate how architectural differences impact performance under production-scale conditions.

References

[1] Project's Github Repository Link: https://github.com/giorgostzamouranis/Vector-DB-Benchmark-milvus-weaviate

[2] Milvus Documentation. [Online]. Available: https://milvus.io/docs

[3] Weaviate Documentation. [Online]. Available: https://docs.weaviate.io/weaviate/concepts

[4] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2018. Available:

[5] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[6] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser, "Consistently faster and smaller compressed bitmaps with Roaring," *Software: Practice and Experience*, vol. 46, no. 11, pp. 1547–1569, 2016.

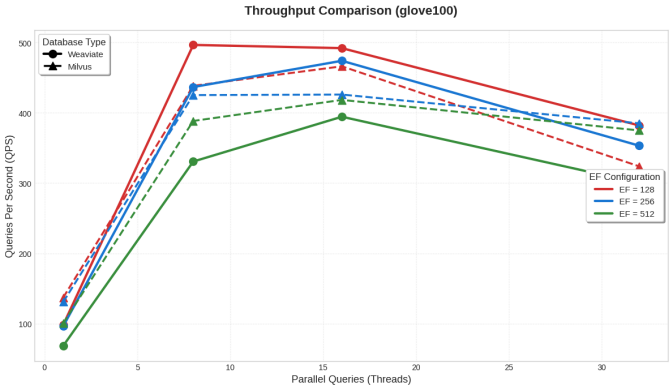
[7] Vector-DB-Benchmark Github Repository: https://github.com/qdrant/vector-db-benchmark

[8] MVTec anomaly detection dataset https://www.mvtec.com/company/research/datasets/mvtec-ad

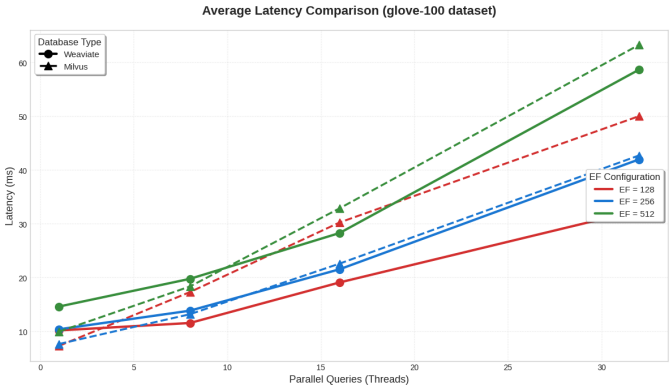
[9] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu, K. Yu, Y. Yuan, Y. Zou, J. Long, Y. Cai, Z. Li, Z. Zhang, Y. Mo, J. Gu, R. Jiang, Y. Wei, and C. Xie, "Milvus: A Purpose-Built Vector Data Management System," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, 2021, pp. 2614–2627. Available:

[10] V. Bansal, "A deep dive: What is LSM tree?", Curious Engineer (Substack), Aug. 18, 2024. [Online]. Available: https://vivekbansal.substack.com/p/what-is-lsm-tree

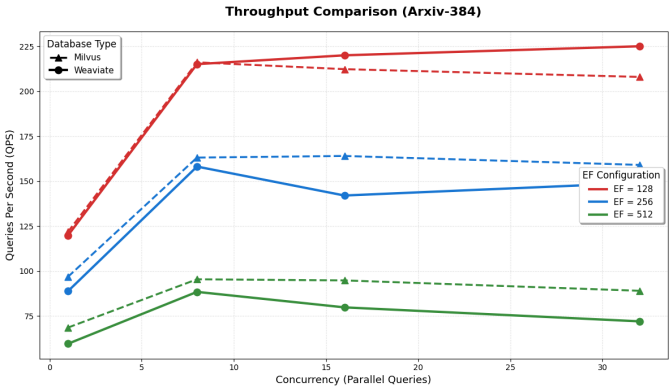
[11] GloVe-100-Angular dataset, TensorFlow Datasets. Pre-trained Global Vectors for Word Representation (100-dim GloVe embeddings) for nearest neighbor search, consisting of 1,183,514 database vectors and 10,000 test vectors. [Online]. Available: https://www.tensorflow.org/datasets/catalog/glove100_angular



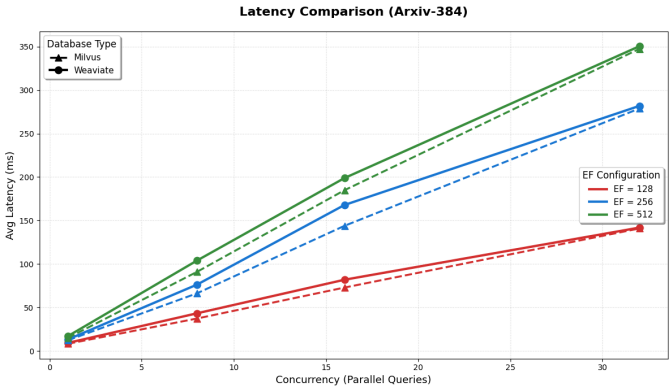
(a) Glove-100 Dataset: Throughput vs EF vs Parallel Requests



(b) Glove-100 Dataset: Latency vs EF vs Parallel Requests



(c) Arxiv-titles Dataset: Throughput vs EF vs Parallel Requests



(d) Arxiv-titles Dataset: Latency vs EF vs Parallel Requests

Figure 26