ChatGPT

Valkey Beginner’s Guide

Concepts and Terminology

What is Valkey? Valkey is an open-source, in-memory key-value data store designed as a drop-in replacement for Redis OSS [1]. It was created as a community-driven fork of Redis 7.2 when Redis Inc. announced future versions would not be fully open source [2]. Backed by the Linux Foundation, Valkey will remain BSD-licensed open source forever [2]. In essence, Valkey retains the core concepts of Redis – it stores data in memory for ultra-fast access, supports persistence to disk, and offers the same rich set of data structures (strings, hashes, lists, sets, sorted sets, streams, bitmaps, HyperLogLogs, geospatial indexes, etc.) [3]. This means you can use Valkey as a cache, message broker, session store, or even a primary NoSQL database, much like Redis.

Key Terminology: Valkey uses modern terminology similar to newer Redis versions. A **primary** is the read-write master node (previously called “master” in Redis), and a **replica** is a read-only copy (formerly “slave”). A **sentinel** is a specialized Valkey process that monitors primaries and replicas for high availability and can automatically perform failover (more on this later) [4]. **Valkey Cluster** refers to running Valkey in a distributed, sharded mode for horizontal scaling and high availability, using multiple nodes and partitioning the keyspace into **hash slots** (Valkey uses 16,384 hash slots, identical to Redis Cluster) [5] [6]. Each node in a cluster handles a subset of slots, and data is automatically sharded among nodes. **Numbered databases** (sometimes called logical databases, indexed by a DB number like 0,1,2,… up to 15 by default) allow separate key namespaces on a single instance. In Redis, numbered DBs were disabled in cluster mode (cluster always used only DB 0), but Valkey 9.0 introduces support for multiple databases even in cluster mode [7], effectively providing namespacing without requiring key prefixes. Other terms you’ll encounter include **TTL** (time to live expiration for keys), **Lua scripting** (using the embedded Lua 5.1 engine or extended function API), and **modules** (plugins that add new data types or commands). Valkey supports Redis’s module system, so you’ll see official Valkey modules for JSON (document support), Bloom filters, vector search, etc., which extend its capabilities.

Valkey’s **data model** and commands are virtually identical to Redis’s. It supports atomic operations on data structures (e.g. pushing to a list, incrementing a counter), pub/sub messaging, and transactions with MULTI/EXEC. The **keyspace** (set of all keys in the database) can be managed with similar commands (SCAN, KEYS, etc.), and Valkey has the same notion of **key eviction policies** when used as a cache with a max memory limit (all Redis eviction strategies like LRU, LFU are available) [8]. Because Valkey is protocol-compatible with Redis, existing Redis client libraries and tools can communicate with Valkey without modification in most cases [9]. In summary, if you know Redis, you already understand most Valkey concepts – the difference lies in new features and the open-source governance of Valkey.

New Features in Valkey (Compared to Redis 7.2)

Valkey’s initial version (7.2.4) started as a fork of Redis 7.2, so it included all Redis 7.2 features, but the community has rapidly extended and improved it. **Open-source forever:** The most fundamental “feature” of Valkey is its license – unlike Redis 7.2 which was the last truly open Redis version, Valkey

will remain permissively licensed (BSD-3-Clause) under community stewardship [2] [10]. Beyond licensing, Valkey introduces several technical enhancements and new capabilities beyond Redis 7.2:

- **I/O Multi-threading and Performance Boosts:** Valkey 8.0 introduced a new I/O threading architecture that dramatically improves throughput and latency. By parallelizing parts of command processing, Valkey 8.0 achieved up to **230% higher throughput and 70% lower latency** compared to Valkey 7.2 (Redis 7.2) under heavy workloads [11]. In Valkey 8.1, these gains were extended: more operations (like aspects of networking and TLS) are offloaded to background I/O threads, further increasing performance. For example, offloading TLS handshake and buffer processing to I/O threads improved new connection acceptance rate by ~300%, and yielded ~10% higher SET throughput and ~22% higher GET throughput in benchmarks [12]. The number of I/O threads is configurable (similar to Redis 6+ `io-threads` setting) – by default Valkey runs in single-threaded mode for execution, but enabling additional I/O threads can improve performance on multi-core systems for network-heavy or TLS workloads. The net result is Valkey can handle significantly more operations per second than Redis 7.2 on the same hardware when tuned, while retaining the single-threaded command execution semantics to avoid data races.

- **Memory Efficiency Improvements:** Valkey 8.0 and 8.1 include optimizations to reduce memory overhead per key. A major change in Valkey 8.1 was a **redesigned hash table** implementation for the keyspace and core data structures [13]. This new hash table uses modern techniques to pack and align data more efficiently, resulting in roughly **20–30 bytes of memory saved per key** (20 bytes saved for keys without expiration, ~30 bytes for keys with TTL) [14]. This optimization means you can store more keys in the same amount of RAM. In Valkey's own benchmark of 50 million entries, Valkey 8.1 showed significantly lower memory usage than 8.0. Additionally, Valkey 8.1 optimized certain data type operations using SIMD instructions – e.g. the HyperLogLog `PFCOUNT` and `PFMERGE` operations are up to **12x faster** using AVX2 acceleration [15], and BITCOUNT on large bitmaps is over **5x faster** with AVX2 [16]. Sorted set operations got targeted speedups too (e.g. `ZRANK` up to 45% faster) [17]. These improvements collectively mean Valkey can deliver better performance and use less memory than Redis 7.2 by default.

- **New Data Capabilities and Commands:** The Valkey community has added features that were not available in Redis 7.2:

- Hash Field Expiration: Valkey 9.0 (upcoming at the time of writing) introduces the ability to set expirations on individual fields within a hash [18]. In Redis/Valkey up to 8.x, expirations (TTLs) could only be set on whole keys. With per-field TTLs in Valkey 9.0, you can have finer-grained cache invalidation or session expiration within a single hash object (for example, auto-expiring certain fields). This feature required significant changes to how expirations are tracked and is a unique Valkey enhancement.
- Conditional Updates (Compare-and-Set): Valkey 8.1 added a new option to the `SET` command for conditional updates. Using `SET ... IFEQ <expected-value>` allows atomic set-if-equals logic on a key [19]. In other words, the `SET` will succeed only if the current value matches the given expected value. This eliminates the need for a separate GET and compare in many cases and saves a round trip. (Valkey also supports the standard NX/XX options from Redis for set-if-not-exists or set-if-exists. The new IFEQ option is analogous to a CAS operation and was not available in vanilla Redis.)
- Cluster Multiple Databases: As mentioned, Valkey 9.0 lifts a Redis Cluster limitation by supporting **numbered databases in cluster mode** [7]. You will be able to `SELECT` a different database index even when connected to a cluster. Keys in different logical DBs are still distributed across the cluster slots the same way (the hash slot is computed from the key name as usual; the DB index

does not affect slot calculation) [20] [21] . Essentially each slot now contains 16 logical sub-databases. This feature is mainly for namespacing convenience – it allows multi-tenant usage of one cluster or separating data without key-prefix conventions. (Note: This doesn't provide resource isolation; it's purely a logical separation [22] . Heavy use of multiple DBs can still have "noisy neighbor" effects, so use case should be considered carefully.)

- Improved Observability: Valkey 8.1 introduced a **COMMANDLOG** feature extending Redis's Slow Log. While the Slow Log tracked only slow commands, Valkey's new command log can record **large or slow** commands, capturing **payload sizes and response sizes** for troubleshooting end-to-end latency [23] . This helps identify commands that might be fast server-side but slow overall due to huge reply size or network. Additionally, the built-in **latency monitor** was enhanced – the `LATENCY LATEST` output now includes the total count of events and cumulative latency for each monitored event type [24] . This gives more insight into how often spikes occur. Logging is more flexible too: you can switch Valkey's log output to structured logfmt format and ISO8601 timestamps for easier ingestion by log aggregators [25] [26] .
- Module Ecosystem (JSON, Search, Bloom, etc.): Because Valkey continues as an open-source project, modules that were formerly proprietary in Redis Stack now have open-source equivalents. The community has released **Valkey JSON** (a module for JSON document storage with JSONPath querying) [27] , **Valkey Search** (for vector similarity search and secondary indexing of data, comparable to Redis Search) [28] , and **Valkey Bloom** (Bloom filters data type) among others. These modules can be loaded into Valkey to extend its functionality with features similar to RedisJSON, RediSearch, etc., but under open licenses [29] . For convenience, there is even a **Valkey-Bundle** Docker image that packages Valkey with official modules like JSON, Bloom, Vector Search, and LDAP auth in one container [29] . This "batteries-included" option makes it easy to deploy a single Valkey instance that has the capabilities of an entire Redis Stack (useful for development and certain production use cases).

Aside from the above, Valkey incorporates all features of Redis 7.2 (ACLs for authentication and authorization, client-side caching support (`CLIENT TRACKING`), Streams with consumer groups, PUB/SUB, Lua scripting, etc.). Valkey's enhancements are largely backward-compatible – you can use it as a direct replacement for Redis but gain performance and new commands. For example, AWS in their ElastiCache service reports Valkey 8.0's threading and memory optimizations yield **20%+ memory savings and substantial throughput gains** over Redis [11] . If you stick to Redis 7.2 features, Valkey behaves the same; if you choose to use new features (like IFEQ or hash field TTLs), ensure your client libraries are updated to handle those new commands.

Installation Guide

You can install Valkey on a CentOS server or in a container with minimal effort. Below are common installation methods and then we'll cover configuring Valkey for different deployment topologies.

Installing Valkey on CentOS (and Linux)

Valkey provides pre-built binaries and is also packaged in many Linux distributions. For CentOS 7/8 (or RHEL and Fedora), the easiest way is to use `yum` if a Valkey package is available. First, enable EPEL (Extra Packages for Enterprise Linux) if needed (Valkey may be packaged there) [30] . Then run:

```
sudo yum install valkey
```

This installs the server (`valkey-server`) and CLI (`valkey-cli`). Optionally, you can install `valkey-compat-redis` which sets up symlinks so that `redis-server` and `redis-cli` commands point to

Valkey (for backward compatibility in scripts) [31]. You may also install `valkey-doc` for local man pages (`man valkey.conf`, etc.) [32]. On Debian/Ubuntu systems the process is similar with `apt install valkey` [33]. On Alpine: `apk add valkey` [34]. In case your OS's standard repos don't have Valkey yet, you can download the official binary tarball from the Valkey GitHub releases page [35] and run the `valkey-server` binary directly (or compile from source with a simple `make` command [36]).

After installation, you'll have a `/etc/valkey.conf` default configuration file (similar to redis.conf). For a quick test, you can launch Valkey in the foreground with default settings by running:

```
valkey-server
```

(This listens on port 6379 by default). To run it as a background service, edit the `valkey.conf` (e.g. set `daemonize yes`) or use your OS service manager (systemd unit files may be included). On CentOS with systemd, you might manage it via `systemctl enable --now valkey`.

Verify the installation: Use the CLI to confirm Valkey is running. The Valkey CLI syntax is the same as Redis CLI. For example, try:

```
$ valkey-cli PING
PONG
```

If you get "PONG", the server is up and responding [37]. Running `valkey-cli` with no arguments enters an interactive shell (REPL) where you can type commands directly [38].

Running Valkey via Docker

Valkey provides official Docker images on Docker Hub for quick deployment. If you prefer not to install directly on the host, you can pull and run an image. For example:

```
docker run -d --name my-valkey -p 6379:6379 valkey/valkey:8.1.4
```

This command will download the Valkey 8.1.4 image and run the server detached, exposing port 6379 to the host [39]. The Docker image tag names usually correspond to Valkey versions (e.g., `valkey:8.1.4` or `valkey:7.2.11` for the LTS 7.2 branch [40] [41]). There are also `-alpine` minimal images. Running Valkey in Docker is convenient for testing and even production, though for production you'll want to mount a volume for persistence (to store RDB/AOF files outside the container) and tune memory limits.

Important: If using Valkey Cluster in Docker, be aware that clustering doesn't work behind NAT or port mapping without additional configuration. Valkey Cluster nodes need to know their externally reachable address and require direct node-to-node communication. Typically you must run with `--net=host` in Docker or use the cluster's ability to set an `announce-ip`. Standard master-replica setups (non-cluster) do not have this issue and can run in Docker with normal port mapping. Always ensure the container's ports (6379 and the cluster bus port 16379) are open as needed [42] [43].

High-Availability Deployment Architectures

Valkey can be deployed in two primary HA modes: a **Primary-Replica (master-slave) architecture** (optionally coordinated by Sentinel or an external tool) or **Cluster mode** (sharded cluster with replicas). We'll overview how to set up each:

Primary-Replica Mode (with Keepalived or Sentinel)

In this mode, you have one primary (master) that handles all writes, and one or more replicas that asynchronously replicate the primary. Replicas can serve read traffic (if your app is read-heavy) and stand by to be promoted if the primary fails. By default, Valkey replication works just like Redis: replication is asynchronous and non-blocking – the primary will send an initial sync (RDB snapshot) to a new replica and then stream updates. Replicas connect to the primary using the `replicaof <host> <port>` configuration (or issue the `REPLICAOF` command at runtime). You can verify replication status via `INFO REPLICATION` – replicas will show `master_link_status:up` when connected.

Keepalived + Virtual IP failover: One simple way to achieve automatic failover in a primary-replica pair is using Keepalived with VRRP. In this setup, the primary and replica are both configured with Keepalived to manage a floating **virtual IP (VIP)** address. The VIP always points to whichever node is the primary. If the primary node goes down, Keepalived on the replica will automatically "takeover" the VIP, so that clients connecting to the VIP seamlessly start hitting the replica. Of course, the replica must also be promoted to primary role in this event – this can be done via a Keepalived notification script. For example, a notify script can call `valkey-cli` to run `REPLICAOF NO ONE` on the replica when it becomes the VIP holder, thereby making it a primary [44] [45]. The original primary (when it comes back) would then need to be set as a replica or kept out of rotation. Keepalived handles IP failover in a few seconds or less, offering a very simple HA solution. **However, this approach does not coordinate data replication state** – it's possible a few latest writes are lost if the primary failed before syncing to the replica (since the failover is not synchronous). As the community notes, VIP failover is best for use cases like caching or session stores where slight data loss is acceptable [46] [47]. It provides high availability without needing a third node (no quorum), but you sacrifice some consistency guarantees.

Sentinel for automatic failover: A more robust, official solution is **Valkey Sentinel**, which is analogous to Redis Sentinel. Sentinel is a separate process (or set of processes) that monitors Valkey primaries and replicas and handles failover logic [4]. You typically run an odd number of Sentinel instances (minimum three for a quorum) on different hosts. They communicate and agree on the state of the primary. If the primary fails (not responding to pings), the Sentinels will elect a leader to coordinate failover: one of the replicas is chosen and promoted to become the new primary, and the other replicas (if any) are reconfigured to follow the new primary [48]. Sentinel also updates clients with the new primary address – client libraries that support Sentinel can query the Sentinels to get the current primary (this avoids needing a VIP). To use Sentinel, you start the `valkey-sentinel` executable with a config file that lists the primary to monitor and quorum settings [49] [50]. For example, a minimal **sentinel.conf** might contain:

```
sentinel monitor mymaster <primary_ip> 6379 2
sentinel down-after-milliseconds mymaster 60000
sentinel parallel-syncs mymaster 1
sentinel failover-timeout mymaster 180000
```

This tells Sentinels to monitor a primary named "mymaster" at the given address, with a quorum of 2 (meaning at least 2 Sentinel instances must agree the primary is down to trigger failover) [50]. The other lines configure timing. When running, Sentinels announce themselves to each other and to replicas. They

will constantly check the health of the primary and also monitor the replication lag of replicas. In a failover event, one Sentinel will send the target replica the command to `REPLICAOF NO ONE` (making it primary) and update the others. Sentinels then inform clients of the new primary via Pub/Sub or direct responses to Sentinel queries.

Sentinel vs Keepalived: Sentinel requires running additional processes and client integration (or a service discovery mechanism) to redirect to the new primary, whereas Keepalived provides a transparent IP-level failover. Sentinel, however, can make a more informed choice of new primary (for example, it will pick the most up-to-date replica), and it avoids split-brain by requiring quorum agreement for failover [51] [52]. Many users in production use Sentinel for Valkey/Redis HA since it's a proven design. Your choice can depend on environment: if using cloud or containerized deployments, Sentinel is usually preferred (or using cluster mode), whereas Keepalived could be simpler for on-prem bare-metal with controlled network.

Deployment tip: If you use Sentinel, run at least **3 Sentinel instances** (they are lightweight). They should run on separate hosts or VMs for fault tolerance [53]. Also ensure your Valkey clients or your connection layer can handle Sentinel; many popular client libraries (Redis-py, Jedis, etc.) have Sentinel support (they'll ask Sentinel for the primary's address on connect, and auto-reconnect on failover). If not, you may need to update client configuration manually on failover, or use a proxy layer. Additionally, secure Sentinel as you would Valkey (Sentinel can be configured with an auth password so that it can authenticate to Valkey primaries and also require auth from clients).

Cluster (Sharded Mode with Valkey Cluster)

For applications that need to scale beyond one node's memory or CPU and want a built-in sharding solution, **Valkey Cluster** mode is ideal. Valkey Cluster allows you to run many Valkey nodes that share data via sharding, and also provides high availability through replica failover within the cluster. A cluster is composed of multiple **shards** (each shard is a set of one primary and N replicas) distributed across nodes. Data keys are partitioned into **16384 hash slots** using CRC16 hashing (same as Redis) [5]. Each primary is responsible for a subset of those hash slots. For example, in a cluster with 3 primaries, one might cover slots 0–5500, another 5501–11000, and another 11001–16383 [54]. The cluster manager ensures all slots are covered and no overlaps.

High availability in cluster: Within each shard, if the primary fails, one of its replicas is automatically promoted by the cluster itself (this is coordinated by the remaining primaries via the cluster bus). So, unlike Sentinel, you don't need external processes – the cluster nodes themselves detect failures and perform failover. However, to avoid total cluster outage, a majority of primaries must be active. (If too many primaries fail, some slots become unavailable and the cluster stops accepting writes until quorum is restored). Typically, you'd deploy cluster with at least 3 primaries and at least 1 replica each for safety [55] [56]. For example, a 3 primary + 3 replica cluster can tolerate up to one primary failure (since its replica will take over). If a primary and its replica both go down, that shard's slots are unavailable.

Cluster setup: To create a Valkey cluster, you start multiple `valkey-server` instances (either on one machine with different ports or on multiple machines). Ensure `cluster-enabled yes` (or `--cluster-enabled yes` CLI flag) is set in their configs, and each node has a unique `cluster-node-timeout` and node ID (Valkey will generate one). Then, you use the `valkey-cli --cluster create` command to connect the nodes together. For example, if you started 6 instances on ports 7000–7005 (3 intended primaries and 3 replicas), run:

```
valkey-cli --cluster create 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 \
--cluster-replicas 1
```

This will auto-configure a cluster with 3 primaries and 3 replicas (the `--cluster-replicas 1` option means one replica per primary) [57] [58]. The tool will assign slots to primaries evenly and pair each replica to a primary. You'll be prompted to confirm, and upon success you should see "[OK] All 16384 slots covered" [59]. After that, the cluster is live; the nodes will know about each other and exchange heartbeat messages over the cluster bus (which uses the port 10000+offset, e.g. 16379 if data port is 6379) [60].

Using the cluster: To talk to a clustered Valkey, your client must be cluster-aware (able to follow redirects). If you use `valkey-cli`, you can specify `-c` flag for cluster mode and connect to any one node: the CLI will automatically follow redirections for commands that are hashed to a different node [61] [62]. Most Valkey/Redis client libraries have cluster support (for example, Jedis or Lettuce in Java, redis-py in Python, etc., can map keys to the right node). The cluster presents a single unified keyspace spread across nodes – but note that multi-key operations are limited to keys in the same hash slot (you can use hash tags `{...}` in keys to force certain keys to the same slot if needed) [63]. If you attempt a transaction or Lua script touching keys on different slots, Valkey will return a cross-slot error.

Cluster management: You can reshard or add/remove nodes from a Valkey cluster using commands like `CLUSTER MEET`, `CLUSTER REPLICATE`, and `CLUSTER RESHARD`. The `valkey-cli --cluster` utility also has subcommands to assist with rebalancing slots and check cluster consistency. In production, you should also set up node configuration such that each node knows its external IP/port (especially if running in different networks or containers). Use the `cluster-announce-ip` and `cluster-announce-port` settings if needed. Monitor the cluster with `CLUSTER INFO` – it reports stats like cluster_state (ok/fail), number of slots, etc. Additionally, each node's `INFO REPLICATION` will show its role in the cluster (a node in cluster still reports itself as master or slave (replica) in the replication section).

Cluster vs Sentinel: Cluster mode is a more scalable solution (it shards data and can handle more data/requests by adding nodes). It's typically used when you have large datasets or need to distribute load. Sentinel (primary-replica) is simpler and might be used when you want strong consistency on a single shard (though keep in mind both Sentinel and Cluster use asynchronous replication, so neither guarantees zero data loss – for absolute consistency, an alternative approach like MemoryDB's multi-AZ transaction log is used, but that's beyond our scope). If you start small, you can begin with a single primary and replicas (Sentinel-managed) and later migrate to cluster if needed. In fact, some cloud services allow migration from a non-clustered setup to a clustered one (with "cluster mode enabled") when scaling up.

Administration and Monitoring Guide

Running Valkey in production requires attention to certain metrics and behaviors to ensure it's healthy and performing well. Here we cover important things to monitor and how to verify normal operation.

Important Metrics to Monitor

You can retrieve a wealth of runtime metrics from Valkey using the `INFO` command. Key sections and metrics to watch include:

- **Memory Usage:** In `INFO Memory`, check `used_memory` (bytes in use) and `used_memory_peak`. Compare `used_memory` to your `maxmemory` configuration (if set). If `used_memory` is near `maxmemory` and you have an eviction policy, keys will be evicted (monitor `evicted_keys` counter in INFO Stats). Also watch `used_memory_rss` (resident set size) versus `used_memory`. A much higher RSS than used_memory indicates fragmentation – Valkey's active defragmentation (if enabled) will try to address this. In Valkey 8.1, active defrag was improved to reduce latency impact [64], but you still want to ensure fragmentation is under control. The `MEMORY DOCTOR` command can provide suggestions if memory usage is suboptimal [65].

- **CPU and Latency:** Valkey itself doesn't expose CPU usage via INFO, so use external OS monitoring for CPU load. High CPU could indicate heavy command processing. **Latency** can be monitored via the internal latency monitor: if `latency-monitor-threshold` is set (e.g. < `100` ms), Valkey logs spikes. Use `LATENCY LATEST` to see recent events; in Valkey 8.1 this output now includes how many spikes and total time spent in them [66]. Also check `slowlog-get` for any slow commands recorded (by default, commands taking over 1 millisecond, configurable via `slowlog-log-slower-than`). A growing slow log indicates some operations are taking long (maybe large data transfers or blocking commands). The new `COMMANDLOG` (if enabled) will help track large payloads too [67].

- **Clients and Connections:** `INFO Clients` shows `connected_clients` and potentially `blocked_clients` (clients waiting on blocking commands like BLPOP). If blocked_clients is high, it may indicate usage of blocking operations – ensure that's expected. If you see a high number of connected clients, monitor for hitting connection limits (`maxclients` setting). Also, look at `connections_received` (cumulative) and `rejected_connections` – any rejected connections could mean you hit maxclients or the backlog queue was full.

- **Persistence:** If you use AOF or RDB snapshots, monitor those. `INFO Persistence` gives `rdb_last_bgsave_status` and timestamp, as well as AOF current size, last rewrite time/status. If `rdb_last_bgsave_status:ok` and `aof_last_write_status:ok`, then persistence is working normally. A failure here (status = err) could indicate disk full or permission issues – critical to catch, because it means your data isn't being persisted. Also check `aof_pending_rewrite` or `rdb_bgsave_in_progress` to see if a background save is currently running. Monitoring the interval between successful snapshots (if using periodic RDB) ensures you have recent backups.

- **Replication health:** On a primary, `INFO Replication` lists replicas and their state. Each replica line has `state=online` and lag info. On a replica, INFO Replication shows `master_link_status:up` when connected. If a replica's `master_link_down_since_seconds` is not 0, it's disconnected – investigate network or primary status. The `repl_backlog_size` and `repl_backlog_histlen` on primary indicate the circular buffer for replication; if a replica is briefly disconnected, a sufficient backlog can avoid full resync. Ensure `master_last_io_seconds_ago` on replicas is low (meaning they're actively receiving data). A large lag can indicate the replica can't keep up or network problems.

- **Evictions and Keyspace:** In `INFO Stats`, watch `evicted_keys` and `expired_keys`. Evicted keys count means your maxmemory policy is evicting data – if this is not expected (e.g., Valkey being

used as cache, some evictions are normal; but if it's a primary data store, evictions could be data loss). `expired_keys` shows how many keys reached TTL expiry. Also, check `keyspace` section at the end of INFO (e.g., `db0: keys=12345, expires=50, avg_ttl=...`). This tells the number of keys per DB and how many have expiry. A sudden drop in keys might mean a flush or big expiration event occurred. Ensuring the key count is within expected range is a good health indicator (for instance, if it unexpectedly goes to 0, that's a serious issue unless a flush was intended).

- **Errors:** Monitor the Valkey log file (usually `/var/log/valkey.log` or stdout). By default, Valkey logs warnings and higher. You'd want to catch things like `OOM command not allowed` errors (if a write was denied due to out-of-memory and no eviction), or replication errors. With Valkey 8.1's new log format options, you can parse logs more easily if structured logging is enabled [25]. Also, consider enabling the `notify-keyspace-events` if you need to monitor key events via Pub/Sub (this allows an external monitor to subscribe to events like expired keys, evictions, etc., which can be useful for auditing).

Ensuring Healthy Operation

To check that Valkey is running normally, you can perform a few basic health checks: - **PING/PONG:** The simplest check is `PING` command as shown earlier. A healthy Valkey should respond with `PONG` immediately [37]. Many load balancers or health-check scripts use this. - **INFO and Stats Check:** A script or monitoring system can periodically retrieve INFO and check key fields. For example, ensure `uptime_in_seconds` is increasing (resetting would indicate a crash and restart). Ensure `mem_fragmentation_ratio` isn't extremely high (above, say, 2 or 3 could indicate fragmentation issues). Ensure replication link is up if applicable, etc. - **Latency Check:** Use the built-in `LATENCY DOCTOR` command which analyzes latency samples and reports if there are issues (like if the latency monitor has detected disk I/O spikes, it will point out "Disk I/O overloaded" or similar). This is a quick way to get Valkey's own assessment of any latency problems [65]. - **Operational Tests:** In a non-production environment or during maintenance, you might do a quick set/get test of critical data. For instance, set a key and retrieve it to ensure read/write path works. If using persistence, you could test a `BGSAVE` and see if it completes successfully (the command replies "OK" and later INFO shows last save time updated). - **Monitoring Tools:** Integrate Valkey with monitoring solutions. Many existing Redis monitoring plugins work out of the box with Valkey (since INFO format is same). Grafana dashboards for Redis can be repurposed for Valkey by just changing the data source. Keeping graphs of memory, ops/sec, connections, etc., over time helps spot anomalies.

If Valkey is not "running normally", symptoms might include: high latency responses, timeouts, memory allocation errors (check log for "Cannot allocate more memory" if overcommit is an issue), failing to persist to disk, or crashing. If a Valkey process crashes, it usually writes a crash log with stack trace – investigate that and upgrade if it's a known bug (Valkey's rapid development means many issues get fixed in newer releases – check the release notes). Running the latest stable 8.x version is recommended for production.

Backup and Recovery

Having backups and a strategy for recovery is crucial if you use Valkey as a data store (beyond a pure cache). Valkey offers two persistence mechanisms: RDB snapshots and AOF logs (or a combination of both), similar to Redis. Choosing one or both depends on your durability needs.

- **RDB Snapshots (Point-in-Time Backups):** RDB (Redis Database) snapshots are binary dumps of the database at a moment in time [68]. By default, Valkey's configuration might save snapshots

at intervals (e.g., the default `save 900 1` means every 900 seconds if at least 1 key changed, etc.). You can customize the `save` schedule in `valkey.conf`. You can also trigger snapshots manually via the `SAVE` (synchronous, blocks server) or `BGSAVE` (background save) commands. **Backup with RDB:** To take a backup, you can run `BGSAVE` and wait for `rdb_bgsave_in_progress:0` in INFO or a log message indicating completion. This produces a `dump.rdb` file (or whatever `dbfilename` is set to) in the configured directory [69]. You should copy this file to safe storage (e.g., off-server or to cloud storage) – RDB files are compact and ideal for archival backup [70]. For example, some setups do hourly RDB snapshots and ship them to S3, keeping a history of 24 hours, etc. The RDB approach is very efficient for disaster recovery since it's a single file you can move around [71]. However, **data loss** can be up to the snapshot interval – e.g., if you snapshot every 5 minutes, a crash could lose up to 5 minutes of latest data [72].

- **AOF (Append Only File) Logging:** AOF logs every write command the server receives to a file, so you can replay it to reconstruct the dataset [68]. AOF can be configured to fsync data to disk every write, every second, or never (let OS flush). The default (every second) strikes a balance – you'd lose at most 1 second of writes on a crash [73]. AOF tends to be larger than RDB and a bit slower on writes (due to logging), but provides better durability (no large gaps in data). **Backup with AOF:** If using AOF, you should still periodically copy the AOF file somewhere safe. Also monitor AOF size; Valkey can auto-rewrite (compact) the AOF in background when it grows too large [74]. In a recovery scenario, the AOF will be replayed automatically on server start to rebuild state. Ensure the AOF file is not corrupted – use `valkey-check-aof` tool if in doubt (it can fix a truncated AOF) [75]. One advantage of AOF: you can open it (it's plain text commands) to extract data or even undo recent mistakes (e.g., if someone ran a FLUSHALL, you could theoretically edit that out of the AOF if you catch it before rewrite) [76].

- **Mixing RDB and AOF:** Valkey allows enabling both RDB and AOF for maximum safety [77]. In this mode, on restart Valkey will prefer to load AOF (for most up-to-date state) but having an RDB gives a fall-back backup. Many production deployments use AOF (appendfsync every sec) plus RDB snapshots every now and then. That way, you have the AOF for point-of-failure recovery (minimal data loss), and RDB for easier offsite backups and faster restart (RDB loads faster than AOF replay) [78] [79]. If using both, be mindful of performance (there's overhead in maintaining two persistence files).

Recovery: To restore from a backup RDB, simply stop Valkey, replace the `dump.rdb` file in its working directory with your backup copy, and start Valkey. It will detect the RDB and automatically load it into memory [80] [81]. If using Docker, you'd copy the RDB into the container's data mount before starting [82] [81]. To restore from AOF, similar – place the AOF file and ensure `appendonly yes` is in config, then start Valkey and it will replay the AOF. Always keep backup copies of your RDB/AOF files **separate** from the live instance, to avoid accidental override. A best practice is to snapshot a backup, transfer it off the server, then verify the backup by loading it into a staging Valkey instance (to ensure it's not corrupt and contains expected data).

Replication as live backup: Another approach to consider is running a replica solely for backup purposes. Because Valkey replication can be used across data centers, you could have a replica in a remote location. That replica could be configured with `replica-read-only yes` (default) and you never promote it unless needed. It will continuously receive updates. You can even periodically `BGSAVE` on that replica to produce RDB backups without loading the primary. In case the primary data center is lost, you have a replica already up-to-date which you can promote (and since it's asynchronous replication, some last transactions might be lost, but similar to AOF 1-second window if network goes down).

Finally, **test your backups**! Nothing is worse than assuming a backup works and finding out it's corrupted or incompatible. Do test restores. Note that Valkey 7.x RDB files are compatible with Valkey 8.x (so upgrading doesn't break persistence), but they are **not** compatible with Redis "Community 7.4+" (the closed-source ones) [83] [84]. As long as you stay within Valkey or Redis OSS, you can load RDB files across versions (forward compatibility is generally maintained; e.g., you can load a Redis 6 RDB into Valkey 8). For AOF, the format is simply commands – also generally compatible.

Performance Tuning

Valkey is fast out-of-the-box, but to get the best performance and avoid common bottlenecks, consider the following tuning tips:

System and OS Tuning: - Memory Overcommit: On Linux, enable memory overcommit (so the kernel doesn't erroneously deny large allocations). Set `vm.overcommit_memory = 1` in `/etc/sysctl.conf` and apply it [85] [86]. This avoids issues when Valkey forks the process for RDB saves – forking can fail if the kernel is strict about overcommit and thinks there isn't enough available memory. - Transparent Huge Pages (THP): Disable THP, as it can cause latency spikes and memory usage bloat for in-memory databases. You can disable it by `echo never > /sys/kernel/mm/transparent_hugepage/enabled` on startup [87]. THP can make memory allocation slower and unpredictable, so it's recommended off for Valkey/Redis. - Swap: Ideally, avoid swapping as it will hurt performance badly. However, it is suggested to have some swap enabled so the kernel doesn't OOM-kill Valkey if memory is exhausted [88]. A small swap can provide a safety net (Valkey will try not to use it much if `maxmemory` is set properly). Monitor for any swapping – if you see it, you need to either lower Valkey memory use or increase RAM. - Networking: For best results, run Valkey on a reliable low-latency network. If using clusters across multiple racks or AZs, network delays can affect throughput. If using high connection counts, consider tuning Linux TCP stack (e.g., ephemeral port range if many short connections, or increase backlog queue via `tcp_max_syn_backlog`). In cloud environments, make sure to use enhanced networking drivers if available.

Valkey Configuration Tuning: - Maxmemory & Eviction: If you use Valkey as a cache, set `maxmemory` to a value slightly below total available RAM (to leave room for overhead and fragmentation) [89]. E.g., on a 10 GB instance, maybe set maxmemory to 8 or 9 GB [90]. Choose an eviction policy (`allkeys-lru` is common for cache, or `noeviction` if you prefer failures over evicting). If Valkey is a primary store, you might run without maxmemory to avoid evictions, and rely on monitoring to not exceed capacity. - Persistence and Disk: If you have heavy write load and persistence enabled, tune fsync appropriately. AOF with `everysec` is usually fine. If latency is critical, use a fast SSD for persistence. For RDB, consider using **diskless replication** (so that when a new replica connects, the primary sends RDB over socket rather than saving to disk first) to reduce disk I/O – enable with `repl-diskless-sync yes`. Also, if using AOF, adjust `auto-aof-rewrite-percentage` and `auto-aof-rewrite-min-size` so that the AOF rewrite (BGREWRITEAOF) happens at a good time (e.g., when AOF has grown 100%+ of original size). Large fork operations can momentarily stop the world, so plan snapshotting during low traffic if possible. - Threads: Valkey 8+ can utilize I/O threads. By default, `io-threads` is 1 (disabled). If you have spare CPU cores and see CPU saturation on a single core (with networking or TLS overhead), you can try enabling, say, 4 I/O threads (`io-threads 4`) and `io-threads-do-reads yes` (to use threads for reads). This can increase throughput on multi-core systems for network-heavy scenarios [11]. But test this under your workload – some workloads might not benefit if they are single-key small operations (as Redis/Valkey are very efficient single-threaded for those). - Client Output Buffer: If you use Pub/Sub or have clients that might slow-consume data, watch the client output buffer limits (configurable per client type). This prevents one slow subscriber from consuming too much memory and potentially crashing the server. You can adjust `client-output-buffer-limit` settings if needed (for example, increasing the limit for pubsub if large

bursts are expected, or lowering it to be safe). - Other Configs: Ensure `tcp-backlog` is set high (the default is 511, you can increase if expecting large connection bursts). If using cluster, ensure `cluster-node-timeout` is tuned – default 5 seconds might be fine, but in highly latent networks you might increase to avoid false failovers. For replication, if the primary is high volume, increase `replication-backlog-size` to a few MBs to handle replicas briefly falling behind.

Application-Level Optimization: - Pipelining: Just as with Redis, using pipelining can greatly enhance throughput by reducing round-trip latency cost [91]. Valkey supports pipelining natively – clients can send multiple commands without waiting for replies one by one, and then read all replies. If your use case allows, batch operations into pipelines (especially in high-latency network situations). - Data Structures: Choose appropriate data types to avoid inefficiency. For example, instead of having a million keys each storing a small counter, consider using a Hash that holds those counters as fields (if it makes sense for your access pattern). This can be more memory-efficient (one key vs many) and certain operations like retrieval of many fields are faster than many GETs. But note that extremely large Hash or Sorted Set objects can also become a bottleneck for operations that traverse them. There is a balance – Valkey's new hashtable design reduces per-key overhead, but having fewer keys might still help reduce overhead operations. - Avoid Hot Keys: If one key is extremely heavily accessed/modified, it could become a throughput bottleneck (since all operations on a given key are processed serially by one thread). Try to distribute load. For example, using sharding (in cluster mode) or logically partitioning data (like user:1001:... keys spread by user ID). If using cluster, the hashing is automatic but ensure your keys have enough diversity in names. - Lua Scripting vs Multi-roundtrips: If you have a sequence of operations that must be atomic, you can either use a MULTI/EXEC transaction or a Lua script (EVAL). Lua scripts run synchronously in Valkey, which can block other operations, so keep them small and efficient. With Valkey's future support for other scripting engines (WASM, etc.), you may have more choices [92], but general advice is the same as with Redis – don't run long loops in Lua that lock the server.

- Active Defragmentation: If you have a long-running Valkey instance where memory allocations/deallocations might lead to fragmentation (e.g., lots of deleting and adding keys), consider enabling `activedefrag yes` in config. The active defrag process runs in the background to compact memory. Valkey 8.1 improved active defrag to cap pause times under 1 millisecond and be smarter about fragmentation checking [64]. This can help avoid wasted memory and performance issues due to fragmentation. Monitor `mem_fragmentation_ratio` to decide if defrag is needed (a ratio >> 1.0 indicates fragmentation).

Benchmarking and Testing: Before going to production, it's wise to benchmark Valkey under expected workload. Use the included `valkey-benchmark` tool to generate test load [93]. It can simulate GET/SET or other command mixes at a given throughput. Valkey-benchmark is the same as redis-benchmark but updated for Valkey. This gives you an idea of achievable throughput and latency on your hardware. Also test failover scenarios: if using replication, intentionally fail the master and see how the replica (or Sentinel/cluster) handles it, measure downtime. If using cluster, kill a node and see if the application experiences errors or recovers properly (make sure your retry logic in the app is in place).

In summary, Valkey can be tuned at many levels – the defaults are reasonable for general use, but for a high-throughput or large deployment, adjusting memory, threading, kernel settings, etc., will ensure smooth operation. The **Performance Optimization Methodology** post on Valkey's blog (May 27, 2025) goes in-depth on systematically identifying bottlenecks [94], which can be a great resource as you fine-tune your Valkey setup.

Migration and Upgrade

This section covers two scenarios: migrating from Redis to Valkey, and upgrading Valkey itself (e.g., from 7.2 to 8.x, or 8.x to newer versions).

Migrating from Redis to Valkey

One of Valkey's goals is to be a seamless replacement for Redis OSS. If you have an existing Redis deployment, you can migrate to Valkey with minimal downtime. The compatibility is very high: Valkey 7.2.x is basically Redis 7.2 under the hood, and Valkey 8.x extends that. According to the official guide, Valkey is **compatible with all Redis open-source versions 2.x through 7.2** – migrating is essentially like upgrading Redis to a new version [83] [95]. (Redis "Community Edition" 7.4 and later are not open source and use a different persistence format not compatible with Valkey, so those require special migration steps not covered here [96].)

Common migration approaches: - **In-Place Physical Migration (Copy RDB):** The simplest method is to take a snapshot of your Redis data and start Valkey with it. For example, if you run `SAVE` on Redis to produce a dump.rdb [97] [98], you can then stop Redis, copy the `dump.rdb` file to Valkey's data directory, and start the Valkey server. Valkey will load the RDB and your data will be available [80] [81]. This method is fast and straightforward – essentially downtime is just the restart time. However, ensure no writes occur during the cut-over, otherwise those will be lost (to avoid that, you'd disable writes to Redis, do a SAVE, then switch). The migration guide notes you should **disconnect all clients** from Redis before creating the RDB snapshot to prevent changes during backup [99]. You can verify the key count before and after to confirm all data migrated (use `INFO KEYSPACE`) [100] [101].

- **Minimal-Downtime Migration via Replication:** This approach treats Valkey as a replica of your Redis instance, then fails over. Valkey's replication protocol is compatible with Redis 7.2 and earlier, so you can connect a Valkey server as a replica to an existing Redis master [102] [103]. The steps would be: start a Valkey server (with `replicaof <redis-ip> <redis-port>`). It will connect to Redis, do a full sync (download RDB from Redis) and then start receiving live replication stream [104]. Let it catch up until it's consistently replicating (check `master_link_status:up` on Valkey INFO). Then, to cut over with minimal downtime, you would pause writes on Redis (if possible, or use a maintenance window with no writes), then promote the Valkey replica to primary (`REPLICAOF NO ONE` on Valkey) and direct your application to Valkey. You could also use a tool like Sentinel to handle this promotion, but since it's a one-time migration, manual is fine. The downtime in this case can be just a few seconds (the time to switch client connections). This approach is more complex but avoids extended downtime and ensures near-zero data loss (since Valkey was live replicating, it has virtually all writes). Note: once Valkey is primary, the Redis master should no longer be written. You can either decommission it or make it a replica of Valkey if you want a rollback strategy. The migration guide provides detailed steps for setting up such replication and then switching over [102] [105].

- **Key-by-Key Migration:** If for some reason binary compatibility is an issue or you only want to migrate a portion of data, you can also script a migration at the application level. For example, using `SCAN` on Redis and piping data into Valkey via `valkey-cli` or a custom script. This is slower and usually unnecessary, but it's an option for complex scenarios (the Valkey docs mention "Migrating specific keys" [102] [104]). There are also tools that can live-copy between Redis instances (e.g., redis-shake, etc.) which might support Redis->Valkey since the protocols are the same.

After migrating, **validate** everything on Valkey: key counts, data correctness (sample some values), performance, etc. If using modules (e.g., you had Redis with JSON module, and now Valkey with Valkey-JSON), ensure the data moved properly – RDB files generally include module data that Valkey can load if the same module (or compatible version) is present.

Upgrading Valkey Versions

Valkey follows semantic versioning (major.minor.patch). Major releases (8.0, 9.0) may have breaking changes or big features, but so far the upgrade from 7.2 to 8.0 was smooth in terms of compatibility (the biggest difference was introducing new optional features). Here's how to approach upgrades:

- **In-Place vs Rolling Upgrade:** If you can afford a small downtime, the simplest is stop the Valkey server and start the new version binary pointing at the same data file. Valkey 8 could directly load a RDB created by Valkey 7.2 (since it was a fork of Redis 7.2, RDB version 9 or 10), so a straight restart with the new binary works [106]. For minimal downtime in a primary-replica setup or cluster, do a rolling upgrade. In replication mode, you can upgrade one node at a time: e.g., bring up a new Valkey version as a replica of the old primary, let it sync, then promote it and switch over, similar to the migration steps. The administration guide explicitly suggests this: **run a new Valkey instance as a replica of the old, then fail it over** as a way to avoid downtime on upgrade [107] [108]. This also serves as a test that the new version can sync from the old one properly.

- **Sentinel or Cluster Rolling Upgrade:** If you use Sentinel-managed HA, you can upgrade one replica at a time: take a replica down, upgrade it, bring it back. Once all replicas are on new version, do a manual failover (so one of the new-version replicas becomes primary), then upgrade the remaining old primary (now a replica) [109]. This ensures you always have the cluster operational. In a Valkey Cluster, you similarly upgrade node by node. Valkey cluster is designed to allow mixed versions temporarily (for example, you can have some nodes on 8.0 and some on 8.1 during an upgrade, but it's best to keep that window short). Always check the release notes for any specific instructions – e.g., if upgrading to Valkey 9.0, there might be special considerations if you used certain features (like new module data types).

- **Backup before upgrade:** Needless to say, backup your data (RDB/AOF) before upgrading. While downgrading is generally possible (Valkey 8 can dump RDB that Valkey 7 could read, etc.), it's safer to have the snapshot. The release notes often indicate "upgrade urgency" and whether you can directly downgrade if needed [110]. For example, Valkey 8 had no changes to RDB format from 7.2, so one could technically switch back if needed. But if you use new features (like when 9.0 adds hash field TTLs), those data structures might not be understood by older versions.

- **Testing new version:** It's wise to test the new version in staging with your application traffic. Ensure that performance characteristics haven't changed unexpectedly or that a new config default won't surprise you. For instance, if 9.0 enables some new feature by default, make sure it doesn't affect your workload. Read the **Release Notes** (Valkey provides them on GitHub for each release [11] [111]).

Valkey being a relatively new fork means the pace of improvements is high, so upgrading frequently can give you big benefits (e.g., memory savings in 8.1, new functionality in 9.0). The community is actively fixing bugs too, so staying on the latest patch release of your major version is recommended. Always monitor after an upgrade for any issues (like memory usage patterns or logs). So far, community reports of upgrades (e.g., some blog posts on "Upgrade Stories from the Community" [112]) indicate the process is quite painless, especially compared to the complexity of upgrading some other databases.

Reference and Utility Commands; Client Tools

This section provides a quick reference to common administration commands and the client libraries available for Valkey in different programming languages.

Common Administration Commands (valkey-cli)

The `valkey-cli` tool is your go-to for managing and troubleshooting Valkey from the shell. Here are some essential commands (many we've mentioned above):

- **PING:** Basic reachability test. `valkey-cli ping` should return `PONG` [37].
- **INFO:** Retrieves server info. You can do `valkey-cli info` or specify a section: `valkey-cli info memory` etc. This prints various metrics and settings.
- **CONFIG GET / SET:** To view or change configuration at runtime. For example, `valkey-cli config get maxmemory` shows the current value, and `valkey-cli config set maxmemory 10gb` would change it (if allowed). Not all configs are mutable at runtime, but many are.
- **CLIENT LIST / CLIENT KILL:** To inspect connected clients or kill a hung client connection. Useful if you suspect a client is stuck or causing issues.
- **SLOWLOG:** `slowlog get 10` shows last 10 slow commands logged.
- **LATENCY:** `latency latest` shows recent latency events; `latency doctor` provides an analysis with tips.
- **BGSAVE / BGREWRITEAOF:** Triggers persistence events manually. Use `BGSAVE` to force an RDB snapshot now. Use `BGREWRITEAOF` to trigger AOF compaction.
- **FLUSHALL / FLUSHDB:** Wipes database (use with caution!). Sometimes used in development or to clear cache entirely.
- **DEBUG:** (Use carefully, mostly in non-production) e.g., `DEBUG SEGFAULT` will crash the server to test persistence (should never run in prod).
- **MODULE LOAD / UNLOAD:** Dynamically load a module (.so file) into a running Valkey. For instance, `MODULE LOAD /path/to/valkeyjson.so`. Note that modules must match the Valkey version's API. Official modules are usually loaded via config at startup rather than CLI in production.

In general, any Redis command you know is available in Valkey (unless it was an enterprise-only command in Redis, which few are). The Valkey Command Reference is comprehensive – it lists all commands alphabetically and by group, including those added by modules [113]. For example, after loading Valkey JSON module, commands like `JSON.SET`, `JSON.GET` become available.

One nice thing: Valkey's documentation is available as man pages if you install `valkey-doc`. For instance, you can do `man HSET` to read about the HSET command usage [32], or `man valkey.conf` for config file syntax.

Client Libraries and Integration

Because Valkey is protocol-compatible with Redis, **existing Redis clients can be used** for Valkey (especially those supporting Redis 7.x features) [9]. However, the Valkey project also provides and recommends certain client libraries that are tested with Valkey and may offer extra features:

- **Python:** You can use `redis-py` (the standard Redis client in Python) with Valkey. In fact, `pip install valkey` provides **valkey-py**, which is essentially the Valkey-specific fork of redis-py [114]. There is also **Valkey GLIDE** for Python (multi-language client; more on GLIDE below) [115]. For most purposes, you can continue using `redis.StrictRedis` or `Redis` client in Python – just point it to

your Valkey endpoint. If you want to use new Valkey-specific commands (like `SET IFEQ` or new modules), make sure your client version is updated if command constants are needed (or use the raw `execute_command` method to send any command string).

- **Java:** Popular Redis clients like Jedis or Lettuce work with Valkey. Additionally, the Valkey project offers **valkey-java** (an official Java client similar to Jedis) [116] [117] and it endorses **Redisson** which has added Valkey compatibility [118] [119]. Redisson is a high-level client that offers distributed objects, locks, etc., and as of v3.48.0 it supports Valkey as well [120]. To use Valkey in Spring or other frameworks, you can typically just change the host/port and it should work, since from the application's perspective it's still Redis protocol.

- **Node.js:** The Node Redis client (`redis` npm package) works with Valkey. There is also an official multi-language client **Valkey GLIDE**, which in Node is available as `@valkey/valkey-glide` [121]. Additionally, a client named **iovalkey** is a Node client optimized for Valkey/Redis (v0.3.1 as of 2025) [122]. It's MIT-licensed and focused on performance – likely analogous to ioredis but for Valkey.

- **Go:** The Go ecosystem has many Redis clients (redigo, go-redis, etc.). Valkey provides **valkey-go** (a Go client that supports auto-pipelining and client-side caching) [123] [124]. You can `go get github.com/valkey-io/valkey-go` to use it. Of course, go-redis (redis/go-redis) should also work as it supports Redis 7.2 features.

- **Other Languages:** PHP has phpredis extension which will work; Valkey also listed a PHP client in their docs. C/C++ can use hiredis (works fine with Valkey). For **C#/.NET**, StackExchange.Redis (the de-facto Redis client) is compatible with Valkey out of the box. In fact, cloud providers like AWS and Azure have started supporting Valkey endpoints and the same clients are used. There's also a Swift client introduced by Valkey ("valkey-swift") for iOS/macOS developers [125], showing the breadth of client support.

The **Valkey GLIDE** deserves a mention: it's a Rust-based client core that provides bindings in multiple languages (Python, Java, Node, Go) [126] [127]. GLIDE is designed for high performance and advanced features like server-assisted client-side caching. For example, GLIDE automatically pipelines commands to reduce latency, and it supports tracking keys to get invalidation messages (the "client side caching" feature using Redis/Valkey's tracking). If you need bleeding-edge performance from your client and are okay with using a newer library, GLIDE is a great option (it's maintained by the Valkey team and Apache-2.0 licensed). But mainstream clients work perfectly well too – so you don't have to change all your app code to adopt Valkey.

Utilities: Apart from client libraries, remember tools like **valkey-benchmark** (to load test) [93], **valkey-check-rdb** and **valkey-check-aof** (to verify and repair persistence files). If you have a corrupted AOF, you can run `valkey-check-aof --fix` on it [128]. For RDB, corruption is rare, but `valkey-check-rdb` can analyze an RDB dump for errors.

Also, **Valkey Sentinel** is effectively a utility when run (`valkey-server --sentinel <conf>` starts it). There aren't separate binaries named differently, except the symlink `valkey-sentinel` which is just the server in sentinel mode [129]. Manage sentinel via its config and the SENTINEL commands (like `SENTINEL FAILOVER <master-name>` to force a failover, etc.).

Finally, Valkey has a strong community – if you run into issues or need help, you can find the team on their Slack/Discord (community links on valkey.io) or ask on Stack Overflow with the tag **[valkey]** [130].

The project being open source means you can also inspect the source code or even contribute. Tools like `valgrind` or `perf` can be used if you're debugging low-level performance, and Valkey's source comes with a test suite (`make test`, etc.) which is useful if you suspect a bug and want to reproduce it.

With these references and tools at your disposal, you should be well-equipped to install, manage, and develop with Valkey effectively. It is a powerful evolution of Redis with the freedom of open source – combining familiarity with new improvements. Enjoy exploring Valkey's features and building upon this high-performance key-value store!

Sources:

1. AWS ElastiCache – What is Valkey? (History and features of Valkey) 2 11
2. Valkey Documentation – Valkey Introduction and Features (Open source status, supported data types) 1 3
3. Valkey Blog – Valkey 8.1 Performance Improvements (New hashtable, threading, optimizations) 14 12
4. Valkey Blog – Introducing Hash Field Expirations (Valkey 9.0 feature for per-field TTL) 18
5. Valkey Blog – Valkey 8.1 Release Highlights (Conditional SET IFEQ, observability improvements) 66 19
6. Valkey Blog – Numbered Databases in Valkey 9.0 (Cluster mode multi-DB support) 7 21
7. Valkey Documentation – Installation (CentOS & Docker) 131 39
8. Valkey Documentation – Replication and Sentinel (Sentinel capabilities and usage) 4 50
9. Community Guide – Keepalived VRRP for Valkey HA (Keepalived failover concept and caveats) 46 44
10. Valkey Documentation – Cluster Tutorial (Cluster ports, hash slots, creation with CLI) 5 57
11. Valkey Documentation – Administration and Upgrade (Linux tuning, no-downtime upgrade via replica) 86 107
12. Valkey Documentation – Persistence (RDB vs AOF, backups) 71 73
13. Valkey Documentation – Migration from Redis to Valkey (Compatibility and migration steps) 83 80
14. Valkey Documentation – Clients (Supported client libraries in Python, Java, Node, Go, etc.) 9 114

---

1 2 3 10 11 What is Valkey? - Valkey Explained - AWS
https://aws.amazon.com/elasticache/what-is-valkey/

4 48 49 50 51 52 53 129 Valkey Documentation • High availability with Valkey Sentinel
https://valkey.io/topics/sentinel/

5 6 42 43 54 55 56 57 58 59 60 61 62 63 Valkey Documentation • Cluster tutorial
https://valkey.io/topics/cluster-tutorial/

7 20 21 22 Valkey • Numbered Databases in Valkey 9.0
https://valkey.io/blog/numbered-databases/

8 91 93 113 Valkey • Topics
https://valkey.io/topics/

9 114 115 116 117 118 119 120 121 122 123 124 126 127 Valkey • Client Libraries
https://valkey.io/clients/

12 13 14 15 16 17 19 23 24 25 26 64 66 67 92 111 Valkey • Valkey 8.1: Continuing to Deliver Enhanced Performance and Reliability
https://valkey.io/blog/valkey-8-1-0-ga/

18 27 28 29 94 112 125 Valkey • Blog
https://valkey.io/blog/

30 31 32 33 34 35 37 38 131 Valkey Documentation • Installation
https://valkey.io/topics/installation/

36 GitHub - valkey-io/valkey: A flexible distributed key-value database that is optimized for caching and other realtime workloads.
https://github.com/valkey-io/valkey

39 40 41 Valkey
https://valkey.io/

44 45 46 47 Simple Keepalived notify script for Valkey/Redis master-replica failover • GitHub
https://gist.github.com/jirutka/20fbe0531099b09c0627bb52f2959aa5

65 85 86 87 88 89 90 107 108 109 Valkey Documentation • Administration
https://valkey.io/topics/admin/

68 69 70 71 72 73 74 75 76 77 78 79 128 Valkey Documentation • Persistence
https://valkey.io/topics/persistence/

80 81 82 83 84 95 96 97 98 99 100 101 102 103 104 105 Valkey Documentation • Migration from Redis to Valkey
https://valkey.io/topics/migration/

106 Supported versions | Memorystore for Valkey - Google Cloud
https://cloud.google.com/memorystore/docs/valkey/supported-versions

110 Releases • valkey-io/valkey - GitHub
https://github.com/valkey-io/valkey/releases

130 Valkey • GitHub
https://github.com/valkey-io