ChatGPT

Evaluation of Hybrid Local-First Object Storage Solutions

Introduction and Context

Modern AI/data pipelines often manage a mix of large media files, model artifacts, backups, and generative outputs. A **self-hosted, local-first object storage** is crucial for low-latency access and data sovereignty, while still allowing remote access or cloud synchronization for collaboration and off-site durability. The chosen object store must seamlessly integrate with the user's existing tech stack – which includes PostgreSQL, Neo4j, Qdrant, Redis (with Lua), Dramatiq, and RabbitMQ – and support **encryption, fine-grained access control, and robust Python SDK/CLI tools** for development. Below we examine a broad range of open-source and hybrid-compatible object storage solutions (beyond just MinIO and SeaweedFS) against these requirements, comparing their durability, performance, deployment, maturity, and integration capabilities.

Requirements and Key Criteria

The object storage solution should:

- *Versatility:* Handle **all file types** (images, video, audio, text, binary blobs, database backups, AI model outputs, etc.) without format restrictions.
- *Local-First & Remote Access:* Primarily operate on local infrastructure for speed, but allow **remote LAN access** and **optional cloud sync/replication** to off-site or cloud storage (hybrid cloud support).
- *Security:* Provide **built-in encryption at rest** (server-side encryption) and in transit (TLS), plus **fine-grained access control** (multi-tenant support, user/bucket policies or ACLs) for secure sharing.
- *Developer Tools:* Offer a **strong Python SDK** (or S3 API compatibility for use with `boto3`) and user-friendly **CLI tools** for management and scripting.
- *Integration:* Optionally hook into messaging or databases – e.g. support event notifications to **RabbitMQ/AMQP**, Redis streams, or webhooks, and be able to work in concert with relational DBs or caches if needed (for example, for metadata storage or triggers).

With these criteria in mind, we evaluated several leading open-source object stores suitable for self-hosting: **MinIO**, **Ceph (RADOS Gateway)**, **SeaweedFS**, **OpenIO**, **Zenko (Scality CloudServer)**, and **Garage (Deuxfleurs)**. Each of these systems is S3 API-compatible to varying degrees, but they differ in architecture and feature sets. We also note any emerging alternatives that fit the hybrid local-cloud paradigm. The sections below compare these solutions on durability, performance, scalability, ecosystem maturity, and integration with the user's stack.

Candidate Solutions Overview

- **MinIO:** A high-performance, distributed object store written in Go. It is S3-compatible and known for its simplicity and speed. MinIO supports erasure coding for data protection and offers advanced features like bucket replication, object locking, and event notifications.
- **Ceph (RADOS Gateway):** Ceph is a mature distributed storage platform; its RADOS Gateway (RGW) provides an S3-compatible object storage interface on top of Ceph's reliable distributed object store (RADOS). Ceph is highly scalable and feature-rich (versioning, tiering, encryption, etc.), albeit with higher complexity.
- **SeaweedFS:** A lightweight distributed file system and object store inspired by Facebook's Haystack design. SeaweedFS emphasizes simplicity and speed for small files, uses a master/volume server architecture, and provides an S3 API gateway. It supports both replication and erasure coding, as well as tiering of older data to cloud storage [1] [2].
- **OpenIO:** An open-source object storage platform with a unique "Conscience" technology for dynamic data placement. OpenIO supports the S3 and Swift APIs, multi-tenancy, and flexible **storage policies** (e.g. mix of replication and erasure coding based on object size) [3] [4]. It is designed for easy scaling with no single point of failure and includes advanced features like lifecycle rules and versioning [5] [6].
- **Zenko (Scality CloudServer):** An open-source **multi-cloud data controller** by Scality. Zenko's CloudServer provides an S3 interface that can *transparently replicate or tier data to multiple backends/clouds**** [7] [8]. It enables a unified namespace over on-prem and cloud storage locations, with policy-based replication (via its Backbeat component). Zenko focuses on workflow automation across clouds, and is typically deployed in Kubernetes for high availability.
- **Garage (Deuxfleurs Garage):** A newer, lightweight distributed object store written in Rust, aimed at **self-hosters and small clusters**. Garage is S3-compatible (subset) and uses a decentralized **DHT-based storage** with configurable replication factor [9]. It includes built-in deduplication and compression [10], and is designed to tolerate geo-distributed nodes (even on low-end hardware or mixed networks) [11]. Garage prioritizes simplicity (single binary, no external DB) [12], though it lacks some advanced S3 features (e.g. object versioning, bucket policies) [13].

Feature Comparison of Solutions

The table below summarizes how each solution meets the key feature requirements:

Solution	Redundancy & Durability	Encryption & Access Control	Hybrid Cloud Support	Dev Tools (SDK/CLI)	Event/ Message Integration
MinIO	Erasure coding across nodes/disks (configurable parity) + optional multi-cluster replication [14]. No SPOF in distributed mode.	TLS in transit; server-side encryption at rest with KMS support [15] [16]. IAM-like policies (RBAC) for users/groups [17] [18].	**Yes:** Tiering rules can move objects to remote S3 or cloud storage [19]; multi-site active-active replication supported.	**Yes:** Full S3 API (use Boto3) + official Python SDK [20]. Rich CLI (`mc`) for admin and automation [21].	**Yes:** Built-in bucket notifications to AMQP (RabbitMQ), Redis, PostgreSQL, webhooks, etc. [22] [23] for event-driven workflows.
Ceph RGW	Pluggable: default 3x replication or erasure coding pools (CRUSH placement) for high durability. Multi-site cluster replication available for geo-redundancy [24].	TLS support; encryption at rest via integration with Vault, Barbican, etc. [25]. Supports S3 bucket policies, ACLs, and user quotas (multi-tenant via Ceph users or Keystone) [26].	**Yes:** Multi-cluster **zone replication** and **cloud tiering** (objects can transition to AWS S3 or tape via lifecycle) [27] [28].	**Yes:** S3 API (compatible with AWS SDK/boto3). Admin CLI (`radosgw-admin`) and Ceph dashboards.	**Yes:** Bucket notifications to HTTP, **AMQP (RabbitMQ)**, or Kafka on object events [29]. Also supports AWS S3-like PubSub API for notifications.

Solution	Redundancy & Durability	Encryption & Access Control	Hybrid Cloud Support	Dev Tools (SDK/CLI)	Event/ Message Integration
SeaweedFS	Volume replication (configurable per volume) for hot data; background **erasure coding** for colder data to save space [2]. Master/volume architecture with no single metadata bottleneck (O(1) disk seeks) [30] [2].	Supports TLS. **Encryption at rest** optional: data on volume servers can be AES-encrypted with keys managed in the Filer DB [31]. Basic auth and S3 key-based access (no native IAM; bucket policies not fully supported [32]).	**Yes: Tiered storage** – can offload old volumes to S3/ cloud for "warm" storage [1]. Cross-cluster replication (active-active or active-passive) via changelog for multi-site sync [33] [34].	**Partial:** S3 API subset (works with AWS CLI, s3cmd) [35]; not all S3 features (no WebUI by default). Has a `weed` CLI for admin. No official SDK, but community client libraries exist.	**Limited:** No built-in S3 event webhook (as of now). However, Filer uses external DBs (e.g. PostgreSQL, Redis) [36] which could be tapped for custom event logic, and SeaweedFS changelogs can be consumed for events [37].

Solution	Redundancy & Durability	Encryption & Access Control	Hybrid Cloud Support	Dev Tools (SDK/CLI)	Event/ Message Integration
OpenIO	Flexible **storage policies**: per-container choice of N-way replication vs erasure coding (e.g. small files replicated, large files erasure-coded) [3] [38]. No single point of failure; background rebalancing and integrity checks ensure durability [39] [40].	TLS support. **Server-side encryption** (SSE-S3 and SSE-C) supported at the S3 gateway [41]. **Identity & Access Management** with multi-tenant accounts, container ACLs and bucket policies (supports S3-style IAM API) [42] [43].	**Partial:** Primarily on-prem, but supports multi-datacenter distribution and async replication to other OpenIO clusters (geo-replication) [44]. (Cloud tiering is not native in open source; usually used for heterogeneous local storage tiers like SSD/ HDD/tape [45].)	**Yes:** S3 API compatibility (works with boto3/CLI). Native Python API and CLI tools (`oio` CLI) are provided [46]. Integration with OpenStack Swift API and others.	**Partial:** No native S3 event notification mechanism exposed. However, OpenIO has an event-driven framework internally (e.g. was used for video transcoding triggers) [47] and supports monitoring via Prometheus. Likely requires custom integration for external notifications.

Solution	Redundancy & Durability	Encryption & Access Control	Hybrid Cloud Support	Dev Tools (SDK/CLI)	Event/ Message Integration
Zenko	**Depends on backend:** Zenko itself stores object data on configurable backends (e.g. cloud buckets or Scality RING). For local storage it can use a file system or an integrated Scality RING for durability (replication factor configurable on RING). Zenko's value is in coordinating replication between sites [7] [8], rather than providing its own storage engine.	TLS for all endpoints. Supports SSE encryption (client-provided keys and KMS) for data before replicating to target clouds [48]. **IAM support** exists (it can manage multi-user accounts via its Vault component) [49] [50]. Bucket-level policies need configuration of its internal IAM system.	**Yes:** This is Zenko's core strength. It offers **Cross-Cloud Replication** and data movement rules: e.g., write once to local storage and Zenko's **Backbeat** will replicate to Amazon S3, Azure Blob, Google Cloud, etc. [51] [52]. Supports bidirectional sync and lifecycle across heterogeneous backends.	**Yes:** Full S3 API compatibility (use AWS SDK/CLI). No separate Zenko SDK needed. Admin UI ("Orbit") and CLI are available for managing replication workflows [53].	**Partial:** Zenko tracks internal events via **Backbeat** (which uses Kafka/Queue internally for replication and lifecycle) [54], but does not directly expose a simple S3 event notification to external systems. Instead, one could consume logs or use custom extensions. (Zenko's focus is on replication rather than triggering user functions on events.)

Solution	Redundancy & Durability	Encryption & Access Control	Hybrid Cloud Support	Dev Tools (SDK/CLI)	Event/ Message Integration
Garage	**Replication**: Configurable `replication_factor` (default 3) for copies across nodes [55]. Supports multiple consistency modes (strong or eventual) depending on latency needs [56]. No central metadata – uses distributed hash table, and automatic repair processes for failed nodes [57] [9]. (No erasure coding yet; relies on replication & deduplication for efficiency.)	**Encryption:** TLS *between nodes* is enforced (all inter-node RPC traffic encrypted) [58]. **At-rest:** No automatic SSE-S3 (they recommend OS-level disk encryption or client-side encryption) [59] [60]. Supports SSE-C (client-supplied keys for encryption per request) [60]. **Access control:** Multi-tenant users with access/secret keys; Garage has an admin API to create users and manage credentials [61]. Lacks fine-grained bucket policies/ACL – a simplified model compared to AWS IAM [13].	**Partial:** Designed for self-hosted **geo-distributed clusters** (e.g. nodes at home and remote sites) – it can place replicas in different locations and is resilient to high-latency links [11]. However, it doesn't natively sync to public cloud storage; it's more for federating your own nodes. (Cloud backup would require manual sync or mounting Garage as a target in rclone, etc.)	**Yes:** S3-compatible API (works with common S3 clients). A lightweight Python library is available for Garage, and it works with tools like s3fs, restic, etc. [62] [63]. A command-line tool `garage` is provided for cluster management [64].	**No:** Garage does **not yet support bucket event notifications** to external systems (no S3 event API). It focuses on storage, so any event handling (e.g. post-upload hooks) would need to be implemented externally (e.g. polling or using the planned K2V metadata store for custom triggers).

※ *Table Legend:* **Hybrid Cloud Support** denotes ability to sync or replicate data to external cloud storage. **Event/Message Integration** refers to out-of-the-box support for sending object event notifications to message queues, webhooks, or other services.

As seen above, all candidates support basic S3 operations and a range of security features, but they differ in completeness. **MinIO and Ceph** stand out with the most comprehensive feature sets (advanced IAM, encryption, notifications), whereas **SeaweedFS and Garage** trade some API completeness for simplicity and performance. **OpenIO** offers a middle-ground with enterprise features like versioning and flexible policies, and **Zenko** targets specialized multi-cloud workflows.

Durability and Redundancy Features

Durability is paramount for storage – each solution uses either **replication, erasure coding, or both** to guard against data loss:

- **MinIO:** Protects data with high-efficiency **erasure coding**. In distributed mode, MinIO splits objects into data and parity shards across multiple drives/nodes; it can tolerate the loss of up to half the drives or nodes in a group [65] [66]. This provides RAID-like resilience at the object level. MinIO also supports **bucket replication** to another MinIO instance (or even to AWS S3) for disaster recovery [67]. However, note that MinIO clusters have a fixed set of nodes for erasure coding – scaling out may require adding in *whole new sets* of nodes to maintain balance. In practice, durability is very high as long as the erasure coding parameters (e.g. 4 data + 2 parity disks, etc.) are chosen appropriately.

- **Ceph (RADOS Gateway):** Ceph is known for rock-solid durability. It uses the CRUSH algorithm to distribute object data across many OSDs (object storage daemons) with either **N-fold replication** or erasure coding. A typical Ceph setup keeps **3 replicas** of each object on different nodes by default. Even more space-efficient erasure codes (e.g. 8+3 or 4+2 schemes) can be used for large volumes of data. Ceph's design has no single point of failure and can withstand multiple node or disk failures by design [24] [40]. Ceph RGW also supports **multi-site replication**: you can configure federated zones where buckets are asynchronously replicated to another Ceph cluster or to tape/cloud via the **Cloud Sync/Transition** modules [27] [28]. This makes Ceph very flexible for durability across data centers. The trade-off is complexity and overhead: Ceph's strong consistency and background recovery processes can consume significant resources.

- **SeaweedFS:** SeaweedFS employs a mix of replication and optional erasure coding to protect data. By default, when you write through the S3 gateway, files are stored in **volume servers** with a replication count (e.g. 2 or 3 copies) as configured. Seaweed's master node quickly assigns a file to a volume and copies it to replicas as needed [68] [69]. For improved storage efficiency, SeaweedFS offers an **erasure coding mode** where older or less-frequently accessed volumes can be erasure-coded (splitting into shards with parity) instead of fully replicated [70] [71]. This way, "hot" recent data can be kept in faster replicated form, and "cold" data can be stored with less redundancy. This two-tier approach gives a balance of performance and durability. Additionally, SeaweedFS can perform cross-cluster replication: it keeps a **changelog** of metadata operations that can be replayed to another cluster for backup or active-active setups [33] [34]. With a modest number of nodes, SeaweedFS is able to survive failures without downtime, though it doesn't have the years of large-scale durability track record that Ceph has.

- **OpenIO:** OpenIO provides very customizable durability policies. You can define **storage policies** for each data container or even per object based on size or other criteria [72] [38]. For example, OpenIO can be set to **replicate small objects** (for high IOPS and low overhead) while using **erasure coding for large objects** to save space [3] [38]. These policies are dynamic and can be changed on the fly.

OpenIO's distributed architecture has no central metadata node – it uses a distributed directory with multiple meta levels and a "Conscience" mechanism where each node knows which nodes are least loaded [73] [74]. This means adding or removing nodes automatically rebalances data without manual intervention [75] [76]. All critical metadata (namespace directories, etc.) can themselves be replicated across nodes synchronously for safety [4]. **Integrity checking** is built-in – the system periodically checks and heals any corrupted chunks [77]. OpenIO is also **geo-distributed capable**, supporting multi-datacenter replication or stretching a cluster across sites for redundancy [44]. Overall, OpenIO aims for durability comparable to Ceph in theory, with more manual tuning available. It is used in production for multi-petabyte setups, indicating proven redundancy.

- **Zenko:** Zenko's durability depends largely on what backend you configure as the "primary storage." Zenko itself doesn't store data long-term except optionally in a **"transient source"** mode (using a Scality RING or local filesystem) [78] [79]. In a typical Zenko deployment, you might designate your on-prem storage (which could be Ceph, MinIO, a NAS, or Scality RING) as the primary, so that system's durability mechanisms apply (replication/erasure as provided by that backend). Zenko then **replicates objects to secondary targets** (like AWS S3, Azure, etc.) as per policy. This means durability is enhanced by having multiple copies in different clouds. If Zenko's *transient storage* is used, it leverages a Scality RING (which itself does replication across nodes) [80]. Zenko is typically run in a **3+ node Kubernetes cluster** for HA of its services (MongoDB, Kafka, etc.), ensuring no single point of failure in the control plane [81] [82]. In short, Zenko can give **multi-location redundancy** (a local copy + cloud copies) but by itself doesn't do erasure coding – it relies on underlying storage engines for low-level data protection.

- **Garage:** Garage's approach to durability is to maintain multiple copies of data across nodes in a decentralized way. You configure a **replication factor (RF)** in the Garage cluster config (the default recommended RF=3 means three copies of each block stored on three distinct nodes) [9] [83]. On upload, data is chunked and each chunk is hashed and stored on a subset of nodes such that the RF is satisfied. A distributed algorithm ensures chunks are placed to meet geo-redundancy constraints if specified (e.g. ensuring copies reside in different locations) [84] [85]. If a node goes down, Garage's background **repair service** will detect missing chunks and re-replicate from surviving copies to new nodes [86] [87]. There is **no master coordinator or consensus leader** in Garage; this lack of a single coordinator (no RAFT) means each node can operate independently for most tasks [88]. It improves performance but still keeps consistency by clever use of hashing and eventual consensus on placement. The downside is **no erasure coding** yet – so Garage uses more raw storage due to full replication (mitigated by block-level deduplication to avoid storing duplicate content twice) [10]. In practice, Garage should handle disk or node failures smoothly if RF$\geq$2, but being a younger project, it doesn't have the long-haul track record of Ceph or the formal proofs of durability. It is designed to run on unreliable nodes (even Raspberry Pis), so resilience is a key goal of the project.

Summary: All solutions can provide strong durability within a local cluster. **Ceph** leads in proven enterprise-grade reliability (many large deployments) and fine-grained durability tuning. **MinIO, OpenIO, and SeaweedFS** all achieve no-SPOF redundancy; MinIO via erasure coding (great space efficiency), OpenIO via a mix of replication/EC per policy, and SeaweedFS via replication and optional EC tiering. **Garage** uses replication and has innovative repair mechanisms, suitable for small clusters including geographically separated nodes. **Zenko** leverages other storage engines for durability but excels in making sure data ends up copied to multiple clouds or sites (which can be the ultimate durability if one site is lost).

Performance on Local Nodes

Performance can mean high throughput for big files, low latency for small files, or ability to handle many concurrent ops. We consider each:

- **MinIO:** MinIO is widely recognized for its **high performance** in throughput-oriented workloads. It is written in Go with careful optimization for modern hardware (SSE instructions, etc.), and can saturate 10 GbE or 40 GbE networks given sufficient backend disk speed. In benchmarks, MinIO can achieve **hundreds of MB/s to GB/s** of throughput, rivaling or exceeding AWS S3 in many cases. It has a small I/O stack (uses direct posix writes or filesystem for storing objects) and keeps metadata in memory or small files, avoiding heavy databases in the datapath. For concurrent operations, MinIO's design is scale-out: you can run multiple nodes/shards to increase aggregate throughput. Its erasure coding does introduce some CPU overhead, but this is mitigated by using hardware acceleration and by operating at the stream level. MinIO also shines for **AI/ML use cases** – the project touts its ability to feed data to GPU clusters at high rates [89]. Latency for small file operations (metadata-heavy ops) is decent, but not as low as a pure in-memory system; still, MinIO's S3 API compatibility ensures typical latencies in the low tens of milliseconds for small object GET/PUT on a local network. **Bottom line:** MinIO offers **excellent raw throughput and good concurrency scaling**, making it suitable for streaming large media or big datasets to AI models.

- **Ceph RGW:** Ceph's performance is often characterized as **"eventually good, but initially complex"**. Raw throughput: Ceph can be very scalable – adding more OSDs (Object Storage Daemons) and more gateways can linearly increase throughput, and Ceph is used in some of the largest storage clusters. However, on a small scale (few nodes), Ceph has more overhead than simpler systems. Every object write may involve replication across the network, and Ceph's strong consistency (with acknowledgments from peers) can add latency. For small random reads/writes, Ceph can be less snappy than MinIO or SeaweedFS because it might need to consult its OSD map or do extra work if data is erasure-coded. Recent efforts like **Crimson OSD (seastore)** aim to significantly improve Ceph's latency and IOPS by rearchitecting its internals [90], but as of 2025, a default Ceph (Bluestore-based) is optimized for *throughput and reliability over raw low-latency*. That said, Ceph can drive a lot of throughput with enough parallelism – it's not uncommon to see Ceph clusters delivering multi-GB/s read/write when spread across many disks. It can handle **billions of objects**, though memory usage will grow accordingly. For AI workflows requiring lots of small object reads (e.g. millions of image files), Ceph might need careful tuning (like using BlueStore on NVMe, etc.) to avoid latency spikes. In summary, **Ceph performs well at scale** but may be **overkill for smaller setups**, where its latency could be higher than simpler solutions due to internal overhead.

- **SeaweedFS:** SeaweedFS is explicitly optimized for **fast access to small files and metadata**. Its architecture eliminates the need to seek a central metadata index on each file access – instead, the master gives a client a volume ID, and the client then directly reads from the volume server in one hop [30] [91]. Each volume server keeps file metadata **in memory (only 16 bytes per file)**, enabling O(1) lookups for file read/write on that node [92] [93]. This design yields very low-latency access for small objects: SeaweedFS can handle **millions of small files** with minimal slowdown, which is a known weakness of many distributed stores. Additionally, writes are append-optimized and sequential on volumes, which helps with write throughput. SeaweedFS's developer claims that for use cases with **lots of small files or thumbnails, it "should beat most other options" in performance** [94]. Indeed, SeaweedFS tends to outperform heavy systems for metadata-intensive

workloads. For large file streaming, SeaweedFS can still perform well (it splits large files into 8MB chunks behind the scenes [95]), but it may not reach the same sequential throughput as MinIO or Ceph with striping – Seaweed focuses more on **concurrency and low seek overhead**. In one user's 2020 tests, SeaweedFS throughput was "mediocre" for parallel loads compared to MinIO [96] [97], but improvements and proper configuration (e.g. using multiple volume servers and tuning networking) likely ameliorated that. Overall, for an AI environment that might involve *millions of tiny embedding files or lots of metadata ops*, SeaweedFS offers **excellent responsiveness**. For pure max throughput on huge files, it's good but might be edged out by raw-erasure systems like MinIO.

- **OpenIO:** OpenIO claims **low latency that even improves as the system grows** [98]. Thanks to its placement algorithm, any object lookup is resolved with a fixed at most 3 hops, regardless of cluster size [73]. This means metadata lookup time remains small even as you scale to many nodes. OpenIO also actively distributes load based on node performance: new hardware is automatically given more work until it equalizes [75]. This dynamic load balancing can improve utilization and performance. In practice, OpenIO has shown **very good throughput and concurrency** in benchmarks; it was designed to run even on mixed hardware without bottlenecks. The absence of a global RAFT or single leader means writes don't all funnel through one server – many ops can happen in parallel on different nodes [88]. One distinctive feature is built-in **content caching**: OpenIO nodes or gateways can cache recently accessed objects in RAM, which benefits repeated reads (useful for AI models reading the same data repeatedly). It also supports **direct streaming** and multi-threaded transfer for large objects, making it competitive in throughput. Some anecdotal evidence: OpenIO was used to back a large email service (for attachments) and video platforms, indicating it handled real-world loads well. Latency for small ops is likely on par with or slightly above SeaweedFS (due to more complex routing), but still very low. In sum, **OpenIO provides strong all-around performance**, excelling at heterogeneous cluster efficiency and maintaining latency as it scales. It may require fine-tuning of policies to get the best speed for a given workload.

- **Zenko:** Zenko's performance needs to be viewed in two parts – the **control/metadata plane** and the **data path**. The data path for Zenko can be efficient: when a user uploads via Zenko, the CloudServer component writes the object to the configured storage location. If that location is local (a filesystem or RING), the data flows through the Zenko instance to disk. If the location is remote (AWS S3), Zenko streams it to AWS. Zenko is implemented in Node.js, which is generally not as fast at raw I/O as Go or C++; however, Node can handle many concurrent connections with its async model. The S3 CloudServer was found to have **very good S3 API compatibility** but slightly lower performance than Ceph or MinIO in some S3-tests [99] [100]. For moderate workloads and multi-cloud scenarios, Zenko's throughput is usually acceptable (it can parallelize transfers for replication). The **metadata operations** (tracking replication status, etc.) are handled by MongoDB, Kafka, etc., so the performance there depends on those components – which are scaled in Kubernetes. In essence, Zenko adds **some overhead** compared to a direct store, because it's doing extra bookkeeping for replication and using a higher-level language. If used just as a local S3 store, it's not as lightweight as MinIO. Its strength is being able to orchestrate copying to multiple targets without the client needing to do it; the cost is some latency as Backbeat queues the tasks. Zenko can still saturate a decent network for large file transfers, but if **maximum local performance** is the goal (with no multi-cloud need), Zenko wouldn't be the first choice. It's more appropriate when performance across **distributed clouds** is the concern (where network or cloud speed is the limiter, not Zenko itself).

- **Garage:** Despite its minimalistic approach, Garage is quite performant for small-to-medium scale. Its use of **no global consensus** in the critical path means writes and reads can be handled by any node that gets the request, coordinating with peers as needed without a central bottleneck [88]. The developers highlight that this is beneficial especially when nodes are **geographically distributed** – no need to wait for a far-away leader on each op, so latency can be low even over WAN links [101]. In testing, Garage can handle **high concurrent PUT/GET rates** on modest hardware, and the absence of heavy metadata databases (it keeps metadata within the chunks and small internal state) reduces overhead. Moreover, the built-in **deduplication** can improve effective throughput if the workload has redundant data (it won't write the same block twice). One should note that **Garage is optimized for smaller clusters** (tens of nodes or less); it's not intended to scale to hundreds of nodes (where a DHT without hierarchical management might face routing overhead). But for a lab or small enterprise setup, it's efficient. It may not reach the raw streaming throughput of MinIO or Ceph if those are deployed on powerful hardware – Garage is more about doing the job on whatever hardware you have, even under suboptimal conditions (e.g., mixing a Raspberry Pi and a desktop – it can adapt). Also, since **Garage currently lacks erasure coding**, every byte is written `RF` times, which could impact write throughput vs. an erasure system that writes fewer bytes – however, in many cases replication can actually be faster (no need to compute parity). In summary, **Garage provides solid performance for its target scope**, especially in scenarios involving edge nodes or varying network latency. It may not be the very fastest for a single large file stream, but it is **fast enough** for most AI/dev workflows and excels when data is accessed from multiple sites.

To illustrate performance differences: for a single-node scenario, one might see MinIO > SeaweedFS > Ceph in raw throughput for large files (MinIO being very optimized in Go, Seaweed avoiding extra overhead, Ceph incurring more checks). For small random reads, SeaweedFS and OpenIO likely outperform Ceph and MinIO due to their metadata strategies. Garage and OpenIO aim to keep performance consistent even as more nodes are added or in less reliable networks, which is a different kind of performance (predictability and low tail-latency).

In an AI context, if the user's workload is streaming large training data, **MinIO** or **Ceph** (with proper hardware) could deliver the highest sustained throughput. If it's lots of small model checkpoints or embedding vectors, **SeaweedFS** or **OpenIO** might serve those faster. If the data processing is geographically split (edge devices collecting data to store centrally), **Garage** could shine by efficiently ingesting from all edges with minimal coordination delay.

Ease of Deployment and Scaling

The complexity of deploying and managing these systems varies widely – from single binary servers to complex multi-service clusters:

- **MinIO (Deployment):** MinIO is famously **easy to deploy**. It's a single self-contained binary; you can start a server with one command (`minio server /data` pointing at a directory). For a distributed setup, you launch MinIO on multiple nodes with the same configuration (e.g., list all node addresses) and it automatically forms a cluster. It has a simple web GUI for initial exploration and a well-documented set of Docker images and Kubernetes operators if needed. MinIO can be up and running in minutes on a single machine and scaled out later by adding drives/nodes (though to expand an existing cluster's size, one typically adds an entire new set of nodes rather than one at a time). Because of its simplicity, **MinIO is often used by developers for quick S3-compatible**

storage testing. Upgrading MinIO is also straightforward – replace the binary and restart (backward compatibility is good). Configuration is via environment variables or a config file, which is relatively minimal. **Scaling:** You scale MinIO by either running it in **distributed mode** from the start (multiple nodes) or using federation (multiple clusters) if needed. A single MinIO instance supports a certain number of drives/nodes (e.g., 16 nodes with 16 drives each in one cluster for optimal erasure coding). If you need to grow beyond that, MinIO recommends federating separate clusters and using e.g. DNS pooling or their federation features. In terms of **operational ease**, MinIO is lightweight – there is no external database or dependencies – making it suitable for a small team to manage. One caveat: MinIO's newer licensing (AGPLv3) means if you heavily customize or redistribute it, you need to comply with open-source license rules, and enterprise features/support require a commercial deal. But from a pure deployment perspective, **MinIO is one of the simplest** among these options.

- **Ceph (Deployment):** Ceph is at the opposite end of the complexity spectrum. A Ceph cluster has multiple components: monitors (for cluster state), OSDs (for storage on each disk), metadata servers (for CephFS, not needed for RGW alone), and RGW instances (for the S3 API). Setting up even a small Ceph cluster manually can be challenging – historically it involved careful config files and executing commands to deploy monitors, prepare disks, etc. However, Ceph has invested in tools like **ceph-ansible and cephadm**, and the Kubernetes **Rook operator**, which greatly simplify deployment. For instance, using Rook, you can spin up a Ceph cluster on K8s with a single custom resource definition. There's also a trend of using containers for each Ceph daemon to ease installation. Still, one must configure networking (Ceph works best with a dedicated backend network), ensure time sync, and have enough nodes (a practical minimum is 3 nodes for monitors and 3 OSD hosts, though you *can* run a test cluster on a single node with multiple OSDs if needed). The **learning curve is steep** – understanding CRUSH maps, pool settings, and how to tune Ceph might require expertise. **Scaling Ceph** is one of its strong points: you can add OSDs (disks) or whole nodes on the fly, and Ceph will rebalance data automatically according to policies. It can scale to **thousands of nodes and exabytes** (as proven by large OpenStack and cloud installations). With scaling, however, operational complexity (monitoring, performance tuning) also grows. Ceph has a rich CLI and dashboard for management. In summary, **Ceph demands more initial work and ongoing administration**, and is best if you anticipate **very large scale or need its unified block/file/object capabilities**. For a small self-hosted AI stack, deploying Ceph might be overkill unless you specifically want to invest in the reliability it offers.

- **SeaweedFS (Deployment):** SeaweedFS is relatively easy to deploy, especially for basic use. The project provides a single binary (`weed`) that can run all roles depending on arguments. For a minimal setup, you can literally run `weed server -s3` and it will start a master server, volume server, and S3 gateway all in one process [102]. This "all-in-one" mode is great for testing and small deployments – you get an S3 endpoint immediately with data stored in the local folder. For production, you'd likely run a dedicated master process, one or more volume server processes (pointing at storage directories on each node), and the **Filer** process if you want POSIX-like filesystem features. SeaweedFS's documentation includes simple examples, and because it doesn't require external databases (unless you use certain Filer stores) it's not hard to get going. If you do use the Filer with an external metadata DB (like PostgreSQL or Redis), then setting those up is an extra step but also straightforward for anyone familiar with those databases. **Scaling SeaweedFS** is one of its highlights – to add capacity, you just start another volume server (pointing it to a new storage directory or host) and **notify the master**, and it's immediately available for new data [103]. There's no huge rebalance cost because new files will simply start filling the new volumes. Existing

volumes aren't split or moved unless you explicitly arrange that (Seaweed doesn't auto-shard old volumes across new nodes – it treats volumes as atomic, 32 GiB units). This means **scaling is linear and simple**, though if a volume gets full, it stays on its original node (you just get more volumes on new nodes for new data). SeaweedFS also supports mounting as a filesystem (via FUSE) and has Kubernetes integration (it can be used as a backing store for K8s with CSI driver). The project maintainer is very responsive and the community small but active. Overall, **SeaweedFS is quite lightweight to operate**; you might run 1 master + 1 filer (with, say, Postgres) + N volume servers. Even a single-node can run all components easily. It's a good choice if you want to avoid complexity but still distribute data across a few servers.

- **OpenIO (Deployment):** OpenIO is more involved to deploy than SeaweedFS or MinIO, but still easier than Ceph. It consists of several daemons (for proxy, for metadata (directory), for storage (called `chunk` or `rawx` servers), etc.). OpenIO provides binary packages and even Docker images. They also had an Ansible playbook for setting up a cluster in their documentation [104] [105]. A basic cluster might require around **6 services**: one (or a couple) of **conscience directory** servers (metadata coordinators), a few **proxy** servers (which handle S3/Swift API requests), and multiple **storage daemons (rawx)** on each storage node, plus some supporting services for management. The configuration can be a bit tricky for newcomers because you need to define the "namespace" and ensure all nodes know about each other (the Conscience makes it somewhat self-discovering, though). The recommended starting point is 3 nodes (to avoid SPOF and leverage Conscience fully) [106]. The OpenIO docs and community forums (and Slack) can help with architecture planning. **Scaling OpenIO** is quite flexible: you can add nodes of any capacity at any time; the Conscience service will detect them and integrate them. However, after adding capacity, you might manually trigger rebalancing for optimal distribution (OpenIO has tools for rebalancing data in the background). Because it supports heterogeneous nodes, you don't need identical hardware to scale – you can add a bigger server and it will just carry more weight automatically [107]. Operationally, OpenIO provides a GUI and metrics which help. Still, compared to one-binary solutions, **OpenIO requires managing multiple services**, which is more work. If an organization is comfortable with Linux systems and config management, it's doable. Given OpenIO's enterprise use, it has features like account management, quota, etc., which come with slightly more complex config. In summary, **deployment is moderate complexity** – not plug-and-play, but far from Ceph's complexity. It's a good candidate if you're willing to spend some time learning it or if you need its unique features like dynamic policies.

- **Zenko (Deployment):** Zenko has a **significant deployment overhead** because it's essentially a **suite of microservices** typically run on Kubernetes. A minimal Zenko deployment includes **MongoDB** (for metadata), **Redis** (for caching some states), **Kafka + Zookeeper** (for the Backbeat queue and coordination), and the **CloudServer and related services** (which themselves can be multiple pods) [53] [108]. Scality provides Helm charts or operators to deploy Zenko on a K8s cluster, but you'll need a K8s environment (on-prem or cloud). They recommend at least 3 nodes (to run MongoDB and Kafka in a HA manner) and in practice often 5 nodes for production [109]. This is clearly heavier than a standalone binary. There was also an **all-in-one Docker Compose** that Scality offered for dev/testing (Zenko "sandbox"), which runs all components in one host – convenient for trial but not for production. Once deployed, Zenko can be managed via its Orbit UI (though note: earlier Orbit was a cloud-hosted UI, one had to sign up; not sure if that changed to a local UI). **Scaling Zenko** means scaling the underlying K8s pods: if you need more throughput, you scale out the CloudServer instances (which are stateless S3 API servers) up to dozens of pods [110]. Mongo can be sharded if

needed, etc. It's capable of handling large environments, but the complexity is obvious – you need to maintain a whole suite of services. This approach is justified if you truly need multi-cloud workflows with complex rules, or if you already have a Kubernetes platform and want to plug in storage management capabilities. For a single admin or small team looking to add object storage to an existing stack, Zenko's deployment may be **too heavyweight** unless the multi-cloud capability is absolutely critical. Essentially, Zenko is best seen as an **orchestrator** on top of storage – if you don't need that orchestration, setting it up is unnecessary work. But if you do, expect a non-trivial time investment to get it production-ready.

- **Garage (Deployment):** Garage is fairly easy and **self-contained**. It also uses a single binary for all functions; you run the `garage` daemon on each node and it will discover peers (via a static config or using a Kubernetes/Consul integration) [111] [112]. The configuration (TOML file) defines the cluster ID, node addresses, replication factor, etc. Starting a cluster is as simple as launching the binary on each node with the same cluster key. Garage was designed to be **easy to run even on minimal hardware** – for example, it could be deployed on a few low-power machines at home without fuss. It does not require any external database; all state is managed by the nodes themselves. The documentation provides a quick start to get a small cluster working, and even covers using it with Docker or Kubernetes if desired. **Scaling Garage** is straightforward: bring up another `garage` node, and use the admin API or CLI to **invite** it into the cluster (or if using K8s, the operator does that). The cluster will then rebalance by copying some data to the new node in the background to satisfy the replication factor and spread capacity [113] [57]. One advantage is you can have *geographically separated* nodes join, and Garage will factor that in when placing data (you can tag nodes by region). This makes scaling across sites very natural. On the flip side, because it is so self-contained, you should monitor the cluster's health – e.g., if a node goes down and doesn't come back, use the admin CLI to mark it removed so that data is re-replicated elsewhere. The **Garage CLI and admin REST API** handle these tasks. There is also a built-in Prometheus metrics endpoint for monitoring [114]. All in all, **Garage is lightweight to deploy and manage**, aligning with its goal of being a "low-tech" storage for self-hosters. The primary caution is to understand it's still version <2.x and actively evolving, so you'll need to stay updated with releases and perhaps do minor migrations as features improve (the project has migration guides for each version update [115]).

Deployment Summary: For a single-developer or small team environment: **MinIO and SeaweedFS** stand out as *very quick to set up* with minimal moving parts. **Garage** is similarly easy for a small cluster and has modern ops niceties (self-discovery, etc.). **OpenIO** and **Ceph** are heavier – OpenIO requires running multiple daemons but still can be done without specialized orchestration, whereas Ceph often justifies a dedicated effort or using Rook on Kubernetes. **Zenko** is the heaviest, essentially a distributed app requiring Kubernetes and multiple dependencies – not typically justified unless multi-cloud management is a primary objective.

The user's existing stack (PostgreSQL, Neo4j, Qdrant, Redis, etc.) suggests they are comfortable managing complex services, but probably they would prefer to minimize overhead for the storage layer if possible. A simpler deployment leaves more resources free for AI model serving and databases.

Ecosystem Maturity and Documentation

Choosing a storage solution also involves evaluating its **stability, community support, and documentation quality**:

- **MinIO (Maturity):** MinIO is a **highly mature project**. It has been under active development since 2015 and has a very large user base (from individual developers to large enterprises). It's considered production-ready and stable; in fact, many cloud-native applications and commercial products embed or rely on MinIO for S3-compatible storage. The **documentation** is excellent – MinIO's docs cover everything from quick-start to advanced security configuration and performance tuning. There is also a rich blog with best practices (e.g., securing MinIO, using it for AI, etc.). The community is active on Slack and GitHub, though note that MinIO's team has, in recent years, steered users towards their **paid support** for deeper issues. Nonetheless, most questions can be resolved via documentation or community forums. MinIO's **ecosystem** includes client SDKs in many languages (Java, Python, JavaScript, etc.) and a CLI (`mc`). It adheres closely to AWS S3 APIs, which means any tool or library built for S3 generally works with MinIO without trouble. Regarding updates, MinIO's philosophy is to roll out frequent updates (sometimes weekly), which add features or improvements. These are usually backward-compatible, but one should keep MinIO reasonably up-to-date to benefit from security patches. Overall, MinIO's **maturity and support are top-notch** in the open-source object storage world, with the slight caveat that some features (e.g., certain admin features or WORM compliance) might be **enterprise-only** as the company focuses on commercial clients. However, all core functionality is open source (AGPL). Users have reported MinIO being rock-solid in production for years.

- **Ceph (Maturity):** Ceph is one of the **most battle-tested** storage platforms. It originated around 2007 from academic research and has been widely deployed in production (often via Red Hat Ceph Storage) for over a decade. It's the backbone of many OpenStack cloud storage services and has a strong track record of reliability at scale. The community is large and global, with monthly releases and an active development mailing list. **Documentation** for Ceph is extensive [116] – almost to a fault: the official docs can be hard to navigate due to the breadth of features. But they cover everything: RGW usage, encryption setups, multi-site, tuning, etc. Additionally, countless blog posts, conference talks, and even books exist on Ceph. If a problem arises, chances are someone has encountered it and discussed it on the Ceph mailing list or tracker. Ceph has a formal release cadence and long-term supported versions. As of 2025, versions like **Quincy or Reef** are in use, each with improvements (RGW in newer versions supports features like tiering and OIDC integration). The **ecosystem** around Ceph includes Rook (K8s operator), integration with Kubernetes CSI, and client libraries for the lower-level RADOS (if one ever needed to bypass S3 and use librados in C/Python directly). Running Ceph for just object storage means you won't necessarily use its file (CephFS) or block (RBD) interfaces, but it's nice that the option is there if your needs expand. One thing to note: Ceph's complexity means there's a **steeper learning curve**, but the community (including many companies like Red Hat, SUSE, and now even IBM via Red Hat) are behind it. Ceph is a safe choice when longevity and community support are concerns – it's not going anywhere and will likely continue to evolve. For the user's AI stack, Ceph might be considered if they foresee needing a highly scalable unified storage that could also do file system duties (for example, CephFS could be used to share files between training nodes in a cluster, alongside RGW for object APIs). If not, the overhead might not be worth it, but maturity is certainly a plus.

- **SeaweedFS (Maturity):** SeaweedFS is a **younger project than Ceph or Swift** but has been around since ~2014 and has steadily grown in features. It's now fairly stable and even offers some commercial support via Chris Lu's company for those who need it. The community is smaller than MinIO or Ceph, but quite enthusiastic – you'll find it on GitHub discussions and a Slack/Telegram channel. SeaweedFS documentation is decent and has improved (the GitHub README and Wiki cover architecture and features, and there's even a recent book by community contributors). However, some parts can be terse, and occasionally one must read issue threads to figure out advanced configurations. The **S3 API compatibility** of SeaweedFS is not complete (no versioning, limited policy support as mentioned) [32], which is a sign it's not as focused on being a drop-in for all AWS S3 features. But it covers the basics well enough for many cases. In terms of usage in the wild, SeaweedFS has been used in scenarios like backing cloud drives, AI annotation platforms, etc., particularly where lots of small files are involved. The **release activity** is continuous; the maintainer is very active in merging PRs and releasing new versions with bug fixes and features (encryption support, tiering, etc. were added over the past couple of years). It's not backed by a big corporation, but it's open and MIT-licensed, meaning you can use it freely. As an open-source project, it has a good trajectory. That said, being more niche, if you run into an obscure bug, you might rely on the maintainer's help rather than a broad user base – though the maintainer is known to respond quickly. Overall, SeaweedFS is **mature enough for medium-sized deployments** (there are reports of clusters with hundreds of millions or billions of files), and its simplicity can be an advantage (less that can go wrong). The documentation plus community will generally get you through any configuration, and the codebase is not as huge as Ceph's, making it more approachable for tweaking if needed.

- **OpenIO (Maturity):** OpenIO, while not as universally known as Ceph, has an interesting history. It was used in production for large systems (for example, it was publicized that **OVHcloud**, a big European cloud provider, adopted OpenIO for some of their storage needs and eventually acquired the company behind OpenIO in 2020). So OpenIO has seen real workloads at scale (hundreds of TBs to PBs). The project is stable (current open-source version is 20.04 as per docs, which suggests a major release in 2020; not sure if newer versions are under a different model with OVH). **Documentation** is quite thorough for the open-source version [42] [45] – it includes an admin guide, quick start, reference for S3 compliance [43], etc. There is an "OpenIO SDS" community, though since the company's acquisition, open-source community activity might have slowed if development went mostly internal. However, OpenIO SDS is Apache-2.0 licensed and the code is on GitHub, so it remains available. One potential concern: if most new development is tied to OVH's internal use, the public-facing releases might come slower. Even so, the existing feature set is rich and was quite forward-thinking (things like the dynamic tiering and policy engine are not found in many other solutions). **Integration**: OpenIO had integrations with other software (e.g., it could plug into OpenStack Swift APIs, and had connectors for NFS, etc.). The **community support** might not be as easy to find as Ceph's – likely one would rely on archived forums or the documentation itself. But given that OpenIO aimed at enterprise use, it's fairly mature. If the user values having a tried-and-true system but wants something lighter than Ceph, OpenIO could be a candidate, acknowledging that it's a bit more niche. The **learning curve** is moderate – you need to grasp concepts like the Conscience algorithm and how to configure services – but once set up it tends to "just work". It hasn't seen the hype that MinIO has in recent years, but it's a solid piece of tech with an **"elegant architecture"** as noted by reviewers [117].

- **Zenko (Maturity):** Zenko (or Scality CloudServer) is an offshoot of Scality's long-standing proprietary storage. Scality has been in the object storage business for a long time (RING is a well-known object storage for enterprises). Zenko was open-sourced around 2017. The core S3 server code (CloudServer) is fairly mature and had contributions from the Scality team and community – it was noted for good S3 API compatibility (even better than MinIO in some aspects of AWS-specific features) [99]. However, some parts of Zenko (like Backbeat for cross-cloud replication) can be complex and were originally tied to Scality's own stack (e.g., using Kafka+Mongo to tail object logs). Zenko had a **latest release 2.x** a couple years ago, and it's unclear how active it is now. Scality shifted focus to hybrid deployments which might still use CloudServer as a component. The **documentation** on ReadTheDocs is detailed [118] [119], indicating a fairly comprehensive guide to using Zenko. The **community** is smaller; a lot of Zenko users were either using it as a tech preview or in very specific use cases. If you encounter issues, you might need to look at Scality's GitHub or forums, but they are not as bustling as Ceph or MinIO communities. In terms of **stability**, running Zenko in production means running multiple moving parts – each of which (Mongo, Kafka, etc.) is individually mature, but the overall orchestration needs to be robust. Provided it's run on a solid K8s platform and not over-stressed beyond intended workloads, Zenko should be stable. But it's a more **complex system to troubleshoot** if something misbehaves (one might have to check logs across several services). If the user's priority is a widely-adopted solution, Zenko might not rank as high as others. It remains a unique offering (multi-cloud control), so maturity in that niche is moderate: it works as advertised, but it's not ubiquitous in the way others are.

- **Garage (Maturity):** Garage is **relatively new** (initial public releases around 2021). As of 2025 it's at a fairly stable version (the presence of a 1.0 and now 2.0 indicates some milestone of stability was reached). The project is maintained by a small collective (Deuxfleurs), and their goal is explicitly to offer a reliable self-hosted storage for the community. The **documentation** is good and user-focused – they have cookbooks for integration with Nextcloud, etc. [62] [120], and a clear explanation of design decisions [121] [84]. Since it's newer, it doesn't have a large user base yet, but it's growing among self-hosters. The features missing (versioning, some S3 operations) are openly tracked [13], so one can see if they are dealbreakers. The **stability** for core functionality (basic PUT/GET, listing, etc.) is reportedly quite good – it's being used by the maintainers themselves to run services (their website is hosted on Garage, etc., proving out real-world use [122]). One can anticipate more features to be added as time goes on (e.g., possibly SSE-S3 encryption or a form of erasure coding in the future if the roadmap allows). The **community** is small but accessible; the maintainers engage on Hacker News, Discord/Matrix, and accept contributions. If an AI architecture needs something reliable but not necessarily enterprise-proven, Garage is a candidate, but being on the cutting edge means you might hit an unpolished area occasionally (though they take security seriously – for example, requiring encrypted node communication by default [58]). The advantage of a newer project is that it's **unencumbered by legacy** – it's written in Rust, uses modern libraries, and may be easier to extend or fix if you have Rust expertise. For the user, adopting Garage would be a bet on an emerging tech: it could pay off in simplicity and just-enough-features, but it's not as time-tested as others.

In summary, **MinIO and Ceph** are the most mature and widely supported by large communities and companies. **OpenIO and SeaweedFS** are solid and have seen real-world use, but their communities are more niche (OpenIO in enterprise, SeaweedFS in open-source circles). **Zenko** is mature in concept but maybe less common in deployment. **Garage** is the newcomer that's promising but still rounding out features. The documentation for all of these is available; MinIO's and Ceph's stand out in completeness,

while the others have adequate docs with some gaps possibly filled by community Q&A. When making a decision, one should consider how much community help or official support they might need. For instance, if the user might need enterprise support contracts, **Ceph (through Red Hat)** or **MinIO (through their subscription)** are options. If they are fine with self-support via open communities, all are viable with varying levels of help available.

Integration with the Existing Stack

Now, focusing on how well these solutions **integrate with the user's current tech stack and workflows** (Postgres, Neo4j, Qdrant, Redis, Dramatiq/RabbitMQ):

- **S3 API and Python integration:** All the evaluated solutions (MinIO, Ceph RGW, SeaweedFS, OpenIO, Zenko, Garage) expose an **S3-compatible API** endpoint. This is key because it means the user can use standard Python tools (like `boto3` or MinIO's Python SDK) to interact with any of these stores just as they would with AWS S3. For example, storing AI model artifacts or datasets can be done with simple Python scripts using boto3 on MinIO or Ceph with no code change, since they accept the same REST calls. That said, **API compatibility varies**: Ceph and Zenko are nearly full implementations of the S3 API (Ceph passes almost all S3 tests [123], Zenko not far behind [100]), whereas SeaweedFS and Garage implement common S3 operations but lack newer or less-used features (like object locking or certain policy types) [13]. In practice, for integration, this means if the user's Python code sticks to basic operations (put, get, list, delete, presigned URLs, etc.), any of these will work. If they plan to use more advanced S3 features (e.g., **multipart upload**, **versioning**, **server-side encryption via API**, **batch delete**, etc.), they should ensure the chosen store supports it. For example, **multipart upload** is widely supported (MinIO, Ceph, OpenIO, etc., all have it; SeaweedFS supports multipart as well), but **object versioning** is not in SeaweedFS or Garage yet [32] [124]. Since the user specifically mentioned *generative AI outputs and backups*, versioning might be a nice-to-have for keeping historical versions; if so, MinIO, Ceph, or OpenIO would integrate better (they all support versioning).

- **Database integration:** The user has PostgreSQL and Redis in their stack. Some object stores can leverage these: **SeaweedFS**, for instance, can use PostgreSQL as its Filer metadata store [36]. If the user wanted, they could configure SeaweedFS's Filer to store directory metadata in PostgreSQL, which might simplify backup of metadata or allow browsing the metadata via SQL (though it's not a typical use case to manually query that). SeaweedFS can also use Redis for certain metadata caching. **OpenIO** doesn't use external DBs for core operation, but it can interface with a relational DB for specific needs (OpenIO's metadata indexing service could potentially push index data to a Postgres, though by default it doesn't require Postgres – it has its internal DB). **Ceph** and **MinIO** run their own metadata internally (Ceph uses its RADOS cluster, MinIO uses embedded key-value stores like BoltDB or LMDB on each node). So direct integration with the user's Postgres isn't needed. However, the presence of PostgreSQL and Redis indicates the user might want to *trigger events or coordinate between object storage and these systems*. For example, an AI pipeline might upload a file and then record a reference in Postgres or send a message via Redis or RabbitMQ to start processing that file.

- **Event Notifications:** Both MinIO and Ceph RGW excel here by providing *built-in event hooks*. **MinIO** can be configured to publish events (like "object created" or "object removed") to a variety of endpoints such as **RabbitMQ (AMQP)**, **Redis** (as a PubSub channel or stream), **PostgreSQL** (it can write events to a table), and others [22] [23]. The Medium example shows how MinIO triggers a

RabbitMQ message when a new file is uploaded [125] [126]. This is directly relevant: the user has **Dramatiq and RabbitMQ** for their task queue, so they could tie MinIO events to RabbitMQ queues – e.g., automatically send a message to process a new uploaded image. MinIO's event system is quite powerful and easy to set up with its `mc` CLI or config file. **Ceph RGW** similarly offers an **S3-compatible bucket notification API** where you can create "topics" for events and subscribe endpoints. Ceph supports sending to **AMQP (RabbitMQ)** and Kafka and webhook endpoints [29]. So Ceph could directly push a JSON event to RabbitMQ when, say, an object is created in a certain bucket. This integration potential is excellent for building event-driven pipelines (like automatically ingesting data into Qdrant or calling AI models when new data arrives).

- **OpenIO integration:** OpenIO doesn't natively emit RabbitMQ events on object ops (at least not in the open version). But OpenIO did have an "event hooks" mechanism for their grid – historically, they demonstrated things like **triggering a video transcoding job when a new video is uploaded**, using their internal messaging. It might not directly speak AMQP, but you could integrate by writing a small service that polls OpenIO's metadata or uses its **metadata indexing** feature to find changes. Additionally, since OpenIO supports **Lambda function execution in its enterprise version** (similar to AWS Lambda on new objects, as per some press releases), it indicates the architecture can handle event-driven tasks. For the open-source, the more straightforward integration is via the **S3 API notifications**: unfortunately, I don't believe OpenIO implements the S3 Bucket Notification API. So you'd likely fall back to something like periodically listing a "staging" bucket for new items. Not as elegant, but feasible.

- **SeaweedFS integration:** SeaweedFS does not provide S3 event notifications out-of-the-box. However, because SeaweedFS has a **changelog** (especially if you use the Filer), one could tail that changelog to detect new files. Also, SeaweedFS allows **scripting with its Filer** – e.g., you can run a meta log consumer that pushes events to an external system. It might require custom coding though; it's not a simple config switch like MinIO's. If the user is comfortable writing a small integration (for example, a Python script that monitors SeaweedFS Filer's Kafka export or uses its gRPC API to get notifications), it can be done. But natively, SeaweedFS's integration will be more manual.

- **Zenko integration:** Zenko's **Backbeat** keeps track of all operations for replication and can even index metadata into ElasticSearch (so you can search objects by metadata across clouds). It likely could be leveraged to send notifications, but that's not its primary goal. Zenko is more about syncing data than notifying external apps. If the user uses Zenko and wants to trigger their own jobs, they might again need to plug into the Kafka topics that Zenko uses internally for replication events. This could be complex and not officially supported. Zenko might have an integration to trigger a **serverless function** (Scality was talking about enabling that) but it would require using Zenko's metadata search or workflow engine, which is beyond basic usage. Given that the user already has RabbitMQ and Dramatiq, simpler direct integration from the storage is preferable.

- **Garage integration:** Garage currently has **no built-in event hooks**, which means integration would be polling or manual. One idea could be to use Garage's **"admin API"** which could list recently changed objects or use its experimental K2V metadata store to track changes, but that would involve additional development. However, Garage does integrate nicely with some existing apps: for example, using Garage as external storage in Nextcloud or as a backend for restic backups (as mentioned in docs) [62] [127]. This suggests it plays well with standard S3 clients at least.

- **Redis integration:** Apart from events, the user has Redis with Lua in their stack – perhaps for caching or quick operations. None of these object stores directly use Redis except SeaweedFS (which *can* use Redis as a metadata store, but that's an internal detail). If the user wanted to cache certain object metadata or small objects in Redis for faster access, they'd have to handle that at the application level. One neat integration: **MinIO** can use Redis as a cache for *gateway mode* (though gateway mode is deprecated as per MinIO docs for some backends). More relevantly, MinIO can send notifications to Redis, as mentioned, effectively turning Redis into a pub-sub broker for events [128]. So if the user's pipeline uses Redis Pub/Sub or Streams, MinIO can natively publish object events there. Ceph RGW doesn't directly do Redis but could do AMQP which can be bridged to Redis if needed.

- **PostgreSQL integration:** MinIO's ability to log events to PostgreSQL [22] might be interesting: it could, for example, insert a row in a "file_uploads" table every time an object comes in. If the user wants to keep an index in Postgres of all stored files (maybe for metadata search or linking with other relational data), this is an elegant solution. Each insert would contain details like bucket, object name, size, event type. Ceph doesn't log to SQL directly, only to AMQP/Kafka. SeaweedFS if using Postgres for Filer, essentially already stores all file metadata in a SQL table (which the user could query, although the schema is meant for Seaweed's consumption mostly).

- **Integration with Neo4j / Qdrant:** These are more specialized – Neo4j is a graph DB; Qdrant is a vector search engine. The object storage might be used to store large blobs (images, audio, etc.), while Neo4j/Qdrant store structured data or vectors referencing those blobs. No object store is going to directly integrate with a graph DB or vector DB, but the typical pattern would be: store the file in object storage, get its URL or key, then store that reference in Neo4j or Qdrant's metadata. For example, an image is stored in MinIO and its vector embedding is stored in Qdrant with a payload that contains the MinIO object URL. Later you might query Qdrant for similar images and get the object keys to fetch from storage. All solutions support retrieving objects via HTTP (S3 API), so as long as the vector DB or your application can fetch from an S3 URL (possibly presigned for auth), integration is fine. One thing to consider is **access control** – if Qdrant or Neo4j processes need to fetch data from the object store, you'll want either the store to be on the same network and allow access, or have a mechanism to generate temporary access tokens. **MinIO** supports **STS and temporary credentials** via its policy API (and can integrate with identity providers), as does **Ceph RGW** (which can tie into Keystone or use its own user accounts). These allow issuing credentials that other services can use. If integration is all internal, one could also give Qdrant a static access key for the object store. **OpenIO** and **Zenko** similarly allow multiple users with separate keys, so integration services can have their own credentials with limited permissions. **SeaweedFS** is a bit less developed in multi-user auth, but one can run multiple S3 keys or use the "–iam" mode for multi-user (still somewhat experimental). **Garage** allows multiple user accounts (it has an admin API to create users and keys [61]), so one could give each microservice a distinct key pair.

- **Messaging and Task Queues:** The user uses **Dramatiq (with RabbitMQ)** for background tasks. We covered that MinIO/Ceph can send to RabbitMQ. Another angle: if the user doesn't use store-generated events, they can still have their application send messages when using the storage. For example, a web app uploads a file to object storage then explicitly enqueues a task in Dramatiq to process it. In that case, the object storage doesn't need to have notification features. But having them can reduce custom code and ensure no events are missed (especially if files might be added by means outside the direct app control). Ceph and MinIO even allow filter by prefix/suffix on events

[125] [129], so you could set up, say, notifications only for objects in a “new-uploads/” prefix to avoid noise.

- **Backup integration:** The user mentioned backups as one file type. Many backup tools (like **Veeam, restic, Borg with rclone, etc.**) can target S3-compatible storage. A mature ecosystem integration point: **MinIO** and **Ceph** are tested with such tools (MinIO’s blog even has articles on using it with backup software [130]). **Garage** explicitly mentions working with restic and similar backup software [131]. **SeaweedFS** could also be used as a restic target via its S3 gateway (though not commonly seen in docs). The benefit of using a well-supported solution is you can plug into these existing backup workflows easily.

- **Logging/Monitoring:** Integration with logging and monitoring tools is also part of the picture. Ceph and MinIO both expose extensive metrics – Ceph in its mgr module with Prometheus exporters, MinIO with built-in Prometheus metrics. Zenko and OpenIO also integrate with Prometheus (as indicated by Zenko’s docs using Prom for monitoring [132] [133], and OpenIO’s metrics endpoints). Garage too provides Prometheus metrics [114]. So whichever is chosen, the user can likely pull metrics into their observability stack (if they have one) to monitor storage performance and usage. This indirectly integrates with their operations.

To wrap up integration: **MinIO offers the most plug-and-play integration features** (native notifications to the exact components the user has: RabbitMQ, Redis, Postgres). This is extremely convenient for building an event-driven AI pipeline. **Ceph** also integrates well but requires a bit more initial setup to enable and perhaps writing a small script to subscribe to its AMQP or HTTP endpoint (not hard). **OpenIO** and **SeaweedFS** can be integrated but need either custom changes or periodic jobs since they lack off-the-shelf event hooks. **Zenko** could integrate but at cost of complexity, and it’s overkill if you just wanted notifications. **Garage** integration capabilities are currently minimal beyond basic S3 API – you’d have to implement any event flow yourself.

Given the user’s stack, if *tight integration* and automation is a priority, MinIO and Ceph stand out as fitting in elegantly with the message queue and database. If the user is okay with simpler integration (like just using it as a store and handling logic in the app), any will do, but MinIO/Ceph still give the option to expand into more automation later.

Operational Comparison at a Glance

Finally, to summarize some of the operational aspects discussed (scalability, ease of management, community, etc.), the table below compares the solutions on these points:

Solution	Local Performance (Throughput/ Latency)	Scalability & HA	Deployment Complexity	Community & Support
MinIO	*High:* Optimized for streaming; can saturate high-bandwidth links. Great concurrency; ~ms latency for small ops (Go optimized) [89].	*Horizontal scale-out* (multi-node clusters up to dozens of nodes; federation for more). No SPOF with distributed erasure coding.	*Easy:* Single binary or Docker. Minimal config. Expand by adding new set of nodes (some planning needed).	*Strong:* Large community, excellent docs. Commercial support available. Frequent updates (feature-rich) [134] [135].
Ceph RGW	*Moderate to High:* Very high throughput when scaled (used in PB-scale ops). Small op latency higher than MinIO (more overhead) [136]. Recent improvements (Crimson) targeting latency.	*Massive scale:* Designed for hundreds of nodes. Auto-rebalancing. Truly no SPOF. Multi-site federation possible.	*Complex:* Requires 3+ nodes for HA. Many services (MON, OSD, MGR, RGW). Best deployed via cephadm or Rook on K8s.	*Excellent:* Long-standing project, extensive docs. Wide adoption in industry. Multiple vendors (Red Hat, SUSE) offer support. Steeper learning curve for admins.
SeaweedFS	*High for small files:* O(1) metadata lookups give low latency [92]. Good read/write throughput; slight limits on parallel large-file throughput (Go, 8MB chunking).	*Linear scaling:* Add volume servers on the fly [103]. Master can handle huge file counts. No single volume >32GB (auto-chunking beyond). HA via master replication and volume replication.	*Easy:* One binary, start all-in-one or separate master/ volume. Minimal config (just point storage dirs). Scales out easily; manual volume move if rebalancing needed.	*Active community (smaller):* Dev is very responsive. Decent docs and examples. Fewer third-party tutorials. Used in open-source and some startups.

Solution	**Local Performance** (Throughput/ Latency)	Scalability & HA	Deployment Complexity	Community & Support
OpenIO	*High:* Consistent low latency even as system grows (3-hop lookup max) [73]. Can exploit node capabilities for performance (new nodes auto-used) [75]. Throughput scales with nodes, minor overhead per op.	*Flexible scaling:* Add nodes of any size at any time [107]. Automatic load balancing via Conscience. No SPOF (redundant directory services) [39]. Multi-site capable.	*Medium:* Several services to deploy (meta, proxy, storage). Provided ansible/docker for setup. Manage via CLI/UI. Some manual rebalancing tools.	*Moderate:* Proven in enterprise (OVHcloud). Docs are comprehensive [42]. Smaller OSS user base, but conceptually stable. Company backing (OVH) albeit less community interaction than MinIO/Ceph.
Zenko	*Moderate:* S3 ops compatibility is excellent; performance adds slight overhead (Node.js) vs native stores [100]. Good multi-stream throughput, but latency depends on Mongo/Kafka pipeline. Best for moderate loads across sites.	*Highly scalable* in design (runs on Kubernetes, scale services as needed) [110]. Handles multi-cloud volumes. Requires 3-5 nodes for HA of control plane [109]. Uses underlying storage for actual data scaling.	*High:* Deploying Zenko means deploying K8s + Mongo, Zookeeper, Kafka, etc. Complex but manageable with Helm charts. More moving parts to monitor.	*Niche:* Backed by Scality (enterprise vendor). Docs available [7], but smaller open-source community. Support mainly via Scality channels. Good for specific multi-cloud use cases.
Garage	*Good:* Fast local reads/writes with replication. Rust code is efficient; no centralized latency. Dedup can reduce I/O. Lacks parallel erasure streams, but can utilize multiple nodes well.	*Easy scaling for small clusters:* Add/remove nodes with minimal ceremony [57]. Auto-rebalance to maintain RF. Designed for geo-distributed small clusters (tolerates WAN). No single metadata server.	*Easy:* Single binary per node. Config is simple (TOML). Can even run on mixed hardware. Low resource requirements. Suitable for self-hosters without devops teams.	*Emerging:* Small but growing community. Maintainers engaged. Documentation is user-friendly [121]. Fewer production references (new project), but rapidly improving. No corporate support yet (community-driven).

As the table indicates, the **operational sweet spot** differs: MinIO is great when you want a quick, powerful setup and don't mind scaling in discrete increments. Ceph is ideal when you foresee massive scale and need proven reliability (if you have the skills to manage it). SeaweedFS and Garage appeal if you prefer *lightweight, ease-of-use* and are dealing with either a special workload (many small files for Seaweed) or lower-end infrastructure (Garage). OpenIO and Zenko cater more to hybrid and policy-driven scenarios; OpenIO being more similar to Ceph in on-prem scaling (with less overhead) and Zenko focusing on cloud workflows.

Top Recommendations and Trade-offs

After considering all factors – feature set, integration, performance, and manageability – here is a **ranked shortlist** of recommended object storage solutions for the user's self-hosted AI architecture, with the rationale and trade-offs for each:

1. MinIO **–** ***Top Choice for Local-First AI Pipelines***. MinIO offers the best blend of **ease, performance, and integration** for the given stack. It's very simple to deploy, yet scales to many nodes with strong erasure-coded durability. Crucially, MinIO's built-in **encryption** (with KMS support) and **fine-grained IAM policies** will secure the varied data types (backups, media, etc.) [17] [18]. Its **Python SDK and S3 compatibility** mean minimal friction in connecting to existing code [20]. Where MinIO truly shines for this user is integration: it can natively trigger **RabbitMQ messages and Redis/Postgres events on object uploads** [22] [125], which aligns perfectly with the user's Dramatiq/RabbitMQ workflow and any logging needs. In terms of performance, MinIO's high throughput ensures even large generative AI outputs (e.g. multi-GB models or videos) are delivered quickly. The **community support and documentation** are excellent, reducing the risk in adopting it. *Trade-offs:* As a lightweight solution, MinIO lacks some of the multi-site federation sophistication of Ceph or Zenko – multi-cluster replication in MinIO is possible but not as automatic as Ceph's. Also, MinIO's licensing (AGPLv3) means any proprietary modification would need to be published, though using it as-is poses no issue. Overall, for a single-site, local-first deployment that may optionally sync to cloud, MinIO is a **clear winner** in simplicity and capability.

2. Ceph (RADOS Gateway) **–** ***Most Comprehensive & Robust (Enterprise-Grade)*****. Ceph RGW is recommended as a close second for those willing to manage a more complex system in exchange for** maximum durability and feature completeness**. Ceph delivers the** strongest S3 API fidelity **(passing the most S3 compatibility tests of all** [123] **) – meaning if the user needs things like object locking, ACLs, or multi-tenancy at scale, Ceph has it. It provides virtually** unlimited scalability and fault tolerance**, which future-proofs the storage backend if the project grows massively. Another big plus is Ceph's** native integrations**: encryption ties into enterprise KMS (e.g. Vault)** [25] **, and bucket notifications can directly feed** RabbitMQ/AMQP events **for the task queue** [29] **, similar to MinIO. The data can also be tiered to cloud or across datacenters using Ceph's multi-zone capabilities** [27] [28] **– useful if the user later wants an off-site mirror or cold archive (Ceph's new *Cloud Transition* module automates moving old objects to AWS S3/Glacier).** ***Trade-offs:*** **The obvious downside is** operational overhead**. Deploying Ceph requires more initial work (e.g., 3 nodes minimum, configuration of OSDs/Monitors) and ongoing tuning. It will consume more resources for equivalent workloads due to its stronger consistency and replication (e.g., a small Ceph cluster might use several GB of RAM just for overhead). In a limited environment, Ceph can be overkill, and misconfigurations can lead to performance issues. However, with proper setup, it is very stable. If the user values** rock-solid reliability and a rich ecosystem** (and possibly wants to consolidate object, block, and file storage in one system), Ceph is worth the investment. It

pairs well with Redis/Rabbit (via notifications) and can store Neo4j/Qdrant backups with absolute confidence in durability. It's the "heavy armor" option – more work to carry, but extremely powerful.

3. SeaweedFS **– *Lightweight & Cloud-Ready, Ideal for Many Small Files*. SeaweedFS is an excellent choice if simplicity and** local-first with cloud tiering **are top priorities. It has a very small footprint and can be set up in minutes, which is attractive for a nimble AI stack. Its performance for** numerous small files **or rapid metadata ops is a big advantage** [30] [2] **– for example, if the AI workflow generates thousands of snippet files or embeddings, SeaweedFS will handle those efficiently in a way that, say, Ceph might struggle without heavy resources. The ability to** automatically offload older volumes to cloud storage (S3) **is a unique feature** [1] **. This means the user can keep recent project data on fast local disks and let SeaweedFS migrate cold data to a cloud bucket periodically, achieving a hybrid storage strategy with minimal effort. SeaweedFS also integrates flexibly with external databases for metadata (including Postgres/Redis) which could align with the user's existing DBs** [36] **. *Trade-offs:* The S3 API support is** not full-featured **– notably, no object versioning or advanced IAM policies in the community edition** [32] **. If the user's security model requires per-user bucket policies or if they need to retain multiple versions of files, this is a limitation. Another trade-off is slightly weaker access control out of the box: SeaweedFS has basic key-based auth, but multi-user role separation isn't as baked-in (you might front it with a proxy for advanced auth, or use the experimental IAM mode). Notifications are not built-in, so hooking SeaweedFS to RabbitMQ would require a custom solution (e.g. tailing the filer log) rather than a config toggle. In summary,** SeaweedFS is fantastic for a small-team, local deployment** that may later leverage cloud storage for expansion – it keeps things simple and fast. Just be aware that you might sacrifice some S3 features and would need to implement any event-driven integration manually.

4. OpenIO **– *Flexible and Feature-Rich, with Dynamic Scaling*. OpenIO merits consideration especially if the user anticipates heterogeneous hardware use or wants sophisticated data placement control. Its** dynamic tiering and policy engine **allow mixing SSDs, HDDs, etc., and applying rules (for instance, important media files get 3 replicas on SSD while bulk data gets erasure-coded on HDD)** [137] [3] **. Such fine control could optimize performance-vs-cost better than one-size schemes. OpenIO is also proven in** production at scale**, and supports all essential S3 features (versioning, lifecycle, compression, encryption)** [5] [6] **. The** no-SPOF architecture **and self-balancing means as the AI data grows, adding nodes is straightforward and doesn't require rearchitecting the cluster** [75] **. *Trade-offs:* Deploying OpenIO is more complex than MinIO/Seaweed – expect to configure multiple services and spend some time learning its tooling. Its community, while solid, is not as large, so finding help might rely on official docs or reaching out to maintainers. OpenIO also doesn't natively push notifications to external queues (though one could possibly use its** metadata indexing or the built-in event framework **with some coding). Additionally, since OpenIO's founding company was acquired, one should verify the latest open-source updates; the project is stable at last release, but future enhancements might be slower unless community-driven. In essence,** OpenIO is a great all-rounder** if you want enterprise capabilities without going full Ceph, and are comfortable with a moderate ops overhead. It will integrate fine with the user's stack via S3 API and is compatible with tools (it even has a Python client library [46]). The trade-off is mainly the learning curve and less out-of-the-box integration for events.

5. Garage **– *Promising Newcomer for Self-Hosted Use*. Garage is an innovative choice for a self-hosted setup, built *by* self-hosters *for* self-hosters. If the user's priority is to have a** simple, decentralized storage **that can run on a few machines (even if spread across home/office), Garage is very attractive. It provides** built-in redundancy and geo-replication **without needing complex config – just run the**

nodes and it takes care of distributing data with the set replication factor [9] . Its lightweight nature means it won't hog system resources, leaving more room for AI services. Also, features like deduplication **can be a boon if the user stores many similar files (common in backup scenarios or versioned data) [10] .** ***Trade-offs:*** **As a newer project, Garage lacks some maturity and features. Most notably,** no server-side encryption or object versioning **yet [13] – so if compliance or backup retention is needed, you'd handle that at the application level. Access control is basic (you can make users and keys, but no advanced policy system). And integration-wise, you won't get automatic RabbitMQ events or the like; you'd need to implement any event triggers in the application. The community is helpful but small, so troubleshooting might rely on direct interaction with maintainers on forums. Choosing Garage is somewhat a bet that its simple approach will cover your needs and that missing features aren't critical for your use case. The trade-off is foregoing the rich feature sets of the older solutions for the sake of** operational minimalism**. For a contained environment (say a lab with a few servers and not a ton of concurrent users), Garage could perform admirably with very low admin effort. Just plan for alternative solutions for encryption (e.g., use LUKS on disks or client-side encryption libraries) if needed, and be prepared to upgrade Garage as it evolves.

Other solutions like **Zenko** did not make the top cut primarily because their value (multi-cloud coordination) isn't a pressing need in the described local-first scenario – the user can achieve optional cloud sync via simpler means (MinIO or Ceph's tiering, Seaweed's offload, etc.) without the overhead of Zenko's infrastructure. **OpenStack Swift** (the classic object store) was also considered but it lags in S3 compatibility and is generally overshadowed by Ceph or MinIO today for new deployments [138] . **Seagate CORTX** is another emerging open-source object store, but it's more oriented toward very large deployments and is relatively unproven in smaller-scale hybrid contexts, so it was not emphasized.

In conclusion, for a self-hosted AI backend in 2025, **MinIO** emerges as the top recommendation due to its superb alignment with the user's requirements (feature-rich, easy, integrable). For those willing to handle more complexity for ultimate scalability and completeness, **Ceph RGW** is a strong choice. **SeaweedFS** offers a compelling "set and forget" local solution with cloud scalability for those who need simplicity and speed with lots of files. The other technologies each have their niche strengths, and the final decision may weigh factors like team expertise, anticipated data growth, and specific feature needs (e.g., multi-tenant support or regulatory compliance). Each of the shortlisted options can fulfill the core need (storing diverse files with remote access and security); the recommendations differ mainly in **operational philosophy and expansion path**. By selecting one of these solutions, the user will equip their AI architecture with a solid foundation for object storage that can keep up with the demands of local-first processing and hybrid cloud collaboration.

Sources:

- Vitastor S3 implementation comparison (2024) – compatibility and feature support of MinIO, Ceph RGW, Zenko, Swift, SeaweedFS, Garage [139] [123] [124] .
- MinIO documentation and blog – details on security (encryption, RBAC) [17] [18] and event notifications (RabbitMQ, Redis, Postgres) [22] [125] .
- Ceph official docs – bucket notification feature (AMQP/Kafka support) [29] and multi-site/cloud tiering capabilities [27] [28] .
- SeaweedFS docs – architecture (O(1) metadata, tiered cloud storage) [1] [2] and encryption support [31] .
- OpenIO docs – dynamic storage policies (replication vs erasure) and encryption/IAM features [137] [41] .

- The Register's review of OpenIO – architecture highlights (no metadata bottleneck, Conscience load balancing) [73] [75].
- Zenko documentation – multi-cloud replication design and required services [7] [108].
- Garage documentation – feature list (replication modes, static website hosting, admin API) [9] [61] and encryption considerations [59] [60].
- Hacker News discussions – user experiences with MinIO, Ceph, SeaweedFS (performance, changes) [136] [140] and MinIO's focus on performance/AI use cases [89].
- JuiceFS vs SeaweedFS blog – confirms SeaweedFS encryption at rest and cross-cluster replication via changelogs [31] [33].

---

[1] [2] [30] [36] [70] [91] [92] [93] GitHub - seaweedfs/seaweedfs: SeaweedFS is a fast distributed storage system for blobs, objects, files, and data lake, for billions of files! Blob store has O(1) disk seek, cloud tiering. Filer supports Cloud Drive, xDC replication, Kubernetes, POSIX FUSE mount, S3 API, S3 Gateway, Hadoop, WebDAV, encryption, Erasure Coding. Enterprise version is at seaweedfs.com.
https://github.com/seaweedfs/seaweedfs

[3] [4] [5] [6] [38] [39] [41] [44] [45] [72] [77] [137] Core management features for your on premise object storage — OpenIO 20.04 documentation
https://docs.openio.io/latest/source/arch-design/data_management.html

[7] [8] [51] [118] [119] Introduction — Zenko 2.8.9 documentation
https://zenko.readthedocs.io/en/2.8.9/reference/introduction/

[9] [10] [11] [12] [57] [61] [64] [83] [84] [85] [88] [101] [111] [112] [113] [114] [121] [122] List of Garage features | Garage HQ
https://garagehq.deuxfleurs.fr/documentation/reference-manual/features/

[13] [24] [32] [48] [49] [50] [54] [99] [100] [123] [124] [138] [139] S3 implementation comparison — Vitastor
http://vitastor.io/en/blog/2024-05-09-s3-comparison.html

[14] [19] Object Lifecycle Management - MinIO
https://min.io/docs/minio/kubernetes/eks/administration/object-management/object-lifecycle-management.html

[15] [16] [17] [18] [130] [134] [135] How to Secure MinIO - Part 1
https://blog.min.io/secure-minio-1/

[20] MinIO Client SDK for Python - GitHub
https://github.com/minio/minio-py

[21] [22] [23] [125] [126] Part [1/6]: Publish Minio events via RabbitMQ | by Atul Jha | Medium
https://medium.com/@koolhead17/part-1-3-publish-minio-events-via-rabbitmq-8dd747632623

[25] [26] [29] [116] [129] Bucket Notifications — Ceph Documentation
https://docs.ceph.com/en/quincy/radosgw/notifications/

[27] Cloud Transition - Ceph Documentation
https://docs.ceph.com/en/latest/radosgw/cloud-transition/

[28] Cloud Sync Module - Ceph Documentation
https://docs.ceph.com/en/latest/radosgw/cloud-sync-module/

[31] [33] [34] [35] [37] [68] [69] [71] [95] JuiceFS vs SeaweedFS - JuiceFS Blog
https://juicefs.com/en/blog/engineering/similarities-and-differences-between-seaweedfs-and-juicefs-structures

40 47 73 74 75 76 98 106 107 117 Openio's objective is opening up object storage space • The Register
https://www.theregister.com/2015/12/02/openio_object_storage_upstart/

42 43 104 105 How to configure data encryption at rest in object storage — OpenIO 20.04 documentation
https://docs.openio.io/latest/source/admin-guide/configuration_encryption.html

46 Object Storage Python client library, SDK docs, API examples
https://docs.openio.io/latest/source/sdk-guide/python_example.html

52 53 78 79 80 81 82 108 109 110 132 133 Architecture — Zenko 1.2.1 documentation
https://zenko.readthedocs.io/en/latest/operation/Architecture/index.html

55 Configuration file format | Garage HQ - Deuxfleurs
https://garagehq.deuxfleurs.fr/documentation/reference-manual/configuration/

56 Garage (https://garagehq.deuxfleurs.fr/) gets pretty close for object ...
https://news.ycombinator.com/item?id=38514654

58 59 60 62 63 86 87 115 120 127 131 Encryption | Garage HQ
https://garagehq.deuxfleurs.fr/documentation/cookbook/encryption/

65 mc ilm tier update — MinIO Object Storage for Linux
https://min.io/docs/minio/linux/reference/minio-mc/mc-ilm-tier-update.html

66 67 Data Lifecycle Management and Tiering - MinIO
https://min.io/product/automated-data-tiering-lifecycle-management

89 90 94 96 97 102 103 136 140 Can someone with experience with Ceph and MinIO or SeaweedFS comment on how they... | Hacker News
https://news.ycombinator.com/item?id=32498332

128 Publish Events to Redis — MinIO Object Storage for Linux
https://min.io/docs/minio/linux/administration/monitoring/publish-events-to-redis.html