Caching as a Best Practice for Microservices-Based Applications

Table of Contents

Abstract

Microservices-based applications have been a game changer. This approach to building applications comes with its own set of tools, technologies and best practices. This paper is focused on how in-memory caching, like Amazon ElastiCache, has an important role in microservices-based applications.

The topics covered in this paper include:

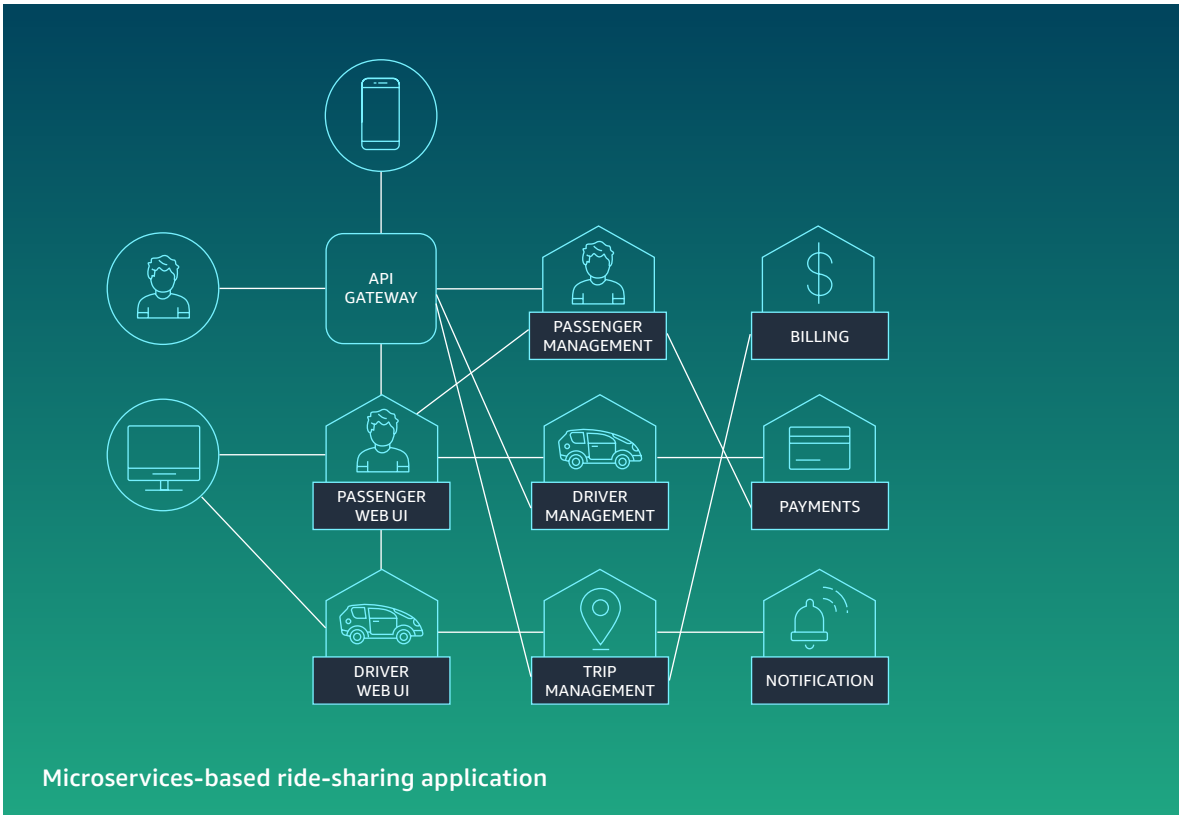
- Microservices-based applications have compelling advantages that are now indispensable. They are a great fit for containerization and for use with AWS cloud infrastructure and container orchestration systems like Kubernetes.
- Microservices introduce network latency. For the overall response-time of the application to be acceptable, other areas of the architecture need to be optimized for performance, placing demands on the latency of each microservice.
- The key to managing reduced microservice latency budgets is to reduce reliance on a backend, high-latency database as much as possible.
- The importance of high availability and how ElastiCache supports this.
- How storing a user's session data in ElastiCache enhances the user's experience.
- How ElastiCache can be used as a message broker or an event store for inter-microservice communications.

In this whitepaper, we examine the role of the data layer, and how the use of Amazon ElastiCache as an in-memory cache can be a critical infrastructure component in support of microservices architectures.

The Rise of Microservices

Microservices-based architectures are now a well-established best practice for building web-scale applications.

> ***The complexities of web-scale applications have made traditional monolithic approaches unsustainable.***



Microservices-based ride-sharing application

With microservices, applications are broken down into smaller, isolated services that are accessed via their APIs.

As shown in the preceding graphic, consider the microservices that constitute a ride-sharing app. The eight microservices are processes that communicate with each other over the network in order to fulfill a goal. In short, the microservice architectural style is an approach to developing a single application as a suite of smaller services. Each microservice runs in its own process and communicates with lightweight mechanisms, often an HTTP resource API. These microservices are built around business capabilities and independently deployable by fully automated deployment machinery.

There is a bare minimum of centralized management of these microservices, which may be written in different programming languages and use different styles of databases.

The API gateway is the single-entry point for all customers. It handles requests by fanning out to multiple services. Rather than providing a one-size-fits-all style API, the API gateway can expose a different API for each client. The API gateway might also implement security, such as authenticate the client (AuthN) and verify that the client is authorized to perform the request (AuthZ).

The Benefits of Microservices

- **Release velocity:** Since microservices are isolated services with minimal interdependence, each microservice team can release their service on their own schedule. This loose coupling is the key enabler of an overall time-to-market acceleration and innovation benefit because team interdependence is a recipe for slow progress. Microservices support incremental, rapid release of software, characteristic of agile DevOps methodologies, and reduce the risk of failure inherent in the "all-or-nothing" monolithic approach.

- **Best fit technology choices for optimized performance:** Working autonomously, each team can make their own best fit technology choices.

> ***At the data layer, each team is empowered to choose purpose-built backend databases that are best suited for their workloads, a practice known as polyglot persistence.***

How the data is modeled is also an autonomous decision, local to each microservice.

- **Simpler and easier to maintain:** Complex monolithic codebases are difficult to maintain. Tightly interwoven code is more brittle. So, even minor code changes may have a broad impact on the entire application needing to go through a lengthy process of regression testing, user acceptance testing, and performance testing.

- **Easier to scale:** Microservices can be scaled independently based on need. This granular scaling approach is more efficient and effective than scaling an entire monolithic application. Microservices frequently run inside of containers, making them fast to start up or shut down, with their lifecycles automatically managed by container orchestration systems.

Microservices are a natural fit for cloud infrastructures that provide dynamic, on-demand capacity.

- **More reliable:** The ability to dynamically add instances of containerized microservices also provides an elegant solution for high availability. If an instance fails, another instance of the same microservice can pick up the load. Whereas, monolithic applications cannot granularly handle component failures.

- **Deeper functionality:** Development teams can focus on a scope that is bounded by a business function which results in more optimized, hardened code with deeper functionality. It is easier for each team to develop expertise on the business function they are automating.

Microservices are often deployed as Kubernetes (K8s) containers, with one or more microservice per container. As containers, K8s can manage the lifecycle of microservices by starting, stopping, and scaling containers as needed. K8s gives us an elegant approach to scaling stateless workloads.

> ***Microservices mark the convergence of an evolution comprised of several industry developments and iterations.***

With all of these high-impact benefits, it's no surprise that microservices have now emerged as the way forward.

Data Fragmentation Challenges with Microservices

Despite the benefits of a microservices architecture, the distributed nature of microservices introduces challenges related to the fragmentation of data and network latency. In many cases, the application logic has to invoke multiple microservices to retrieve data. One service might send a task to another, which must then query its database to get the result before the first service can finish up.

These cascading service calls can quickly become problematic, creating a snowball effect that increases latency and slows down overall performance.

Consider the following examples of cascading microservices:

Example 1: What is my insurance co-pay for a medical procedure?
The responding microservice would first look up the patient's policy by sending a request to the patient microservice. It would then have to look up the details of the policy by sending a request to the policy microservice. Other required lookups include the coverage, provider, and procedure. All of this would have to be composed to answer the co-pay query. Optimizing lookup speeds pays dividends because of the sheer frequency with which lookups occur.

Example 2: What is the status of my restaurant order?
The orders microservice receives this request and serves as an aggregator by calling other microservices to pull order status information together. The kitchen microservice returns the status of the order from the restaurant's perspective and the estimated time it will be ready for pickup. The delivery microservice returns delivery status, estimated delivery information, and its current location. The accounting microservice returns the order's payment status. All of these microservices would need to respond to fulfill the user's query.

The overall application's service-level agreement (SLA) on latency puts inordinate pressure on microservices, with each microservice being granted only a fraction of the latency budget of the overall application.

In the remainder of this white paper, we describe how Amazon ElastiCache for Redis is a great way to boost the performance of microservices based applications. We start by describing the service, focusing on the main features that are well aligned with the needs of microservices, followed by common patterns and use cases.

Amazon ElastiCache for Microservices

The ElastiCache in-memory caching service is easy to deploy, operate, and scale in the cloud. It is designed for improving the performance of applications by retrieving information from fast, fully managed, in-memory caches instead of relying on slower disk-based databases.

ElastiCache offers two in-memory engines that are compatible with popular open-source technologies, Redis and Memcached. ElastiCache for Redis is a Redis-compatible in-memory service that delivers the ease-of-use and power of Redis along with the availability, reliability, and performance suitable for the most demanding applications. ElastiCache for Memcached is a Memcached-compatible caching service that works seamlessly with popular tools that you use with existing Memcached environments. Both engines are widely used key-value stores, and you can choose which engine you want to use depending on your specific needs.

ElastiCache clusters can scale up to 500 nodes, up to 340 TB of in-memory data, and up to 1 PB of total data using the data tiering feature. Data tiering provides a new price-performance option for Redis workloads by using lower-cost solid state drives (SSDs) in each cluster node, in addition to storing data in memory.

ElastiCache as a Fully Managed Service

ElastiCache is a fully managed service, making it easier to deploy, operate, and scale. AWS takes care of the maintenance and management tasks for you. This includes patching, backups, and failure recovery. You can simply create a cache cluster, and AWS will handle nearly all of the rest.

Automated Capacity Management with ElastiCache

ElastiCache reduces or eliminates capacity management tasks by automating the allocation or de-allocation of capacity based on changing workloads. ElastiCache has two deployment options: serverless caching and self-designed clusters.

Serverless caching

Serverless caching simplifies cache creation and instantly scales to support your most demanding applications. With ElastiCache Serverless, you can create a highly-available and scalable cache in less than a minute, eliminating the need to provision, plan for, and manage cache cluster capacity. Serverless caching automatically scales both vertically and horizontally without any capacity management. Clients connect to a single endpoint. Serverless caching is the easiest way to get started with a cache when you have unpredictable application traffic or when you are creating a new cache for new or unknown workloads.

Self-designed clusters

For more fine-grained control over your ElastiCache for Redis cluster, you can choose to design your own Redis cluster with ElastiCache. Self-designed clusters give you the flexibility to initially choose and later change the node-type, number of nodes, and node placement across AWS Availability Zones for your cluster. With the Auto-Scaling feature you can also configure scaling based on a schedule, or scale based on metrics like CPU and Memory usage on the cache. Self-designed clusters are good option when you don't expect your application traffic to fluctuate much or you can accurately predict your application traffic peaks and troughs. The ability to forecast your capacity requirements enables you to choose reserved node pricing which provides a significant discount off the ongoing hourly usage rate for the node(s) you reserve in one-year or three-year terms.

Self-designed clusters are also a better option if you need to use ElastiCache Global Datastore for low-latency reads and disaster recovery across AWS Regions. Also, if you want to use ElastiCache with Kubernetes, you can manage a self-designed cluster as an external AWS managed resource directly from Kubernetes. (see Bridging Kubernetes Workloads to ElastiCache). Self-designed cluster also support data tiering, which provides price-performance option for Redis workloads by utilizing lower-cost solid state drives (SSDs) in each cluster node in addition to storing data in memory.

ElastiCache as a Remote Cache

A number of caching topologies can be deployed across your IT infrastructure. These topologies include database-integrated caches, local caches, and remote caches.

ElastiCache is a remote cache. As such, it is separate and decoupled from any application or database instances, and it is dedicated to storing and sharing the cached data in-memory.

> ***Remote caches are a good fit for microservices-based applications because the separation of the cache server from the microservice provides the flexibility to independently scale up the cache service.***

For a full discussion of caching topologies and their pros and cons, see Appendix A: Caching Topologies and Their Pros and Cons.

High Availability with ElastiCache

Distributed architectures have dozens of application and database instances that all communicate over a network. That means maintaining acceptable levels of availability with distributed architectures can be more challenging than with monolithic architectures. The reliance on a network increases the importance of network availability in the equation.

> ***The increase in the number of components that comprise an application increases the likelihood of failure.***

Developers need to write extensive error or exception-handling code. Not only do they need to detect microservice failures, but they also need to restart your microservice.

In addition to the microservice code, the microservice data and state also need resilience. As microservices are stopped and started, the state of a stopped microservice is needed for a new microservice instance to recover successfully.

Amazon ElastiCache for Redis provides high availability, with automated failover and recovery.

> ***When a primary node fails, ElastiCache detects the failure and promotes a replica node to be the new primary. This process usually takes less than 30 seconds.***

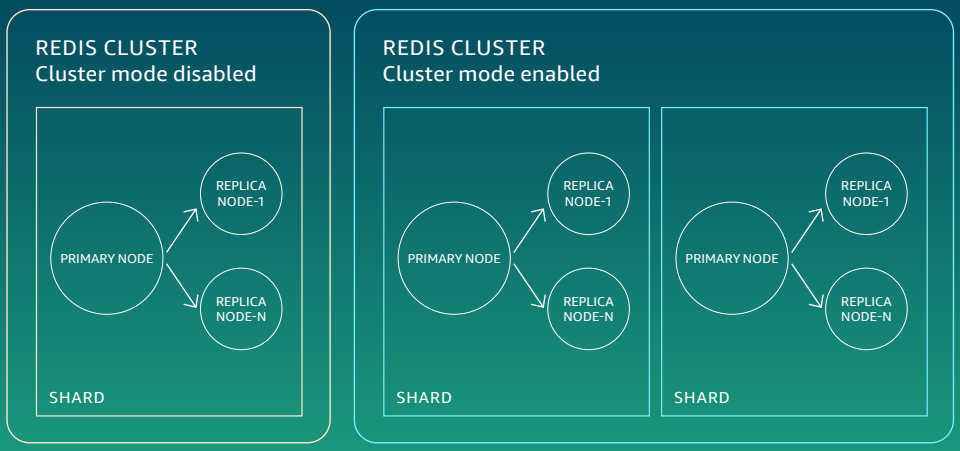
The failed node is then replaced and returned to the cluster as a replica node. For multi-AZ deployments, ElastiCache for Redis provides a 99.99% SLA for high availability.

The Performance-Durability Tradeoff

Like other caches, ElastiCache is an ephemeral cache. As such, it's susceptible to data loss in the event of a cache failure. The high-availability feature in ElastiCache mitigates this exposure considerably. ElastiCache asynchronously replicates data from the primary instance to read replicas. If the primary instance of an ElastiCache fails, ElastiCache will detect the failure and promote the least lagging read replica to the role of a new primary. Data loss is limited to the changes that were 'in-flight' just before the failure. These are the changes that were made in the primary before the failure, but the failure prevented the outbound replication of these changes.

If zero data loss is a requirement, then Amazon MemoryDB for Redis, which is a durable in-memory database, is a better fit. When an application writes to MemoryDB for Redis, the write is not acknowledged until it is synchronously written into both the memory layer as well as a durable multi-AZ transaction log.

For read performance, both ElastiCache and MemoryDB provide ultra-fast microseconds response time and throughput improvements over disk storage. For write performance, ElastiCache takes microseconds, whereas MemoryDB can take a few milliseconds.



Sharding and read replicas with and without cluster mode

Sharding and Read Replicas for Improving Read and Write Performance

Sharding is a technique that splits up a large data set by partitioning data along a keyspace. A keyspace refers to the full range of a key that is partitioned and distributed across different shards. This results in data segments (shards) that cover a bounded range within the keyspace. By spreading shards across multiple instances of ElastiCache, each instance can independently operate in parallel. Since shards are mutually exclusive, each shard can be updated without any update collisions with other shards. Reads also run faster because of the power of parallelism. For further acceleration of reads within each shard, read replicas can be created by replicating the shard's primary instance.

> ***Shards and replicas can be added or removed online, while the cluster continues serving incoming requests.***

Each node in a cluster has the same compute, storage, and memory specifications. The ElastiCache API lets you control cluster-wide attributes, such as the number of nodes, security settings, and system maintenance windows.

A Redis cluster, with cluster mode disabled, will never have more than one shard. With cluster mode enabled, you can create a cluster with a high number of shards and a low number of replicas totaling up to 500 nodes per cluster. For example, you can choose to configure a 500-node cluster that ranges between 83 shards (one primary and 5 replicas per shard) and 500 shards (single primary and no replicas).

Re-sharding is an online process that allows scaling in/out while the cluster continues serving incoming requests. Autoscaling automatically adds shards and replicas to your cluster when traffic spikes and your cluster is under load.

Conversely, your cluster can automatically scale-in when traffic subsides. Using pre-defined metrics is a fast and easy way to define autoscaling policies. Alternatively, you can use Amazon CloudWatch for crafting custom metrics.

You can scale on a set schedule. This is useful when your workload is predictable and you have a solid understanding of what your cluster capacity needs are as a function of time.

With ElastiCache Serverless, scaling is handled transparently by the service using a combination of vertical and horizontal scaling for variable workloads. With a serverless cache, you do not need to add or remove shards or replicas, as the cache will scale to handle your dataset size and request rate.

Bridging Kubernetes Workloads to ElastiCache

As an AWS managed service, ElastiCache mirrors Kubernetes (K8s) by providing a similarly fluid ability to scale the cache workload. But how do the worlds of K8s and fully managed services on AWS come together?

> ***Fortunately, you can manage ElastiCache as an external AWS managed resource directly from K8s.***

The key cluster management actions are supported, such as scaling up, down, in, or out. You can also scale the number of read replicas and create other ElastiCache resources, such as snapshots, parameter groups, and subnet groups. Managing ElastiCache in a K8s world relies on a controller that extends K8s. Controllers use the K8s API to control the lifecycle of custom resources that are not native to K8s, like databases and caches.

> ***By using a controller for ElastiCache, the management of ElastiCache can be automated, much like a native K8s resource.***

AWS Controller for Kubernetes (ACK) for ElastiCache adopts the approach of managing ElastiCache as an external resource. With ACK, you can use AWS managed services for your K8s applications without needing to define resources outside of the K8s cluster. Or you can run services that provide supporting capabilities like databases, caches, or message queues within the K8s cluster. Each ACK service controller manages resources for a particular AWS service, and is packaged into a separate container image that is published in a public repository.

> ***Developers can leverage their knowledge of the K8s resource model for working with ElastiCache, just like any other K8s resource.***

ACK enables K8s users to describe the desired state of AWS resources using the K8s API and configuration language. ACK resources are defined using YAML-formatted manifest files to both initially define the resource configuration and also to modify it. Once the manifest file is created, the resource it defines is initially created by using the file name as the input argument to the Kubernetes "kubectl apply" command. To change a resource configuration, you simply edit the appropriate parameters in the existing resource manifest file, then call the "kubectl apply" command in the same manner as the initial resource creation.

Also, ACK is declarative, so you can define the desired state and allow the controller to take the necessary steps without defining an imperative list of steps. The K8s control loop manages the state of your cluster as well the configuration you passed in for your AWS resource. Periodically, an ACK service controller will look for any drift and attempt to remediate.

> ***A single consolidated approach using ACK makes it easier to adopt a GitOps based approach to automating your deployments.***

ACK for Amazon ElastiCache itself runs in a container on any K8s distribution on-premise or in the cloud. Hence it is not limited to Amazon Elastic Kubernetes Service (Amazon EKS).

ElastiCache for Cost Optimization of Performance Improvements

Developers and architects often try to address performance requirements by scaling the database used by each microservice. This involves tuning, indexing and applying various performance optimization techniques. Performance optimizations of backend databases require increasingly more effort and become increasingly more expensive. In the case of legacy databases that were not built for the cloud, the only option often available is to vertically scale the database. This is usually extremely costly and disruptive.

> ***Ultimately for any disk-based database, the performance of the database hits a limit because of the physics of retrieving data from disk.***

The use of ElastiCache for achieving performance targets is a simpler and more cost-effective approach. ElastiCache for Redis supports a throughput of 1 million requests per second per node or 500 million requests per second per cluster with microsecond response time. With ElastiCache serverless caches, you only pay for the resources you use, including the data stored and requests made against the cache. You do not need to provision capacity. Alternatively, with self-designed clusters you have full flexibility and control over horizontal scaling using sharding and read replicas (as previously discussed) and vertical scaling by changing the node type to resize the cluster. Vertical scaling allows scaling up/down online while the cluster continues serving incoming requests.

> ***The ease with which ElastiCache scales, both vertically and horizontally, results in a more cost-effective approach to achieving performance at scale than trying to scale backend databases, many of which were not designed for this purpose.***

Low-Latency Access to Operational Data

Why Low-Latency Access to Operational Data is Critical

Whether serving the latest news, displaying items in a product catalog, or selling tickets to an event, speed is of paramount importance. For example, a typical web application opening page is comprised of dozens of calls to several microservices to collect the data needed for the display. This can include calls to microservices for user management, billing, subscription management, personalized recommendations, and more. A large number of users can hit the opening page concurrently placing demands on the throughput of the application.

> ***Supporting high throughput with low-latency responses is critical to achieve customer satisfaction.***
>
> ***The New York Times found that users can register a 250-millisecond (1/4 second) difference between competing sites. Users tend to opt out of the slower site in favor of the faster site.***

Tests done at Amazon revealed that for every 100 ms (1/10 second) increase in load time, sales decrease by 1%.

Today's enterprises need to be more agile to improve their customers' experience, respond to changes quickly, and have the freedom to innovate. As part of their digital transformation journey in the cloud, these businesses need to scale quickly to potentially millions of users, have global availability, and provide ultra-fast response times for customers.

Addressing the Impact of Exploding Data Volumes

Increasing data volumes exacerbates the performance bottlenecks of disk-based backend databases, even more so with relational databases. When the database management system has to sift through terabytes of data, the response time is inadequate for many modern applications.

Data volumes have exploded in the last two decades, and this trend will continue. This growth in data increases the need to distinguish between hot data and cold data so that caching can be applied to hot data. Although the lines are shifting so that more data can be cached in memory, often the sheer volume of data imposes a need to store data based on a tiered approach.

Addressing the Impact of Hot Data

In-memory caching has been a long-standing solution for addressing slow response times from disk-based data stores, but it has often been relegated to data that is most frequently and concurrently accessed, also know as hot data. Segmenting data by the frequency of access gives us the ability to allocate data layer resources based on the 'temperature' continuum—hot data can be stored in ultra-fast in-memory caches, while warm data can be stored in traditional disk or SSD storage, and cold data can be archived in the least costly storage. This approach optimizes the overall cost of storage.

Relegating caches to only hot data has been a common practice as a result of several factors. The cost of memory was prohibitive. Caches were limited in capacity and difficult to scale. Also, the applications of the past didn't have the same response time requirements of today's applications, so relegating caching to hot data was a workable solution.

All of these limiting factors are continually changing in favor of broader adoption of caching.

> ***Memory costs have declined substantially. The ability to scale caches horizontally has reduced capacity limitations. And a growing share of applications now require near-real-time response.***

With these shifts, a re-examination of old assumptions is warranted. The less frequently accessed data still contributes to the overall responsiveness of your application.

> ***It is now reasonable to consider caching much more of your application's data, such as your warm data or perhaps even all of your application data.***

In the case of very high data volumes a tiered approach is still needed. But shifts in technology provide the opportunity to re-draw the lines of your data segmentation strategy.

ElastiCache can significantly lighten the load on the database by servicing most of the requests for data. With lazy loading, only the first instance of a request is subject to the performance of backend databases. With the write-through pattern, each write is subject to the performance of backend databases after which each subsequent read request can be served by ElastiCache. You can cache anything that can be queried where the underlying source of the data could be from relational databases or non-relational databases. You can even cache data that can be accessed through an application programming interface (API).

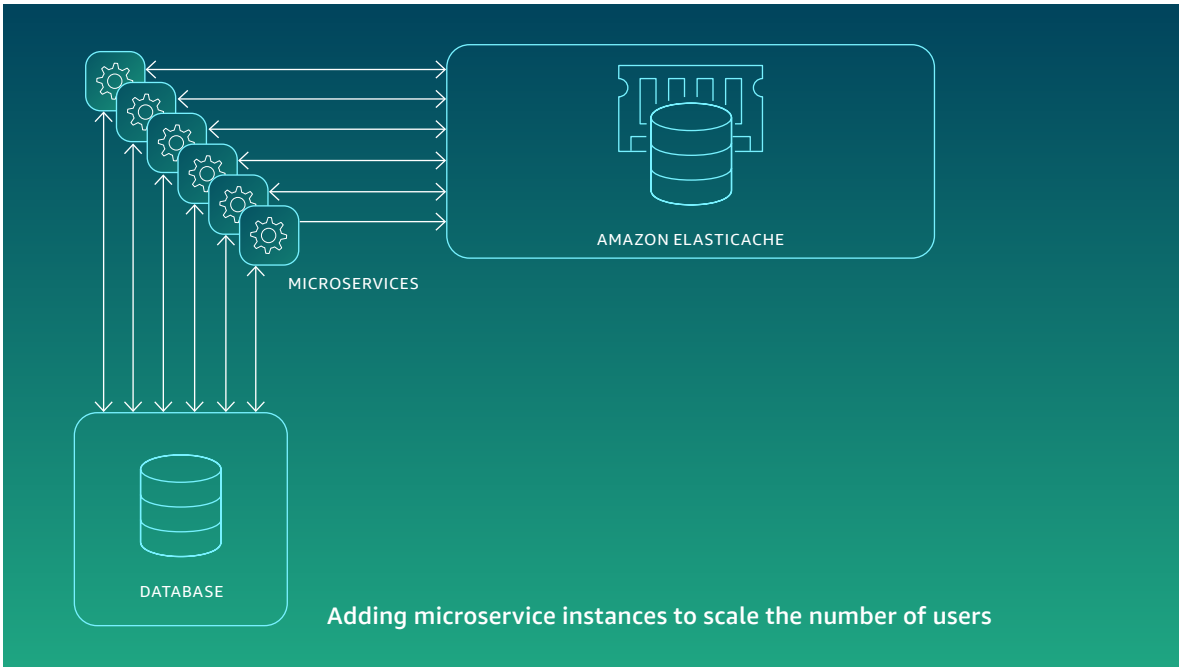
ElastiCache for Complex Queries

Querying a database is always slower and more expensive than locating a key in a key-value pair cache. Some database queries are especially expensive to perform. Examples include queries that involve joins across multiple tables or queries with intensive calculations. By caching such query results, you pay the price of the query only once. Then you can quickly retrieve the data multiple times without having to re-execute the query.

> ***Various types of custom analytics are a common category of use cases because they often involve some data consolidation. A cache can serve as the aggregation layer for this data.***

For example, determining whether there is any relationship between customer satisfaction and your suppliers requires a unified view of data. The customer satisfaction information and the supplier information are likely handled by different microservices. Unifying this information in a cache makes this analysis easier. Other common analytic use case examples include the detection and prevention of security infractions and fraud, and creating consolidated information snapshots to allow for point-in-time queries. Moreover, snapshots can be pieced together to build trend lines of data changes.

Addressing the Need for User Growth and High Throughput



Adding microservice instances to scale the number of users

User growth is a common key performance indicator (KPI) for businesses as applications mature, needing to support up to millions of new customers and expansion into new regions.

> ***Expanding an application's user base can generate higher levels of requests for data. These workloads need to handle high levels of throughput and respond in microseconds while sustaining high performance at scale.***

A common approach to scaling microservices for increasing users is by adding additional microservice instances. Adding instances scales the business logic of the microservice. This is one of the benefits of microservices—you can scale different services independently. Large-scale deployments are likely to require many instances of each microservice, and microservice instances can be added or removed as needed. This level of dynamism is easier if your microservices are designed to be stateless. All forms of data are stored externally in a database, hence no state information is lost when microservice instances are removed.

As the microservice instances increase, the number of times the business logic is executed also increases, with commensurate increases in requests for data. The elasticity and scalability of a microservices architecture is impacted by the throughput constraints of the database. Caching can be used to alleviate this pressure.

Multiple instances of a microservice can access different nodes of an ElastiCache cluster.

> ***The horizontal scalability of ElastiCache makes it well suited for a distributed architecture.***

As a remote cache, ElastiCache presents the application layer with a single view of data accessible by any instance, and updates are available to all microservice instances. Just like multiple instances of a microservice can scale the number of users of the business logic, multiple replicas of data can scale the number of simultaneous reads and writes ElastiCache can process.

Communication Between Microservices Using Event-Based Architectures

Some communication between microservices is often needed, even when they're isolated. Since applications consist of several microservices, the microservices will need to function together as an application. Changes in the state of a given microservice may be of interest to other microservices. Data from one microservice may be needed by another microservice. There are many reasons for microservices to communicate.

Organizations often use specialized integration services to build event-based architectures for inter-microservice communication (Amazon Simple Notification Service, Amazon Managed Streaming for Apache Kafka, Amazon Kinesis, Amazon EventBridge, and more.).

In many organizations, ElastiCache for Redis is also often used for this purpose. The other options provide delivery guarantees and durability. Nonetheless, Redis has broad adoption because of its lightweight ease of use, and it's an attractive choice for organizations that are already familiar with Redis. As a cache, ElastiCache for Redis stores ephemeral data, and for use cases that can tolerate the possibility of a loss of events, the performance of ElastiCache for Redis makes it an attractive choice. For example, if your use case involves analytics on counts or aggregations of events, statistics, and trendlines rather than individual events, then a small loss of events in a large total count of events is immaterial. The loss of events is minimized by ElastiCache's high availability and failover capabilities. If durability of events is of utmost importance, then Amazon MemoryDB for Redis is a better choice. The same performance vs. durability trade-off discussed earlier applies to this use case as well.

There are two approaches to building event-based architectures with ElastiCache for Redis centered around two features—using Redis Pub/Sub or building an event store using Redis Streams.

Using Redis Pub/Sub as an Asynchronous Message Broker

The Redis Pub/Sub server is often used for orchestration of messages. In contrast to point-to-point communication, with the Pub/Sub approach, the message publisher sends messages to channels which serve as a mechanism for organizing messages. To receive messages on a channel, the Redis instance on the receiving side subscribes to the channel. In addition, a subscriber can specify a pattern to subscribe to all messages that match the pattern. Messages are sent and received by the Redis caching layer underneath each microservice, via the Redis Pub/Sub server.

The Pub/Sub mechanism in Redis is inherently asynchronous.

> ***An asynchronous, message-based, event-driven system minimizes the inter-dependence between microservices by making the required communications between them non-intrusive.***

Microservices can produce events without needing to be aware of which microservices are consuming these events and how the events are being handled. For microservices development teams, an event-driven architecture allows each team to focus on their own problem domain.

> ***"Build systems that can evolve. And the best way to make evolvable systems is to focus on event driven architectures."***
>
> Werner Vogels
> VP and CTO at Amazon.com

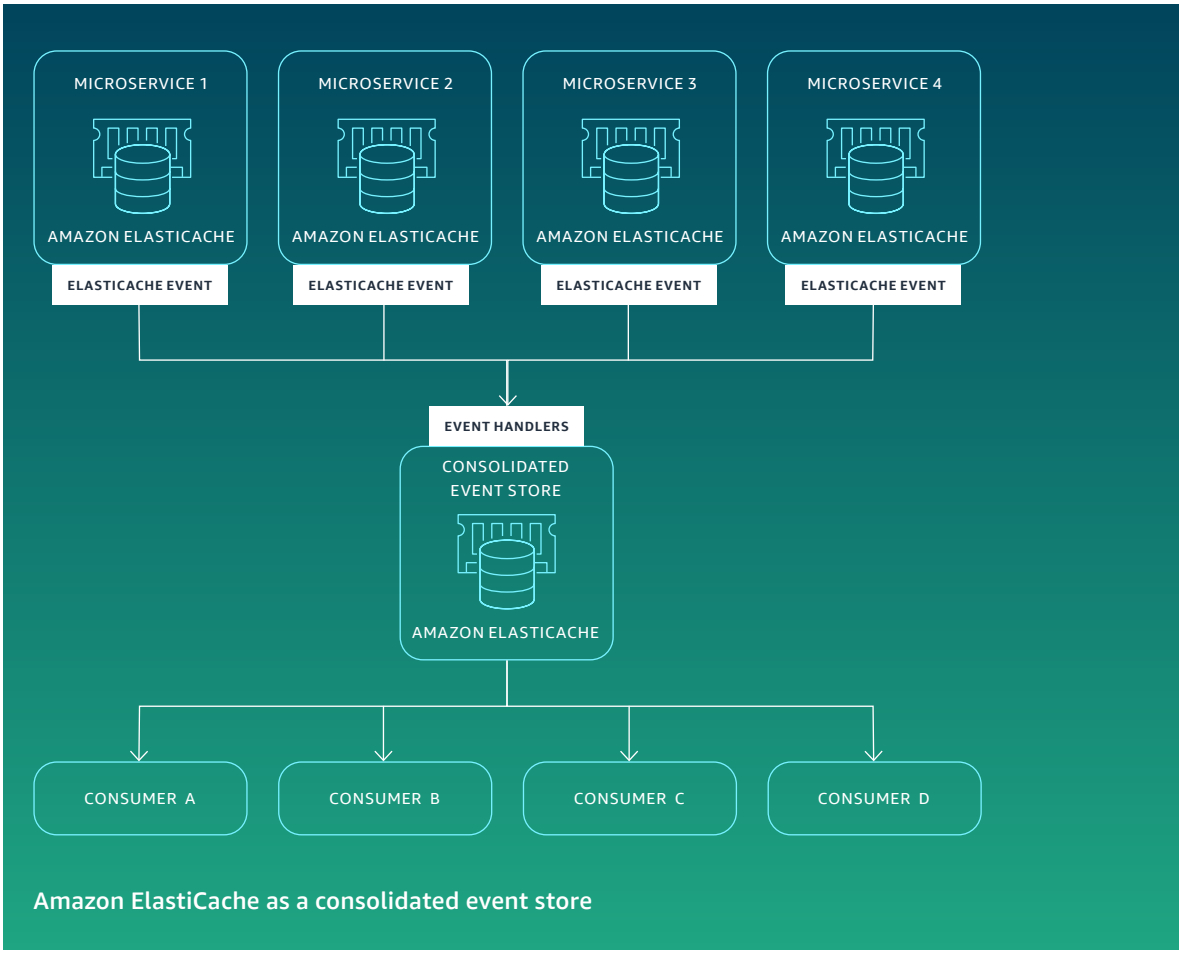
Redis Streams: Sharing Events Across Microservices Using an Event Store

An alternate approach is to build an event store using Redis Streams. With Redis Streams, events are generated directly from the cache without requiring developers to build major pieces of an event-driven system within the application.

> *The event store is a sequential log, and it serves as the destination for the event streams from each microservice. Event streams are propagated from source microservices to the event store.*

Consumers of these events can then subscribe to and read the events of interest. The event store essentially serves as an event source for each consumer. Consumers maintain their own logic related to the filters that will be applied to determine whether an event is of interest. Each consumer also maintains their own pointer/offset into the event store to serially process the events. The consolidation of events across microservices opens up an entire category of use cases—those having to do with cross-domain information that spans microservices.

> *The complete view of all events presented in the unified event log can be used to play back selected events and create a projection of the information in any way desired.*



Amazon ElastiCache as a consolidated event store

Consumers of a stream can be data warehouses like Amazon Redshift for analytics, Amazon OpenSearch for search indexing and searching across microservices, Amazon CloudWatch for application-level monitoring, consolidation for debugging, and many more.

A unified log can have demanding requirements for performance and scalability given the large number of microservices that can source event streams.

> ***ElastiCache for Redis' design for speed, scale, and massive concurrency, together with its Streams data type, makes it an increasingly popular choice as a unified log.***

Redis can support a large number of consumers and retain large amounts of data with very little overhead. Based on their use case, event consumers can generate data projections by replaying the appropriate events in the log. As data flows through the system, it can be validated and enriched.

Projections from the unified event log can be stored in the most appropriate data management technology for the anticipated workload. For example, with analytics use cases, creating projections in Amazon Redshift is a good option. For high-volume concurrent lookups with low-latency responses, another separate instance of ElastiCache can be used as a target. Amazon Neptune can be used if queries are run based on graph-oriented relationships between the entities in the domain model. Amazon Aurora is a good fit for relational workloads. AWS provides a full portfolio of purpose-built databases, and the choice of a database can be based on anticipated workload. Both the underlying data management technology and the data model can be specialized for the use case.

Comparing Redis Pub/Sub and Redis Streams

Redis Pub/Sub is an at-most-once "fire and forget" model using a push protocol, and it cannot protect against network disconnections or lapses in the subscriber's availability. All parties need to be active at the same time to be able to communicate. If a message is published and there are no subscribers listening, the message is lost and cannot be recovered. Redis Streams offers both at-most-once or at-least-once (explicit acknowledgement sent by the receiver) communications. If a subscriber is unavailable when a message is published, it can connect later and read all messages since it last checked.

Pub/Sub is blocking-mode only. Once subscribed to a channel, the client is put into subscriber mode, and it has become read-only. It can only issue limited commands related to subscribing and unsubscribing. Redis Streams allows consumers to read messages in blocking or non-blocking mode. Pub/Sub uses a fan-out mechanism only. All active clients get all messages. Redis Streams allows fan-out, but it also allows you to route different messages to different consumers using consumer groups.

Redis Streams offers a lot more flexibility and reliability than Redis Pub/Sub. Redis Streams provide many more features, like time-stamps, field-value pairs, ranges, and more. On the other hand, Redis Streams consumes a lot more memory capacity. The needs of your use case can guide your choice.

Conclusion

Microservice-based application architectures are the de facto approach to building modern applications. However, the performance of microservices-based applications is highly sensitive to network latency, the weakest link in the performance of distributed systems. Ensuring that the overall application latency meets the needs of modern applications puts pressure on data access latency. ElastiCache for Redis is a horizontally scalable cache that provides ultra-fast in-memory read and write performance, with support for hundreds of millions of operations per second. ElastiCache for Redis is a fully managed, real-time, cost-optimized solution that transforms the performance of microservices-based applications.

ElastiCache for Redis is a good fit for improving the user experience of applications by storing session data. Externalizing session data in ElastiCache for Redis provides resilience from failures of components that impact user sessions. For a more detailed discussion, see Appendix B: Using ElastiCache for Redis as a User Session State Store.

For a collection of microservices to function as a unified application, microservices need mechanisms for communication with each other. ElastiCache for Redis is a popular choice as a message broker or as an event store.

To get started, you can gain free, hands-on experience with ElastiCache - at the free-tier page. AWS also offers an Optimization and Licensing Assessment (OLA) to help you evaluate options for migrating to the cloud. Sign up here so the AWS OLA team can help you. You can also learn more about ElastiCache by heading over to the ElastiCache for Redis page where you'll find related content and additional documentation.

Appendix A: Caching Topologies and Their Pros and Cons

A number of caching topologies can be deployed across your IT infrastructure. The topologies and their pros and cons are as follows.

Database-Integrated Cache

These caches are tightly coupled and built into each database instance. They have built-in write-through capabilities, and can significantly boost database performance. Database integrated caches are much easier to set up. However, integrated caches are typically limited to a single node with memory allocated by the database instance. Therefore, data can't be shared with other instances across nodes. Database-integrated caches are a good choice when the data access pattern has affinity with a database instance.

Database-integrated caches are limited in capacity. For example, on the largest instance supported by Amazon RDS the maximum instance memory size is 3.9 TiB (db.x1e.32xlarge).

Local Cache

These caches are coupled with applications, meaning they are local to each application node and its web server. This makes data retrieval faster than with other caching topologies because it removes network latency that is associated with retrieving data. A major disadvantage of a local cache is that each application instance has its own resident cache working in a disconnected manner.

This creates challenges in a distributed environment where most applications use multiple instances of applications, each on their own application server. If an application instance mostly consumes only the data it creates and the need for data sharing is limited, then a local cache can provide a high cache hit ratio. If the architecture can be set up so that any given user is always routed to the same instance, then the desired locality of reference can be achieved and a local cache can be effective. However, this results in a more complex solution. It also results in a bad user experience when an instance to which that user has affinity is unexpectedly lost.

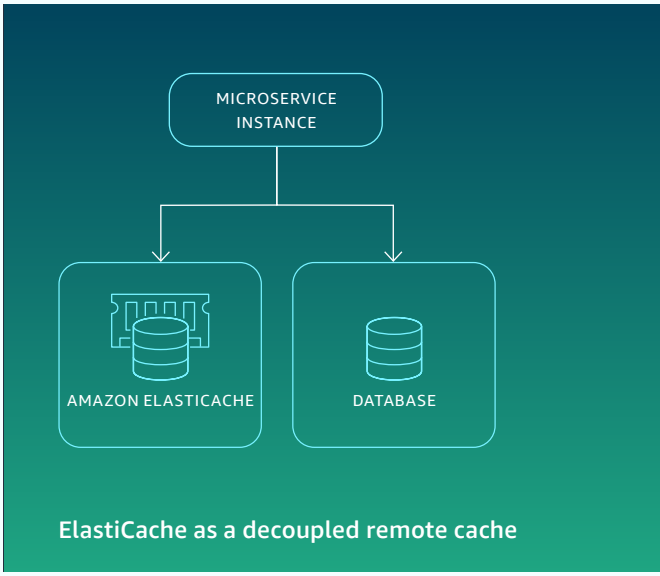
ElastiCache as a Remote Cache

A remote cache (or side cache) is separate and decoupled from any application or database instances, and it is dedicated to storing and sharing the cached data in-memory. ElastiCache is a remote distributed cache offering multi-node scalability (up to 500 nodes in a single cluster). The largest memory capacity available for an ElastiCache for Redis cluster is 310TiB.

ElastiCache stores data in key-value format on dedicated servers. It supports very high throughput with greater than 1 million requests per second, per cache node.

The average latency of a server-side request to a remote cache is on a microseconds timescale, which is faster than a request to a disk-based database by an order of magnitude.

With ElastiCache, the cache itself is not directly connected to the database, but is optionally used adjacent to it. The database is optional because a large class of use cases do not require a backing database in conjunction with the cache. For example, if the cache stores responses from calls to other services or if the cache aggregates data from multiple databases for running queries against consolidated data, then a database is not a requirement.



ElastiCache as a decoupled remote cache

Appendix B: Using ElastiCache for Redis as a User Session State Store

Microservices-based application architectures fragment user sessions across multiple microservices. This can be addressed by externalizing session state information in a single and shared remote cache. This is a key pattern for improving the user's experience, hence is often a starting point for using caching as part of distributed application's infrastructure.

Session data is different from application data (or user data) in some important ways. Application or user data populates the application's databases. It is used by the application to do its job, such as product data or pricing. It can also consist of data about each user, such as user preferences.

On the other hand, session data is data about the user's session or application state. It consists of metadata about the application's interactions stored as session variables. Session data is ephemeral. When the session ends, either from the user closing the browser or due to a time-out, the session variables are cleared. Examples of session variables are user id, password, preferences, and privileges.

There are various ways to manage user sessions. You can store those sessions locally to the node responding to the HTTP request or designate a layer in your architecture which can store those sessions in a scalable and robust manner. Common approaches include using Sticky sessions or using a Distributed Cache for your session management.

The Limitations of Sticky Sessions

Sticky sessions, also known as session affinity, allow you to route a site user to the particular web server that is managing that individual user's session. The session's validity can be determined by a number of methods. This includes client-side cookies or by configurable duration parameters that can be set at the load balancer, which routes requests to the web servers. While this approach is cost effective and fast because it minimizes network latency relative to remote caches, it has key limitations. In the event of a failure, you are likely to lose the sessions that were resident on the failed node. An ElastiCache for Redis cluster, with replicas across multiple AZs for high availability, is unlikely to lose session state information.

If the number of web servers changes, for example a scale-out scenario, it's possible that the traffic may be unequally spread across the web servers as active sessions may exist on particular servers. If not mitigated properly, this can hinder the scalability of your applications.

Sticky sessions are also subject to capacity limitations. For nodes that handle a large number of sessions, the memory capacity of a web server may not be able to store all the session data.

The use of ElastiCache as a highly available remote cache is a much more reliable way of storing both application data and session information, boosting the overall performance and availability of your applications.

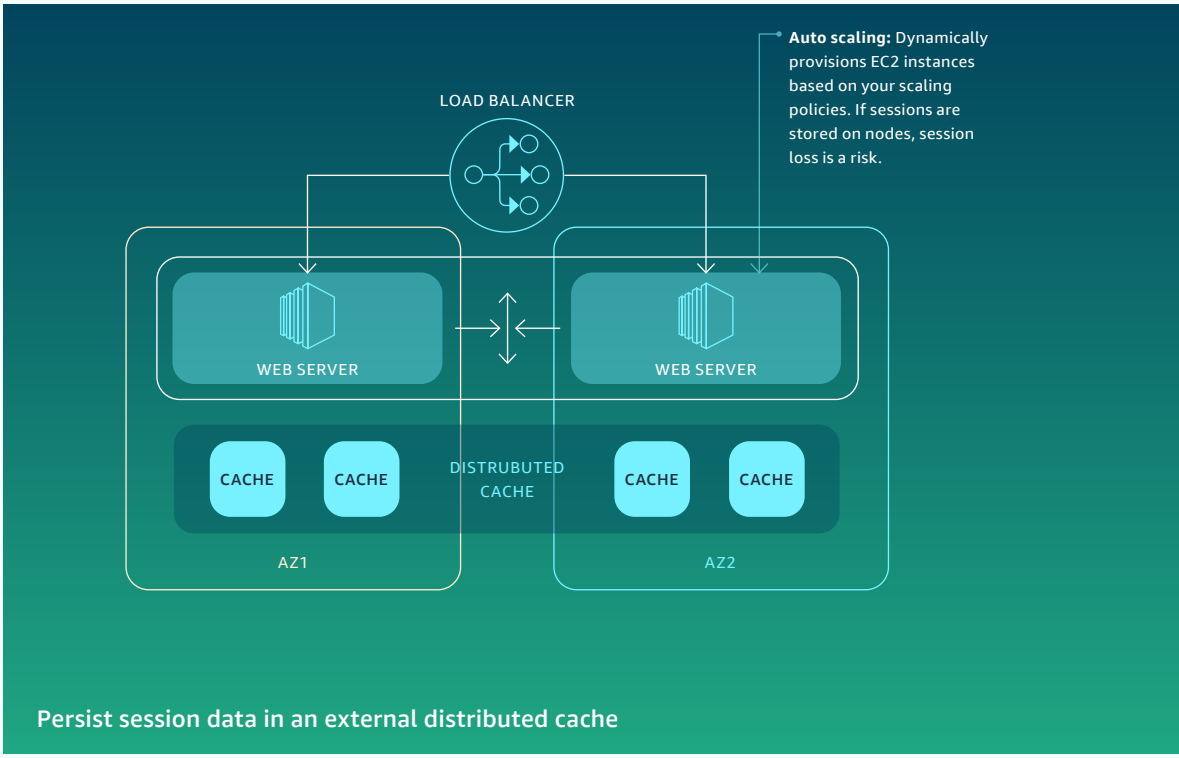
Distributed Session Management

To address scalability and provide shared data storage for sessions that can be accessible from any individual web server, you can abstract the HTTP sessions from the web servers themselves. A common solution is to leverage ElastiCache as a remote key-value cache for its in-memory performance.

ElastiCache is perfectly suited for session data because it provides fast, highly available access to state information, which is critical for elastic operations. Session state information can be used for improving the user's experience by providing resilience to failures.

If a microservice instance fails, another instance can be spun up and the new instance can read state information from the cache and continue processing from where the failed instance left off. For example, in an e-commerce application, the user's shopping cart and browsing history can be recovered even after a failure.

Externalized and shared session data provides support for the high degree of elasticity that comes with microservices. Multiple ephemeral microservice instances or multiple disparate microservices can share the same user session context. Shared session state caches are a key enabling technology for maintaining the infrastructure's scalability.



Persist session data in an external distributed cache