Solution Brief

Cache and Message Broker for Microservices

Speed up and simplify microservice applications using Redis Enterprise

redis

Redis Enterprise was built with many of the core principles that guide microservice architectures: agility, resilience, scalability, and flexibility. This alignment makes Redis Enterprise an ideal caching and asynchronous messaging solution for microservice applications.

But Redis Enterprise does more than align with the strengths of microservices. It also helps to overcome three microservice challenges: complexity, eventual consistency, and latency.

What are microservices?

A microservice architecture breaks an application into a collection of decoupled, lightweight services. Each microservice is built around a specific business context—a goal, focus, or problem area—known as its *domain*. That practice permits domain experts to create services that support their business logic needs.

Individual development teams are empowered to own and operate their specific microservices. Each service is isolated around its domain context so that each team is responsible for choosing the technology stack that best fits that domain. This ensures that each microservice is individually deployable, scalable, resilient, and fully owned by each team.

Microservices benefits

The agile and bounded nature of microservice architecture brings significant advantages. The benefits include:

- **Team-empowerment:** Small, independent teams can quickly deploy code to adapt to changing conditions
- **Flexibility:** Each service is built with the technology that best fits its unique needs
- **Reusability:** Modular code can be applied for multiple purposes, which enables faster development
- **Isolation:** Application service components are individually operable and scalable, with fault isolation to prevent failure from one microservice impacting another

Key microservices challenges

Despite their many benefits, microservice architectures have drawbacks. The most critical challenges are:

- **Increased complexity**
  A multitude of small and independent services means more services to manage, monitor, and maintain. Each microservice is operated independently yet must communicate with others. Microservices often require unique data models to support application needs, each of which may require its own database or data management solution to support.
- **Eventual consistency**
  Data in one microservice may need to be accessed by another service. Data consistency must be maintained as data is shared and processed among dozens or even hundreds of individual microservices.
- **Latency**
  Each independent service communicates via API calls. Calls to a multitude of different services introduces the problem of network latency, which may cause large, complex microservice applications to face issues of slow response time. In addition, databases within each microservice may not be fast enough to meet requirements.

Isolation is a key microservice principle. It calls for completely decoupled code, teams, databases, and deployment cycles. The goal is to increase scalability, agility, and fault isolation in order to produce fast, resilient applications. Each of these isolated services is operated by an empowered team that can act quickly to deploy new features or to respond quickly to issues.

Caching delivers fast and consistent data to microservices

Performance always matters, but data performance is **especially** critical to microservice applications. Caching can counteract the network latency that microservices produce as a result of the multiple API calls required for interservice communication. Decreasing data latency by using caching can give you back critical response time.

Caching is also an excellent way to distribute data that is shared by multiple domains from a system of record without breaking the scope of each individual microservice domain context.

How Redis Enterprise works with microservice caching

Microservice caches typically implement one of the following patterns based on the scope of data access across the architecture:

- **API gateway caching** for globally shared data that must be accessed by all microservices (session data, authentication tokens, etc.)
- **Command Query Responsibility Segregation (CQRS)** for data shared by multiple microservices, but not needed by all at the global level (cross-domain data)
- **Query caching** for data within a single microservice (domain-specific)

API GATEWAY CACHING

Cache globally shared user session and authentication data at the API gateway

Microservice applications often cache globally accessed data at the API gateway level to distribute and speed up data access by the entire underlying architecture. One example is caching session and authentication data. This approach makes frequently needed session data available in real time to all services. Doing so reduces application latency without breaking the bounds of each microservice's individual business context.

When a user logs in, Redis Enterprise's cache stores the session data (such as user ID or preferences) or authentication status (such as permissions). This data is checked by the API before the user interacts with any individual microservice.

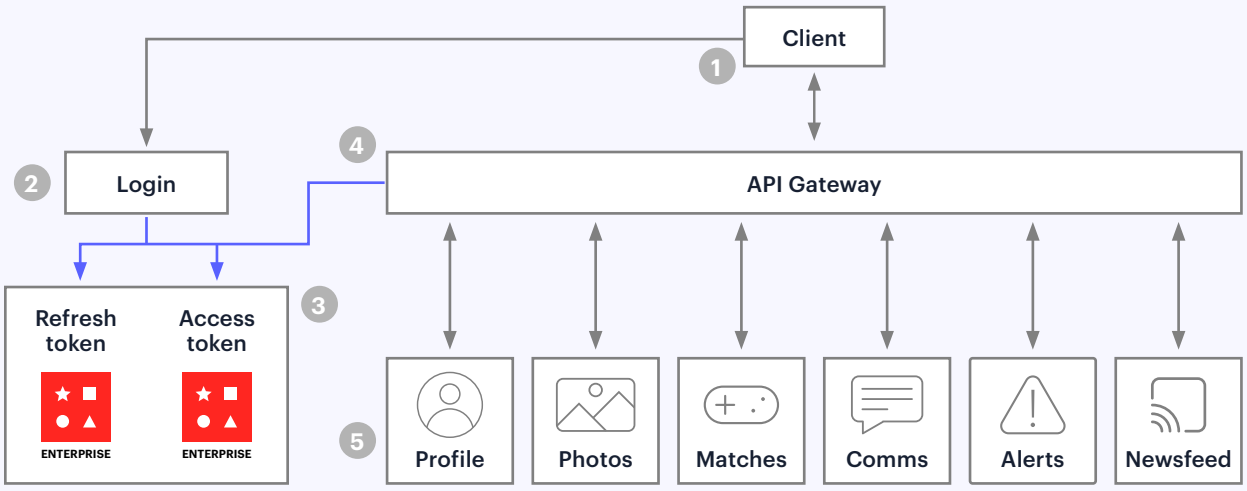
Customer story: Online dating application

Let's look at a customer example from a popular online dating application. This customer supports millions of users daily. In a given day, the dating site's users update hundreds of thousands of photos, send millions of messages, and match with tens of millions of other users.

The application has several microservices, each of which performs specific tasks. Among the tasks:

- Manage user profiles (profile microservice)
- Upload and manage photos (photo microservice)
- List and manage matches with other profiles (match microservice)
- Communicate with other users and matches (communication microservice)
- Send notifications to users (alerts microservice)

To reduce latency during user sessions, this customer uses Redis Enterprise to cache authentication data. The token is pulled by the API gateway to authenticate users and to relay key information about the user's session, such as user settings and permissions.



Let's explore how a customer online dating application can work using Redis Enterprise to cache session data.

1. The client is the user interface. The application is available on desktop, mobile web, Android, and iOS.
2. A user logs in with their credentials.
3. Two tokens are created with user authentication and session data: an access token and a refresh token. These tokens are cached in Redis Enterprise. The access token contains authentication data, user information, and permissions; the API uses it during the user session. The token has a two-day time-to-live (TTL). Once the authentication token expires, a new access token is (or can be) generated from a refresh token, which has a longer TTL; it keeps users logged in if they have enabled the "keep me logged in" setting.
4. The API gateway manages calls from the many microservices that power the online dating application. When a request is made, the API gateway checks the session token cached in Redis Enterprise to see if the user is authenticated. Assuming the user is authenticated, the API lets the transaction occur, and it passes on session information including user data and permissions.
5. Each individual microservice needs to interact with the API gateway, whether to manage dating profiles, upload and manage pictures, view and manage dating matches, send communications to other profiles, send alerts to users, or display relevant news in a newsfeed. In each case, the API gateway first checks if a session is valid before it allows the microservice call to go through.

How Redis Enterprise made a difference

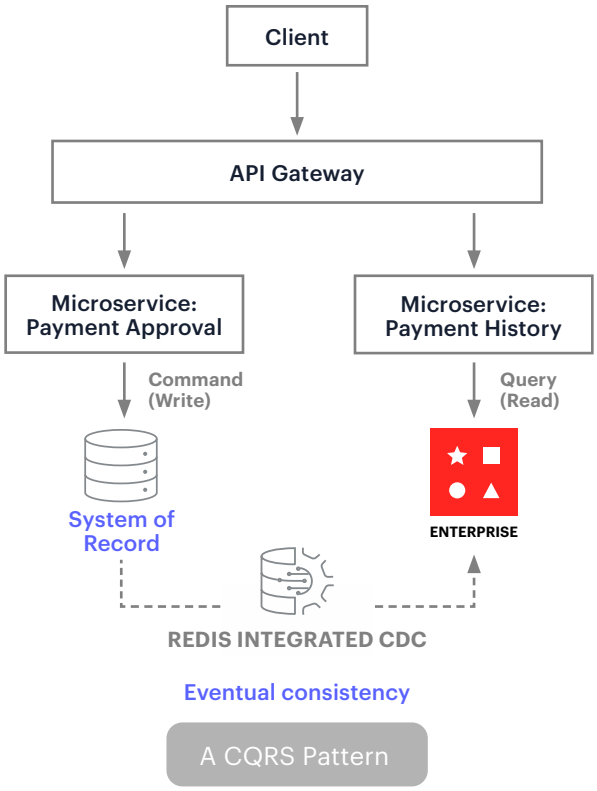
- Using Redis Enterprise as a cache to store user session and authentication data enabled low-latency customer experiences by ensuring that widely accessed data was available, with a response time in sub milliseconds.
- Redis Enterprise's scalability ensured that application latency remained low even during periods of peak user activity.
- As an outage of the cache at the API gateway level would cause the entire system to go down, all applications would be unreachable. Redis Enterprise's 99.999% uptime ensured that no customer disruptions would occur due to lost data.

CQRS PATTERN

Caching cross-domain shared data

Microservices need fast access to data. However, this can be challenging when dozens or hundreds of microservices try to read from the same slow disk-based database. Cross-domain data needs to be available to each microservice in real time, without creating dependencies and breaking isolation.

CQRS is a critical pattern common to many customers using Redis Enterprise as a cache in microservice environments. It enables a service to write data to a slower disk-based database, which remains the system of record, while pre-fetching and caching that data in Redis Enterprise for fast queries. Doing so makes the information immediately available to additional microservices that must read that data.



Customer example: Payment processing microservices application

Let's consider a financial services microservice application. The application has individual microservices to perform specific tasks, such as:

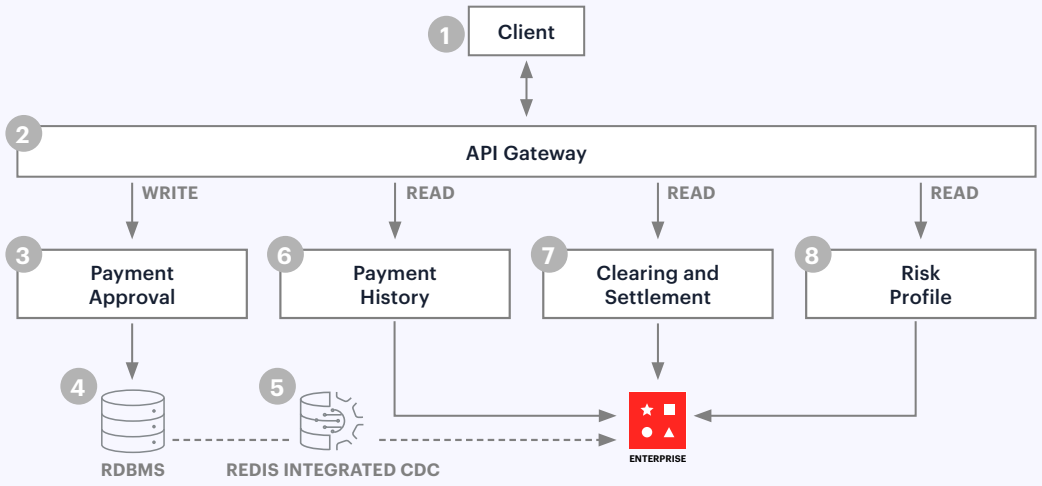
- Approve or decline payments (captured in a payment approval microservice)
- View payment history (payment history microservice)
- Clear and settle payments (clearing and settlement microservice)
- Update a customer's risk profile (risk profile microservice)

The payment history, clearing and settlement, and risk profile microservices are each dependent upon the data from the payment approval microservice. That payment approval microservice must write the outcome of each processed payment (whether approved or declined) to the database that acts as the application's system of record.

A CQRS pattern helps to avoid the latency of all of these microservices calling on a slow database. Additionally, CQRS eliminates the dependency on the database by replicating data and freeing each microservice to operate, scale, and deploy code independently.

Using the CQRS pattern, the microservice API provides the payment approval microservice with write-only access to the database so it can record whether a transaction was approved or denied.

This data is then pre-fetched into a Redis Enterprise cache with the Redis integrated Change Data Capture (CDC) capability that is read and used across domains by the payment history, clearing and settlement, and risk profile microservices.



Consider the process in a customer payment processing application using Redis Enterprise as a cache in a CQRS pattern:

1. The client is the user interface (mobile application, web app, etc.).
2. The API gateway handles all communication between the client and the microservice applications.
3. The payment approval microservice determines whether a payment is approved or declined and then writes that decision to the application database.
4. The application database is a persistent disk-based SQL database acting as the system of record for all payments. It contains the customer's approval status and metadata associated with each payment, such as date and account number. The API gateway only allows writes to the payment approval microservice; all others are read only.
5. The Redis Enterprise cache is pre-fetched using the Redis integrated CDC capability with payment approval data. New payments written to the application database are replicated to the Redis Enterprise cache with eventual consistency.
6. The payment history microservice reads data from Redis Enterprise to view payment dates, status, and other metadata.
7. The clearing and settlement microservice reads data from the Redis Enterprise cache. It moves funds between a sender and recipient account for transactions approved by the payment approval microservice.
8. The risk profile microservice reads data from the Redis Enterprise cache to update customer risk profiles after a transaction is approved or denied.

How Redis Enterprise made a difference

- Using Redis Enterprise as a cache reduced application latency by ensuring that payment approval was cached and available across domains to downstream microservices in sub milliseconds.
- Caching cross-domain data ensured that the payment processing application could eliminate critical business dependency on the database. Each microservice could operate on its own release cycle with autonomy without compromising data integrity.
- The Redis integrated CDC capability transformed the system of record data from a write-optimized data structure (SQL table) to a read-optimized data structure (Redis Enterprise cache) while maintaining isolation.
- Because it could scale up to 200 million operations/second, Redis Enterprise effortlessly handled increases in application processing while maintaining sub-millisecond latency.
- Redis Enterprise was also resilient enough to ensure critical payment data wouldn't be lost due to cache outage.
- Everything got simpler and cheaper. Using a single data management solution for caching and CDC was far less complex than deploying multiple different technology products that would require integration effort.

QUERY CACHING

Query caching for data within a single business context

Building a microservice architecture from the ground up requires following the principle of domain-driven design to split applications into isolated and logical areas of focus. This approach typically leads architects to pursue CQRS and globally shared data at the API gateway.

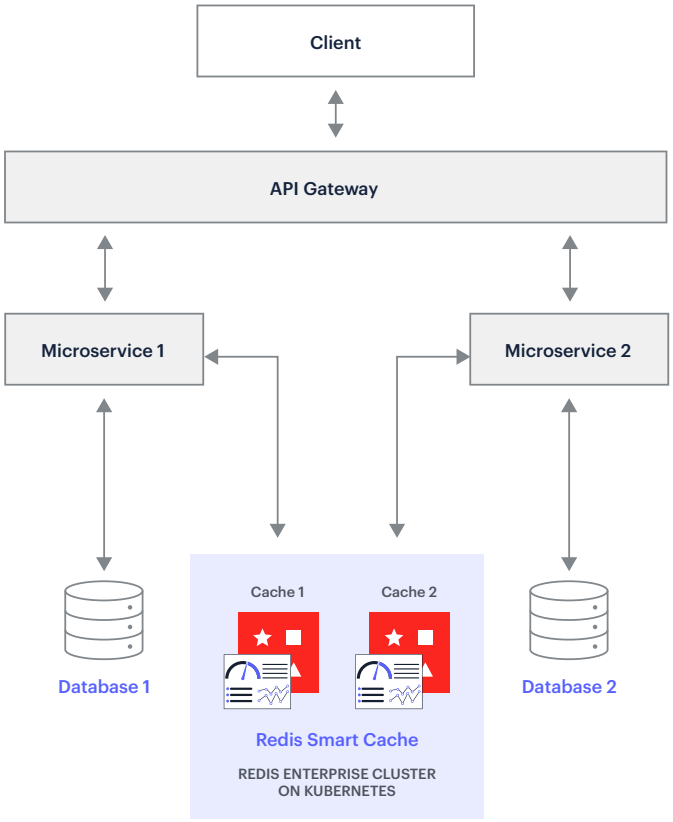
However, in many real-world scenarios enterprises do not start with a blank slate. They must work around the constraints of legacy architecture and technical debt.

For example, an existing relational database may already support multiple microservices. While that database may meet most of the organization's needs, one particular service may have issues meeting its performance service level agreements. Instead of replatforming an entire system, Redis Enterprise can simply be deployed into the existing architecture using a cache-aside or sidecar pattern by caching the results from the relational database. With Redis Enterprise, each query cache can be deployed in a multi-tenant cluster that provides physical isolation to maintain domain independence.

This sidecar architecture uses the Redis Smart Cache client library to standardize cache-aside patterns across all the microservices that require query caching, with no need to update code. This cache speeds up data that an individual microservice needs within a single domain context.

Data queries from the microservice are first sent to the Redis Enterprise cache. If data is present, the results are delivered (at sub-millisecond speed). If the data does not exist in the cache, it is delivered by the primary database and stored in the cache to lower the latency of future requests.

Redis Smart Cache also provides the ability to monitor the cache, thus providing insights into usage without the need for a separate analytics tool.



How Redis Enterprise is used for inter-service communication between microservices

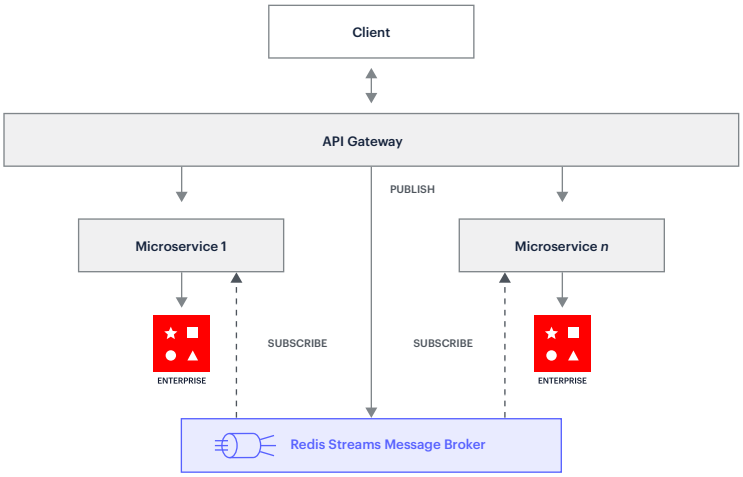
Building loosely coupled microservices is an extremely lightweight and rapid development process. However, building inter-services communication patterns to share state, events, and data between these services is not as simple.

One easy and fast communication pattern is direct (point-to-point) inter-service communication between services, wherein you use RESTful HTTP or gRPC calls. However, synchronous communication assumes both endpoints are available. That contradicts the paradigm of isolation and decoupling that is inherent in microservice architectures, wherein the premise is that a service can be unavailable at any time.

When you have hundreds of microservices, there is also the issue of scale and complexity with implementing retry logics. Hence you should avoid using direct, synchronous communication unless there is a specific use case that requires it.

The recommended pattern is to use an asynchronous broker approach where each service connects a centralized messaging system (a broker) – such as generic message queues like RabbitMQ and Redis queues (lists) – to complex Kafka event streams. In an asynchronous approach, the service doesn't wait for the response coming from another service, thus allowing for that service to be unavailable and respond when recovered and embracing eventual consistency.

An alternative solution is to use Redis Streams, a native, immutable, time-ordered log data structure in Redis Enterprise, as a lightweight, event-driven, asynchronous publish-subscribe message broker and event store. In this pattern, a producer (one service) can publish an event without awareness of whether any consumer (another service) is listening. Consumers of that event can react to it when they are ready or they can ignore it altogether.



This ensures the microservice that is publishing events remains decoupled from the microservice(s) consuming them. The result is that there are no cross-dependencies on availability and release cycles.

Redis Streams supports consumer groups, message persistence, primary/secondary data replication functions, varying delivery guarantees, and other features that are similar to what Apache Kafka topic partitions deliver. However, since you only need a key to deploy Redis Streams, it is much easier to set up than a Kafka Topic (you don't need an expert administrator, for one). Along with Active-Active capabilities and simple deployment, Redis Streams is an ideal option for managing microservice communication at scale.

Redis Enterprise makes microservice applications faster and easier to operate

Reduced operational complexity

Redis Enterprise offers simplified management to reduce the operational complexity that comes with microservice architectures. It enables you to easily deploy and manage dozens to hundreds of multi-tenant and multi-model caches, all in one data platform.

Redis Enterprise caches can be customized for each microservice based on durability, throughput, persistence, and replication requirements. The same data platform can also be used for inter-service communication to simplify coding and to reduce the number of tools needed. The ability to deploy and manage individual caches based on the unique needs of each domain plus act as message broker enables the microservice principles of isolation, fast time to market, and team empowerment.

Real-time speed for faster microservices

Network latency and slow legacy databases can drag down application performance. With sub-millisecond performance for data queries and messaging, Redis Enterprise provides a way to gain back much of the time lost to greatly improve performance.

All the benefits of a full house (without noisy neighbors)

A multi-tenant architecture allows individual resources (databases or virtual machines) to be shared by multiple separate users. These users could be individual customers or business units that share access. Think of a single tenant system like a stand-alone home, while a multi-tenant one is like an apartment: the building is shared but each tenant has their own individual living space. Multi-tenancy brings obvious benefits including cost efficiency, simplified architecture, and better resource utilization.

However, multi-tenancy runs afoul of the microservice practice of completely isolating individual application components. Multi-tenant systems often run into challenges competing for resources, and individual services may overconsume a resource shared by other microservices. This problem is commonly referred to as a noisy neighbor.

Redis Enterprise gives you the best of both worlds: the benefits of multi-tenancy with the level of isolation that microservices require. Because its cluster architecture provides isolation at every level, it avoids noisy neighbors. As individual Redis processes are single threaded, they are inherently bounded by, at most, a single CPU. Redis Enterprise's approach solves the problem of isolation without the trade-off of management complexity.

But that's not all...

Redis Enterprise brings real-time speed for all your data needs

You can also use Redis Enterprise for other important tech challenges. Extend real-time performance by using it beyond caching, e.g., using it for event sourcing and as a lightweight primary database to support individual microservices.

Want to learn more about caching, using Streams as a message broker in an event sourcing pattern, and deploying microservices on Kubernetes?

Read the definitive guide to caching with Redis: Download the *Caching at Scale with Redis* e-book.
Download now

Learn how to develop and operate a high-performance microservice architecture with Redis in our *Redis Microservice for Dummies* e-book.
Read now

Learn about event-driven streaming architectures and the difference between Kafka and Redis Streams. Download the *Understanding Streams in Redis and Kafka—A Visual Guide* e-book.
Download now

Learn how DevOps teams can easily manage and administer and automatically deploy Redis clusters on Kubernetes platforms by watching *Deploying a Microservice Data Layer on Kubernetes* webinar.
Watch now