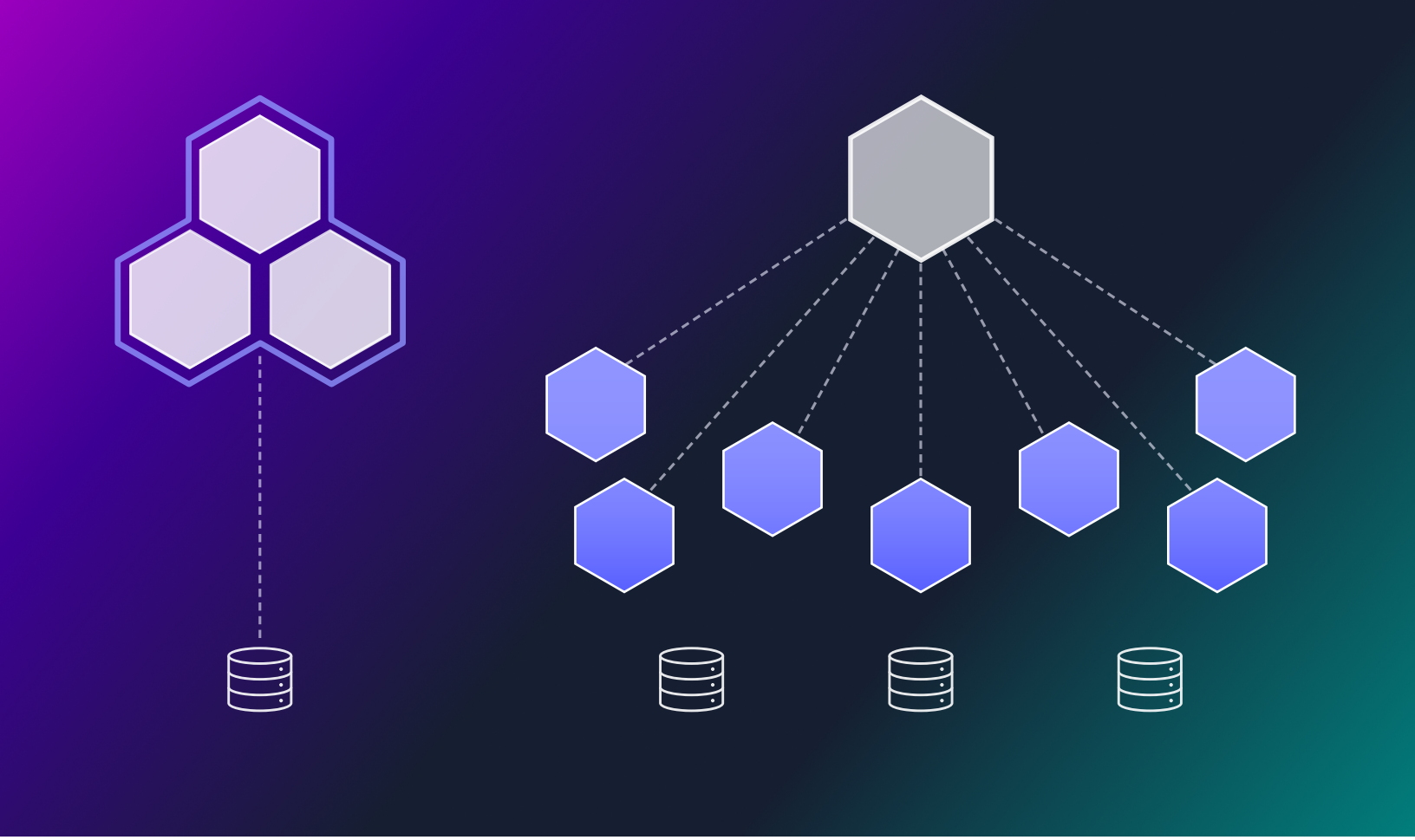


Solution Brief

Caching for Microservices

Speed up and simplify your microservices applications with Redis Enterprise

redis

Redis Enterprise was designed and built with many of the same core principles that guide microservices architectures: agility, resilience, scalability, and flexibility. This alignment makes Redis Enterprise an ideal caching solution for microservices applications. But not only does Redis Enterprise align with the strengths of microservices – it also helps overcome two key microservices challenges: complexity and latency.

What are microservices?

A microservices architecture breaks an application into a collection of decoupled and lightweight services. Each microservice is built around a specific business context – a goal, focus, or problem area – that is known as its domain. These modular services communicate via API to perform their own functions within the application. Each service is isolated around its domain context and supported by individual teams that are empowered to own and operate their own microservices. This ensures that components are individually deployable, individually scalable, resilient, and fully owned by an agile team.

Microservices benefits

The agile, isolated, and focused nature of microservices applications brings significant advantages. Some of these benefits include:

- **Team-empowerment:** Small independent teams can quickly deploy code to adapt to changing conditions with great agility
- **Flexibility:** Each service can be built with the technology that best fits its unique needs
- **Reusability:** Simple and modular code is reusable and can be applied to multiple purposes to enable faster development.
- **Isolation:** Isolation ensures that application components are individually operable and scalable, and provides fault isolation to prevent failure with microservices from impacting one another.

Key microservices challenges

Despite the many benefits, microservices architectures are not without drawbacks. The most common and critical microservices challenges are:

- **Increased complexity**
  Complexity is the downside that comes with a multitude of small and independent services. Each of these individual microservices must be operated independently and each has unique data needs that introduce a tremendous amount of complexity. Different microservices often have unique data use cases or require unique data models, each of which may require its own databases or data management solution to support. Additionally, data consistency must be maintained as data is shared and processed amongst dozens or even hundreds of individual microservices.
- **Latency**
  Latency is another downside. Each of these independent services must communicate via API calls. Calls to a multitude of different services introduce the problem of network latency, which can leave larger and more complex microservices applications facing issues with slow response time.

Isolation is a key microservices principle that calls for completely decoupled code, teams, databases, and deployment cycles to increase scalability, agility, and fault isolation and produce faster, more resilient applications. Each of these isolated services is operated by an agile and empowered team that can act quickly to deploy new features or respond quickly to potential issues.

Caching delivers fast and consistent data to microservices

Data performance is especially critical to microservices applications. Caching to decrease data latency is a great way to counteract the network latency that often builds due to the multiple API calls required for interservice communication - and gain back critical response time.

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

How is Redis Enterprise used for microservices caching?

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

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

Caching globally shared user session and authentication data at the API gateway level

Microservices applications often cache globally accessed data at the API gateway level to distribute and speed up data that is accessed by all services. A perfect example of this is caching session and authentication data. This approach makes frequently needed session data available in real-time to all services, reducing application latency without breaking the bounds of each microservice's individual business context.

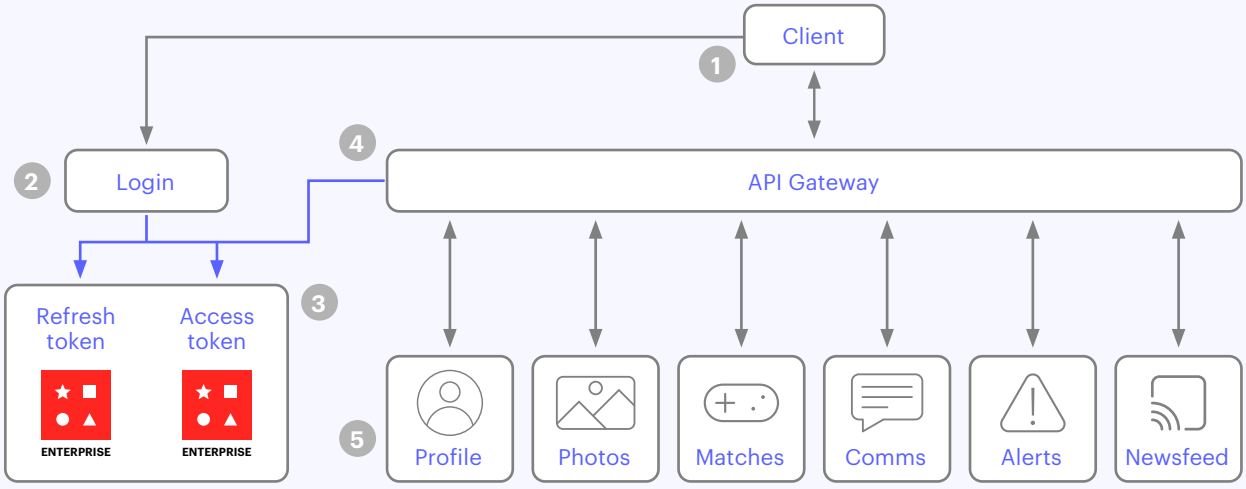
When a user logs in, key session data (user ID, preferences, etc.) or their authentication (authorization status, permissions, etc.) are cached in Redis Enterprise and checked by the API before the user interacts with individual microservices.

Customer story: Online dating application

Let's take a look at a customer example from a popular online dating application. The application has individual microservices to perform specific tasks like:

- 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)
- Surface important notifications to the user using alert (Alerts microservice)

This customer application supports millions of users daily to facilitate activity as users update hundreds of thousands of photos, send millions of messages, and match with tens of millions of other users. In order to reduce latency during user sessions, Redis Enterprise is used to cache authentication data in a token that can be quickly pulled by the API gateway to authenticate users and relay key information about their session like user settings and permissions.



Explore a customer online dating application using Redis Enterprise to cache session data.

1. The client is the user interface. This application was 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. These tokens are cached in Redis Enterprise. One is an access token and the other is a refresh token. The access token is used by the API during the user session, and it contains authentication data, user information, and permissions. This token has a two day time to live (TTL). Once the Authentication Token expires a new access token can be generated from a Refresh Token, which has a longer TTL and is used to keep users logged in should they have the "keep me logged in" setting enabled.
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 and lets the transaction occur, while passing on key session information like user data and permissions.
5. Each of the individual microservices to manage dating profiles, upload, and manage pictures, view and manage dating matches, send communications to other profiles, surface alerts to users, and display relevant news in a newsfeed need to interact with the API gateway. The API gateway first checks if a session is valid before allowing the microservice call to go through.

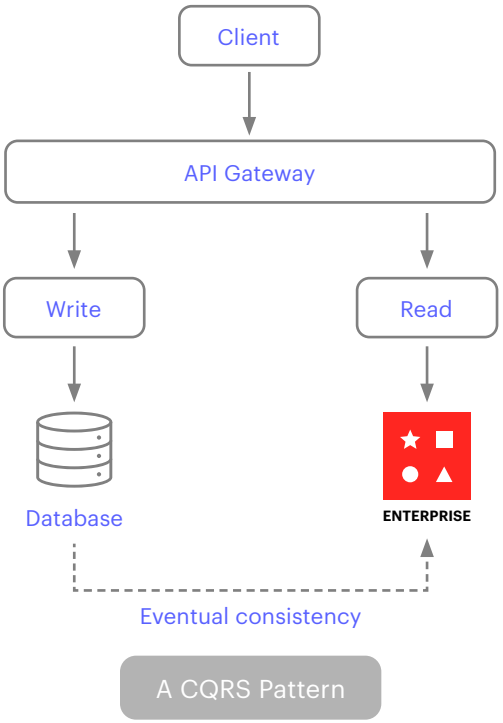
Redis Enterprise benefits:

- Using Redis Enterprise as a cache to store user session and authentication data enabled low-latency customer experiences by ensuring widely accessed data was available in sub-milliseconds
- Redis Enterprise's scalability ensured that application latency remained low even during periods of peak user activity
- Redis Enterprise's 99.999% uptime ensured that the customer could store critical data in memory with confidence, avoiding customer disruptions due to lost session data

Caching cross-domain shared data via CQRS

Microservices need fast access to data but this can be challenging when there are dozens or hundreds of microservices trying to read from the same slow disk-based database. Cross-domain data needs to be made available to each microservice in real-time - and without breaking the scope of its focused business context and goal.

Command Query Responsibility Segregation (CQRS) is a critical pattern common to many customers using Redis Enterprise as a cache in microservices environments. It enables an application to write data to a slower disk-based database, while pre-fetching and caching that data in Redis Enterprise to make it available in real-time to additional microservices that must be able to read application data.



Customer example: Payment processing microservices application

Let's take a look at a customer example from a financial services microservices application. The application has individual microservices to perform specific tasks like:

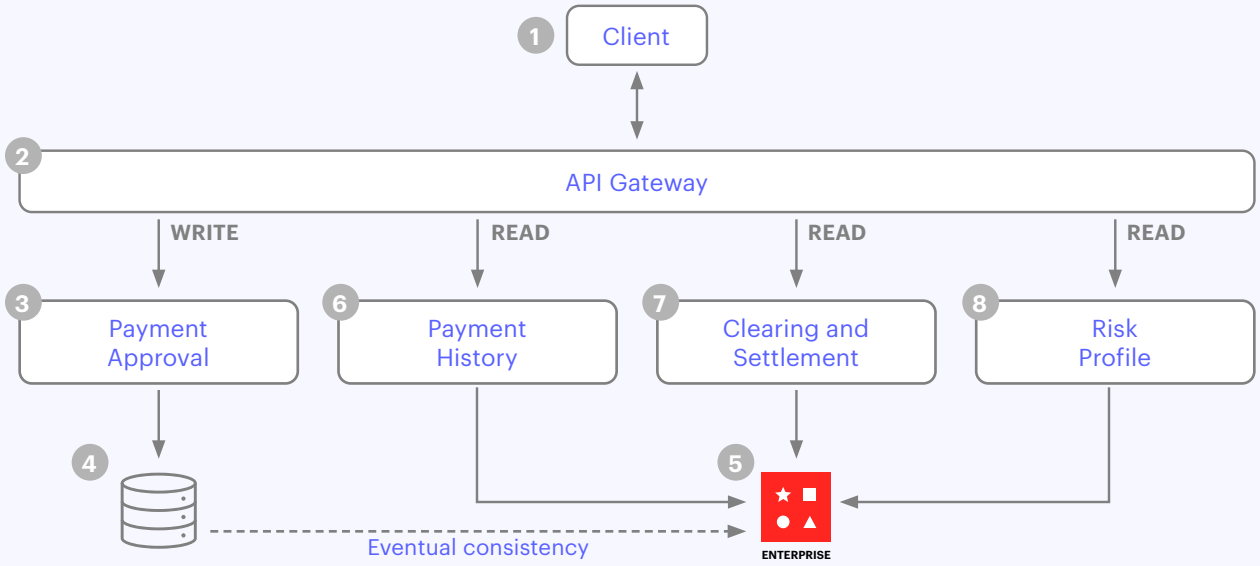
- Approve or decline payments (Payment Approval microservice)
- View payment history (Payment History microservice)
- Clear and settle payments (Clearing and Settlement microservice)
- And update a customer's risk profile (Risk Profile microservice)

The Payment History, Clearing and Settlement, and Risk Profile microservices are each dependent upon the data from the Payment Approval microservice.

The Payment Approval microservice must be able to write the outcome of each processed payment (whether it was approved or declined) to the database acting as the application's system of record. In order to avoid the latency of all of these microservices calling on a slow database, a CQRS pattern is utilized. Additionally, CQRS eliminates the business dependency on the database, replicating data and freeing each microservice to operate, scale, and deploy code independently.

Using the CQRS pattern, the microservices API provides the Payment Approval microservice with write-only access to the database so it may record if a transaction was approved or denied.

This data is then pre-fetched into a Redis Enterprise cache that is read and used across domains by the Payment History, Clearing and Settlement, and Risk Profile microservices.



Explore 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 each of the microservices applications, in addition to interservice communication.
3. The Payment Approval microservices determines whether or not a payment is approved or declined and then writes that decision to the application database
4. The application database is a persistent disk-based database acting as the system of record for all payments. It contains their approval status and metadata associated with each payment, like date, account number, etc. The API gateway only allows writes to the Payment Approval microservices, all others are read only.
5. The Redis Enterprise cache is pre-fetched with payment approval data, and any 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 and moves funds between a sender and recipient account for any 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.

Redis Enterprise benefits:

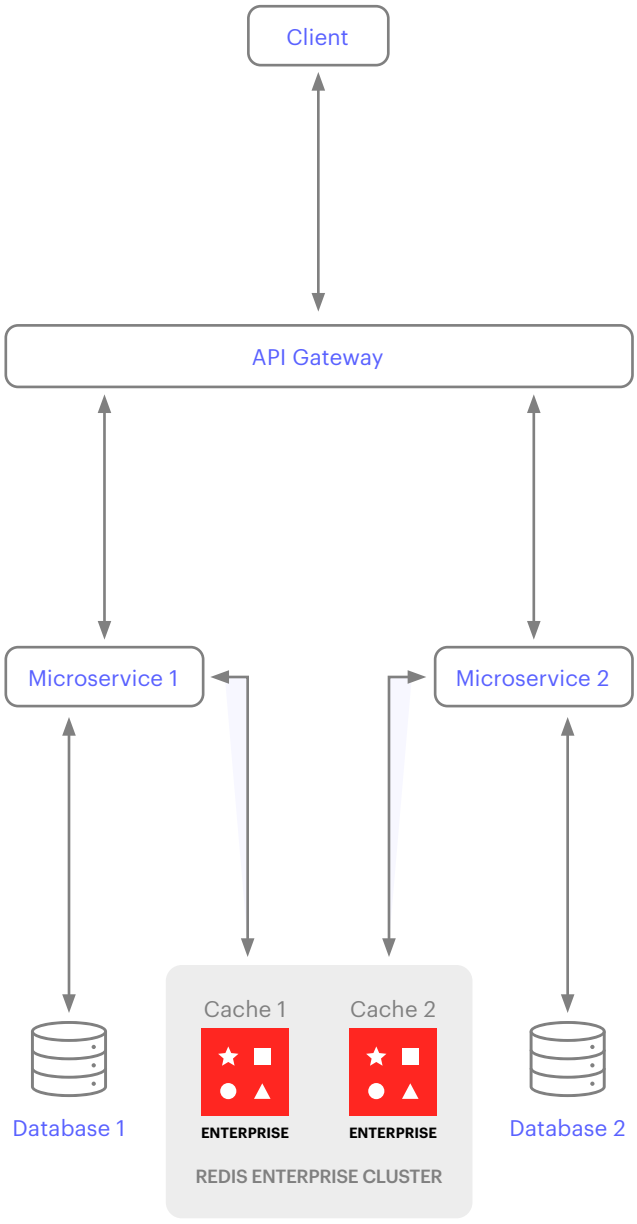
- Using Redis Enterprise as a cache reduced application latency by ensuring that payment approval was cached and available across domains to downstream microservices in sub-milliseconds
- Caching cross-domain data also ensured that the customer was able to eliminate critical business dependency on the database, allowing each microservice autonomy to operate on its own release cycle, without compromising data integrity
- With the ability to scale up to 200 million operations/second, Redis Enterprise was able to effortlessly handle increases in system activity while maintaining sub-millisecond latency. It was also resilient enough to ensure that critical payment data wasn't subject to loss due to cache outage

Query caching for data within a single business context

Building a microservices 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 the benefits of CQRS and globally-shared data at the API gateway. However, in many real-world scenarios enterprises do not start with a blank slate and must work around the constraints of their legacy architecture and technical debt.

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

A sidecar architecture can be deployed to speed up data that is needed by an individual microservice with a single domain context. It acts with a cache-aside pattern where data queries from the microservice are first sent to the Redis Enterprise cache. If data is present, 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 Enterprise can also be deployed with multi-tenancy, enabling multiple completely isolated caches hosted on the same cluster to each support their own single domain microservices.



EXAMPLE OF A SIDECAR PATTERN

Redis Enterprise enables faster and easier to operate microservices applications

Reduced operational complexity

Redis Enterprise offers simplified management to reduce the operational complexity that comes with microservices architectures. It provides the ability to manage dozens of multi-tenant and multimodal databases and their unique data needs in one platform. Aspects of each Redis Enterprise cache can be customized for each microservice with their own durability, throughput, persistence, and replication requirements. The ability to deploy and manage individual caches based on the unique needs of each domain enables the key microservices principles of isolation and team empowerment.

Real-time speed for faster microservices

All of that network latency 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 by interservice communication 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 standalone home, while a multi-tenant one is like an apartment - the building is shared but each tenant has its own individual living space. Multi-tenancy brings obvious benefits like cost efficiency, simplified architecture, and better resource utilization.

However, multi-tenancy runs afoul of the microservices 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 allows for the best of both worlds, providing the benefits of multi-tenancy with the level of isolation required for microservices. Because its cluster architecture provides isolation at every level, it avoids the common problem of noisy neighbors. Redis Enterprise's approach solves the problem of isolation without the tradeoff of management complexity.

But that's not all...

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

Redis Enterprise can also be used to extend real-time performance to a number of microservices use cases beyond caching, like; interservice communication and event sourcing and is also commonly used as a lightweight database to support individual microservices.

Want to learn more about caching and microservices?

Read the definitive guide to caching with Redis. Download the Caching at Scale with Redis ebook.
Download now

Learn how to develop and operate a high-performance microservices architecture, with Redis in our Redis Microservices for Dummies ebook.
Read now

Learn from Allen Terleto, Field CTO at Redis, and Viren Baraiya, the co-creator of Netflix Conductor and CTO at Orkes, how you can overcome key microservices challenges and scale reliably using an in-memory data layer and workflow orchestrator.
Watch now