
yugabyteDB
Modern Design Patterns:
From Disaster Recovery to
Ultra-Resilience

Table of Contents

An Introduction to Modern Design Patterns

Outages are inevitable in a complex global business environment. Many companies are now exploring how they can transition from reactive incident management to proactive architectural resilience.

In the modern era of accelerating digital scale, traditional methods of building on customized, high-end hardware have become financially unsustainable, leading to poor SLAs due to technical complexity and manual issue resolution.

To address these challenges, many enterprises have switched to cloud-native infrastructure (commodity hardware and API-driven automation). However, cloud-native infrastructure comes with its own set of challenges. Over the years we have seen an increase in reported outages and failures, for multiple reasons, including cascading failures, system complexity, network issues, and the sheer number of components involved.

Enterprises are having to reimagine their architectural needs and build highly reliable services on top of a failure-prone infrastructure layer.

This shift has intensified further as we transition from a “cloud-native” to a “cloud-native-post-AI” world.

This whitepaper outlines some key strategies, including:

- How to build ultra-resilient systems
- Integrating failure handling into the core architecture
- The continuous reevaluation of cost-resilience trade-offs
- How the strategic use of distributed technologies achieves high availability and scalability

> *“Everything fails, all the time.”*
>
> **- Werner Vogels, Amazon Chief Technology Officer**

The Case for Ultra-Resilience

Companies that adopt ultra-resilience principles minimize costly outages while enhancing innovation opportunities, customer satisfaction, and operational efficiency.

Ultra-resilient systems deliver measurable business value by protecting revenue, preserving reputation, and reducing technical debt. Below, we detail why ultra-resilience is important for modern applications.

Outage Types and Frequency

As organizations move to cloud-native architectures, the types, frequency, and complexity of outages have evolved. We now grapple with many more cloud regions than in traditional data centers. Outages include:

- **Frequent failures inside a region**
- **Temporary failures (network partitions)**
- **Permanent failures (entire persistence layer failure)**
- **Failures that can be quickly recovered from (create another instance when one fails)**

The Impossibility of Eliminating Outages

Complete elimination of outages is an unattainable goal. The focus must instead be on reducing service disruptions and failures, and, more importantly, on building robust systems that withstand failures and recover quickly.

Failure will eventually materialize given sufficient scale and exposure, so it's crucial that prevention and mitigation measures are in place.

Random Yet Repetitive Outages

Outages are characterized by seemingly random, one-off causes. These range from physical infrastructure failure to unexpected system overload during recovery. Failures may include unresponsive machines, disk failures, or even someone tripping over a network cable in a data center.
Despite having varied origins, failures are a statistical certainty in large-scale systems.

Traditional High Availability and Disaster Recovery Limitations

Standard alerting, observability, and human- or AI-powered runbooks cannot adequately address the vast, unpredictable scope of potential failures.

The pressure to quickly resolve outages, coupled with often inconvenient timing, makes reactive approaches ineffective for maintaining continuous service. Traditional approaches generally involve triaging the outage before addressing underlying causes. A recovery runbook must be executed immediately after an outage to rectify the situation.

In these cases, an outage is treated as an exception event, where "something bad has happened that results in a negative SLA/business consequence." A modern mindset is, "failure is now the norm, and protecting the business from outages is a key part of service/operation/business SLA as usual." This is what an ultra-resilient architecture offers.

The Ultra Resilience Imperative

The solution lies in embedding failure handling directly into system architecture rather than layering it on top. Services must be designed with resilience in mind.

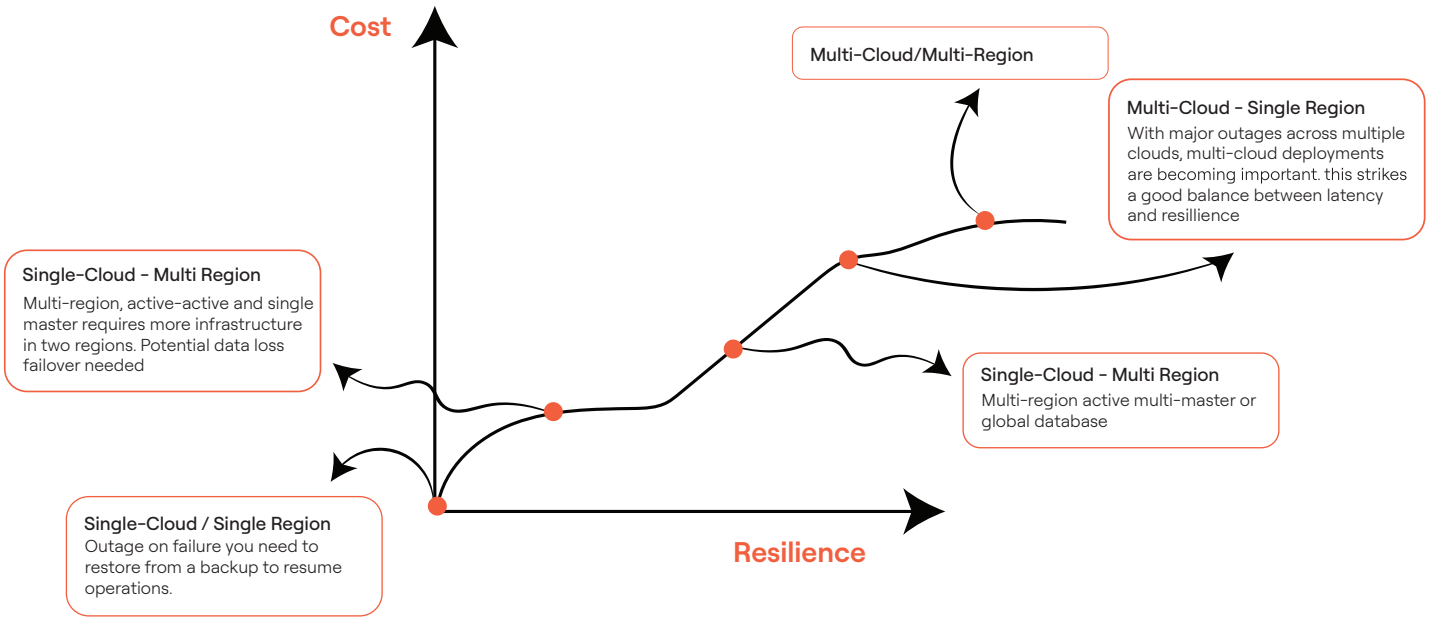
This necessitates a move towards ultra-resilient designs for business-critical services. This often includes Business Continuity and Disaster Recovery (BCDR) strategies across multi-region, multi-cloud, or hybrid environments.

The ultra-resilience approach requires different architectural choices for different failure types. For example, you may implement active-active-active deployment within a region, and use either active-active-active, active-active, or active-replica for cross-region disaster recovery. Application failover design must be dictated by business SLAs.

The Cost-Resilience Trade-off

Achieving higher levels of resilience invariably incurs greater costs. The right approach is to make overall architectural and deployment choices "ultra-resilient" rather than focusing solely on optimizing a specific configuration.

Regulatory requirements, such as country-specific multi-cloud deployments of critical services (like payments), underscore that design choices are essential investments rather than arbitrary architectural decisions.



Mismanaging or poorly understanding the cost-resilience trade-off can result in increased expenditure without effectively addressing underlying resilience issues. To pivot from expensive and time-consuming traditional resiliency approaches, you need to consider picking an architecture for ultra-resilience and standardizing design patterns after rationalizing the cost implications and resilience SLAs.

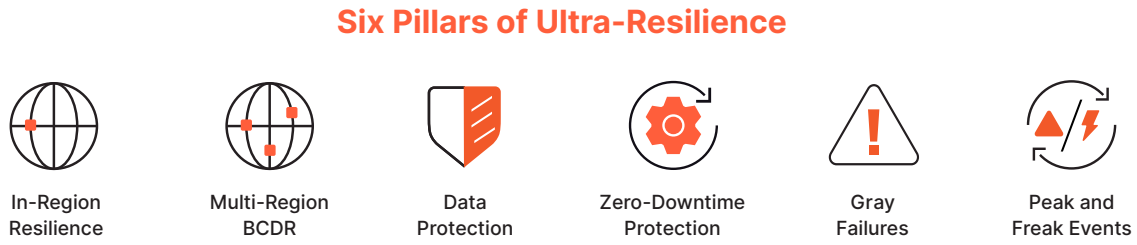
Continuous Cost-Resilience Evaluation

Organizations must constantly assess their resilience requirements against the associated costs. The need for resilience can evolve over time, particularly if business operations become significantly impacted by frequent outages.

Cloud-native distributed SQL databases are designed to help businesses dynamically balance the cost-resilience trade-off at the data layer. They offer standard PostgreSQL APIs and automated day-2 operations to simplify adoption without the need for an extensive, expensive, and time-consuming system redesign.

The Six Pillars of Ultra-Resilience

The ultimate goal of ultra-resilience is ensuring the uptime of critical services. This is not just preparing for the possibility of failure, but guaranteeing operations "no matter what happens."



1. **In-Region Resilience:** Ensuring system availability within a single region or data center.
2. **Multi-Region Business Continuity and Disaster Recovery (BCDR):** Planning for any incident and ensuring data availability across multiple regions.
3. **Data Protection:** Implementing strategies for backups, point-in-time recovery, software rollbacks, and blue/green upgrades.
4. **Zero-Downtime Operations:** Performing routine maintenance without impacting application availability.
5. **Addressing Gray Failures:** Detecting and handling subtle, intermittent issues that degrade performance. Planning for Peak and Freak Events: Scaling quickly and elastically to handle usage spikes and unforeseen disasters.

For more insights, check out the white paper "Architecting Apps for Ultra-Resilience with YugabyteDB."

Introducing Ultra-Resilient Design Patterns

In an interconnected world, every second of downtime can result in significant losses. This makes high availability and resilience crucial for any application.

Many enterprises running mission-critical systems on a legacy two-data-center model are now opting for multiple availability zones (AZ) and cloud-native architecture. This shift is further driven by the need to meet evolving data residency laws and to improve system resilience.

As data storage and processing regulations become more stringent and localized, a distributed architecture ensures compliance by ensuring data resides closer to its origin. While two data centers offer some redundancy, a 3-AZ (availability zones) model provides greater fault tolerance, minimizing the impact of localized outages.

In section, we examine design patterns that are crucial to achieving modern, highly available infrastructure. Discover the benefits, methodologies, and practical applications of utilizing these patterns to enhance resilience and scalability in global systems.

The transition from a centralized data center model to a 3-AZ or globally distributed architecture is a significant undertaking. This move is not merely an upgrade; it's a fundamental rethink of how core applications and data are deployed and managed.

The In-Region Resilience Gold Standard: Active-Active-Active

Making an informed architectural choice requires:

- **Understanding your application needs**
- **Categorizing applications based on uptime requirements**
- **Knowing how to manage database downtime**

You may have one or more of these application types, with various levels of downtime acceptance:

Mission-critical Applications: With a near-100% uptime requirement, these business-critical applications include mobile banking, payments, and e-commerce order processing, where just seconds of downtime directly impact revenue. These use cases often employ a cache to buffer database downtime and ensure continuous operation.

Minimal Downtime Applications: Applications like accounts payable or payroll processing are not mission-critical but still require high uptime (99.999%). One option is to use an event queue between the application and data storage to buffer database downtime. In other situations, companies will use something like a Redis cache for hot data and persistent SQL for all data.

Acceptable Downtime Applications: Some applications, like administration and internal business operations, have a higher tolerance for downtime. These generally do not implement a cache or queue layer between the application and the database.

However, there are better options available when designing with cloud-native, distributed SQL databases like YugabyteDB.

Modernizing to a cloud-native architecture involves several key considerations:

- **In-Region Resilience and Availability:** This focuses on ensuring the application can withstand failures within a specific geographical region, including disk/node/zone failures, network partitions, and back-end process failures.
- **Data Protection and Disaster Recovery (DR):** This encompasses strategies for handling region or cloud failures, implementing robust backups, and ensuring point-in-time recovery (PITR).
- **Operational Simplicity:** Streamlining operations through automated software upgrades, machine/SKU/AMI refreshes, and the ability to dynamically change zones/regions.
- **Developer Productivity/Application Complexity:** Optimizing for ease of development and reducing application complexity through clear read/write endpoints, simplified schema changes, and managing upstream dependencies.

"The growing adoption of cloud applications is generating massive amounts of data around user access. We need a database that can deliver the scale, performance, and cloud native capabilities required for our data infrastructure to meet the growing needs.

The core capabilities of YugabyteDB allow us to easily scale and improve our reliability and performance without the tradeoffs of time and effort required by legacy solutions."

- Jake Roersma, Netskope Vice President of Platform Engineering

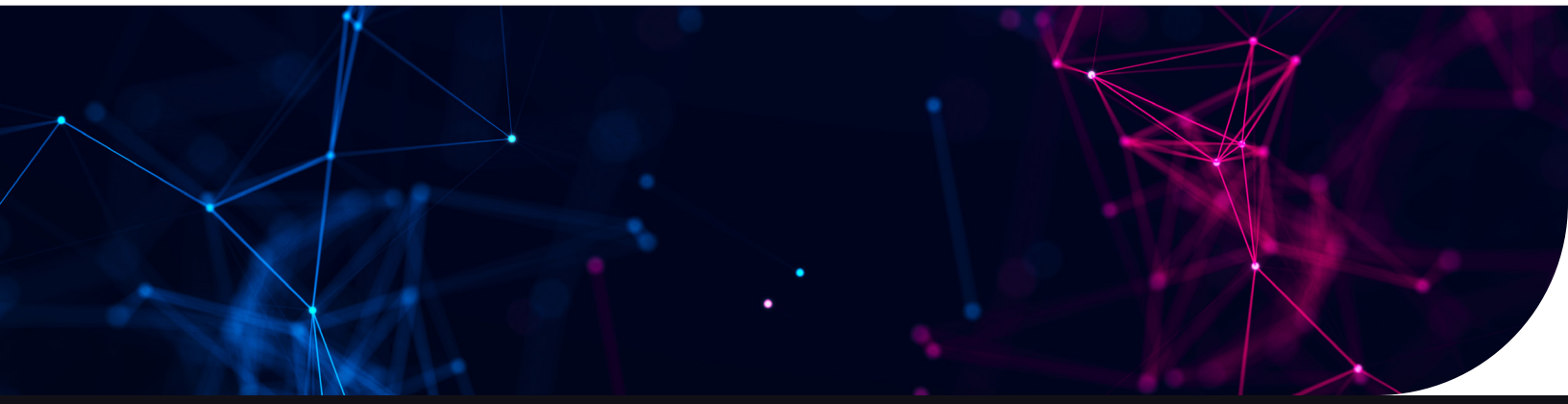
Seven Essential Active-Active-Active Design Patterns

Active-Active-Active design patterns are crucial for achieving high availability and scalability in distributed systems across multiple zones or regions.

This approach ensures that all instances are actively handling traffic, minimizing downtime and enhancing overall system resilience. This pattern is particularly beneficial in distributed YugabyteDB deployments, as it supports both synchronous and asynchronous replication across different regions.

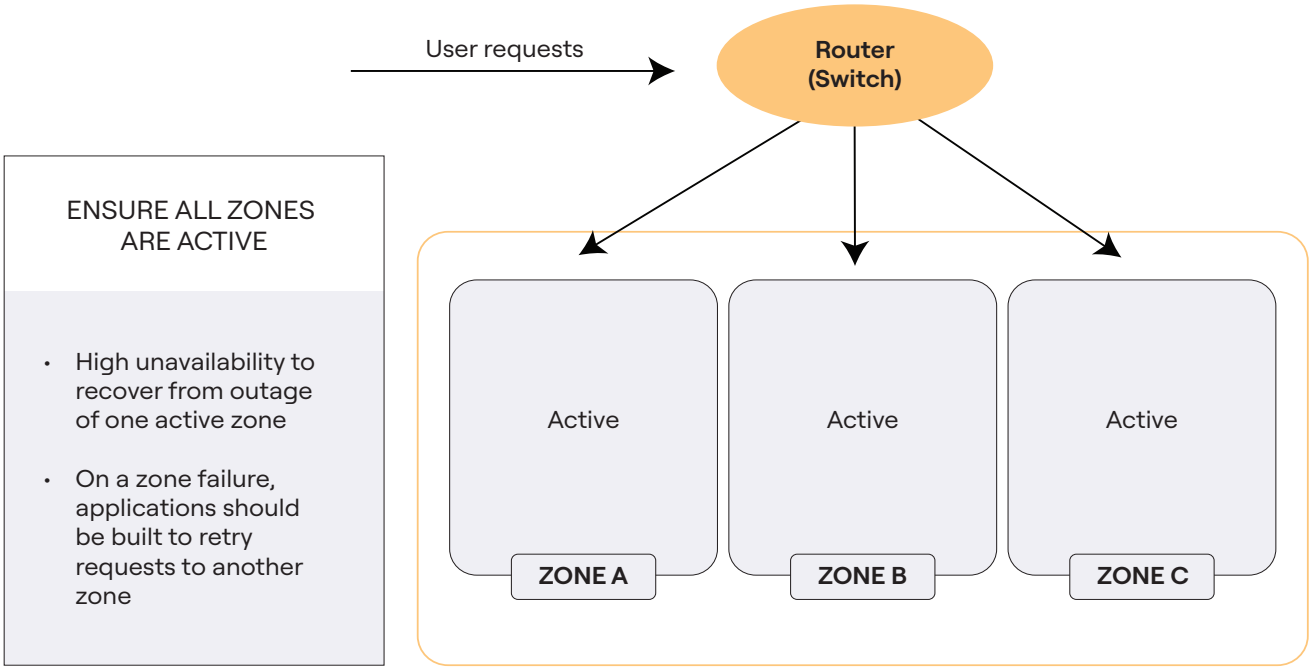
The main goal of Active-Active-Active design patterns is to ensure that applications remain operational even in the event of failure. Unlike traditional single-master or standby designs, where only one node is active at a time, all nodes in an Active-Active-Active setup serve traffic simultaneously.

This approach is useful for cloud-native applications, where node failures are common and must be quickly recovered from. To achieve true in-region resilience, consider the following recommendations while keeping the cost-resilience trade-off in mind.



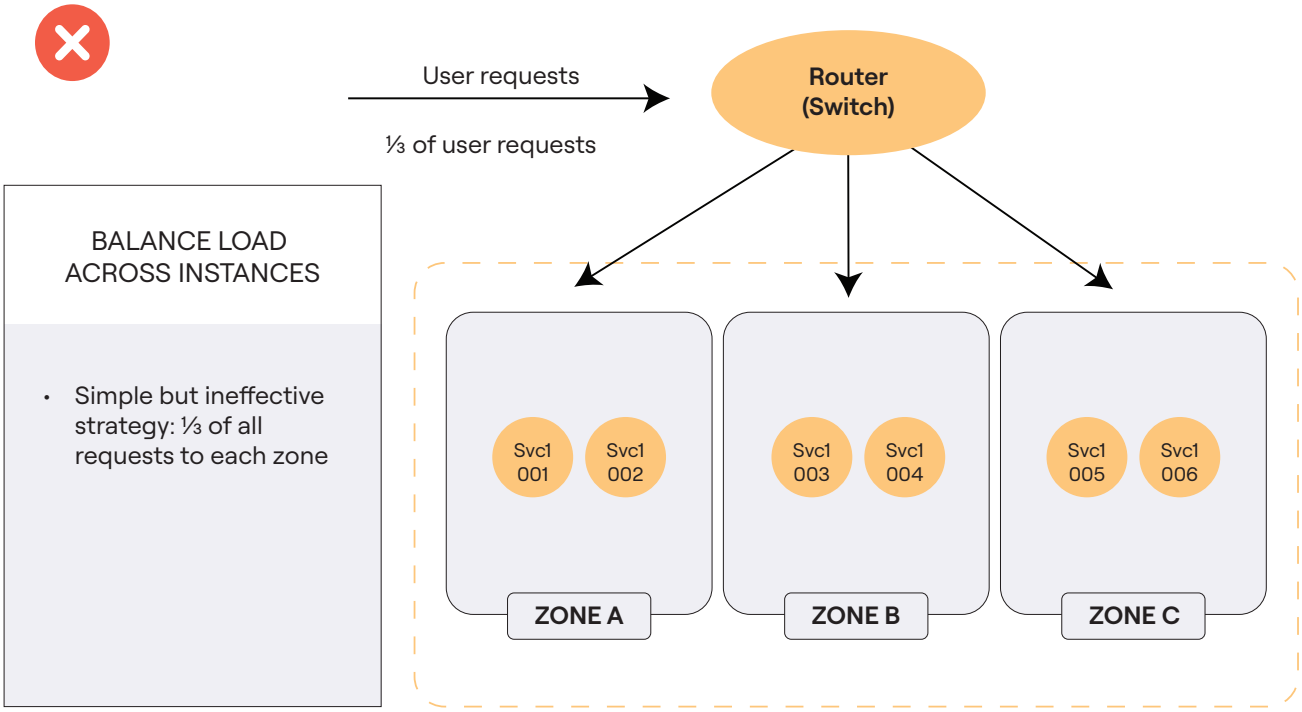
Ensure Active-Active-Active Architecture Across Zones

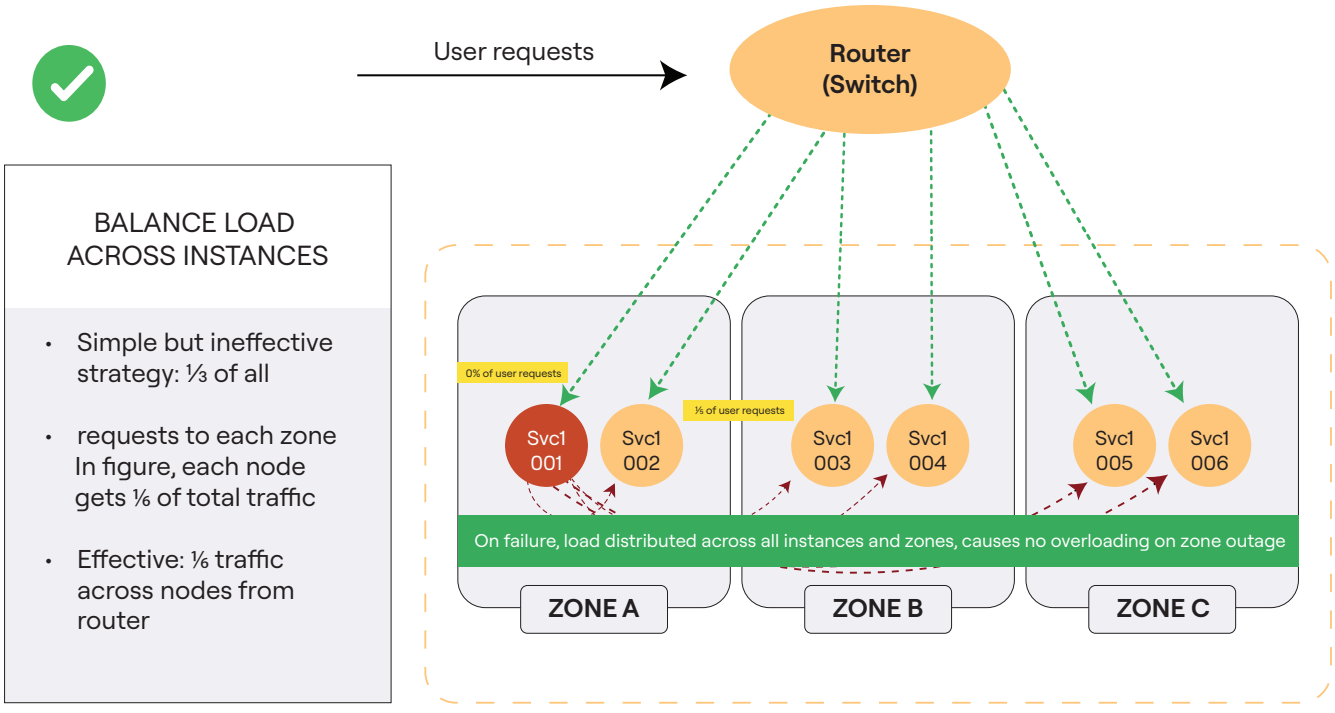
All services in the stack should be active in all zones and load-balanced across instances, not just within zones.



Load Balance Across Instances, Not Zones

Distribute requests across individual instances rather than entire zones to prevent overload in the event of a zone outage. If traffic is routed to zones, and one zone fails, the remaining zones will experience a 100% increase in load, leading to cascading failures. Load balancing across instances ensures that if an instance or zone fails, the load is efficiently redistributed across the remaining active instances and zones to prevent overloading.





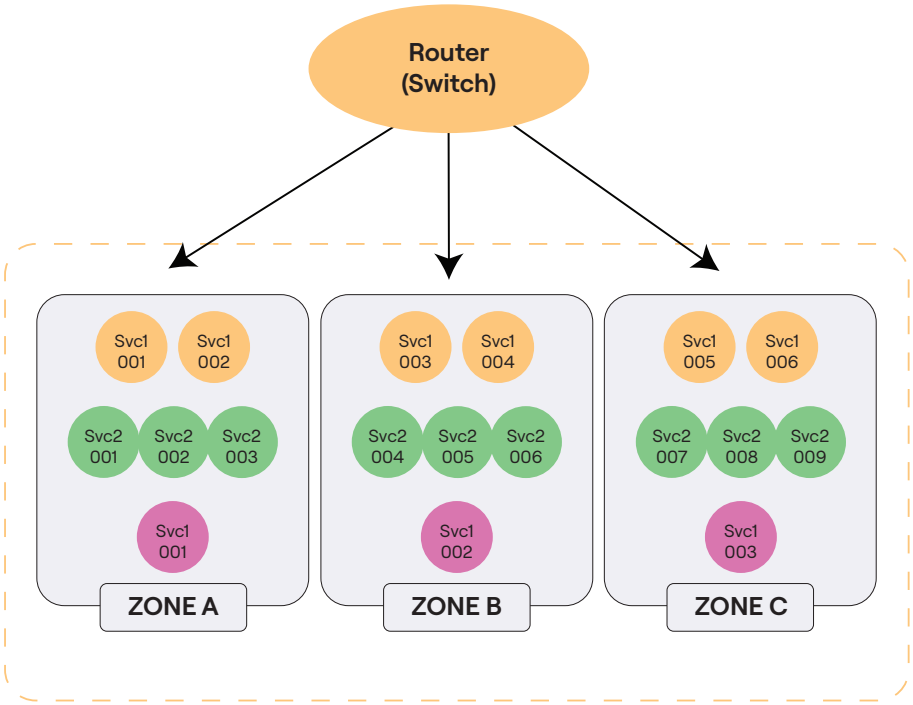
Zones Should Not Be Treated as Silos

Avoid architectures that handle requests within a single zone without cross-zone routing. This can result in increased unavailability and data inconsistency if a service within that zone fails. Instead, architect services as logical entities that span all zones, enabling per-instance routing across zones at each service layer for high availability.

APPLICATIONS IS STACK OF SERVICES

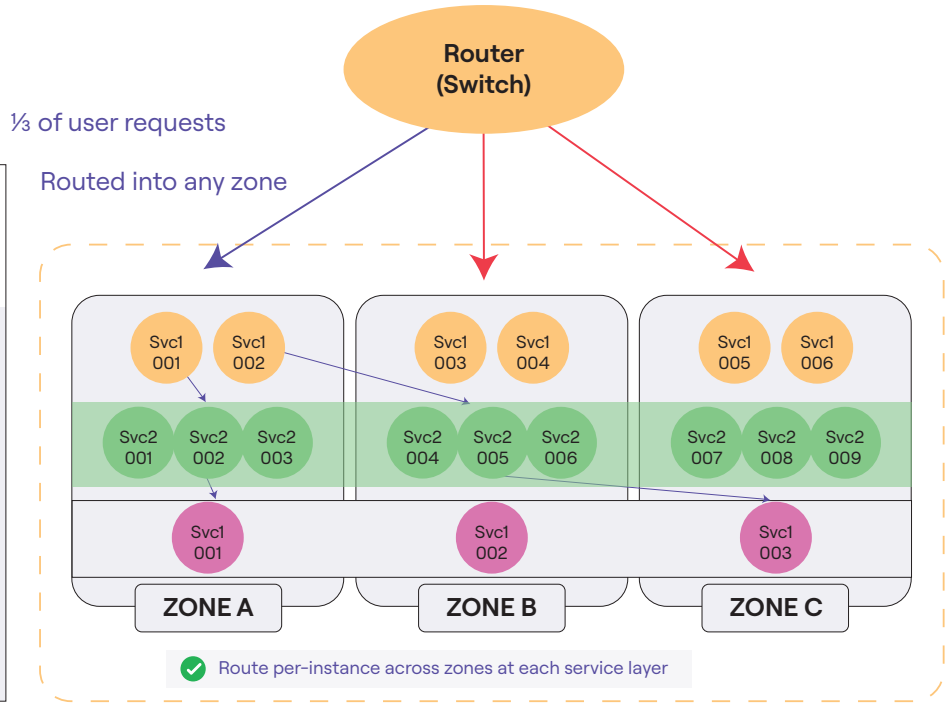
- Examples of services:
1. Queues
2. Message busses
3. App servers
4. Caches
5. Databases

- Service layers depend on each other
- Form "stack" of dependencies



Route across zones for each service

- Let's redo that scenario by routing the request across zones
- No availability or consistency issues
- If done right, each service layer can keep requests in zone as best effort



Avoid Bespoke Architectures and Component Misuses

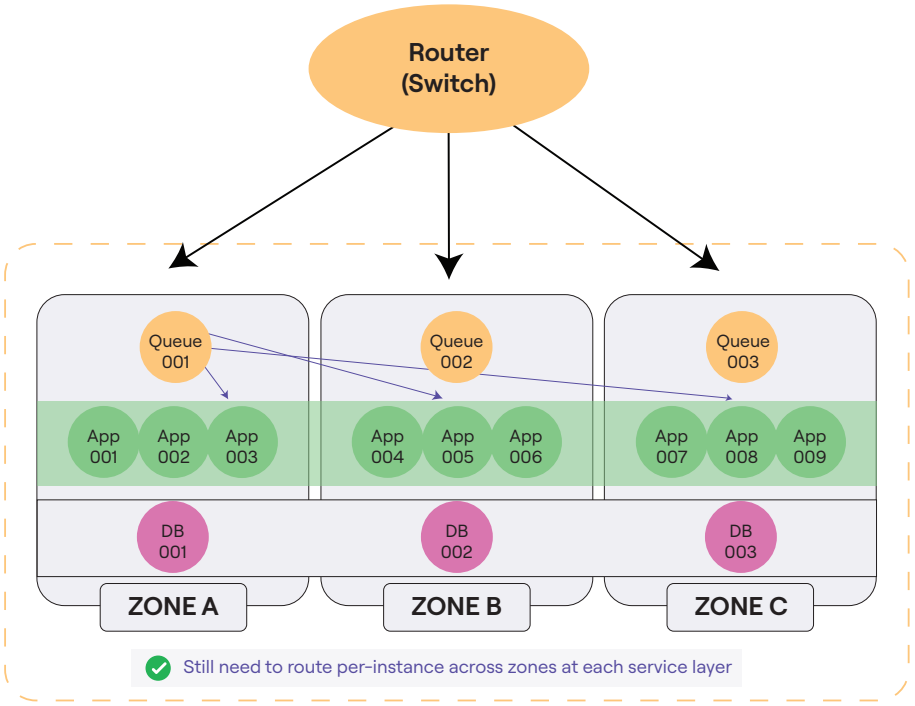
Message Queues: While useful for ordering log data and pub-sub notifications, queues are not a database replacement. They lack critical features like indexes, check constraints, foreign keys, and transactions. This makes it difficult to build flexible applications or enforce data integrity.

Caches: Caches introduce significant complexity, particularly when maintaining cache-database consistency for transactional workloads. Choosing the right caching strategy, handling invalidation, and ensuring data durability are challenging tasks. If a database is scalable, a cache may not be necessary solely to protect I/O operations.

⚠ Message Queues

MSG QUEUES AS ENTRY POINT TO ZONE

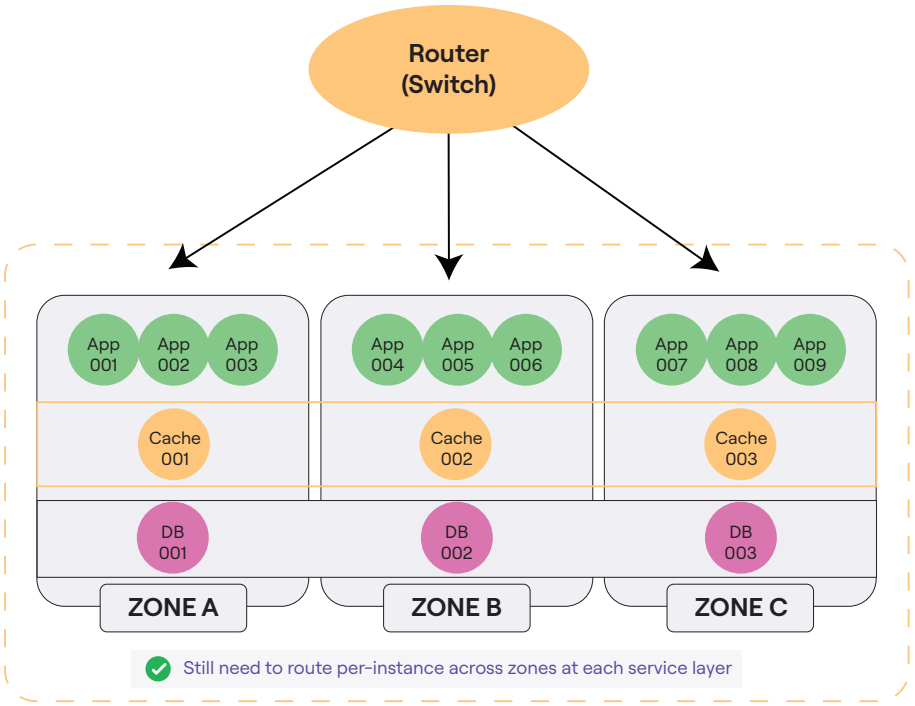
- Sometimes end users need low latency operations but transaction processing has high latency
- App may add transaction to queue and process transaction async
- But not for HA or scale



⚠ Caches

MSG QUEUES AS ENTRY POINT TO ZONE

- Sometimes end users need low latency lookups on the same key over and over
- A cache is a good option for these cases
- However caches introduce high level of complexity



Plan for Short-Term Failures

Ensure automatic and efficient handling of short-duration failures. If recovery takes too long, data availability can be compromised, and the system remains vulnerable to subsequent failures during the recovery window.

Plan for Long-Term Failures

For persistent failures, such as machine replacements, automatic re-replication is critical to restore resilience. Without it, the system is at risk of data inconsistency or unavailability. The system should automatically redistribute data and processing to the remaining nodes.

Identify Global Application Patterns and Architect From the Ground Up

Proactive designs aligned with global application needs are essential for building robust and resilient systems. Active-active-active consideration is not just for regional resilience. It also impacts how you design your operational continuity, such as zero-downtime upgrades or key rotations.

Multi-Region Deployment BCDR Topologies

How to Design Global Applications

As There are many critical decisions to consider when designing global applications.
The three key requirements are:

Architecture

App deployment location and geo-location capabilities

Availability

App behavior in the event of failure and the scope of data operations

Data Access

Read/write instances and tolerance for stale reads

Application Architecture Options

Single-Active
In a single-active architecture, application instances are operational in a single designated region (e.g., 'us-central'). The associated data is co-located with the application within that same region. Applications deployed in other regions remain passive and serve as standby instances, ready to fail over in the event of a primary region failure.

Multi-Active
In a multi-active architecture, all application instances are active. These instances perform both reads and writes, but are all directed to the leader, which serves as the single source of truth.

Read-Only Multi-Active
In a read-only multi-active setup, applications are active in a single designated region (e.g., 'us-central') while applications in other regions are configured to perform reads directly from local followers. A key trade-off of this approach is that read data may be stale, as followers might not be fully up to date with the leader.

Availability Architecture

Follow the Workload
In follow the workload availability architecture, data is intentionally placed close to its respective applications. This means the application and its associated data are designed to move together in the event of a failure. For example, if the US-Central region fails, the application in another region becomes active, and the data replicas in that new active region are promoted to leaders.

Geo-Local Dataset
In geo-local dataset architecture, all application instances are active across multiple regions. They operate only on a subset of data local to a specific geographic region (for example, US user data in the US and European user data in the EU). In the event of a failure, local standby instances become active, and their local replicas are promoted to leaders. This approach is particularly useful for enforcing data-residency laws, such as GDPR.

Data Access Architecture

Consistent Reads
In a consistent reads architecture, all application instances read from the source of truth, meaning they read only from the leaders. A consequence of this approach is that applications running in other regions will incur cross-region latency.

Follower Reads
In follower reads, applications are configured to read directly from follower replicas instead of the leader. While this approach dramatically reduces read latencies by allowing applications to read locally, the data may be stale because follower replicas might not be fully up to date with the leader.

Read Replicas
Read replicas are similar to followers but are replicated asynchronously. Their location has no impact on write operations on the primary cluster. For example, you can maintain your primary data in a region such as the USA, but set up a read replica cluster in Europe to achieve lower read latencies for local users. While this allows for very fast, local reads, the trade-off is that read data might be stale, as replicas may not be fully up to date with the primary data.

Take a holistic view of your SLAs when designing a multi-region application. In addition to disaster recovery:

- **Is latency critical for your application?**
- **How about compliance?**
- **Will there be more considerations as your application scales?**

It is neither practical nor cost-efficient to use three different architectures to deliver three SLA outcomes (for example, disaster recovery, latency, and compliance). Considerations need to be built into the same architecture, and determine where you want to be on the cost-resilience curve.

Beyond in-region resilience and global multi-region BCDR, your design considerations may include managing peak/freak events and gray failures through auto-sharding and elastic scalability, as well as ensuring data protection with point-in-time recovery requirements.

The Architecture-Pattern Matrix

We have mapped the various concepts observed in the application, availability, and data access architectures to the design patterns as depicted in the matrix below:

		Availability	
		Follow the workload	**Geo-local dataset**
Application	**Single Active**	• Global database • Active-active single master	N/A (app active in only one geo)
	Multi Active	• Global database • Duplicate indexes	• Global database • Active-active multi master
	Partitioned Multi Active	• Latency-optimized geo-partitioning	• Locality-optimized geo-partitioning
Data	**Data Access Architecture**	• **Consistent reads** • **Follower reads** • **Read from the nearest region** • **Read replicas**	

Table Definitions:

1. **Global Database:** A single, globally distributed database.
2. **Duplicate Indexes:** Maintaining redundant indexes for faster data retrieval.
3. **Active-Active Single-Master:** Multiple active instances with a single master for writes.
4. **Active-Active Multi-Master:** Multiple active instances, all capable of handling writes.
5. **Latency-Optimized Geo-Partitioning:** Data is partitioned geographically to minimize latency for users in specific regions.
6. **Locality-Optimized Geo-Partitioning:** Data is stored in specific geographic areas to meet data residency requirements.
7. **Follower Reads:** Reads are directed to follower replicas to offload the master.
8. **Read Replicas:** Dedicated replicas for read operations.

To use the above, consider an application that requires instances to be active in a single region that must failover with its data: a **"single-active"** application architecture combined with the **"follow the workload"** availability architecture.

In this scenario, you could choose the **"global database"** or **"active-active single master"** design patterns. If an application requires data to be partitioned and localized to a specific geographic area, the locality-optimized geo-partitioning pattern is suitable. The **data access architecture** patterns are applicable to both availability models.

These patterns offer different availability, consistency, and performance trade-offs. Traditional single-node RDBMS systems typically support active/passive and standby concepts, which are similar to the active-active single master and multi-master patterns previously detailed. However, implementing design patterns such as read replicas can be more challenging with these traditional databases.

In the next section, we examine additional patterns, including global database, duplicate indexes, latency-optimized geo-partitioning, locality-optimized geo-partitioning, follower reads, and read replicas.

Global Database (Single-Active, Follow the Workload, Geo-Local)

In a global database pattern, applications are active in only one region, and the application, along with its workload, moves to another region upon failover.

For example, consider deploying the database across three regions:

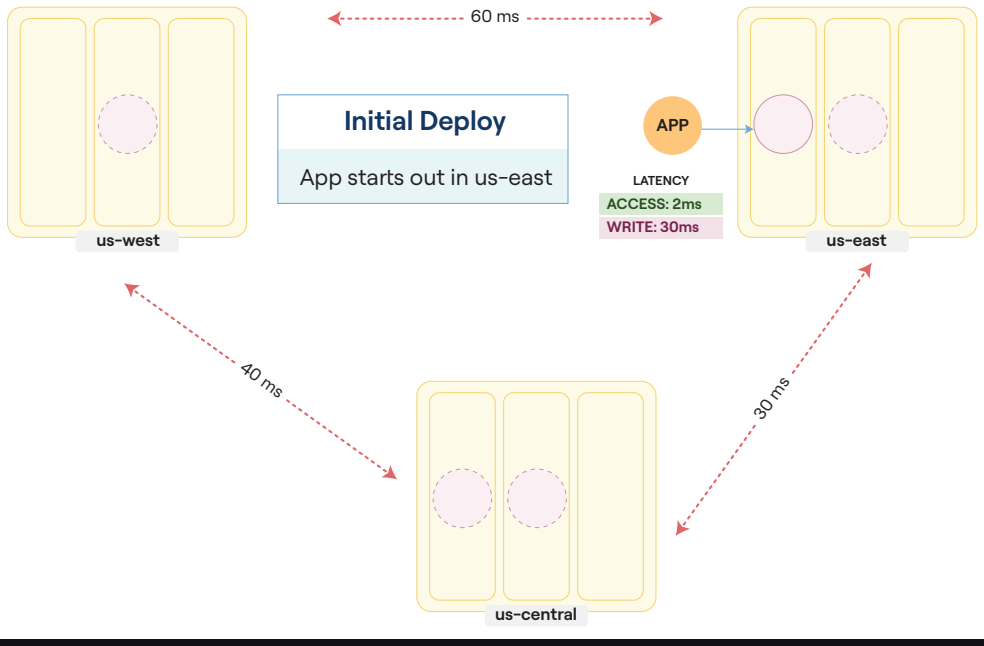
- **us-west**
- **us-central**
- **us-east**

If the application is running in us-east and configured to failover to us-central during an outage, you would set up your leaders in us-east, placing two copies there and two copies in us-central. This results in a Replication Factor of five (RF5).

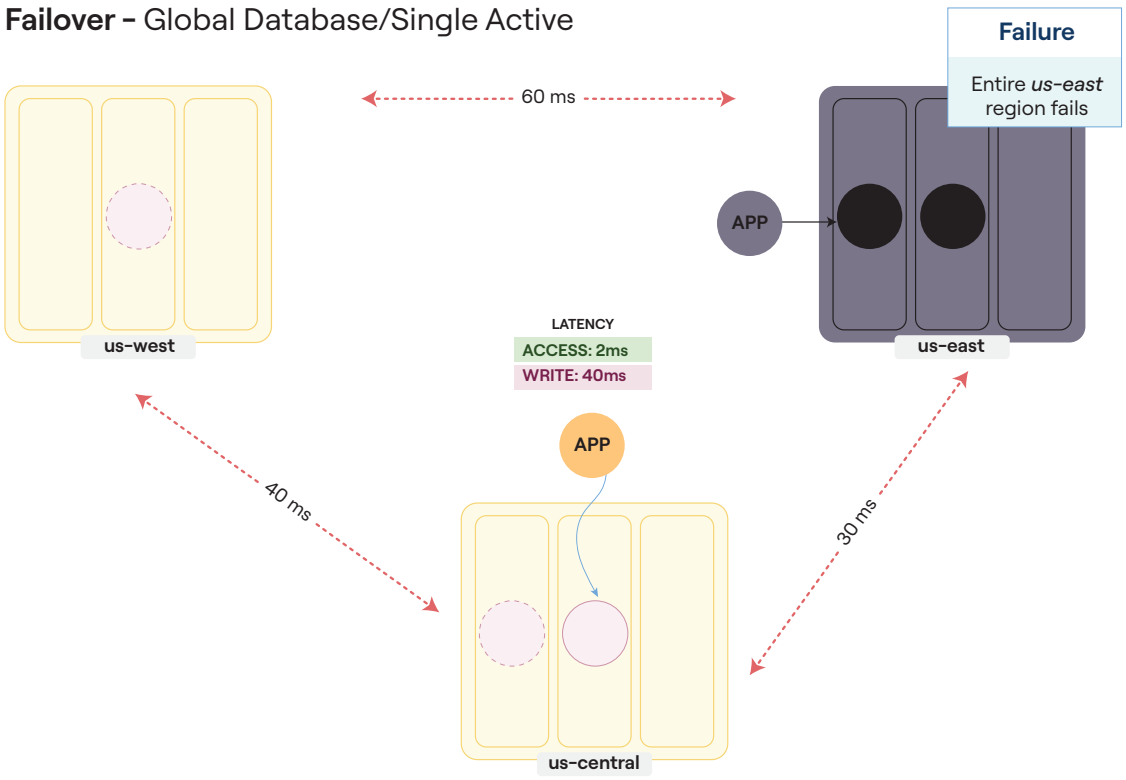
A key advantage of RF5 is having a replica in the same region. This is beneficial for rolling software upgrades and node swap-outs, as it doesn't impact the application if a local node goes down.

Initially, the read latency is low (around 2ms). However, the write latency is about 30 ms. This is because the write must be replicated to at least two replicas, with the second closest replica being 30 ms away in us-central.

App in East – Global Database/Single Active



When an entire region fails, the application moves to us-central, along with the database leaders. This increases write latency by 10 ms (to 40 ms) as a result of the greater latency between the west and central regions.



This latency increase can be mitigated by choosing closer regions during deployment; for instance, choosing east-1 and east-2 can reduce write latency to 10 ms.

When to Choose Global Database Single Active

The global database - single active pattern is a standard deployment best suited when most users are located near a single region (though it can accommodate distributed users by setting preferred regions). In this architecture, applications are active in only one region and utilize replicas (with a recommended replication factor of 5) for high availability and automatic failover with zero data loss.

This pattern offers low read latency, but write latency can be higher. This can be minimized by selecting closer regions for deployment. Preferred regions should be set to match the failover plan.

Duplicate Indexes for Multi-Active Applications (Multi-Active, Follow the Workload)

For the duplicate indexes for multi-active applications pattern, a cluster is set up across multiple regions (e.g. west, central, and east), with preferred leaders initially set in one region (e.g., east) where the application starts.

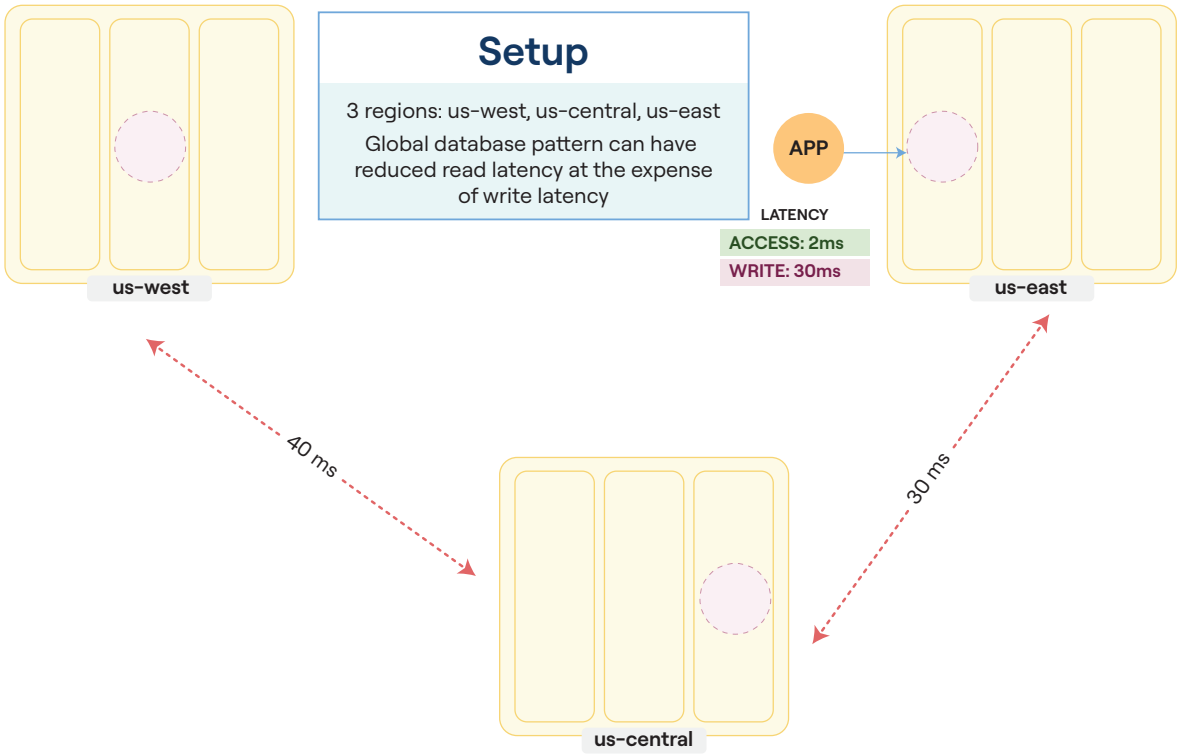
Duplicate indexes, which are essentially identical to the main table, are created for each region, and their leaders are set locally. This optimization dramatically reduces read latency in all regions to a very low level (e.g., 2ms), achieving the goal of low read latency everywhere.

However, this result comes at the expense of high write latency across all regions. This is because:

- **Every write must update the main table**
- **It must be replicated to its followers**
- **Each regional duplicate index must also be updated and replicated to its respective followers**

The core benefit of this pattern is enabling consistent reads in all regions.

Setup – Duplicate Indexes



When to Choose Duplicate Indexes

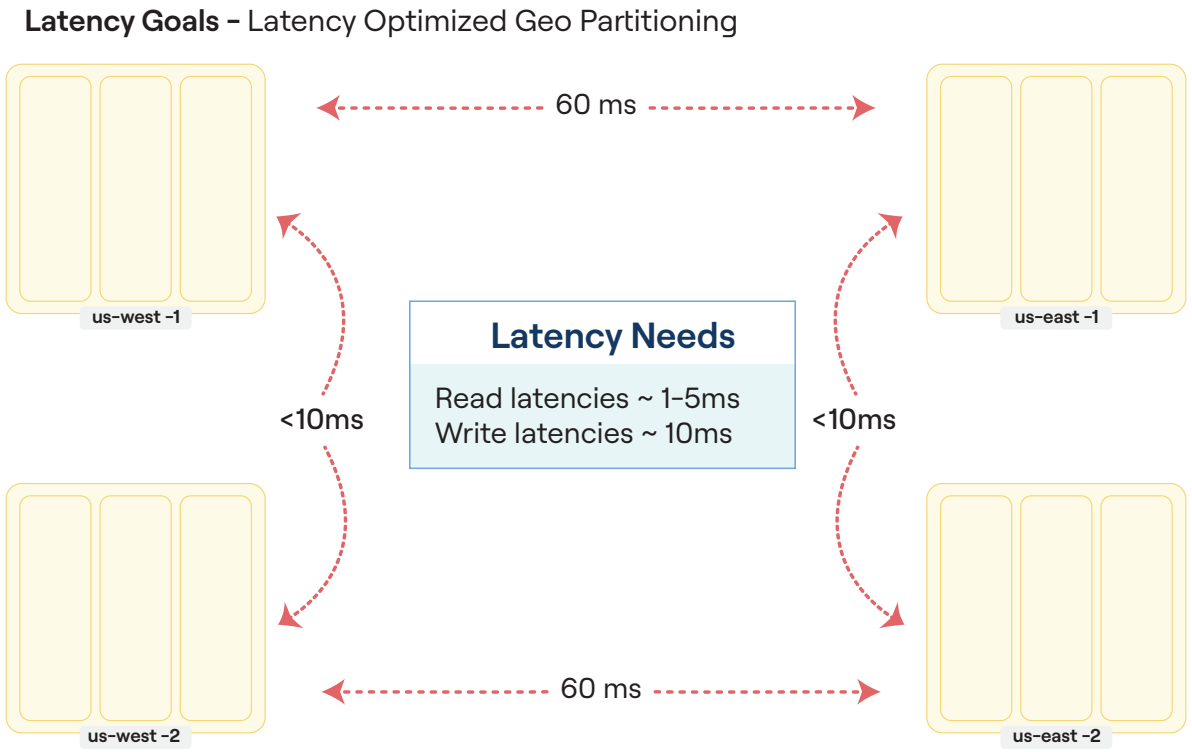
This pattern is best suited for applications that are active in all regions and require consistent reads everywhere. Example use cases are reference tables that don't often change, or quicker lookups of user information. You achieve low-latency reads across all regions by having one duplicate index per region, but this comes at the expense of high write latency.

Latency Optimized Geo Partitioning (Partitioned Multi-Active With Follow the Workload)

In the latency-optimized geo-partitioning pattern, applications are active across different geographical regions and strategically placed near the data subsets they operate on to improve user latency.

For a scenario with users in us-west and us-east, the database is partitioned into an 'east' partition and a 'west' partition, with user data stored locally (e.g., east users in us-east). To achieve low latency, two regions close to each other are selected within each geo. For the west partition, leaders are placed in one west region, and a replica is placed in a second, closer west region. This setup ensures very fast writes due to quorum from nearby regions. A third replica is placed in the east, allowing east applications to perform fast, albeit stale, reads. The same is done for the east partition.

This design results in low read and low write latencies for local users. It requires configuring DNS or load balancers to correctly direct user traffic to the appropriate geo-partition.



When to Choose Latency Optimized Geo Partitioning

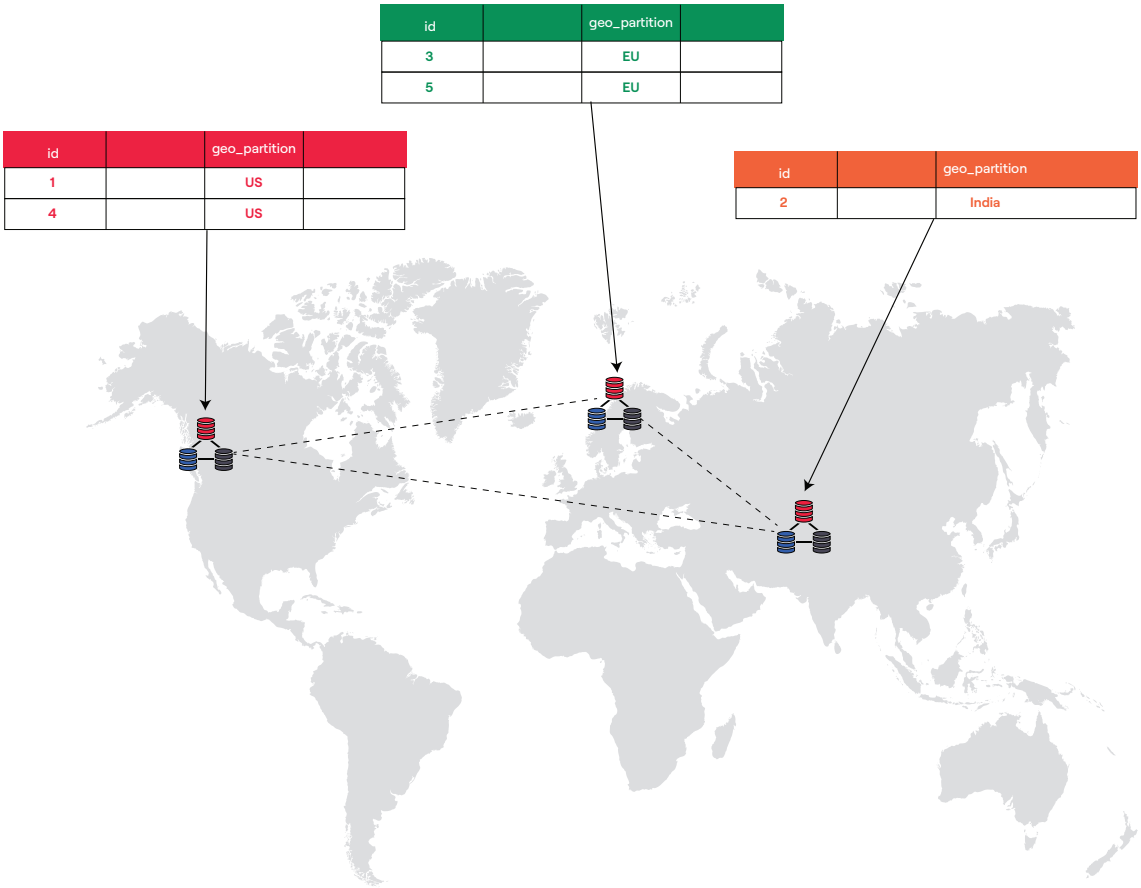
This pattern is ideal for applications with users across multiple geographies that require faster access for local users.

The applications must be active in all regions, and both the database and the application should be partitioned, with one table partition per geographic region. The replication factor can be 3 or 5, and it is advisable to use two regions per geo to achieve low-latency reads and lower write latency with closer regions.

Examples of suitable workloads include ecommerce shopping carts and financial data.

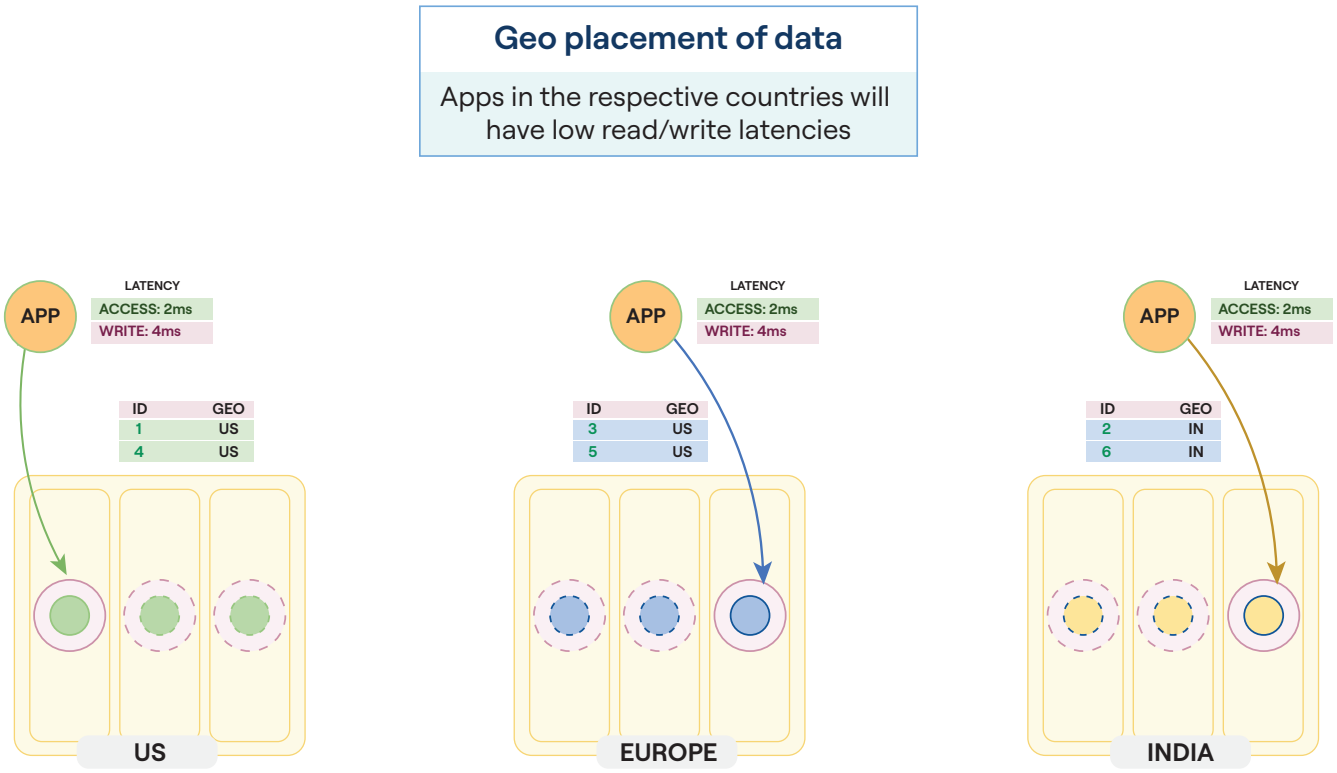
Locality Optimized Geo Partitioning (Partitioned Multi-Active / Geo-Local Dataset)

It is important to ensure you implement partitions in accordance with local laws, such as the GDPR. In the l**ocality optimized geo partitioning** pattern, applications are active in different regions and operate on different subsets of data.



In locality optimized geo partitioning, applications are active in different regions and operate only on a subset of data specific to the geographic area.

This pattern is designed to comply with local data-residency laws, such as GDPR. For example, all rows related to US users are placed in the US, Indian users in India, and European users in the EU.



The database is partitioned, and all replicas, including both leaders and followers, are placed within their respective geographies to meet local regulations.

When to Choose Locality Optimized Geo Partitioning

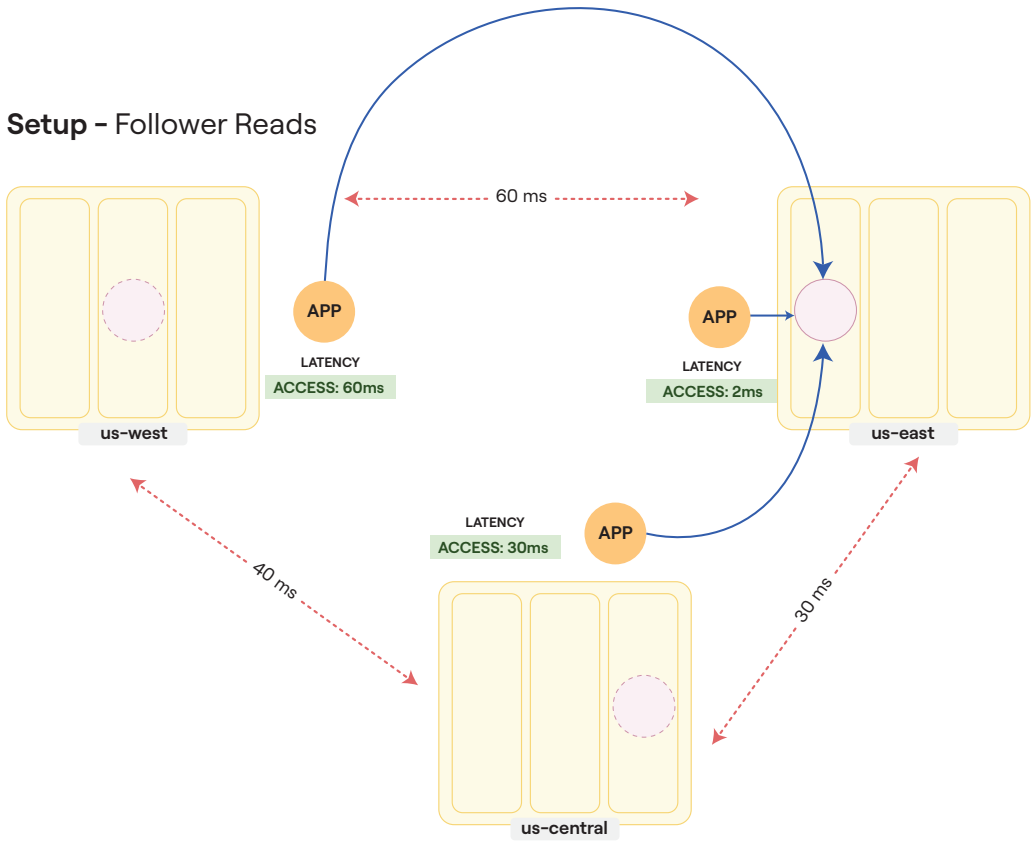
This pattern is designed for applications that must comply with local privacy laws, such as GDPR, LGPD, and PIPL, by storing data in specific geographic areas (geo-localization).

It is suitable for applications that have a mix of global and localized tables (e.g., global product catalog, local user info, and orders). The architecture requires applications to be active in all regions, with both the database and the application partitioned, with one table partition per geo. A key benefit is achieving low latency reads and writes.

Improving Performance Through a Customized Data Access Architecture

Follower Reads

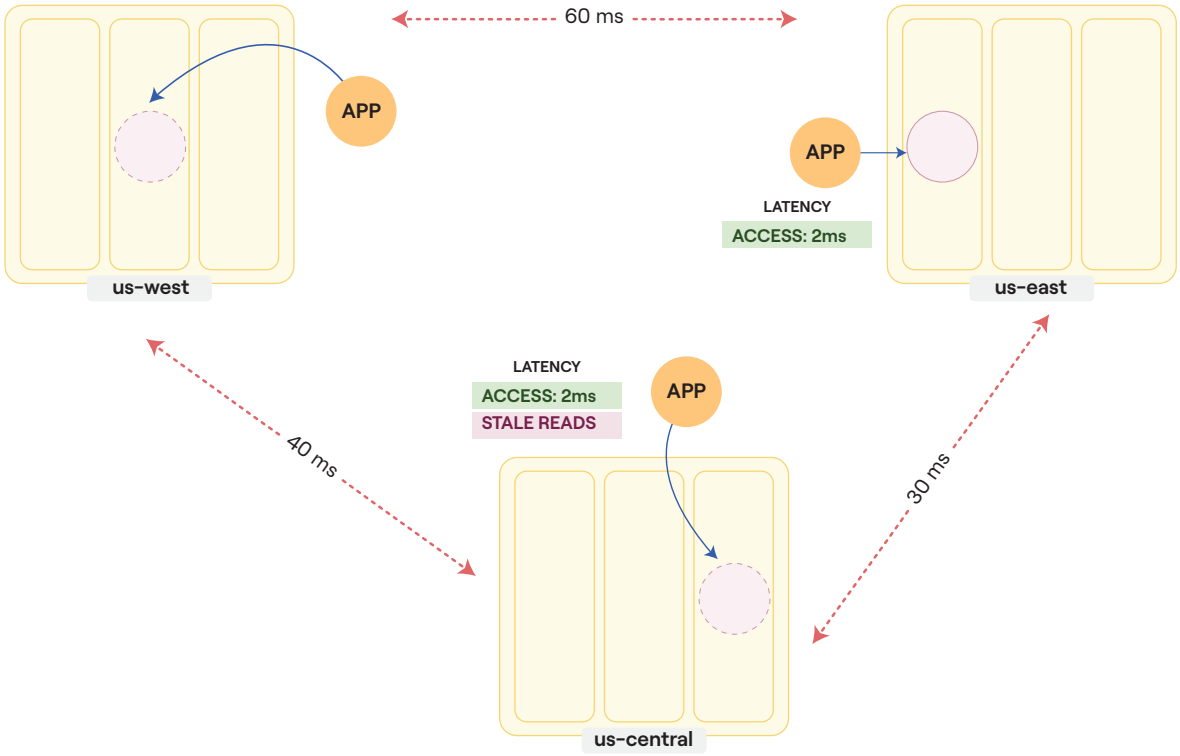
As the name suggests, the application will read from followers rather than leaders.



In follower reads, applications are configured to read directly from follower replicas rather than the leaders.

For example, imagine a global database spread across regions like us-west, central, and east, with leaders in east. Assume that you have applications deployed in all the regions. For reads, each app must go to the leader in us-east. The west app's read latency is ~60ms.

Low Read Latency – Followers Reads



When apps read from locally available followers, read latencies are drastically reduced (2ms across all regions). However, the trade-off is that the read data will be stale.

In the event of a zone or node failure, the application automatically performs a follower read from the next-closest region, where read latencies are still significantly better than performing a full round-trip to the leader.

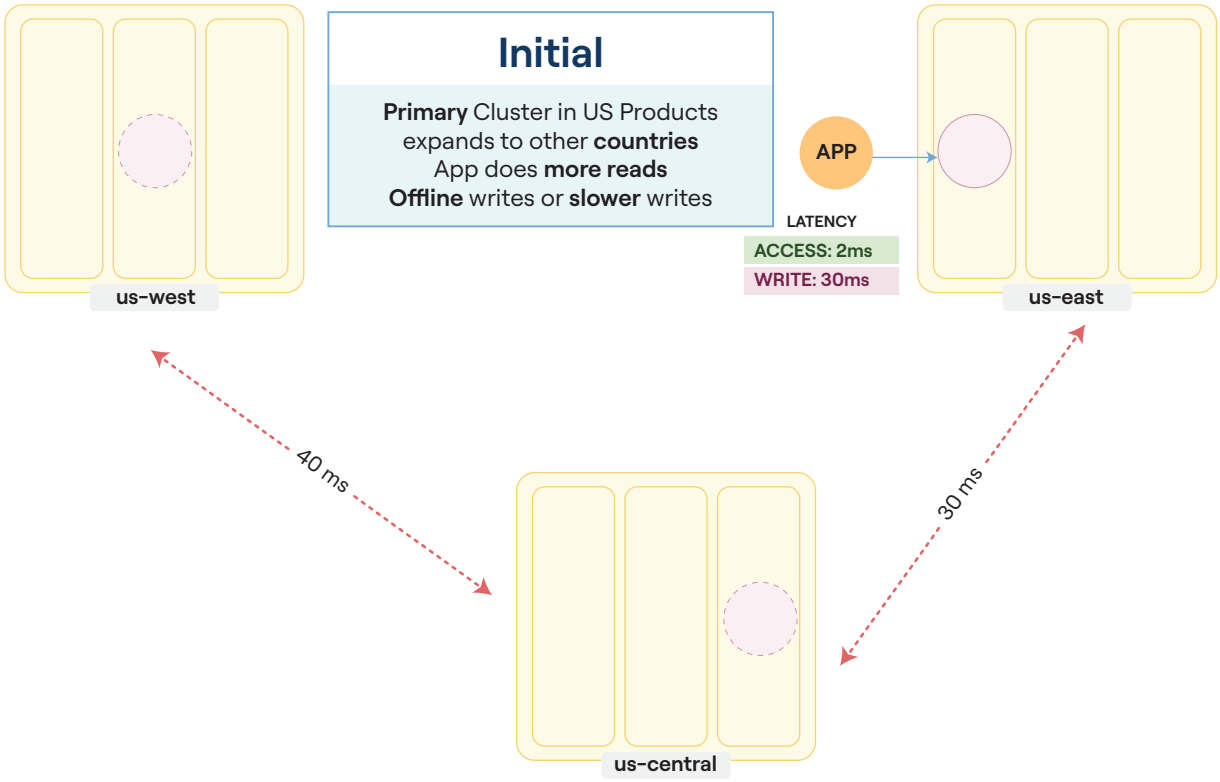
When to Choose Follower Reads

This pattern is suitable for applications that are active across all regions and require low-latency reads, even if the latest data is not required. This makes it ideal for use cases such as offline operations (e.g., report generation) or for data that changes infrequently (e.g., a movie database or book catalog). A key advantage of this pattern is that writes are unaffected by the read approach, as they still go to the leaders.

Read Replicas

Read Replica is a data access pattern that can be applied to both follow the workload and geo-local availability needs.

Primary Cluster – Read Replicas



A read replica is a data access pattern that can be used to follow both workload and geo-locality availability needs. Consider a product initially used in the US with a standard global database cluster, where reads are much more frequent than writes (e.g., a banking app with slow transactions but fast history viewing).

When expanding to a distant region like Europe, users initially face high read latency because they must access the US cluster. To provide fast local reads for European users without incurring the high cost of setting up a separate cluster, you can opt for a read replica. This approach reduces latency for US users during a region failure by moving followers or by prematurely partitioning the data. A read replica is similar to a follower but is replicated asynchronously, and crucially, does not participate in the RAFT consensus or leader election. Consequently, its replication factor can differ from the primary cluster (and may be an even number).

While its location has no impact on write operations, the trade-off is that read data can be stale because replicas may not be fully up-to-date with the primary data.

When to Choose Read Replicas

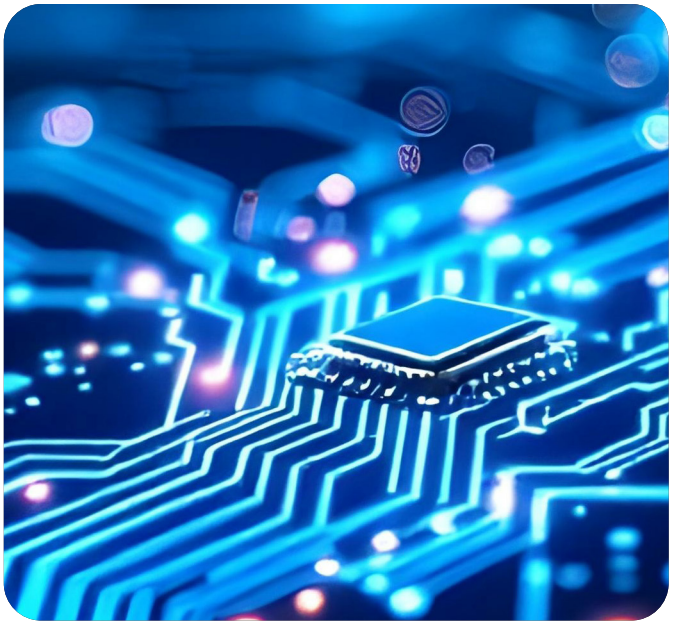
This pattern is ideal for applications that are active in all regions and need to provide low-latency reads for users in faraway regions (where setting up a full cluster would be expensive or challenging).

It is best suited for scenarios where slow writes are acceptable, and the application's read operations significantly outnumber writes (e.g., a banking app's fast history viewing versus slow transactions, or fast local viewing of a news/social feed).

A key advantage of this pattern is that it has no effect on write operations (writes still go to the leaders), and the replica can have a separate replication factor from the primary cluster. However, the trade-off is that the reads are stale due to the asynchronous replication.

Global applications are a necessity in the modern era. While managing applications across multiple regions is a non-trivial task, it can be simplified and accelerate development if you adopt clearly understood design patterns and ensure robust failover scenarios.

This process can be made easier with YugabyteDB.



Real-World Use Cases

Global E-commerce Platforms

These platforms require high availability and low latency to ensure seamless user experiences across different regions. An active-active-active setup with YugabyteDB ensures that database operations are distributed efficiently. Here is a possible design pattern for a global e-commerce platform.

		Availability	
		Follow the workload	**Geo-local dataset**
Application	**Single Active**	• Global database • Active-active single master	N/A (app active in only one geo)
	Multi Active	• Global database • Duplicate indexes	• Global database • Active-active multi master
	Partitioned Multi Active	• Latency-optimized geo-partitioning	• Locality-optimized geo-partitioning
Data	**Data Access Architecture**	• **Consistent reads** • **Follower reads** • **Read from the nearest region** • **Read replicas**	

Read more about YugabyteDB's retail and ecommerce capabilities.

Financial Services

Banks and financial institutions need robust systems that can handle massive volumes of transactions without downtime. Deploying an active-active-active pattern with YugabyteDB helps maintain data consistency and availability.

There are various design patterns that a financial services banking platform can deploy. A multi-active, geo-local dataset, global database is suitable when you require zero data loss in case of regional failure, but you can tolerate some latency when it occurs. Multi-active, active-active multi-master is suitable for predictable performance and latency, even in the case of failure, though there may be some data loss. These are examples - specific business needs will determine the most suitable design pattern.

		Availability	
		Follow the workload	**Geo-local dataset**
Application	**Single Active**	• Global database • Active-active single master	N/A (app active in only one geo)
	Multi Active	• Global database • Duplicate indexes	• Global database • Active-active multi master
	Partitioned Multi Active	• Latency-optimized geo-partitioning	• Locality-optimized geo-partitioning
Data	**Data Access Architecture**	• **Consistent reads** • **Follower reads** • **Read from the nearest region** • **Read replicas**	

Read more about YugabyteDB for financial services here.

Why Choose YugabyteDB?

The journey to a truly active-active-active architecture is complex but essential for modern, resilient applications. Active-active-active design patterns offer a robust solution for achieving high availability and scalability in distributed systems, particularly when combined with AI-ready, modern database solutions like YugabyteDB.

By ensuring all nodes are actively serving traffic, these design patterns minimize downtime and enhance system resilience.

YugabyteDB's support for distributed transactions, geo-distribution, and PostgreSQL compatibility makes it the ideal choice for implementing such patterns in real-world applications.

Distributed, PostgreSQL-compatible, cloud-native YugabyteDB is ideal for achieving active-active-active configurations. Its use of the RAFT protocol for data replication ensures strong consistency and high availability, making it suitable for applications requiring low latency and high uptime.

In a multi-master setup, each cluster is responsible for serving reads and writes independently. This setup is particularly useful for global applications that require data to be accessed locally.
YugabyteDB offers several additional benefits for implementing active-active-active patterns:

- **Distributed Transactions:** Support for ACID-compliant transactions across multiple nodes, ensuring data integrity.
- **Geo-Distribution:** Allowing data to be distributed across multiple regions reduces latency and improves availability.
- **PostgreSQL Compatibility:** Applications built on PostgreSQL can easily migrate to YugabyteDB and leverage its distributed capabilities.

Conclusion

As cloud-native applications continue to grow, adopting active-active-active architectures will become increasingly important to ensure seamless user experiences across multiple regions.

By understanding and implementing these design patterns and recommendations, you can build highly available, fault-tolerant systems that meet the demands of a modern, global, always-on world.

Get Started in Minutes

Ready to take your database to the next level? Embrace our fully managed DBaaS, YugabyteDB Aeon, in minutes with a free cluster in AWS, Azure, or GCP.

Get started at cloud.yugabyte.com.

For more information, contact a YugabyteDB expert at yugabyte.com/contact.

yugabyteDB

FOLLOW US