Enterprise Caching: Strategies for Caching at Scale

Part 2 of the Enterprise Caching Series

Based on *Caching at Scale with Redis*
by Lee Atchison

Contents

1. What is Enterprise Caching and Why Do You Need It?

Enterprise caching is essential for application performance at scale. Without it, your application is destined for the rocks as it grows beyond its abilities. In today's marketplace where fast and consistent user experiences are required, businesses can't afford to launch apps without an enterprise-grade cache.

Basic caches bring speed, supercharging application response time by storing the most frequently used data in front of slower primary databases. Think of them as storage units that liberate original data stores from having to do a lot of the heavy lifting, freeing up more resources for them to process other incoming queries, and keeping the most important data available in-memory to provide real-time experiences.

What does real time mean?
Research into human response indicates applications have roughly 100 milliseconds (ms)—one third of the time it takes to blink—before users feel like they're waiting for a response. If digital interactions are going to be seamless, then they need to happen in real time. That means every request needs to be sent, processed, and responded to in less than 100ms.

But speed alone isn't enough. Modern enterprises require data that's always on and available to every user at any place at any time. In order to perform at scale, an enterprise cache requires the ability to:

- Seamlessly scale to meet peaks in user demand
- Deliver consistent and accurate data regardless of where the cache is deployed or where users are located
- Support modern cloud-based, multicloud, or hybrid application architectures
- Deliver instant experiences at all times

If the enterprise cache struggles with any of the above, it will hamper application performance, causing flawed digital experiences and kickstarting a chain of events that begins with a drop in users and ends with plummeting revenue. To avoid this scenario, your cache needs to scale with ease—a characteristic bespoke to enterprise-proven caches.

But how do you optimize caching as you scale? And how are you able to expand your caching footprint to multi cloud or hybrid environments? These are important questions that cannot be overlooked by anyone looking to scale at an enterprise level.

Using Lee Atchison's book, *Caching at Scale With Redis*, we'll reveal the answers to these questions to help you ensure that your cache is optimized to scale.

1. What is Enterprise Caching and Why Do You Need It?

How is Enterprise Caching different from Basic Caching?

While both are fast, enterprise caches have to perform in larger and more complex environments—with much more at stake. Enterprise caching provides sub-millisecond responses while processing phenomenally high volumes of data that span across different geographical locations and deployment environments.

Enterprise applications are complex and the data requirements to keep them performing optimally at all times are even more so. Because of this, the architectural makeup of enterprise caches differs significantly compared to standard caches, offering unparalleled levels of consistency, availability, performance, scalability, deployment flexibility, and geo-distribution.

Why is Enterprise Caching important?

There are 3.48 million apps on Google Play and 2.22 million apps on IOS, each sparring for the user's attention and commitment to their app. Why would someone stay committed to yours?

The user experience is the number one priority, and to keep them committed to your application, your application needs to be firing on all cylinders 24/7. But this is easier said than done. Enterprise applications are bigger, more complex, and pull in more people across different geographical locations, requiring caches to be dynamic and powerful enough to maintain optimal performance levels on a global level.

The stakes couldn't be any higher: performance needs to be flawless to meet user expectations. Lag burns holes in the user's experience and it only takes one for everything to go up in flames. Users demand real-time responsiveness and won't settle for anything less.

These demands make the advanced data-processing capabilities of enterprise caches fundamental to the longevity of any application. Failing to have one will swamp the main database with an unsustainable amount of data to process, forcing it to work in overdrive. Eventually, when traffic spikes high enough, the database will falter and that's when the house of cards gives way, punctuating the user experience with lags which only creates frustration.

Engagement will plummet. Brand reputation will take a hit. And users will flock to your competitors' applications that can promise a seamless experience.

2. Cache Scaling

"How many seconds does it take to change a lightbulb? Zero – the lightbulb was cached."

While a basic cache can bring speed, an enterprise cache provides additional features required to meet modern user expectations. One key feature is the ability to scale to meet increased demand. Enterprise caches must perform seamlessly regardless of sudden and unexpected surges in application demand or gradual and predicted growth in usage. In fact, because of the volume of customers impacted, it's during these times of peak demand that the seamless performance enabled by an enterprise cache matters most.

And so this poses the big question: how can you scale your cache? Well that's what this chapter from *Caching at Scale with Redis* is all about. Atchison goes through this topic with a fine-tooth comb and highlights all of the important factors needed to supercharge scaling to an enterprise level.

The chapter first highlights the different ways a cache typically reaches the limits of its performance, then digs into the two different ways you can scale your cache: vertical scaling and horizontal scaling. From here, you'll get a clear insight into what technique is best suited to your individual circumstances as well as the required steps to carrying them out.

Atchison finally reveals the important characteristic of Redis Enterprise that allows it to scale on a mass level, across different regions, and with ease. By the end of the chapter you'll understand the core fundamentals of how to scale your cache to an enterprise level.

"Chapter 6: Cache Scaling" from *Caching at Scale with Redis* by Lee Atchison

Caches are hugely important to building large, highly scalable applications. They improve application performance and reduce resource requirements, thus enabling greater overall application scalability.

But what do you do when the cache itself needs to scale?

Once your application has reached a certain size and scale, even your cache will meet performance limits. There are two types of limits that caches typically run into: **storage limits** and **resource limits**.

Storage limits are limits on the amount of space available to cache data. Consider a simple service cache, where service results are stored in the cache to prevent extraneous service calls. The cache has room for only a specific number of request results. If that number of unique requests is exceeded, then the cache will fill, and some results will be discarded. The full cache has reached its storage limit, and the cache can become a bottleneck for ongoing application scaling.

2. Cache Scaling

Resource limits are limits on the capability of the cache to perform its necessary functions—storing and retrieving cached data. Typically, these resources are either network bandwidth to retrieve the results, or CPU capacity in processing the request. Consider the same simple service cache. If a single request is made repeatedly and the result is cached, you won't run into storage limits because only a single result must be cached. However, the more the same service request is made, the more often the single result will be retrieved from the cache. At some point, the number of requests will be so large that the cache will run out of the resources required to retrieve the value

Improving cache scalability

In order to improve the scalability of a cache, you must increase storage limits and/or resource limits, as necessary, to avoid either of these limits impacting your cache's performance. Similar to how parts of a computing system are scaled, there are two primary ways to increase the scale of your cache: **vertical scaling** and **horizontal scaling**.

Vertical scaling, or scaling up and scaling down, involves increasing the resources available for the cache to operate. Typically, this involves moving to a more powerful computer running the cache. For a cloud-operated cache, this often means moving to a larger instance.

Vertical scaling can increase the amount of RAM available to the cache, thus reducing the likelihood of the cache reaching a storage limit. But it can also add larger and more powerful processors and more network bandwidth, which can reduce the likelihood of the cache reaching a resource limit.

Horizontal scaling, or scaling out and scaling in, involves adding additional computer nodes to a cluster of instances that are operating the cache, without changing the size of any individual instance. Depending on how it's implemented, horizontal scaling can also improve overall cache reliability, and hence application availability.

In other words, vertical scaling means increasing the size and computing power of a single instance or node, while horizontal scaling involves increasing the number of nodes or instances.

Horizontal scaling techniques

There are many different ways to implement horizontal scaling, each with a distinct set of advantages and disadvantages.

2. Cache Scaling

Read replicas

Read replicas are a technique used in open source Redis for improving the read performance of a cache without significantly impacting write performance. In a typical simple cache, the cache is stored on a single server, and both read and write access to the cache occur on that server.

With **read replicas**, a copy of the cache is also stored on auxiliary servers, called read replicas. The replicas receive updates from the primary server. Because each of the auxiliary servers has a complete copy of the cache, a read request for the cache can access any of the auxiliary servers—as they all have the same result. Because they are distributed across multiple servers, a significantly greater number of read requests can be handled, and handled more quickly.

When a write to the cache occurs, the write is performed to the master cache instance. This master instance then sends a message indicating what has changed in the cache to all of the read replicas, so that all instances have a consistent set of cached data.

A large Redis implementation consisting of at least three servers is illustrated in Figure 6-1. All writes to the Redis database are made to the single master. This single master sends updates of the changed data to all of the replicas. Each replica contains a complete copy of the stored Redis database. Then, any read access to the Redis instance can occur on any of the servers in the cluster.

This model does not improve write performance, but it can increase read performance for large-scale implementation by spreading the read load across multiple servers. Additionally, availability can be improved—if any of the read replicas crash, the load can be shared to any of the other servers in the cluster, so the system remains operational. For increased availability, if the Redis master instance fails, one of the read replicas can take over the master role and assume those responsibilities.

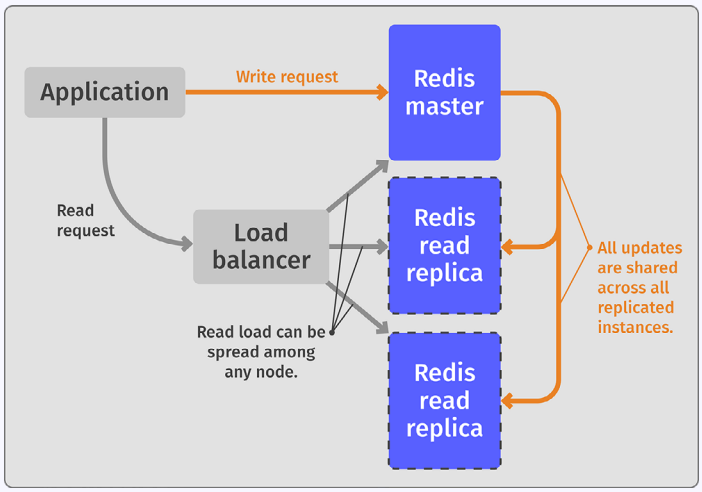

Figure 6-1. Horizontal scalability with read replicas

2. Cache Scaling

Sharding

Sharding is a technique for improving the overall performance of a cache, along with increasing both its storage limits and resource limits.

With sharding, data is distributed across various partitions, each holding only a portion of the cached information. A request to access the cache (either read or write) is sent to a shard selector (in Redis Enterprise, this is implemented in a proxy), which chooses the appropriate shard to which to send the request. In a generic cache, the shard selector chooses the appropriate shard by looking at the cache key for the request. In Redis, shard selection is implemented by the proxy that oversees forwarding Redis operations to the appropriate database shard. It then uses a deterministic algorithm to specify which shard a particular request should be sent to. The algorithm is deterministic, which means every request for a given cache key will go to the same shard, and only that shard will have information for a given cache key. Sharding is illustrated in Figure 6-2.

Sharding is a relatively easy way to scale out a cache's capacity considerably. By adding three shards to an open source Redis implementation, for instance, you can nearly triple the performance of the cache, and triple the storage limits.

But sharding isn't always simple. In generic caches, choosing a shard selector that effectively balances traffic across all nodes can require tuning. It can also lower availability by increasing dependency on multiple instances. Failure of a single instance can, if not properly managed, bring down the entire cache.

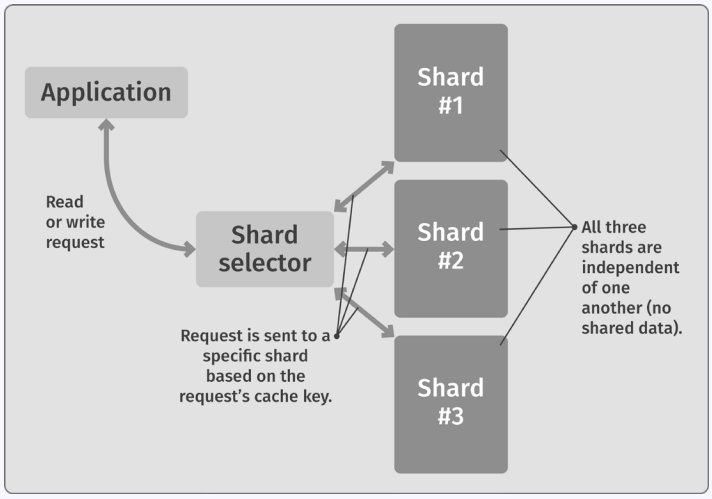

Figure 6-2. Horizontal scaling via sharding

Redis Clustering addresses these issues and makes sharding simpler and easier to implement. Redis Clustering uses a simple CRC16 on the key in order to select one of up to 1,000 nodes that contain the desired data. A re-sharding protocol allows for rebalancing for both capacity and performance management reasons. Failover protocols improve the overall availability of the cache.

In open source Redis, clustering is implemented client-side in a cluster-aware client library. This works, but requires client-side support of the clustering protocol. Redis Enterprise avoids these issues by implementing a proxy protocol to provide clustering server side, allowing any client to utilize the clustered cache.

2. Cache Scaling

Sharding is an effective way to quickly scale an application, and it is used in a number of large, highly scaled applications. While the concept of sharding has inherent advantages and disadvantages, Redis Clustering eliminates much of the complexities of sharding and allows applications to focus on the data management aspects of scaling a large dataset more effectively.

Active-Active (multi-master)

Active-Active, i.e. multi-master, replication is a way to handle higher loads of both the write and the read performance of a cache.

As with read replicas, Active-Active adds multiple nodes to the cache cluster, and a copy of the cache is stored equally on all of the nodes. Because each node contains a complete copy of the cache, this has no impact on the storage limit of a cache. A load balancer is used to distribute the load across each of the nodes. This means that a significantly larger number of requests can be handled, and handled faster, because they are distributed across multiple servers.

When a write to one of the cache nodes occurs, the instance that receives the write sends a message indicating what has changed in the cache to all of the other nodes, so that all instances have a consistent set of cached data.

In the large cache implementation illustrated in Figure 6-3, the cache consists of at least three servers, each running a copy of the cache software and each with a complete copy of the cache database. Any of them can handle any type of data request.

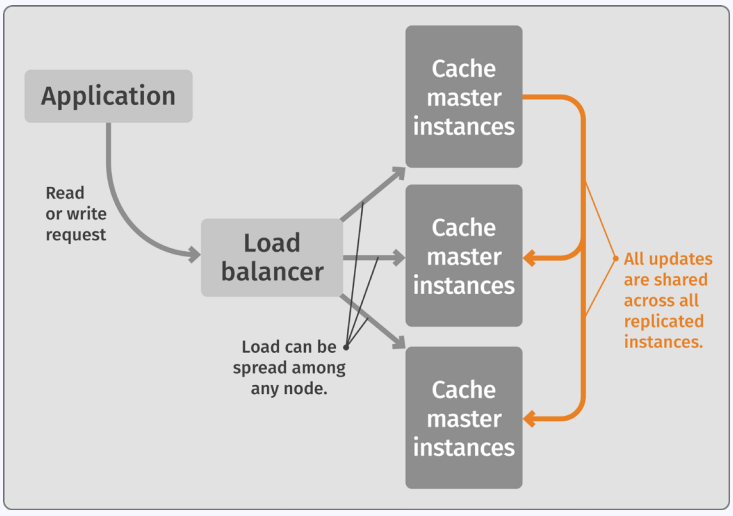

Figure 6-3. Multi-master replication

This model also increases overall cache availability, because if a single node fails, the other nodes can take up the slack.

But what happens when two requests come in to update the same cached data value? In a single-node cache, the requests are serialized and the changes take place in order, with the last change typically overriding previous changes.

2. Cache Scaling

In a multi-master model, though, the two requests could come to different masters, and the masters could then send conflicting update messages to the other master servers. This is called a **write conflict**. An algorithm of some sort must be written to resolve these conflicting writes and determine how the multiple requests should be processed (e.g. which one or ones should be processed, and which should be ignored or in what order should the requests be processed). Additionally, **data lag** can occur, meaning that when data is updated in one node, it may take a bit of time before it's updated in all the nodes. Hence, for a period of time, different nodes may contain different data values. This algorithm can be error-prone and result in an inconsistent cache. Care has to be taken that these sorts of problems do not occur, or at least can be successfully repaired when they do.

Redis Enterprise's Active-Active Geo Distribution

Open source Redis does not natively support multi-master redundancy. However, Redis Enterprise does support a form of multi-master redundancy called Active-Active Geo-Distribution.

In this model, multiple master database instances are held in different data centers which can be located across different regions and around the world. Individual consumers connect to the Redis database instance that is nearest to their geographic location. The Active-Active Redis database instances are then synchronized in a multi-master model so that each Redis instance has a complete and up-to-date copy of the cached data at all times. This model is called Active-Active because each of the database instances can accept read and write operations on any key, and the instances are peers in the network.

Redis Enterprise Active-Active Geo-Distribution has sophisticated algorithms for effectively dealing with write conflicts, including implementing conflict-free replicated data types (CRDTs) that guarantee strong eventual consistency and make the process of replication synchronization significantly more reliable. Note that the application must understand the implications of data lag and resulting write conflicts, and must be written so that these issues aren't a problem.

Cache scaling technique summary

Table 6-1 shows a summary of the scaling techniques discussed in this chapter, along with their advantages and disadvantages.

Which technique, if any, you choose to use depends on your application's architecture and requirements, and your organization's goals.

Technique	Storage limits	Resource limits	
Read replica		↑	Easy to configure in open source Redis, but does not improve write performance
Sharding	↑	↑	Improves database capacity as well as performance. Easy to configure with Redis Enterprise Cluster. Requires tuning.
Active-Active		↑	Ensures each Redis master has an up-to-date copy of the cached data at all times. Delivers local latency on read and write operations in multiple regions. Can continue to handle read and write operations even when the majority of geo-replicated regions are down.

Table 6-1. Scaling technique summary

3. Cache Consistency

"Cache consistency isn't everything. A cache can be wrong, yet consistently so."

As businesses and their applications scale, a new complication is introduced - complexity. And with complexity comes data inconsistency. Cache consistency refers to a cache's ability to consistently retrieve and supplement users with the right data values. A mismatch between the value stored in the cache and value required by the user can hinder application performance.

Enterprise caches need to be accurate in their data retrievals to effectively tailor the experience to each user. Since there will be more queries to process, scaling with an inability to cache consistently will lead to more errors, resulting in large segments of the target market being provided with the wrong data.

This limits an app's ability to tailor the experience to the user, which is absolutely fundamental to building rapport and instilling loyalty amongst consumers.

This chapter goes into the nuts and bolts of cache consistency by highlighting the different variables that make a cache inconsistent, along with a dissection of the underlying chain of events that occur within a cache when trying to achieve cache consistency.

Guiding you through steps A-Z, Atchison breaks everything down chronologically and pinpoints exactly when cache consistency is likely to occur. By the end of the chapter, you'll understand the ins and outs of cache consistency and how it determines your ability to scale effectively.

"Chapter 7: Cache Consistency" from *Caching at Scale with Redis* by Lee Atchison

Maintaining cache consistency is one of the greatest challenges in managing the operation of a cache, so choosing the right caching pattern is essential.

In Chapter 3, "Why Caching?", we introduced a multiplication service and demonstrated how caching could be used to improve the performance of this service. Going back to that example, what happens if the multiplication service doesn't have the value "12" stored as the result of "3 times 4", but instead has the value of "13" stored?

In that case, when a request comes in to return the result of "3 times 4", the cached value will be used rather than a calculated value, and the service will return "13", an obviously incorrect result that would mostly likely never occur in the real world.

3. Cache Consistency

This is an example of a cache that is inconsistent, because it has stored an up-to-date response to a request. Sometimes, it can be difficult to realize that this is happening, and even more difficult to remove these inconsistent results. This may or may not be a major problem for the application, depending on how the application uses the data.

How does a cache become inconsistent?

There are many ways a cache can become inconsistent, including:

1. When the underlying results change, and the cache is not updated
2. When there is a delay in updating cached results
3. When there is inconsistency across cached nodes

Let's look at each of these issues independently.

The underlying results change and the cache is not updated

Consider Figure 7-1, which shows a service requesting a result to be read from a presumably slow data source, such as a remote service or database. In order to speed up reading of the data, a cache is used to make access to frequently used results quicker.

The user requests a specific value to be read from the slow data source. The cache is consulted. If the value is not in the cache, the slower backing data source is consulted, the result is stored in the cache, and the request returns. However, if the result is stored in the cache, the cached value is returned directly.

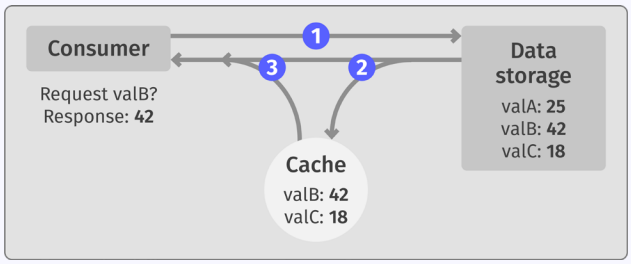

Figure 7-1. A simple cache fronting a slow data source

But what happens if the value in the underlying data source has changed? In a cache-aside pattern, if the old value is still in the cache, then the old value will be returned from future requests, rather than the new value. The cache is now inconsistent. This is illustrated in Figure 7-2. In order for the cache to become consistent again, either the old value in the cache has to be updated to the new value, or the old value has to be removed from the cache, so future requests will retrieve the correct value from the underlying data store.

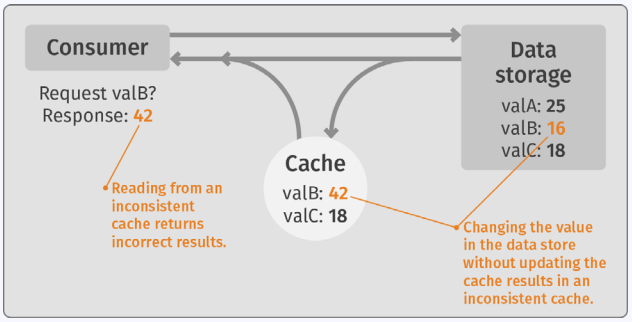

Figure 7-2. Updating the data store without updating cache results in an inconsistent cache.

3. Cache Consistency

A delay in updating cached results

When an underlying data store changes a value, and it needs to update the cache about the changed value, often it sends a message to the cache telling it to either remove the old value or update the cached value to the new, correct value. This processing takes some time, during which the cache still has the old, inconsistent value. Any request that comes in after the data value has changed, but before the cache has updated its value, will return the inconsistent, incorrect value.

This delay could, and should, be quite short—hopefully short enough so that the delay does not cause any serious problems. However, in some cases it can be quite lengthy. Whether or not this delay causes a problem is entirely dependent on the use case, and it is up to the specific application to decide if the delay causes any issues.

Additionally, some caches can be used to cache data from a dynamically changing data store. This is often the case in database caches, for instance, where the underlying data changes occasionally yet regularly.

In these cases, one strategy is to set an expire time on the cache, requiring the cached values to be thrown away and reread from the underlying data store at regular intervals, limiting the amount of time the cached value may be inconsistent. While this strategy can reduce the length of time a cached value is inconsistent, it doesn't remove the inconsistency entirely. As such, this strategy is used only for caching dynamically changing data, where some amount of variation from returning an accurate result is acceptable. An example of this type of cache might be caching the number of likes on a social media post.

In this case, the number changes continuously, but if the cache is set to expire every, say, 15 minutes, you can guarantee the cached value is always accurate to within the most recent 15-minute value. The cache value is still inconsistent, but the inconsistency is minimal, and within the bounds of acceptability for that application use case.

Inconsistency across cached nodes

In highly scaled or highly distributed caches, multiple computer nodes are often used to implement the cache. This was discussed in greater detail in chapter 6, "Cache Scaling."

In many scenarios, multiple cache nodes will have duplicate copies of all or part of the cache. An algorithm is used to keep the cache values up to date and consistent across all of the cache nodes. This is illustrated in Figure 7-3.

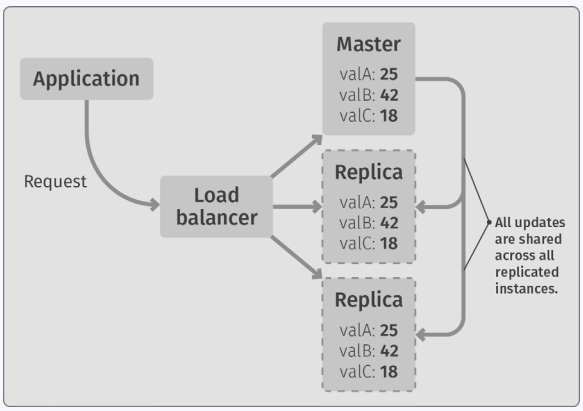

Figure 7-3. Replicated cache with consistent data

3. Cache Consistency

However, updating multiple nodes, especially if there are a lot of them or if they are geographically distributed, can take a significant amount of time. During the time the nodes are being updated, different nodes may contain different values. As a result, depending on which cache node receives a request, the result returned can be inconsistent until eventual consistency is reached. This is illustrated in Figure 7-4.

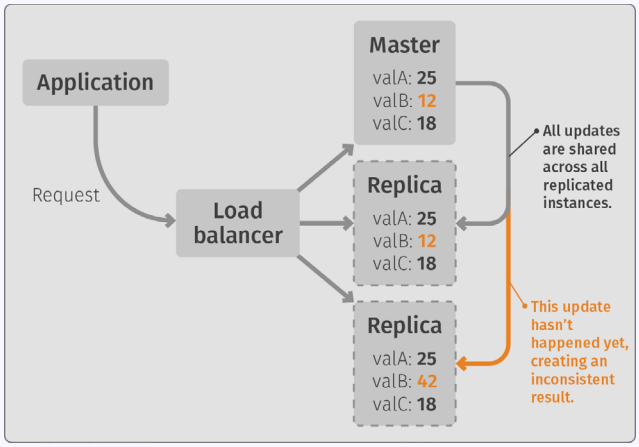

Figure 7-4. Replicated cache with delayed (inconsistent) data

A real-world example of this is when a website is cached at edge locations around the world, in order to speed up access to various portions of a website, such as images, diagrams, and photographs. Often, there are many of these caches around the world, and updating all of them to include updated information, such as an updated diagram, can take a long time. The result is that, for some period of time after an update to a website is made, the "old" website will still be returned for some people in some parts of the world, until all the caches have been updated with the new content. There are various application-specific strategies to address this issue, and how important of a problem this is depends on the application and the application's needs.

4. Caching and the Cloud

"Cloudy with a chance of caching."

Cloud caching is a fundamental part of the scaling process. It boosts application performance by reducing the number of database round trips, creating a more responsive and agile application that boosts engagement.

However, there are many different ways you can expand your caching footprint to the cloud and the route to doing so may not be so straightforward. In this chapter, Lee Atchison walks you through the different ways you can set up and configure Redis as a cache server.

You'll be introduced to the different major cloud providers that are compatible with Redis, and Atchison will examine Redis Enterprise's powerful cloud caching capabilities, breaking down the different ways you can connect Redis to the cloud which include: self-hosting, multi-cloud deployments, and cluster deployments.

Implementing these steps will amplify your ability to scale and launch a powerful application that can process millions of queries simultaneously on a global level. It's a crucial component to achieving enterprise level caching and meeting user expectations.

"Chapter 8: Caching and the Cloud" from *Caching at Scale with Redis* by Lee Atchison

Running a single open source Redis instance on premises is rather straightforward. There are a couple of options for how to set it up, and none of them are complex.

However, in the cloud, there are many different ways to set up and configure Redis as a cache server. In fact, there are more ways to set up a Redis cache than there are cloud providers. This chapter discusses some of the various cloud options available.

Redis services on major cloud providers

All the major cloud providers offer Redis databases, often configured primarily for caching purposes, that are easy to set up and easy to administrate. They can be utilized just like any of a given provider's other cloud services.

4. Caching and the Cloud

Cloud providers' caches

All major cloud providers, including Amazon Web Services (AWS), Google Cloud, Microsoft Azure, and IBM Cloud offer services that include the open source version of Redis. These are available in a wide variety of sizes. Often, they are available as either single instances or with load-balanced read replicas included. They can be set up in a variety of regions across the globe.

These instances are easy to set up and use, can be turned on/off very quickly, and are typically charged by the hour or the amount of resources consumed. This makes them especially well-suited for development, testing, and autoscaled production environments.

Several other service providers offer preconfigured, cloud hosted versions of Redis instances, including:

- **Redis To Go**
- **Heroku**
- **ScaleGrid**
- **Aiven**
- **Redis Cluster** (specializes in Redis-hosted on Kubernetes)
- **Digital Ocean**
- **cloud.gov** (specializes in governments and government contractors)

Redis Enterprise Cloud

Redis offers a cloud-hosted version of its enterprise-grade, fully supported, enhanced Redis Enterprise software. Redis Enterprise Cloud is the fully managed DBaaS that provides transparent high availability and supports Active-Active Geo-Distribution as well as hybrid and multicloud deployments. Redis Enterprise Cloud is available from Redis on the three major cloud providers, AWS, Azure, and Google Cloud. On Microsoft Azure, for example, Redis Enterprise is offered as a fully supported first-party offering called Azure Cache for Redis Enterprise.

In addition to the usefulness of Redis Enterprise Cloud for high-performance production environments, there is also a free tier that can be used for test drives and development purposes. If you are serious about using Redis for high-end production workloads, Redis Enterprise Cloud could be right for you.

4. Caching and the Cloud

Self-hosted in the cloud

You can, of course, install and run Redis yourself, either the open source or the Redis Enterprise version, on standalone cloud compute instances, just as you can install and run Redis on computers in your own on-premises data centers. Nothing magical is needed. However, it is important to make sure security configurations are created and handled correctly. All the major cloud providers, and Redis Enterprise Cloud, offer standard security capabilities based on their normal cloud-security settings—all set up and pre-configured for you. If you set up and run the instances yourself, however, you have to make sure the security settings are correct, certificates are valid, and security is maintained.

Hybrid and multicloud deployments

As discussed in Chapter 6, "Cache Scaling," you can set up an open source Redis instance to be composed of one Redis primary instance, and one or more Redis read replicas. This setup is shown in Figure 8-1.

In a typical cloud setup, these read replicas are put into separate availability zones for improved system availability. However, it is possible to put the read replicas in entirely different cloud regions in order to provide improved read performance at various geographic locations. This is shown in Figure 8-2.

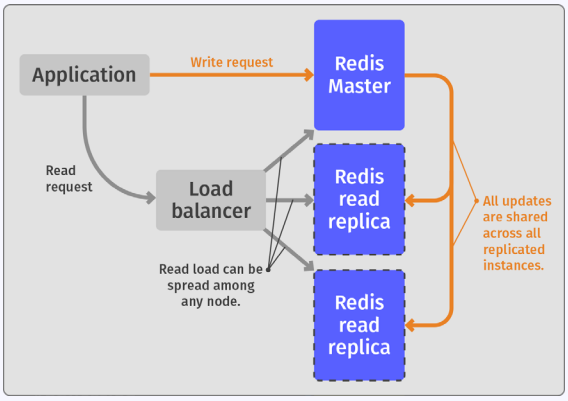

Figure 8-1. Read replicas

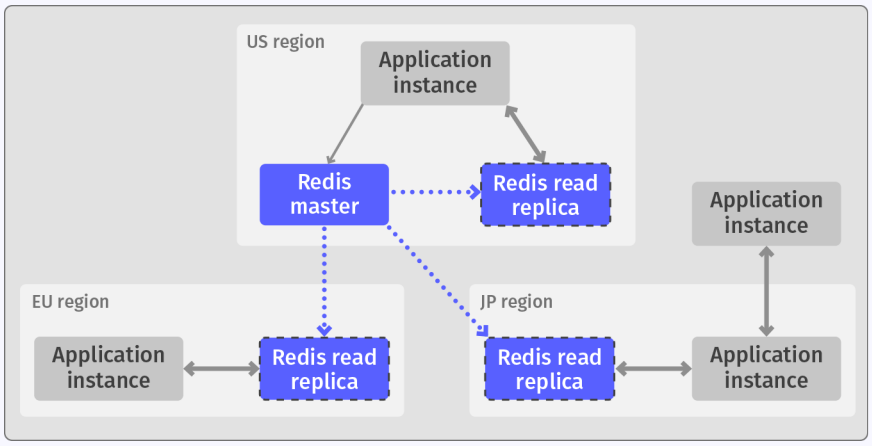

Figure 8-2. Multi-region read replicas

4. Caching and the Cloud

In theory, this model works even in cases in which the different regions are provided by different cloud providers. The only caveat is that the replication setup commands must be available for configuration by the cloud provider, and those commands can be restricted on some levels of service from some providers.

Nonetheless, you could set up Redis manually on separate compute instances in multiple cloud providers and then configure the replication so that your read replicas from a given cloud provider are connected to a master in another cloud provider. This is shown in Figure 8-3.

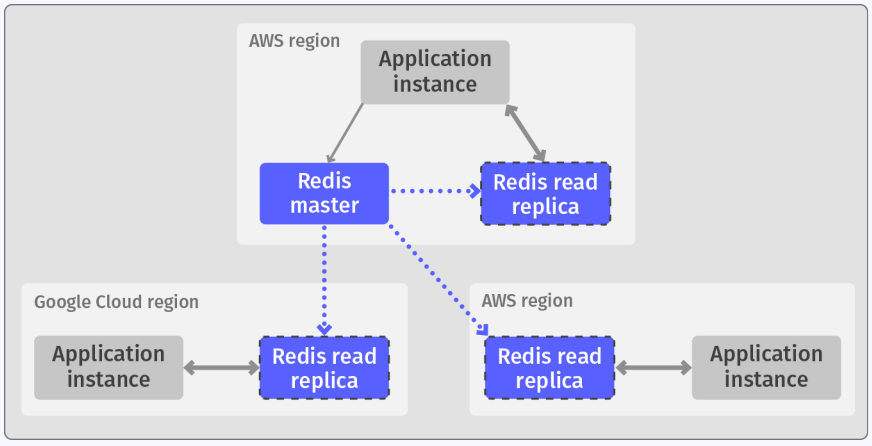

Figure 8-3. Multicloud read replicas

Redis Enterprise cluster deployments

Using basic open source Redis, you are limited to a single master and any number of read replicas. All writes must be sent to the single Redis master. This limits the usefulness for multi-region deployments, and multi-provider, multicloud deployments. This is because when an application is spread across multiple regions and/or multiple providers, only the read performance can be improved by specifying a local read replica, as shown in Figures 8-2 and 8-3.

Write requests must still go back to the single master instance. So, in the Figure 8-2 example, if the application in the JP region wants to write to the Redis database, it must send that write to the Redis master instance in the US region. This can have a significant impact on write performance.

4. Caching and the Cloud

In order to improve both write and read performance in a multi-region or multicloud deployment, you must use a different replication architecture than the simple master-replica architecture described here. Instead, you must create a multi-cluster topology as shown in Figure 8-4. This requires multi-master capability, which is not available out of the box in open source Redis, but in Redis Enterprise, multiple masters can be deployed across multiple regions using an Active-Active deployment.

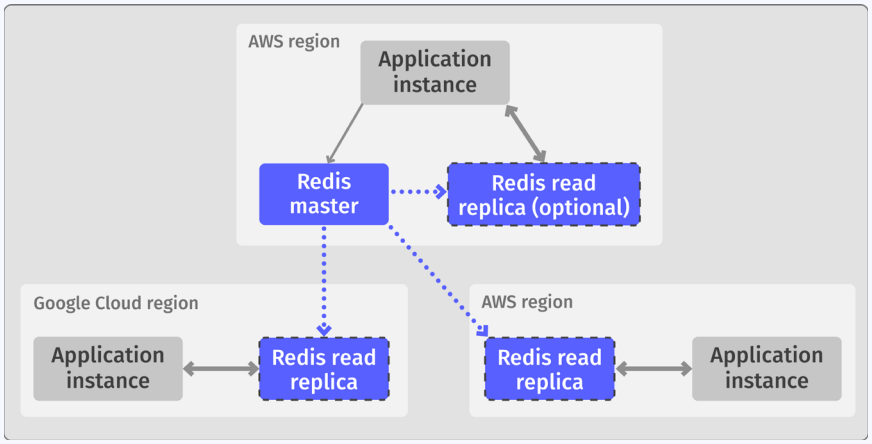

Figure 8-4. Multi-region Active-Active replication with Redis Enterprise

In this model, both reads and writes can be processed from any of the Redis master instances in any region or with any provider. This improves application performance dramatically. After a write occurs in a given region, it is automatically replicated to all the other masters in the cluster. To take advantage of this type of cluster replication, you must use Redis Enterprise.

For cloud-hosted databases, that means you must either use Redis Enterprise Cloud instances, or you must roll your own self-hosted Redis instances on cloud compute instances or container images. None of the major cloud providers offer multi-region Active-Active deployments natively, nor do they offer deployments across cloud providers. For this type of large-scale, highly available distributed architecture, you must use Redis Enterprise.

5. Cache Performance

"A cache can improve performance... or ruin it."

Performance is at the crux of any caching strategy. Whether caching for one small application or across a global enterprise, a flawless user experience starts with sound performance. In this chapter, Atkinson dives into the heart of this topic by first highlighting the characteristics of enterprise caching and the steps you must take to optimize performance levels.

Important components of enterprise caching are analyzed in detail and examples are provided to help visualize their functionality. It's a chapter that reins in all of the drivers behind enterprise caching and one that will help you sharpen areas that are lagging.

By implementing these principles, your cache will be optimized for scaling and will give your application the launchpad needed to operate on a global level.

"Chapter 9: Cache Performance" from *Caching at Scale with Redis* by Lee Atchison

Caches are useful only if they are effective. What do we mean by effective? Well, it depends on why you are using a cache. A cache is most commonly used to decrease the number of long or expensive operations that need to be performed, increasing the speed or efficiency of request processing.

Consider the multiplication example described in Chapter 2, "Why Caching?"In that section, we described a simple multiplication service with a cache in front of it. Whenever a request is made to perform a multiplication, first the cache is consulted to see if the request has already been processed. If not, the multiplication service is called, and the result is returned and stored in the cache for future use. That way, the next time the same request is made, the result is already stored in the cache.

How does the cache improve performance? The diagram in Figure 9-1 shows the same cached multiplication service introduced in Chapter 2. This version, though, shows how much time it takes to retrieve a value from the cache (hypothetically, 1 millisecond), compared with having the multiplication service calculate the result (25ms). Put another way, the first time the request is made, the multiplication service has to be consulted, so the entire operation takes approximately 25ms. But each subsequent equivalent operation can be performed by retrieving the value from the cache, which takes only 1ms, in our example. The result is improved performance for cached operations.

This is how a cache improves performance. Notice that talking to and manipulating the cache also takes time. So requests that must call the service also have to first check the cache and add a result to the cache. This additional effort is called the **cache overhead**.

5. Cache Performance

In a cache-aside strategy, the cache overhead includes the cache check time (to see if the result was previously in the cache) and the cache write time (to store the newly calculated result in the cache). Requests that end up having to call the service anyway take more effort than simply calling the service. Put in cache terms, a cache miss (that is, a request that cannot be satisfied by results stored in the cache) incur additional overhead compared to just calling the service. By corollary, a cache hit (that is, a request that can be satisfied by results stored in the cache) are satisfied significantly faster using only the results stored in the cache.

The total time a request takes to process as a result of a cache miss is:

```
Request_Time = Cache_Check + Service_Call_Time + Cache_Write
```

In our multiplication service, let's make the following assumptions:

```
Cache_Check = 1ms # How long to get a value from the cache
Cache_Write = 1ms # How long to write a new value into the cache
Service_Call_Time = 25ms # How long service takes to process
multiplication request.
```

Therefore, the `Request_Time` for our multiplication service, when we have a cache miss is:

```
Request_Time = 1ms + 25ms + 1ms
Request_Time = 27ms # For a cache miss
```

Notice that this total time is greater than the time it takes for the service to process the request if there was no cache (25ms). The additional 2ms is the cache overhead.

But requests that can be fulfilled by simply reading the cache take less effort, because they do not have to make any calls to the service. Put in cache terms, requests that cache hit take significantly less time by avoiding the service call time.

The total time a request takes to process as a result of a cache hit is:

```
Request_Time = Cache_Check
```

Therefore, the `Request_Time` for our multiplication service, when we have a cache hit is:

```
Request_Time = 1ms
```

So, some requests take significantly less time (1ms in our example), while other requests incur additional overhead (2ms in our example). Without a cache, all requests would take about the same amount of time (25ms in our example).

In order for a cache to be effective, the overall time for all requests must be less than the overall time if the cache didn't exist.

5. Cache Performance

This means, essentially, that there needs to be more cache hits than cache misses overall. How many more depends on the amount of time spent processing the cache (the **cache overhead**) and the amount of time it takes to process a request to the service (**service call time**).

The greater the number of cache hits compared with the number of cache misses, the more effective the cache. Additionally, the greater the service call time compared with the cache overhead, the more effective the cache.

Let's look at this in more detail. First, we need to introduce two more terms. The **cache miss rate** is the percentage of requests that generate a cache miss.

Conversely, the **cache hit rate** is the percent of requests that generate a cache hit. Because each request must either be a cache hit or cache miss, that means:

```
Cache_Miss_Rate + Cache_Hit_Rate = 1 (100%)
```

Now, let's use these rates to determine the efficiency of our service's cache.

When using our multiplication service without a cache, each request takes 25ms. With a cache, the time is either 1ms or 27ms, depending on whether there was a cache hit or cache miss. In order for the cache to be effective, the 2ms overhead of accessing the cache during a cache miss must be offset by some number of cache hits. Put another way, the total request time without a cache must be greater than the total request time with a cache for the cache to be considered effective. Therefore, in order for the cache to be effective:

```
Request_Time_No_Cache >= Request_Time_With_Cache
Request_Time_With_Cache =
( Cache_Miss_Rate * Request_Time_Cache_Miss ) +
( Cache_Hit_Rate * Request_Time_Cache_Hit )
```

And since:

```
Cache_Hit_Rate = 1 - Cache_Miss_Rate
```

You can rewrite this as:

```
Request_Time_With_Cache =
( Cache_Miss_Rate * Request_Time_Cache_Miss ) +
( (1 - Cache_Miss_Rate) * Request_Time_Cache_Hit )
```

Therefore:

```
Request_Time_No_Cache >= Cache_Miss_Rate * Request_
Time_Cache_Miss + (1 - Cache_Miss_Rate) * Request_Time_
Cache_Hit
```

5. Cache Performance

For our multiplication example, that means:

```
25ms >= Cache_Miss_Rate * 27ms + (1-Cache_Miss_Rate) * 1ms
25ms >= (Cache_Miss_Rate * 27ms) + 1ms – (Cache_Miss_Rate * 1ms)
25ms >= 1ms + Cache_Miss_Rate * 26ms
24ms >= Cache_Miss_Rate * 26ms
Cache_Miss_Rate <= 24/26
Cache_Miss_Rate <= 92.3%
```

Given that cache hit rate + cache miss rate = 1, we can do the same calculation using the cache hit rate rather than the cache miss rate:

```
25ms >= (1 - Cache_Hit_Rate) * 27ms + Cache_Hit_Rate *1ms
25ms >= 27ms - Cache_Hit_Rate * 27ms + Cache_Hit_Rate *1ms
25ms >= 27ms - Cache_Hit_Rate * 26ms
2ms <= Cache_Hit_Rate * 26ms
Cache_Hit_Rate >= 2/26
Cache_Hit_Rate >= 7.7%
```

In other words, in this example, as long as a request can be satisfied by the cache (cache hit rate) at least 7.7% of the time, then having the cache is more efficient than not having the cache.

Doing the math the other way, you could ask a different question. If the average request time is 25ms without a cache, what would be the average request time if the cache hit rate was 25%? 50%? 75%? 90%?

For our multiplication service, we use this equation:

```
(1-Cache_Hit_Rate) * 27ms + Cache_Hit_Rate * 1ms
```

For cache hit rate of 25%:

```
(1 - 0.25) * 27ms + 0.25 * 1ms
0.75 * 27 + 0.25 * 1
20.25 + 0.25
20.5ms
```

The average request time assuming a cache hit rate of 25% is 20.5ms. Much faster than the 25ms for no cache!

But it gets better, using our other cache hit rate assumptions:

```
Cache_Hit_Rate = 50%:
(1 - 0.5) * 27ms + 0.5 * 1ms
0.5 * 27 + 0.5 * 1
13.5 + 0.5
= 14ms
```

5. Cache Performance

```
Cache_Hit_Rate = 75%:
(1 - 0.75) * 27ms + 0.75 * 1ms
0.25 * 27 + 0.75 * 1
6.75 + 0.75
= 7.5ms

Cache_Hit_Rate = 90%:
(1 - 0.9) * 27ms + 0.9 * 1ms
0.1 * 27 + 0.9 * 1
2.7 + .9
= 3.6ms
```

As the cache hit rate increases, the average request time improves dramatically. So if the cache hit rate increases from 25% to 90%, the average request time drops from 20.5ms to 3.6ms—86% lower than without a cache (3.6ms compared with 25ms)!

In other words, the higher the cache hit rate, the more effective the cache.

These calculations are all based on the amount of time it takes for the request to be processed by the multiplication service without the cache (25ms in our example). But this value is just an assumption. What happens if that value is larger, say 500ms?

```
Cache_Hit_Rate = 25%:
(1 - 0.25) * 502ms + 0.25 * 1ms
0.75 * 502 + 0.25 * 1
376.5 + 0.25
= 376.75ms

Cache_Hit_Rate = 50%:
(1 - 0.5) * 502ms + 0.5 * 1ms
0.5 * 502 + 0.5 * 1
251 + 0.5
= 251.5ms

Cache_Hit_Rate = 75%:
(1 - 0.75) * 502ms + 0.75 * 1ms
0.25 * 502 + 0.75 * 1
125.5 + 0.75
= 126.25ms
```

5. Cache Performance

```
Cache_Hit_Rate = 90%:
(1 - 0.9) * 502ms + 0.9 * 1ms
0.1 * 502 + 0.9 * 1
50.2 + .9
= 51.1ms
```

We can see that for a service that takes more resources and has a larger request time without a cache, the impact of the cache on the request time becomes much greater. In particular, at a cache hit rate of 90%, the average request time is 89.8% better than without a cache (51.1ms compared with 500ms).

In other words, the greater the cost of calling the un-cached service, the greater the effectiveness of the cache for a given cache hit rate.

These calculations can and should be performed on each caching opportunity to determine whether or not—and to what extent—the application can effectively utilize a cache.

6. Criteria for Evaluating Enterprise Cache

Caching at scale is no small feat. Enterprise-grade caches alleviate much of the risk and operational burden of effectively caching at scale. But choosing the best enterprise-grade cache for your application can be tricky. It requires evaluating different t caching services and their capabilities, features, and components. How do you find the right solution for your applications?

To clear the fog, we've created a checklist to help you to identify the most optimal enterprise-grade caches available on the market:

- Provides consistent high throughput and sub-millisecond latency
- Highly available (offers 5-9s availability)
- Globally available with local low-latency
- Has automatic failover and failback
- Linearly scales to optimize infrastructure resources
- Scales with no performance degradation
- Backup with no performance degradation
- Provide intelligent tiering to manage large data sets economically
- Multi-tenancy to enable 100% asset utilization
- Run on-premises, in a hybrid environment, and multiple clouds without data transfer issues
- Enterprise-gradle support
- Offers the innovation of an open source foundation and community

6. Criteria for Evaluating Enterprise Cache

Basic Caching vs. Enterprise Caching

Below is a chart that will help you differentiate between attributes of a basic and an enterprise-gradle cache. As you'll discover, there's a range of features you need to take into consideration when looking for an enterprise-proven cache.

	Basic Caching	Enterprise-Grade Caching
High throughput	✕	✓
Low latency	✓	✓
Cloud DBaaS	✓	✓
Hybrid deployment	✕	✓
Multi-cloud deployment	✕	✓
5-9s' High availability	✕	✓
Geo-distribution	✕	✓
Guaranteed local low-latency	✕	✓
Data consistency with no performance degradation	✕	✓
Linear scaling	✓	✓
Infinite scaling	✕	✓
Basic clustering	✓	✓
Advanced clustering	✕	✓
Intelligent tiering	✕*	✓
Multi-tenancy	✕	✓
Geo-distribution	✕	✓
Eventual consistency	✕	✓
RBAC support	✕	✓
Support for Kubernetes as a native service (AKS, GKE, EKS, OpenShift)	✕	✓
Upgrade with no downtime	✕	✓
Auto failure detection	✕	✓
Enterprise-grade support	✕	✓

** Partial for AWS*

6. Criteria for Evaluating Enterprise Cache

Real-time Enterprise Caching with Redis

To guarantee real-time data and stay afloat in today's competitive digital environment, applications must be powered by an enterprise-grade cache. But this can be tricky because for a cache to operate on the enterprise level, it needs to meet a long list of criteria.

As a result, companies all over the world have turned to Redis Enterprise to supercharge application performance, maximize engagement, and power the world's best digital experiences. Redis Enterprise is loved by over 8,000 customers because: it's powerful, it's consistent, and it can be run anywhere.

Below are some of the technical features that allow Redis Enterprise to scale and operate on an enterprise level:

- **High availability:** Redis Enterprise provides 5-9s SLAs across multiple geographies or clouds. This includes a number of features, including backups and automated cluster recovery – both of which are crucial for business critical apps.
- **Global Distribution:** Active-Active Geo Replication enables globally distributed applications that guarantee sub-millisecond local latency across the globe with data consistency.
- **Scalability:** Redis Enterprise easily handles usage spikes with automatic dynamic scaling that works behind the scenes to perform consistently and without error.
- **Performance:** Redis Enterprise provides high-throughput and sub-millisecond performance to eliminate the possibility of latency. Users are guaranteed real-time responsiveness which is what ultimately drives engagement.
- **Multi-cloud & hybrid:** The flexibility that Redis Enterprise provides allows you to choose between a range of different deployment options to ensure that you find the right fit for your business and applications

About Lee Atchison

Lee Atchison is an established thought leader in cloud computing and application. Having dedicated over thirty years to working in product development, architecting, modernizing, and scaling, Lee has worked for some of the world's biggest heavyweights, including Amazon and Amazon Web Services.

To discover more about Lee and some of his other recent books, visit his website at leeatchison.com and make sure to follow him on Twitter.

Atchison is widely referenced in many publications and is invited to share his insights at public speaking events all over the world.

About Redis

Data is the lifeline of every business, and Redis helps organizations reimagine how fast they can process, analyze, make predictions, and take action on the data they generate. Redis provides a competitive edge to any business by delivering open source and enterprise-grade data platforms to power applications that drive real-time experiences at any scale.

Developers rely on Redis to build performance, scalability, reliability, and security into their applications. Born in the cloud-native era, Redis uniquely enables users to unify data across multi-cloud, hybrid, and global applications to maximize business potential. Learn how Redis can give you this edge at redis.com.

Final Thoughts

Caching at scale is a requirement for modern digital businesses. Without enterprise caching, growth is a challenge and applications will be hampered by slow performance and sluggish databases that will buckle under today's data demands.

As Atchison highlighted, augmenting your database to scale at an enterprise level can be complex. There's a long list of criteria that a cache must meet to be able to process millions of queries simultaneously at any given time.

And it's not just about optimizing performance for speed—it's consistency too. Having the agility to respond to unforeseen and sporadic surges in traffic is a prerequisite to keeping everything in real time, all the time.

But finding the right enterprise-proven cache that ticks all the boxes can be difficult. And trying to find the one that's the best fit for your business can also feel challenging.

So what's next?

Based on what we've covered, you'll have some insight into the advanced capabilities of Redis Enterprise and why it's the world's favorite cache.

If you want to understand more about Redis Enterprise and its powerful caching capabilities, then make sure to download *The Buyer's Guide to Enterprise Caching*.

You can also discover more about Redis Enterprise at https://redis.com/.