WHITE PAPER

Why use Redis Enterprise as a Database for Your Microservices?

Roshan Kumar, Redis Labs

CONTENTS

Background

Microservices are in the spotlight as infrastructure building blocks because they offer benefits such as the decoupling of services, data store autonomy, miniaturized development and testing set up, as well as other advantages that facilitate faster time-to-market for new applications or updates. The availability of containers and their orchestration tools has also contributed to the increase of microservices adoption. One of the core principles of microservices architecture is the rejection of a monolithic application framework, which in turn favors the sharing of data between services rather than the use of a single large database. Microservices embrace independent, autonomous, specialized service components, each with the freedom to use its own data store. In general, microservices are designed to be agile, lightweight processes.

An e-commerce solution, for example, may employ the following services: application server; content cache; session store; product catalog; search and discovery; order processing; order fulfillment; analytics and many more. Rather than use a large, single database to store all of the operational and transactional data, a modern e-commerce solution may use a microservices architecture similar to the one depicted in Figure 1, in which each service has its own database.

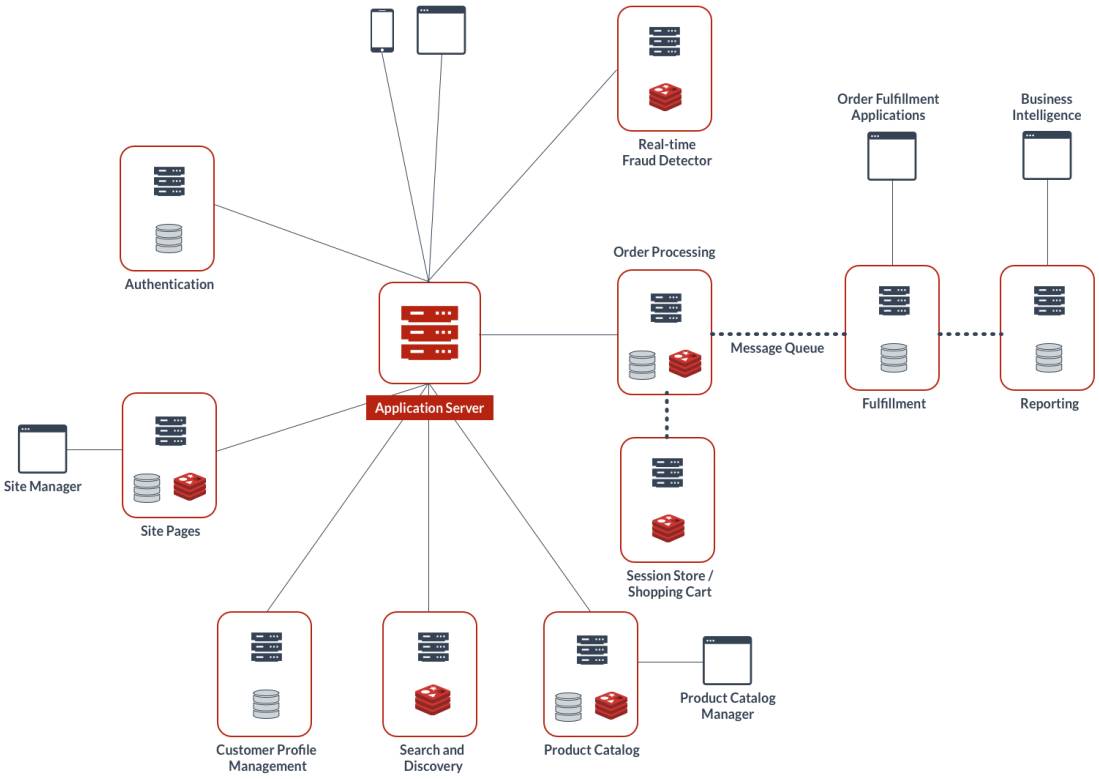


Figure 1. Microservices in a sample e-commerce solution

Understanding the data store requirements for your microservices

One of the most important questions to answer while designing microservices is, "How does one choose the right data store(s) for each microservice?" You should incorporate performance, reliability and data modeling requirements into your selection process.

Performance requirements

In microservices, it's important to design every service to provide the best throughput. If one microservice becomes a bottleneck in the flow of data, then the whole system may collapse.

1. **Read performance:** A commonly used metric for read performance is the number of operations per second. However, in many cases this metric is a combination of how fast you can run queries and how fast you can retrieve results. The speed of retrieving results is dependant upon how well you can organize and index data. A product catalog microservice, for example, may run queries that apply multiple parameters such as product category, price, user rating, etc. The database that you choose for such a microservice must first allow you to organize the data to run your queries faster, and then be able to accommodate the number of operations-per-second requirement as well.
2. **Write performance**: The easy metric here is to determine the number of write operations your microservice performs per second. Disk or network-storage-based databases are inherently slow compared to in-memory databases. Microservices that collect and process transient data need databases that can perform thousands, if not millions of write operations per second.
3. **Latency:** Microservices that deliver instant user experiences require a low-latency database. Deploying a microservice close to its database will minimize the network latency. Database architecture and its underlying storage may add additional latency. Low-latency databases are needed for microservices that perform real-time analytics (such as fraud mitigation).
4. **Resource efficiency:** Microservices are lightweight by design. The database footprint must be minimal while retaining the ability to scale on demand, reflecting the design principles of microservices and their agility to the greatest extent.
5. **Provisioning efficiency:** Microservice components need to be available for rapid development, testing and production. Any database service used will often be required to support the on-demand creation of hundreds of instances per second.

In our example e-commerce solution, we can roughly ascribe the below performance and latency needs to each microservice.

Microservice	Read Performance	Write Performance	Latency
Cache Server	Very high*	High	Low*
Session Store	Very high	Very high	Low
User Activity Tracker	Moderate	Very high	Low
User Comments and Ratings	Moderate	Very high	Low
Product Catalog	Very high	Moderate	Low
Real-time Fraud Detector	Very high	Very high	Low
Reporting	Moderate to very high	Moderate to high	Moderate to low
Order Processing	Moderate to high	Moderate to high	Moderate to low
Order Fulfillment	Moderate to high	Moderate to high	Moderate to low

*For read and write operations, these are the typical numbers for operations per second:

- Very high - Greater than 1 million
- High - Between 500K and 1 million
- Moderate - Between 10K and 500K
- Low - Less than 10K

*For latency, the typical numbers are:

- Low - Less than 1 millisecond
- Moderate - 1 to 10 milliseconds
- High - Greater than 10 milliseconds

Data modelling requirements

The advantage of microservices over monolithic architectures is that each service can choose a database that suits its own data model. A monolithic architecture often depends on relational databases where data is force fit into tables. In a microservices architecture, the data model is not limited to tables; they may employ a data model based on key-value, graph, hierarchical, JSON, streams, search engine, and so on. It's even better for the microservices if the databases offer the flexibility of storing data in the data structures that the program logic is based on.

In our example e-commerce solution, data modeling needs could look like the below table:

Microservice	Read Performance
Cache Server	Key-value
Session Store	Key-value, hash, JSON
User Activity Tracker	Key-value, hash, JSON, sorted set
User Comments and Ratings	Key-value, hash, list, JSON, graph
Product Catalog	Hierarchical (tree), graph, search
Search Engine	Search, graph
Real-time Fraud Detector	Stream, queue, set, sorted set, graph
Order Processing	Relational, stream, queue
Order Fulfillment	Relational, stream, queue

Type of Data Processing

Not all microservices process or manage data at the same stage in its lifecycle. For some microservices, the database could be the source of truth, but for others it may just be a temporary store. To understand the data needs of your microservices better, you can broadly classify the data in the following categories based on how it is processed:

1. **Transient data:** Data such as events, logs, messages and signals usually arrive at high volume and velocity. Data ingest microservices typically process this information before passing it to the appropriate destination. Such microservices require data stores that can hold the data temporarily until it can be stored. These data stores must support high-speed writes. Additional built-in capabilities to support time-series data and JSON are a plus. Since transient data is not stored anywhere else, high availability of the datastore used by your microservice is critical—this data cannot be lost.

2. **Ephemeral data:** Microservices that deliver instant user experiences often rely on a high-speed cache to store the most accessed data. A cache server is a good example of an ephemeral data store. It is a temporary data store whose sole purpose is to improve the user experience by serving information in real time.

   While a data store for ephemeral data does not store the master copy of the data, it must be architected to be highly available, as failures could cause user experience issues and subsequently, lost revenue. Separately, failures can also cause "cache stampede" issues as your microservices try to access monolithic backing databases.
3. **Operational data:** Information gathered from user sessions— such as user activity, shopping cart contents, clicks, likes etc.—are considered operational data. These types of data power instant, real-time analytics, and are typically used by microservices that interface directly with users. This type of data is frequently used in real time to improve user experiences or provide intelligent responses, and later aggregated to support longer-term trend analysis. Even though the data stored in the datastores for this type of processing is not a permanent proof of record, the architecture must make its best effort to retain the data for business continuity and analytics. For this type of data, durability, consistency and availability requirements are high.
4. **Transactional data:** Data gathered from transactions, such as payment processing and order processing, must be stored as a permanent record in a database. The datastores used must provide strong ACID controls and must employ cost-effective means of storage, even as volumes of transactions grow.

In the e-commerce solution example shown in Figure 1, you can classify the microservices and their respective data processing needs as shown in the table below:

Microservice	Type of Data Processing	Objective
Cache Server	Ephemeral	Low-latency user experience
Session Store	Operational	Low-latency user experience
User Activity Tracker	Transient, Operational	Fast data ingest--collect all activities and perform real-time analytics
User Comments and Ratings	Transient, Operational	Data ingest, display, support escalation, communication
Product Catalog	Operational	Low-latency user experience with near-accurate, updated product information
Search Engine	Operational	Low-latency user experience, show up-to-date results in the search
Real-time Fraud Detector	Operational	Detect and report fraud in real time
Reporting	Operational	Gather historical information
Order processing	Transactional	Optimized user experience, permanent store of record
Order Fulfillment	Transactional	Permanent store of record

The other important requirement for your data is to find out whether two or more microservices need to share a common data set. This may happen to all types of data—ephemeral, transient, operational, or transactional. In such scenarios, if you design your microservices to access a common database, you risk reverting back to the monolith architecture, and lose the flexibility and local data access advantages of microservices. In a microservices architecture, you can achieve the sharing of data between databases by utilizing messaging systems, or by employing a multi-master, active-active distributed database with "shared nothing" architecture. The latter will provide a local data copy to your microservices and converge all the updates behind the scenes.

Challenges in selecting the database to meet your requirements

With over 300 databases available in the market, selecting the right database for your microservice may sound like a daunting task. You may find databases from the list that offer all the features and functions you need. However, the challenge is to find a database that not only meets your criteria, but is also lightweight. For efficient orchestration and management, microservices are typically containerized with a light memory footprint. A database with a large binary image and large memory footprint will make it harder to orchestrate as a container.

A note of caution here: watch out for "microservices washing," a concept similar to cloud washing. A few years ago when cloud technology became popular, many companies branded their on-premises solutions as "cloud" solutions without actually making them cloud-native. Microservices-washing is when solutions that were designed to run as large monoliths are erroneously rebranded as microservices, often utilizing legacy service-oriented architecture (SOA) concepts. Packaging a database as a container alone doesn't make it best suited for microservices; it must be lightweight and tunable to meet your data needs.

Operational considerations often drive architects to settle for the lowest common denominator for all use cases—usually a relational database. The slow performance of a relational database doesn't suit the microservices that rely on accessing the data with sub-millisecond latency. On top of that, the rigid schema doesn't allow for flexibility with data models. Document-based NoSQL databases endow the schema with some flexibility. However, performance with respect to number or read/write operations per second is comparable with that of a relational database.

The Redis Enterprise Advantage

Redis Enterprise is designed to suit all your microservices needs. It offers a flexible deployment model—you could deploy Redis Enterprise close to your microservice:

- In your own data center, be it on-premises(VMs or bare metal) or in the cloud
- In a containerized environment, orchestrated by Kubernetes or other container orchestrators
- In a cloud-native/PaaS environment like PCF or Openshift

Being a multi-tenant solution, Redis Enterprise isolates the data between microservices. Most of all, it allows you to tune your database to maintain a trade off between performance and data consistency/durability.

High performance with low latency

Redis Enterprise outpaces the other popular NoSQL databases in the market both in higher throughput (number of operations per second) and lower latency. Its shared-nothing database architecture can perform a million read/write operations per second with just two commodity cloud instances, while making sure the data is also durable. In comparison, other databases in the market require significantly more servers/cloud instances to deliver the same throughput.

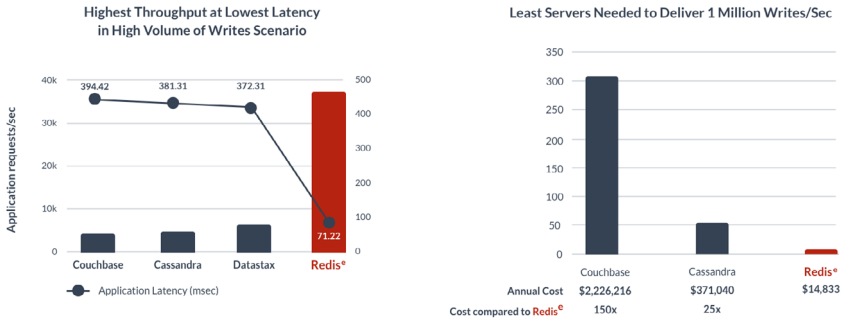


Figure 2. Redis Enterprise performance charts

Redis Enterprise as a multi-model database

Redis Enterprise is a multi-model database that comes with RediSearch, ReJSON, Redis Graph and Redis ML modules built in. Redis Enterprise is built over open source Redis, and supports all of its data structures—Strings, Hashes, Lists, Sets, Sorted Sets, Geo-spatial Indexes, Hyperloglogs, Bitfields, Streams, etc.—so microservices designers can organize their data using the data structure that best suits their performance requirements.

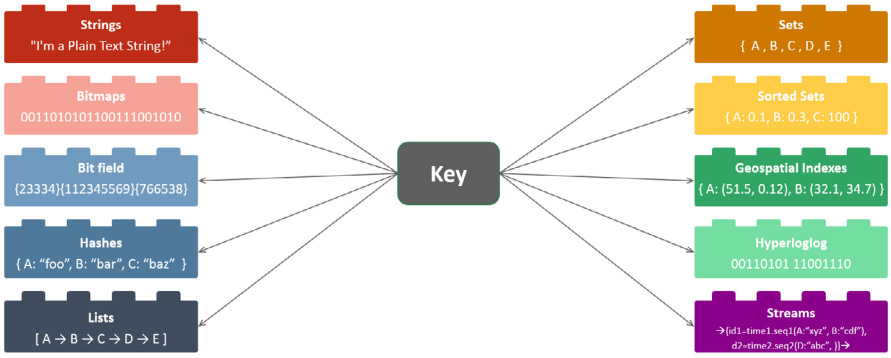


Figure 3. Built-in data structures in Redis

Redis Enterprise simplifies your application and data architecture. It simplifies your polyglot architecture (shown below) into a simple streamlined one.

Microservices Architecture and Polygot Persistence

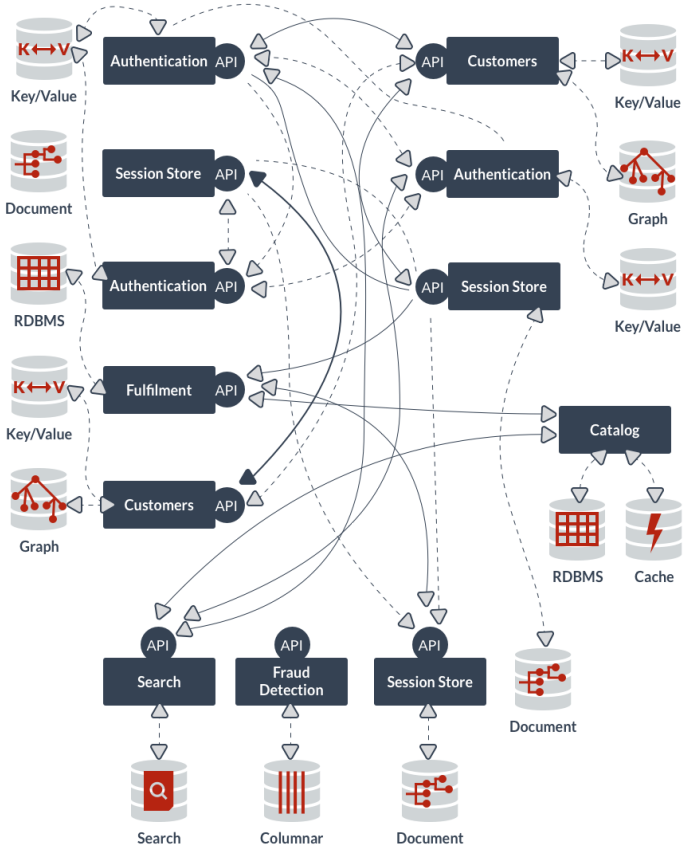


Figure 4a. Microservices - Spaghetti architecture

Redis Enterprise: A Multi-model Database for Microservices

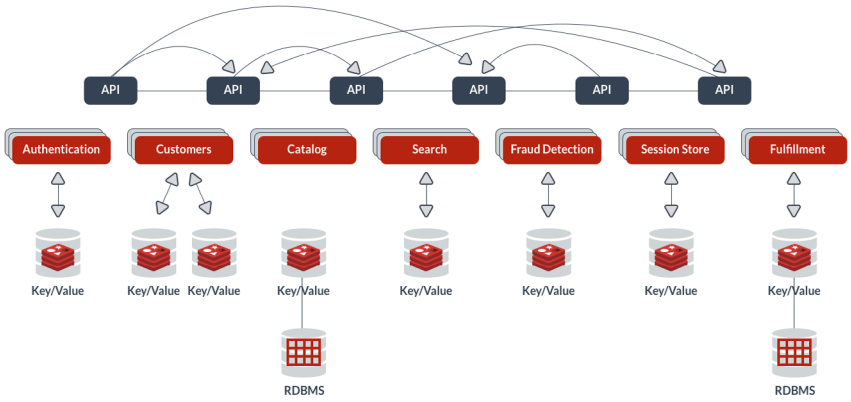


Figure 4b. Multi-model database architecture

High Availability with automatic failure detection

With Redis Enterprise, your microservices get a highly reliable database with always-on availability. Redis Enterprise's architecture offers automated failure detection and zero-downtime scaling completely transparent to the microservices that use it. Failure detection and failover are accomplished within a few seconds, and without data loss.

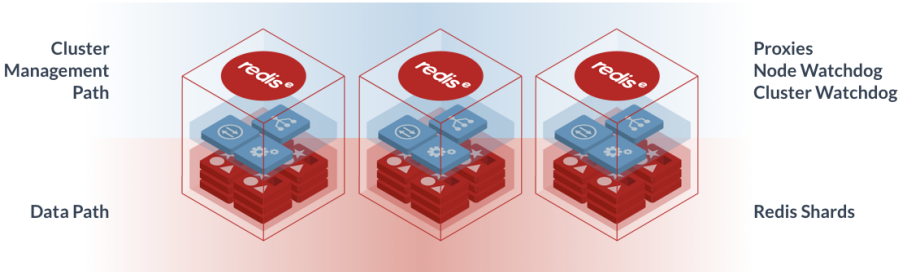


Figure 5. Redis Enterprise architecture for high availability

As shown in Figure 5, the architecture consists of a cluster of container-like Redis Enterprise nodes where each node has an enterprise management layer built on top of open source Redis shards. The management layer has a cluster manager, cluster repository, watchdog entities and UI/CLI/API interface to manage resource provisioning, failovers and several other cluster operations. The data layer includes zero-latency proxies running on every node of the cluster, and diskless replication between the cluster nodes. Should there be a node failure, the nodes establish quorum by electing the secondary database as the new primary. The architecture supports failover across racks, zones and geographies.

Durability options with Redis Enterprise

Redis Enterprise offers durability options ranging from hourly snapshots to log changes every second (with Append Only File (AOF) every second) or every write (with AOF every write). You can configure Redis Enterprise to have a combination of in-memory replication and persistence. These configurable options help you optimize the performance of your microservices for the type of data they handle—ephemeral, transient, operational or transactional. For example, you may choose in-memory replication with no persistence for a caching use case. For a transactional use case, you may choose a combination of in-memory replication and persistence with AOF (append only file). Appendix A lists the popular durability configurations with Redis Enterprise.

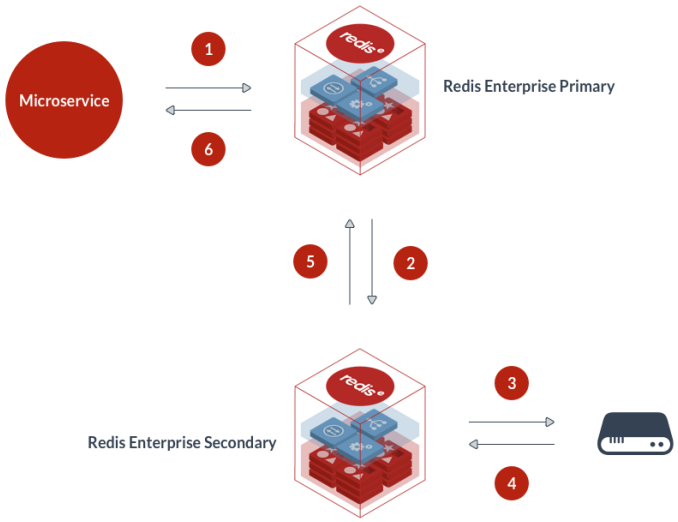


Figure 6. Strong durability configuration with replication and persistence (AOF every write) on secondary with "wait"

Synchronizing data across microservices

Despite all its challenges, a monolithic architecture does guarantee one thing: data consistency. Operational deficiencies are limited, as you only have to deal with one set of data. In the microservices approach, you have to pay significant attention to how data is shared between microservices. You can either use a message stream or a message bus to share messages between microservices. However, if you want to share the dataset across different microservices you'll need a conflict-free solution to ensure the datasets stay in sync and consistent. Redis Enterprise offers you both options, as shown below:

1. Shared datasets between microservices - Redis Enterprise is a CRDTs (conflict-free replicated data types) based, active-active database. When you have multiple instances of a microservice, each with its own database, an active-active distributed database based on CRDTs is especially handy. Each microservice can perform reads/writes with local latencies, and the databases perform the heavy lifting of resolving conflicts.

   Various distributed databases offer different solutions for achieving data consistency. For example, two-phase commit is the most common way to maintain consistent data across distributed datasets in relational databases. However, such databases won't be available for updates should there be a network partition. Other eventual consistency models offered by popular NoSQL databases suffer from the limitations of longer latencies or quorum needs and require a complex application design to provide consistent behavior. They also do not have the rules-based conflict resolution that allows all replicas to converge on the same correct state over time, and cannot guarantee that events will arrive at each replica in the same order across all replicas.

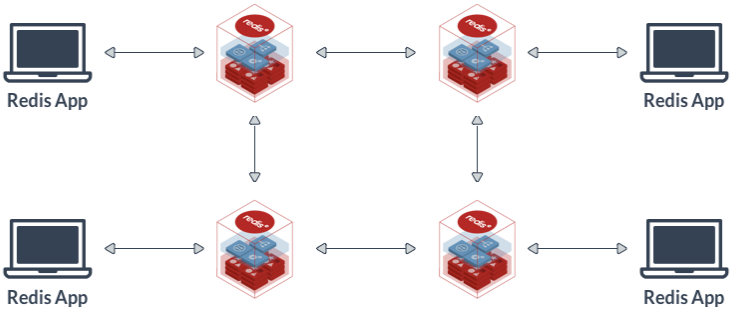


Figure 7. Active-Active database for microservices

CRDT-based, distributed databases deliver strong eventual consistency with causal consistency while maintaining local response times (even in geo-distributed scenarios). Strong eventual consistency allows each replica in the active-active deployment to operate like an isolated local Redis deployment that's completely unaffected by the network latency. Causal consistency guarantees that all replicas will see the updates in the exact same order, thus providing similar characteristics to those of a strong consistency system, but without the overhead associated with running a strong consistency system over multiple-geo regions. The database is available for updates even during network partitions, when the network latency between the data centers is very high, and even in the absence of quorum between the active-active servers.

With Redis CRDTs, each microservice can connect to the local instance of a distributed Redis Enterprise database. The underlying CRDT technology will automatically ensure that all replicas of the data eventually converge to the same consistent state across all microservices.

In the e-commerce example shown in Figure 1, multiple instances of the user ratings microservice will need to keep a common dataset up to date or feed information into the user activity tracker. An active-active, distributed database will aggregate likes/ratings for the same items for different sets of users or users in different locations, in order to display totals or trending items.

2. **Data transfer between microservices** - Redis Enterprise offers three easy ways to transfer data from one microservice to another:
    a. **Pub/Sub:** This technique employs the publish-subscribe model for asynchronous communication between microservices. It's extremely easy to use, and very memory efficient. This technique requires all the subscribers to be up all the time to receive the data.
    b. **Lists:** The List data structure in Redis supports a blocking call for asynchronous data transfer. This is an extremely efficient way to transfer data from one microservice to another. It persists data in Redis if the receiving microservice goes down.
    c. **Sorted Sets:** This data structure is good for time-series data. However, this data structure does not enable asynchronous data transfer.
    d. **Streams:** Streams data structure combines the benefits of Pub/Sub, Lists and Sorted Sets. It allows asynchronous data transfer, supports connectors for multiple clients, and stays resilient to connection loss.

Operationalizing your microservices with Redis Enterprise

Lastly, it is also important to evaluate your solution's deployment and orchestration options to ensure that all microservices are deployed and managed in a homogenous environment. Some key criteria to look for include:

1. **Multi-tenant support:** Redis Enterprise offers a multi-tenant solution for your Redis deployment, in which your Redis endpoint is a database instance that's created inside a Redis Enterprise cluster. You can deploy multiple database instances for all your microservices on a single Redis Enterprise installation, and the data remains isolated between databases. Redis Enterprise's built-in mechanisms protect the instances from noisy neighbors by triggering re-sharding or movement across nodes to balance throughput/latency. The multi-tenancy also provides very efficient creation of databases to the tune of hundreds of instances per second.

2. **Availability as a container:** Microservices deployed as containers and managed by orchestration tools offer a great deal of operational efficiency, but deploying a database as a container can be tricky. Redis is the most popular choice for a database used as a container, as it is already part of the application stack. Redis Enterprise is available as a Docker container.

3. **Layered orchestration in Redis Enterprise:** Popular orchestration systems such as Kubernetes and BOSH are built to manage large numbers of application components, but do not suffice if your applications require a stateful, persistent, highly available database layer. Redis Enterprise integrates with such orchestrators and offers its own internal orchestration tools to assure the high availability needed for a stateful, persistent, Redis database layer. This additional orchestration detects process failures and triggers failover within seconds, so your applications don't experience

outages. The additional orchestration also allows for greater resource efficiency by maximizing the number of databases you can run on the same set of servers in an isolated and multi-tenant manner.

4. **Kubernetes support in Redis Enterprise:** You can orchestrate a Redis Enterprise container as a cloud-native database service in your Kubernetes environment. Redis Enterprise's Kubernetes integration enables:
   - Auto-discovery of Redis Enterprise service with StatefulSet and a headless service to handle DNS resolution
   - Managing Redis Enterprise licenses inside the Kubernetes secrets primitive
   - Bootstrapping multi-node Redis Enterprise clusters using Kubernetes secrets
   - Fully utilizing Redis Enterprise's multi-tenant architecture by creating Redis databases in the multi-node cluster
   - Maintaining and upgrading Redis Enterprise

5. **Cloud/on-premises options:** Whether your microservices run in a private cloud environment, on-premises or in the cloud, Redis Enterprise's flexible deployment options enable you to keep your microservice as close to the data as possible. Redis Enterprise's VPC deployment option delivers a fully managed cloud service within your VPC, allowing you to take full control of your data.

6. **Redis Enterprise on Pivotal Cloud Foundry and Red Hat OpenShift:** With BOSH integration, you could easily deploy, manage and scale Redis Enterprise on the Pivotal Cloud Foundry platform. Since Red Hat OpenShift uses Kubernetes as the primary orchestration platform, you can take full advantage of Redis Enterprise's Kubernetes support to manage your database and microservices in OpenShift environments.

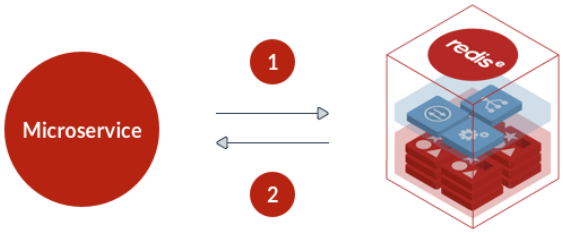
Conclusion

A microservices architecture offers the flexibility and agility that a monolithic architecture cannot deliver. On the flipside, the idea of having a specialized database for each microservice may lead way to a polyglot architecture, bound to become expensive and difficult to operate over a period of time. Redis Enterprise offers a high-speed, multi-model database with high availability and durability options that are crucial for your microservices, enabling you to get rid of your monolith architecture while avoiding a polyglot system.

Redis Enterprise is easy to operate. It's available as a cloud service on all popular public cloud platforms and in VPC/on-premises environments, and as a container that can be orchestrated using Kubernetes, BOSH and Docker Swarm. Redis Enterprise's multi-tenant support enables you to run up to a few hundred databases on a simple three-server cluster, maximizing the use of your resources.

Appendix A. Redis Enterprise configurations with different durability options

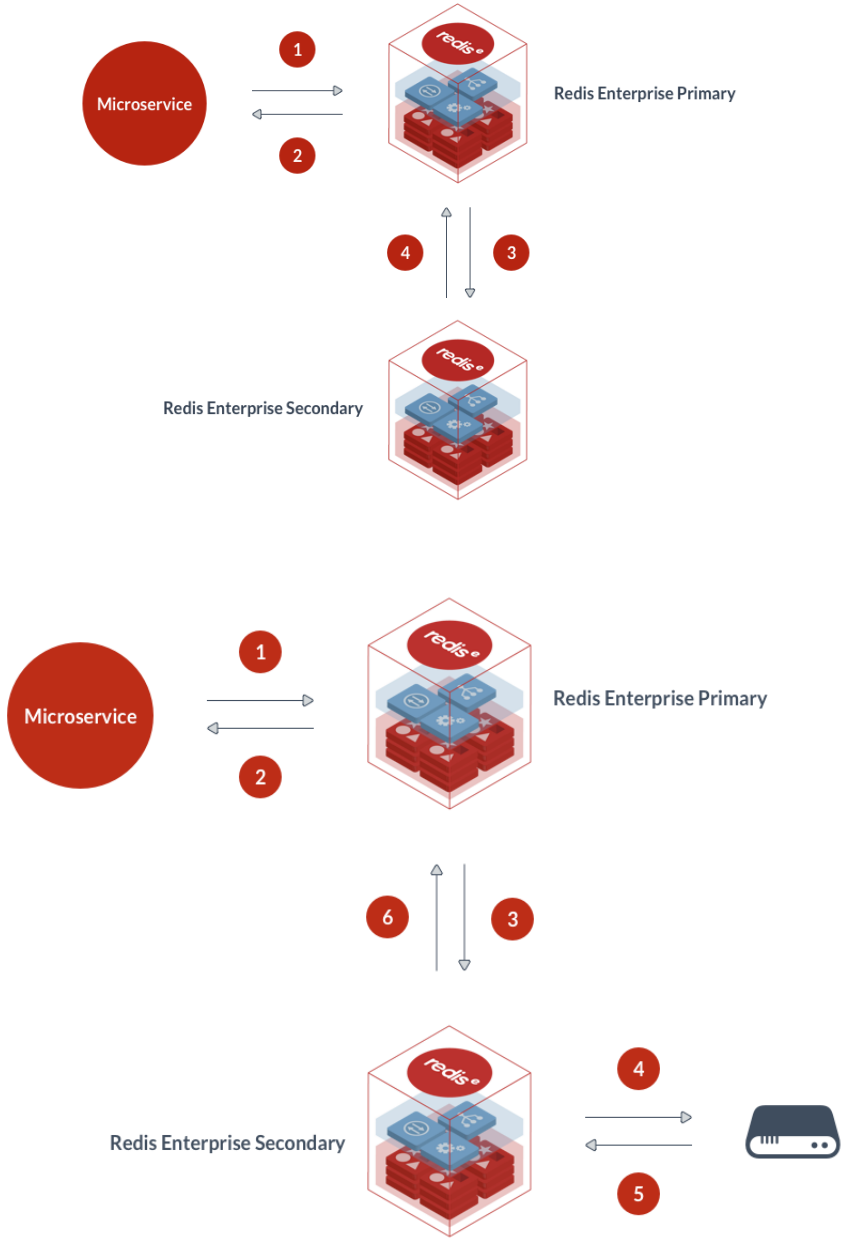
1. Ephemeral data store—only RAM data; no replication



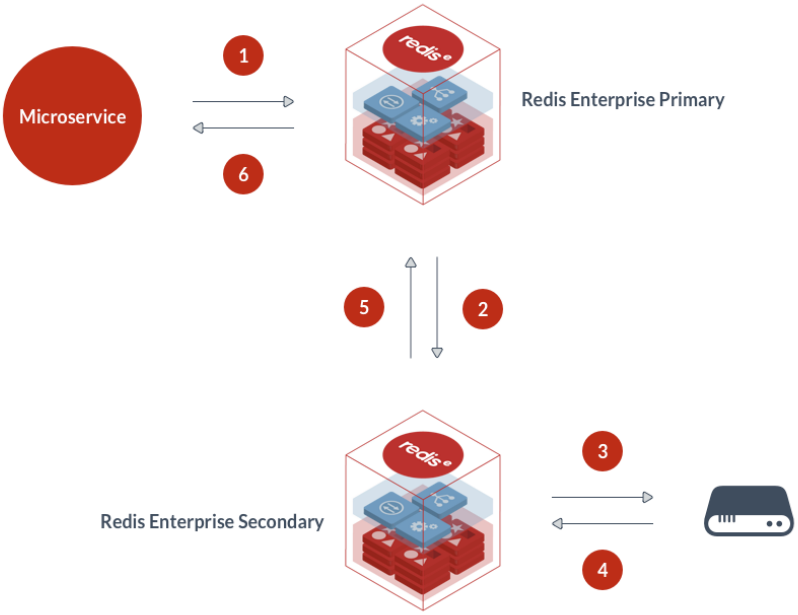
2. Persistence to disk with AOF every second; no replication

3. Persistence to disk with AOF every write

4. Replication; no persistence



6. Replication and persistence (AOF every write) on secondary with wait


Microservice
1
6
Redis Enterprise Primary
5
2
Redis Enterprise Secondary
3
4

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redislabs.com