ChatGPT

Durable, Scalable Cache Options for a Python Backend Stack

Requirements and Context

Before comparing solutions, it's important to recap the requirements for the cache/key-value store in this backend stack:

- **Self-Hosted & Distributed:** The cache must be self-hosted (no cloud services) and capable of running across multiple machines as a unified distributed cache layer. This implies support for clustering or replication to scale horizontally.
- **High Durability:** Durability is prioritized over raw speed. The cache should reliably persist data to survive process crashes or reboots, making features like disk persistence essential.
- **Mixed Workloads:** It will handle both ephemeral data (transient cache like real-time tokens or session state) and more persistent data (resumable job states, cached computation results, possibly caching vectors from Qdrant). Thus it should support optional persistence on a per-use-case basis (e.g. some keys with TTL for ephemeral cache, others persisted).
- **Ecosystem Compatibility:** The stack uses Python (with SQLAlchemy, Dramatiq, etc.), plus PostgreSQL, Neo4j/Neomodel, Qdrant, and RabbitMQ. The cache should integrate smoothly with Python libraries and not conflict with these components.
- **High Concurrency & Observability:** The system is agent-based with high concurrency, so the cache should handle many simultaneous connections/operations. Monitoring and integration with observability tools (like Prometheus/Grafana) is also desired for maintenance.

With these in mind, let's evaluate leading candidates: **Redis**, **Dragonfly**, and other alternatives (such as **KeyDB**, **Memcached**, **Aerospike**, etc.), focusing on reliability, maintenance, Python compatibility, and monitoring.

Redis (Open-Source Redis 7.x)

Redis is a proven in-memory key-value store that checks many of the requirement boxes:

- **Durability Options:** Redis offers robust persistence settings. You can use point-in-time snapshotting (RDB files) and/or an Append Only File (AOF) log that records every write [1] [2]. RDB snapshots are taken at intervals, whereas AOF logs every operation (with configurable fsync policies). In practice, enabling both gives a balance of fast restarts and minimal data loss. For example, AOF with `appendfsync everysec` limits data loss to ~1 second in worst case [2]. This full range of persistence options is a key advantage of Redis [3]. It means Redis can be tuned for high durability (at some cost to throughput when fsyncing) to safely store critical state like job progress or cached results.

- **Distributed Scalability:** Redis supports **clustering** to spread data across multiple nodes (shards) while also allowing replication for high availability. In Redis Cluster mode, the keyspace is partitioned into slots and distributed across nodes (each shard has a primary and replicas) [4]. This allows horizontal scaling to larger datasets and throughput by adding shards, and ensures no single machine is a bottleneck. For high availability without sharding, Redis also offers **Sentinel** for automatic failover in a master-replica setup [5]. In short, you can scale beyond one instance and have replicas for failover. This meets the "multiple machines" requirement: e.g. a cluster of 3 masters + replicas can spread load and survive node loss. (Do note that Redis Cluster has some constraints – certain operations aren't multi-key across shards – but most cache use-cases are shard-friendly.)

- **Performance & Concurrency:** Redis is single-threaded for execution (uses one core per instance for most operations), but it can still handle very high throughput (hundreds of thousands of ops/sec) on a decent CPU. For read-heavy loads, Redis 7 added I/O threading to parallelize network handling. If one instance isn't enough for extremely high concurrency, Redis Cluster (sharding) can divide load, or multiple Redis instances can be used for different purposes. Latency is sub-millisecond for in-memory reads/writes in typical cases. While single-threading can be a limitation on CPU-bound workloads, it also simplifies consistency (no locking needed internally). Given the emphasis on durability over raw speed, Redis's performance is usually "good enough" for many scenarios, but it's not as scalable on one node as some multi-threaded designs (addressed below under alternatives).

- **Ease of Maintenance:** Redis is very mature (over a decade in production use) and well-supported. There's a large community and a wealth of tools and documentation. Operating Redis is straightforward for basic setups; running a cluster is more involved but well-documented. There are stable releases and the software is known for its reliability. Maintenance considerations include monitoring memory usage (to avoid eviction if maxmemory is hit) and managing persistence files (AOF rewrite and RDB snapshots scheduling). Many companies successfully self-host Redis with minimal issues, especially when configured conservatively for durability. Backing up RDB/AOF files is easy for disaster recovery. **High availability** can be achieved either with Sentinel (for a primary-replica pair) or with Cluster mode which has automatic failover if replicas are configured. Sentinel is simpler if you don't need sharding, while Cluster handles both HA and scaling. In either case, the operational burden is well-known and supported by many orchestration tools (e.g., Kubernetes operators for Redis, etc.).

- **Python Ecosystem Compatibility:** Redis has *excellent* integration in the Python world. The official `redis-py` client is feature-complete, and many Python frameworks and libraries support Redis out-of-the-box. For example, Django and Flask can use Redis as a cache or session store easily. Dramatiq (the task queue) can use Redis as a result backend or broker (though in this stack it uses RabbitMQ as broker, Redis could still be used for storing task results or rate limiting if needed). Libraries like `rq` (Redis Queue) or `celery` often use Redis. Neo4j/Neomodel likely doesn't directly tie to Redis, but nothing prevents using Redis for caching Neo4j query results or maintaining fast lookup tables to complement Neo4j relationships. SQLAlchemy can benefit from caching query results in Redis, etc. In short, using Redis will feel natural in Python – data can be `pickle`d or JSON-serialized into it as needed, and there are adapter libraries for caching layers (like `dogpile.cache`, Flask-Caching, etc.) that support Redis.

- **Monitoring & Integration:** Redis has numerous monitoring solutions. A popular approach is using the Redis **Prometheus Exporter**, which exposes Redis's internal INFO metrics to Prometheus/

Grafana [6]. This covers stats like memory usage, ops per second, connections, etc. There are also UIs like **RedisInsight** for interactive monitoring and debugging. Given Redis's popularity, virtually all APM or monitoring stacks have integrations (Datadog, New Relic, etc., all support Redis monitoring). You can set up alerts on memory or persistence lag (e.g., how far behind AOF fsync is, etc.). Redis's own `INFO` command provides a trove of metrics that can be polled. Overall, integrating Redis into your observability stack is straightforward.

Summary (Redis): Redis is a **reliable, well-rounded choice**. It provides strong durability (with AOF) [1], proven cluster scalability, and seamless Python integration. The trade-off is that it's single-threaded, so a single node can be CPU-bound if you have extremely high concurrent throughput needs – but this can be mitigated by sharding or using alternative builds (or by considering the next options). Given the requirements, Redis covers durability and ease of use best, making it a top contender for the primary cache store.

Dragonfly

Dragonfly is a newer in-memory datastore designed as a drop-in replacement for Redis (compatible with Redis protocols and data structures) but built for modern performance. It's worth evaluating for this scenario, especially because of its **performance and scaling model**:

- **Performance and Concurrency:** Dragonfly's claim to fame is its **multi-threaded**, shared-nothing architecture that can utilize multiple CPU cores efficiently. In benchmarks, a single Dragonfly node can handle significantly higher throughput than a single Redis instance (which is limited by one core). For example, on an AWS m5.xlarge (4 vCPU) test, Dragonfly achieved ~279k writes/sec vs Redis's ~190k on the same hardware [7], and the gap widens with more CPUs (reaching millions of ops/sec on large instances) [8]. This means Dragonfly can **vertically scale** to use a big multi-core machine, often matching or exceeding what a sharded Redis cluster could handle, but in one process. In a high-concurrency, agent-based system, this design is attractive because it can reduce bottlenecks – more simultaneous operations can proceed without queuing behind a single thread. Latencies remain low (sub-millisecond) even at high throughput, comparable to Redis in most cases [9]. Essentially, Dragonfly aims to provide Redis-like functionality with much higher performance headroom on modern servers.

- **Durability and Persistence:** This is where Dragonfly currently lags behind Redis. **Dragonfly supports only RDB-style snapshot persistence at the moment**, not a true AOF. By default it operates purely in-memory (like a cache), but you *can* configure periodic snapshots to disk. However, snapshots save state only at intervals, so recent writes since the last snapshot would be lost on a crash. The Dragonfly team has acknowledged that this is "not very durable" since the latest changes may not be captured [10]. They do *not* yet implement an append-only log of each write. As of mid-2025, Dragonfly **does not support AOF** persistence due to the complexity of implementing it in their multi-threaded engine (they have stated it's being explored for future releases) [11]. This means if durability is paramount, Dragonfly has an inherent risk – if the process or machine crashes, you could lose data since the last snapshot. You can mitigate this a bit by snapshotting very frequently, but that can impact performance and still isn't as safe as an AOF that logs every write. For ephemeral cache data that might be acceptable, but for "resumable jobs" or any state that absolutely must survive failures, this is a notable drawback. In short, **Dragonfly prioritizes speed and throughput, whereas durability is a weaker point currently**.

- **Distributed Operation:** Dragonfly can operate across multiple nodes, but it approaches clustering differently than Redis. It actually has two modes: an "emulated cluster" mode (where a single node pretends to be a Redis Cluster for client compatibility) and a true **multi-shard cluster** mode [12] [13]. In multi-shard mode, you can run multiple Dragonfly servers each responsible for a subset of the keyspace (similar to Redis Cluster's hashing of slots) [14]. However, *Dragonfly's server does not include the full cluster management logic.* The nodes don't automatically discover each other or failover on their own; there's no built-in gossip or configuration consensus. Instead, you have to configure the cluster manually (or use their enterprise "Swarm" service) [15] [16]. You assign hash slots to each node via admin commands and set up replication for failover manually [17] [18]. This means that while Dragonfly **can** scale horizontally, it might require more operational effort to set up and maintain compared to Redis's native cluster. On the plus side, Dragonfly does support replication for high availability – you can have replica nodes and use a similar failover approach as Redis (via a "replicaof" configuration) [17]. But since the cluster isn't self-managing, an external orchestrator or careful scripting is needed to monitor and handle failovers. The **bottom line**: Dragonfly is excellent for vertical scaling on one big node (potentially reducing the need to shard at all), but if you do need to shard or have multi-node clusters, it's currently a more manual process. If your dataset fits in a single machine's memory, Dragonfly could simplify things by avoiding a multi-node cluster in the first place (just use one strong node and perhaps one replica for HA).

- **Maintenance & Reliability:** Being a newer project, Dragonfly hasn't reached the same level of production mileage as Redis. Its codebase is young, though backed by a dedicated team. In practice, many are excited about it but it's still gaining trust for mission-critical deployments. One consideration is that Dragonfly's performance optimizations assume **modern hardware and kernel versions** (e.g., it benefits from Linux kernel 5.x features). If your environment is up-to-date, that's fine, but on older systems it may not realize its full potential [19]. The maintenance of Dragonfly itself (configuring snapshot persistence, replication, etc.) is not too complex, but since fewer people have run it in production, community knowledge is more limited. There is an active Discord and forum, and documentation is decent. Still, you may encounter edge cases or bugs that the Redis ecosystem ironed out long ago. It's also evolving quickly, so one must keep up with updates. **Monitoring** Dragonfly is fortunately not difficult: it exposes Prometheus-compatible metrics natively (on an HTTP `/metrics` endpoint) [20]. You can thus scrape internal stats directly and integrate with Grafana dashboards similar to Redis. Dragonfly also supports the Redis `INFO` command for basic stats via Redis client. So observability is well-covered.

- **Python Integration:** One of Dragonfly's design goals is to be a *drop-in Redis replacement*, so it speaks the same protocols. This means you can use the standard Python Redis client (`redis` or `rediscluster` library) and just point it to your Dragonfly server – no code changes needed [21]. From the application's perspective, it behaves just like Redis. This compatibility extends to data structures (strings, hashes, lists, sorted sets, etc.), so any caching logic written for Redis will work on Dragonfly. For example, if Dramatiq or other components have features expecting Redis (perhaps for locks or rate limiting), they should work similarly with Dragonfly. The key caveat is that Dragonfly is still catching up on less common Redis commands – most major ones are supported, but extremely new or obscure commands might be missing (check their docs if you use any advanced Redis features). For mainstream cache usage, you likely won't hit a difference. So from a development standpoint, Dragonfly fits very well into a Python stack that could otherwise use Redis.

Summary (Dragonfly): Dragonfly offers **superior performance and multi-core scalability** relative to Redis – potentially very useful in a high-concurrency environment – and it's compatible with your Python tools. However, it currently trades off some durability (only periodic snapshots, no true AOF logging [10]) and has a less mature clustering/HA solution. If raw throughput or handling extreme concurrency is the primary concern, Dragonfly is compelling. But given that *durability is a top priority*, adopting Dragonfly would require caution: you'd need to accept the risk of losing recent data on crashes or implement workarounds (very frequent snapshots, or application-level mitigation). It might be a better fit for purely ephemeral caching scenarios where losing data is tolerable (like transient caches that can be recomputed), rather than for persistent job state. Many teams currently stick to Redis for guaranteed persistence and use Dragonfly experimentally or for read-heavy caches where its speed shines.

Other Candidate Solutions

Aside from Redis and Dragonfly, there are a few other caching/key-value stores to consider, each with their own pros/cons:

- **KeyDB (Redis Fork):** KeyDB is a multithreaded fork of Redis that emerged a few years ago as a way to get Redis-like features with better vertical scaling. It retains full compatibility with Redis APIs and persistence mechanisms (supports RDB snapshots and AOF logging just like Redis) [22]. The big difference is performance: KeyDB can use multiple threads, so on a machine with many cores it can achieve higher throughput than vanilla Redis (the project advertises up to **~5x faster** throughput in some cases) [23]. One KeyDB instance can often replace several grouped Redis instances or remove the need for sharding on moderate workloads [24]. It also introduced features like active-active replication (multi-primary nodes) where you can have two or more nodes accepting writes (with conflict resolution by last-write-wins) [25] – this can simplify HA setups (no Sentinel needed for failover, since all replicas are writable and sync asynchronously [26]). **For this use case**, KeyDB's appeal would be better multi-core utilization *without sacrificing durability*, since it still offers AOF for full data safety. It's backed by Snapchat and has been used in production, though not as widely as Redis [27]. Maintenance of KeyDB is similar to Redis (it's basically Redis under the hood), but one should follow the KeyDB-specific releases for bug fixes. Some users have reported the occasional instability (e.g. rare crashes) [28], so it hasn't completely proven itself to be as rock-solid as Redis – but it's fairly mature at this point and actively maintained. Python integration is identical to Redis (use redis-py client). Monitoring can use the same Redis exporter or info command approach. **In summary**, KeyDB could be seen as a middle ground: it gives much of Dragonfly's performance boost (though Dragonfly may still win at extreme scales) while keeping Redis's durability and features. The trade-off is a smaller community and potentially fewer adoption references than Redis. It would be a **strong alternative** if you find vanilla Redis isn't handling the concurrency load but you still want AOF persistence and familiar management.

- **Memcached:** Memcached is a classic distributed cache, often compared alongside Redis. It is very simple, in-memory only, and multi-threaded. While memcached is *extremely fast* for basic get/set operations and can be sharded across nodes, it **does not meet the durability requirement**. Memcached has **no persistence** or built-in replication – if a memcached server restarts or crashes, all data in it is lost [29] [3]. It's essentially an ephemeral cache. This makes it unsuitable for storing any data that you cannot afford to recompute or lose. In our scenario, using memcached for ephemeral token caches *could* be fine (since tokens or ephemeral results could be regenerated or fetched from a database if the cache is lost), but it would be dangerous for things like resumable job

states. Additionally, memcached lacks native clustering; you can distribute keys across multiple instances, but the client or some proxy must implement the hashing (Ketama consistent hashing is commonly used on the client side). There's no automatic failover – if one node goes down, you lose that portion of the cache until it's back (and then cache misses will repopulate). **Maintenance** of memcached is minimal (it's lightweight and stable), but the burden is on the application to handle node coordination. Given that Redis/KeyDB/Dragonfly all provide more features (and Redis can be configured to behave like memcached if needed), memcached's only advantage here would be simplicity and raw speed for ephemeral data. However, since durability is a priority, memcached on its own is not an ideal primary solution. It might only be considered as a secondary cache layer for purely ephemeral, non-critical data if one wanted to offload that from the primary durable cache.

- **Aerospike:** Aerospike is an **enterprise-grade distributed key-value store** that can be used as a cache or NoSQL database. It's built for high performance at scale and has a strong focus on persistence and replication. Aerospike uses a hybrid memory model (indexes in RAM, data optionally on SSD or in RAM) and can achieve very high throughput with multi-threaded performance. For a caching layer, Aerospike offers some interesting features: you can run it in an in-memory mode with persistence to disk or even in a memory-only mode with persistence to shared memory for fast restarts [30] [31]. It automatically handles clustering, sharding, and replication across nodes – designed for multi-node operation out of the box. In terms of durability, Aerospike can be configured to commit to disk (or SSD) with strong consistency if needed; it is often touted as more *predictable* and *reliable* under heavy loads than Redis when scaling up, due to its optimized engine [32] [33]. **The downsides** for our use case are complexity and ecosystem mismatch. Aerospike is a heavier system to operate (it has its own config model, requires managing cluster consistency, etc.), and while it has a Python client, it's not as commonly used in typical Python web stacks as Redis is. Monitoring Aerospike is possible (it has monitoring tools and Prometheus integrations) but you'll have a smaller community to draw on for help. Aerospike shines in *mission-critical, large-scale* scenarios (e.g. caching for ad platforms, financial systems) where you need **very high throughput with strong durability and clustering** – essentially it could act as a combined cache+database. In a research-oriented environment, adopting Aerospike might be overkill unless the volume of data or throughput truly demands it. It's a solid option if you require cache-store behavior with no single point of failure and are willing to take on the operational overhead.

- **Hazelcast / Apache Ignite (In-Memory Data Grids):** These are Java-based in-memory data grid platforms that provide distributed caching with persistence options. They allow you to run a cluster of nodes (which can be co-located with your app or separate) and store key-value data (along with querying capabilities). They support writing data to disk or backing up to databases for durability. **Hazelcast** has a Python client, and **Ignite** has a thin Python client as well. Their strengths are easy horizontal scaling (data is partitioned automatically) and additional features like SQL-like queries, but their use in a Python-centric stack is less common. Maintenance can be significant: you'd be running a Java cluster (which means JVM tuning, etc.), and the community usage skews to Java ecosystems. If your system were polyglot or needed an in-memory compute grid, these would be more attractive. For a pure caching layer, they are likely more complex than needed. They could be considered if, for instance, you wanted an **embedded cache in each agent process with near-cache** capabilities (Hazelcast can do that) or if you already had a Java component to integrate. However, given the existing stack doesn't include JVM components, introducing Hazelcast/Ignite might add unnecessary complexity. Python's compatibility is fine (client API), but you won't find as much built-in support in Python frameworks compared to Redis.

- **Cassandra / ScyllaDB:** These are distributed databases (based on the DynamoDB model) rather than pure in-memory caches, but some architectures use them as a **distributed key-value store with high write throughput and persistence**. They excel at durability and multi-datacenter replication. However, they operate on disk (with caching layers) so they have higher read/write latencies (millisecond-level) compared to an in-memory cache like Redis. If the requirement for durability outweighed latency completely, one might consider using a fast NoSQL store like Scylla (a high-performance C++ implementation of Cassandra) to store the "cache" data so that it's always persistent. But this would complicate the design and likely be over-engineering if the goal is sub-millisecond responses. These systems are typically used for primary data storage at massive scale. For a cache layer, sticking to in-memory systems is preferable unless you truly need to cache *on disk* due to extremely large data sizes. (For completeness: **Couchbase** is another option bridging cache and database – it has an in-memory caching layer and disk persistence, and supports key-value operations as well as JSON querying. It's often used as a distributed cache with persistence. But Couchbase, like Cassandra, is heavier to maintain than Redis and usually only worth it if you need its additional document DB capabilities or scaling to huge sizes.)

In summary, **Redis, Dragonfly, and KeyDB** are the primary contenders for a durable, distributed cache in a Python environment. Memcached is out for durability reasons [3] (though fine for a secondary ephemeral cache if needed). Aerospike and others offer strong durability and scalability but at the cost of complexity and less Python-native feel.

Monitoring & Maintenance Considerations

Whichever solution is chosen, it should integrate with your monitoring/alerting. Redis/KeyDB/Dragonfly all expose metrics that can be captured. Redis and KeyDB can use the Redis Exporter for Prometheus, and Dragonfly provides a built-in metrics endpoint [20]. Ensure to monitor key stats like memory usage (to avoid OOM or eviction), persistence health (AOF fsync latency or RDB snapshot intervals), and throughput/latency (to catch any bottlenecks). Also plan backup strategies for persisted data (e.g., off-site backup of AOF files or RDB snapshots if using Redis/KeyDB). Maintenance also involves software upgrades: Redis is very stable across versions; Dragonfly is newer so updates might bring big improvements (e.g., if/when they add AOF support in future) – keep an eye on release notes.

Recommendation

Top Recommendation: Given the requirements, **Redis (open-source)** is the safest and most robust choice as the primary cache/storage layer. It offers the best durability guarantees (snapshots plus AOF for minimal data loss) [1], proven stability, and smooth integration with the Python stack and existing tools. Redis Cluster or Sentinel-based replication can fulfill the multi-machine scalability and high availability needs [4]. You will benefit from a large ecosystem of management and monitoring tools, and your team is less likely to hit surprises in production. By tuning Redis persistence (e.g. AOF with `everysec` or even `always` fsync if truly needed), you can ensure reliability of critical cached state. Redis's minor performance limitations (single-threaded) are not likely to outweigh its strengths in durability and support for most scenarios – especially since you can scale out with clustering if needed. In short, **Redis is a durable workhorse** that fits well.

Backup Alternative: As a secondary option, **KeyDB** is worth considering if you find Redis's single-thread performance to be a bottleneck. KeyDB would give you a boost in throughput on multi-core machines while still retaining Redis's durability mechanisms and operational familiarity [22] [34]. It could be a drop-in improvement if write volume or concurrency grows beyond what a single Redis instance can handle, before committing to a sharded Redis cluster. The trade-off is a smaller community and the need to verify stability for your use case – but it has been used in large-scale environments, and it simplifies scaling by using one node more effectively.

What about Dragonfly? Dragonfly is an exciting new technology and might become a top choice in the future, especially if/when it adds full AOF persistence. At present, however, its **lack of true durability** is a serious concern for the use cases described (where we don't want to lose resumable job data, etc.) [10]. If you have portions of your caching use-case that are purely ephemeral (where losing that cache is acceptable and only results in recomputation or slight disruption), Dragonfly could be introduced for those specific cases to leverage its speed. But for the primary cache where you need **reliable recovery**, it's risky to rely on snapshot persistence alone. One strategy could be to use Redis/KeyDB as the primary durable cache, and experiment with Dragonfly in non-critical pathways to see its benefits (since it speaks Redis protocol, this is feasible to A/B test). As Dragonfly matures and perhaps implements append-only persistence, it could be re-evaluated.

In conclusion, **Redis** (with disk persistence enabled) is the recommended choice due to its proven reliability, ease of maintenance, rich Python ecosystem support, and extensive monitoring integrations. **KeyDB** can be a strong alternative if you require higher single-node throughput without losing durability. These solutions will give you a durable and scalable caching layer that integrates well with your stack. Dragonfly, while offering phenomenal performance, should be approached cautiously for durable use cases in 2025 – it can be a backup candidate for high-throughput scenarios where durability is secondary, or kept on the radar for the future once it meets the durability bar.

Sources: Redis and KeyDB documentation, Dragonfly docs and community insights, and industry comparisons have been cited throughout to support this evaluation [1] [10] [11] [4], among others. These underscore the differences in persistence, scalability, and performance between the options discussed.

---

[1] [2] [Answered] What is an in-memory database with persistence and how can it be implemented?
https://www.dragonflydb.io/faq/in-memory-database-with-persistence

[3] [6] Redis vs Memcached | Redis
https://redis.io/compare/memcached/

[4] [5] Understanding Redis High Availability: Cluster vs. Sentinel | by Praful Khandelwal | Medium
https://medium.com/@khandelwal.praful/understanding-redis-high-availability-cluster-vs-sentinel-420ecaac3236

[7] [8] [9] GitHub - dragonflydb/dragonfly: A modern replacement for Redis and Memcached
https://github.com/dragonflydb/dragonfly

[10] Deploy Dragonfly on Northflank with Docker — Northflank
https://northflank.com/guides/deploy-dragonfly-on-northflank-with-docker

[11] Append Only File (AOF) | Dragonfly
https://www.dragonflydb.io/docs/managing-dragonfly/aof

12 13 14 15 16 17 18 21 Cluster Mode | Dragonfly
https://www.dragonflydb.io/docs/managing-dragonfly/cluster-mode

19 23 24 27 28 A Battle of Lightning-Fast In-Memory Databases in the Real World ⚡ - Blog by Saifeddine Rajhi
https://seifrajhi.github.io/blog/in-memory-databases-comparison/

20 Monitoring | Dragonfly
https://www.dragonflydb.io/docs/managing-dragonfly/monitoring

22 25 26 34 KeyDB - The Faster Redis Alternative
https://docs.keydb.dev/

29 Memcache(d) vs Redis? I'm particularly interested in clustering, but ...
https://www.reddit.com/r/PHP/comments/264s1b/memcached_vs_redis_im_particularly_interested_in/

30 31 caching - Redis vs Aerospike vs Couchbase as a cache use cases? - Stack Overflow
https://stackoverflow.com/questions/78911682/redis-vs-aerospike-vs-couchbase-as-a-cache-use-cases

32 What is Redis cache? - Aerospike
https://aerospike.com/compare/redis-vs-aerospike/redis-cache/

33 MemcacheD vs Redis vs Aerospike — Which One to Choose for ...
https://www.linkedin.com/pulse/memcached-vs-redis-aerospike-which-one-choose-your-design-saxena-zk6uc