**Cluster Architecture**

**A screenshot of a computer screen

Description automatically generated**

**A diagram of a database

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer screen

Description automatically generated**

* **Each database can be accessed in multiple ways:**
* **Database endpoint:** Simply connect your application to your database endpoint (a unique URL and port on the fully qualified domain name), and Redis Enterprise will transparently handle all the scaling and failover operations.
* **Sentinel API:** Use sentinel protocol to connect to the correct node in the cluster to access your database.
* **OSS Cluster API:** Use the cluster API to directly connect to each shard of your cluster without any additional hops.
* **High availability:**

Redis Enterprise **ensures zero downtime** by using replica sets across one or multiple regions.

* If a server fails, another takes over instantly.
* Replica shards are always placed on different nodes for failover.
* The system automatically switches to a backup to keep services running.
* **Redis Enterprise Cluster: Separation of Data Path and Control Path**

**What Are the Two Paths?**

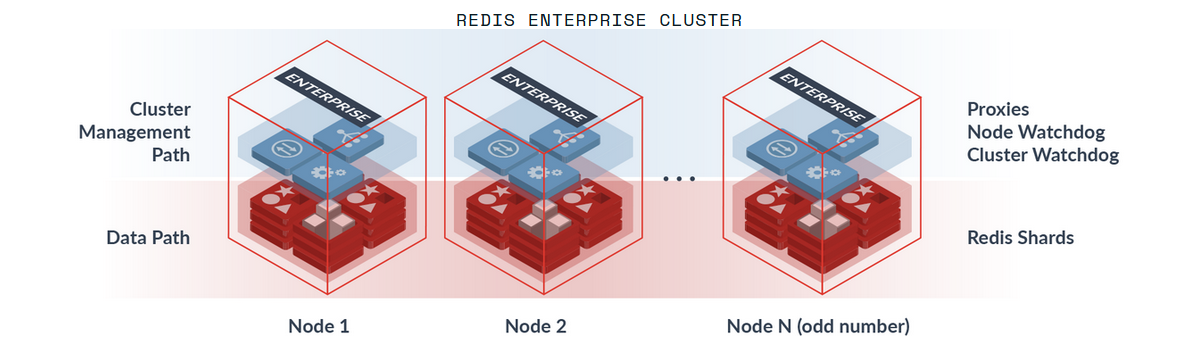
**1. Data Path (Proxies & Shards):** Handles actual read/write requests from clients.

**2. Control Path (Cluster Management):** Manages cluster coordination, monitoring, and failover.

**A screenshot of a computer

Description automatically generated**

**Cluster Component consist of Nodes:**

****

**Benefits of This Separation:**

**1. High Availability & Fault Isolation**

* Even if a management node goes down, Redis continues processing client requests.
* Ensures zero downtime in case of node failures.

**2. Performance Optimization**

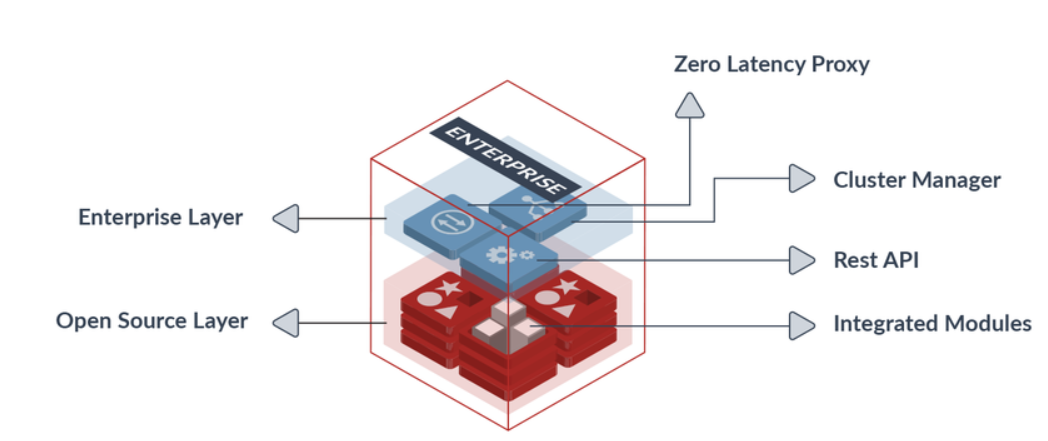
* Control operations (like cluster balancing) don’t slow down normal Redis queries.
* Load is distributed efficiently across shards, avoiding bottlenecks.

**3. Horizontal Scalability**

* Easier scaling because new nodes can be added dynamically.
* Cluster management handles rebalancing and resharding without downtime.

**4. Intelligent Traffic Routing**

* Proxies route client requests to the right shard, preventing direct client-shard communication.
* Reduces client-side complexity (no need for cluster-aware clients).
* **Node Architecture:**

****

* **Redis shard:** A Redis instance with either a primary or replica role that is part of the database.
* **Zero-latency proxy:**The proxy runs on each node of the cluster. It is designed for **fast request processing** and **efficient connection handling** without adding noticeable delay (**zero-latency**). The proxy handles the following primary functionalities:

**1. Hides Cluster Complexity**

* Applications interact with **a single database endpoint** rather than dealing with multiple Redis shards.

**2.** **Maintains the Database Endpoint**

* The proxy ensures that applications can always connect to **the same endpoint**, even if shards are rebalanced or fail over to a new node.

**3. Requests Forwarding**

* Routes **client request** the appropriate **shard or node** efficiently.

**4. Manages Data Encryption through SSL**

* Provides **secure communication** between clients and Redis by **encrypting data** in transit using **SSL/TLS**.

**5. Redis Acceleration via Pipelining & Connection Management**

**Pipelining:-** Bundles multiple commands together, reducing **round-trip time (RTT)** between client and server.

A screenshot of a computer

Description automatically generated

**Connection Management:-** Redis Enterprise uses **persistent connections** instead of opening and closing them frequently. **Connection pooling** allows multiple clients to share a limited number of connections. The **zero-latency proxy** helps distribute and optimize client requests.

* **Cluster Manager:** It’s a critical control component that is independent from the data path components. It’s key functionalities:

1. **Database Provisioning & De-Provisioning**
   * Creates and deletes databases dynamically based on demand.
2. **Automatic Scaling**
   * Detects increasing workloads and adds more resources automatically which prevents performance degradation during traffic spikes.
3. **Automatic Re-Sharding**
   * Redistributes data across shards when a shard gets overloaded.
4. **Automatic Re-Balancing**
   * Prevents certain nodes from being over-utilized while others remain idle.
5. **Resource Management & Health Monitoring**
   * Continuously monitors the cluster's health (CPU, memory, network).
6. **Node Watchdog**
   * **Monitors all processes on a Redis node.**
   * Detects shard failures and triggers automatic recovery.
7. **Cluster Watchdog**
   * **Monitors the entire Redis cluster** to ensure all nodes are functioning.
   * Detects node failures and initiates failover to maintain availability.

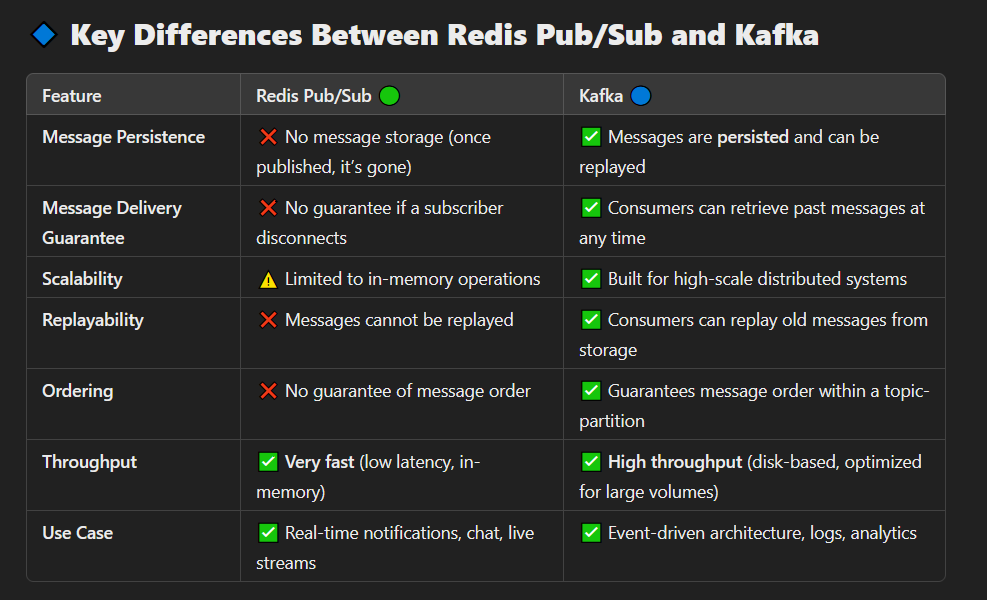
* **Secure REST API:**All management and control operations on Redis Enterprise are performed through a dedicated and secure API that is resistant to attacks and provides better control of cluster admin operations.

**Reasons to use a cache memory:**

1. Data stored and retrieved much faster than when in DB
2. Cache memory is highly volatile thus if the server crashes all data will be gone that’s why we need a DB as well (but now Redis evolved and can be used as the primary DB)

**What is new?**

* 1. Data persistence and replication to make sure that the data is durable and available
  2. JSON support and search which made it much easier to store complex data
  3. Reddis OM – object mapping library

****

**Caching strategies**

**A black and white text

Description automatically generated**

**Redis Pagination**

**A screenshot of a black and white screen

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A black background with white text

Description automatically generated**

**A screenshot of a computer error

Description automatically generated**

**A black background with white text

Description automatically generated**

**Frequently Used Commands**

**Data type:**

* Data is store in Key-value pair eg. Name 🡪 ‘Aya’
* Integers are stored in string format
* Has different data types as Strings, lists, sets, hashes, etc.

**Commands:**

**SET** name ‘mario’ or SET name mario

**SET** name2 Chun li 🡪 error, it must be SET name2 ‘Chun li’

**GET** name

**DEL** name2

**MSET** name2 Yoshi color green rating 10 -> Set multiple key-value pairs

name -> Yoshi

color -> green

rating -> 10

**MGET** name name2 rating -> Get multiple key-value pairs

**GETRANGE** name 0 3 -> get from first char to 4th (mari)

**GETRANGE** name -3 -1 -> get from the last char to the third from last (rio)

**SETRANGE** name 2 abc -> will overwrite the name from 3rd char (mario -> maabc)

**INCR** rating -> will increase the rating by 1 (10 -> 11)

**DECR** rating -> will decrease the rating by 1 (11 -> 10)

**INCRBY** rating 5 -> will increase the rating by 5 (10 -> 15)

**DECRBY** rating 10 -> will decrease the rating by 10 (15 -> 5)

**SET** name mario XX -> will set name with mario if name already exist

**SET** name mario NX -> will set name mario if name does not exist

**SET** name3 Peach GET -> it will set name3 with Peach and get the previous value of name3 if it existed

**Sets:** (Unordered collection of strings, where all values must be **unique**)

**SADD** names mario -> added one element to names set

**SADD** names Yoshi Peach -> added two more elements to names set

**SREM** names Yoshi -> removes element Yoshi from names set

**SUNION** names moreNames -> Retrieves all elements in both names and moreNames sets without altering in it

**SISMEMBER** names link -> sees wither link exists in names set or not, if it exists it returns 1, if it doesn’t it returns 0

**Lists (Linked Lists):**

* Ordered strings based on when added, their values don’t have to be unique.
* Its downside is in searching as it takes time to find an element in the middle as not all elements are stored in the same place but rather each point to the place where the other is stored
* Its upside is that any operation that involves the head or tail is processed fast.

**LPUSH** orders ryu -> push from the lift in the linked list

**RPUSH** orders li-chun -> pushes from the right in the linked list

**LPOP** orders 1 -> pops one from the left side

**RPOP** orders 1 -> pops one from the right side

**LRANGE** orders 0 2 -> get from index 0 to index 2 from the left side

**LRANGE** orders -2 -1 -> get first and second elements in orders from right side

**LINDEX** orders 1 -> get element in index 1

**LPOS** orders ryu -> get the index of element ryu

**Hashes (nested properties, each can have different datatype):**

**HSET** book:1 title “where in life”-> Create book hash if not created, and inserts a field in book, but if it exists it will be overwritten

**HSET** book:2 title “Here we are” author “Patrick” rating 9 -> will create a book hash with three attributes

**HGET** book: 1 title -> get the value of title in the hash

**HGET** book:2 -> gets all attributes in this hash

**HEXISTS** book:1 title -> returns 1 if property exists otherwise returns 0

**HDEL** books:1 author -> delete a field from the hash

**DEL** books:1 -> delete the entire hash

**HVALS** books:2 -> get all different values without the properties