fauna

Balance global user experience and data sovereignty in your database

The architect’s guide to building scalable global applications fauna

Overview

Cloud-based applications drive today’s economy. Increasingly, global applications are becoming more desirable for their speed and the superior user experience they provide, and more of a requirement for businesses to comply with regulations concerning personally identifiable information (PII) and data sovereignty, such as the EU’s General Data Protection Regulation (GDPR).
While cloud provider offerings — notably, serverless edge compute — have made building global applications easier than ever, managing application data remains a significant bottleneck. Until now, globally scaling an application’s database has involved:

- Significant database infrastructure with poorly understood multi-region replication schemes, brittle failover strategies, and custom tooling.
- Increased complexity in the application layer as it takes on database concerns, leading to lowered developer productivity and worse user experience in the form of data correctness bugs.
- Added exposure to multiple sets of privacy and data sovereignty regulations, and the resultant need for even more sophisticated data routing solutions and access controls.

What if there were a platform that addressed the above concerns with the same ease of use and developer productivity as serverless compute platforms have done for the application layer?
This whitepaper outlines why Fauna is the right database to meet these needs.

Building a truly global application

A truly global application requires a great user experience (UX), regardless of where users are located. From day one, the application (i.e., client) must be intuitive and clear, reach users everywhere, and maintain consistency. Unfortunately, long distances create latency challenges for client-server interactions.

Architectural Challenges

Many applications, even modern ones, use a legacy three-tier architecture where a client application connects to a single regional application server and database:

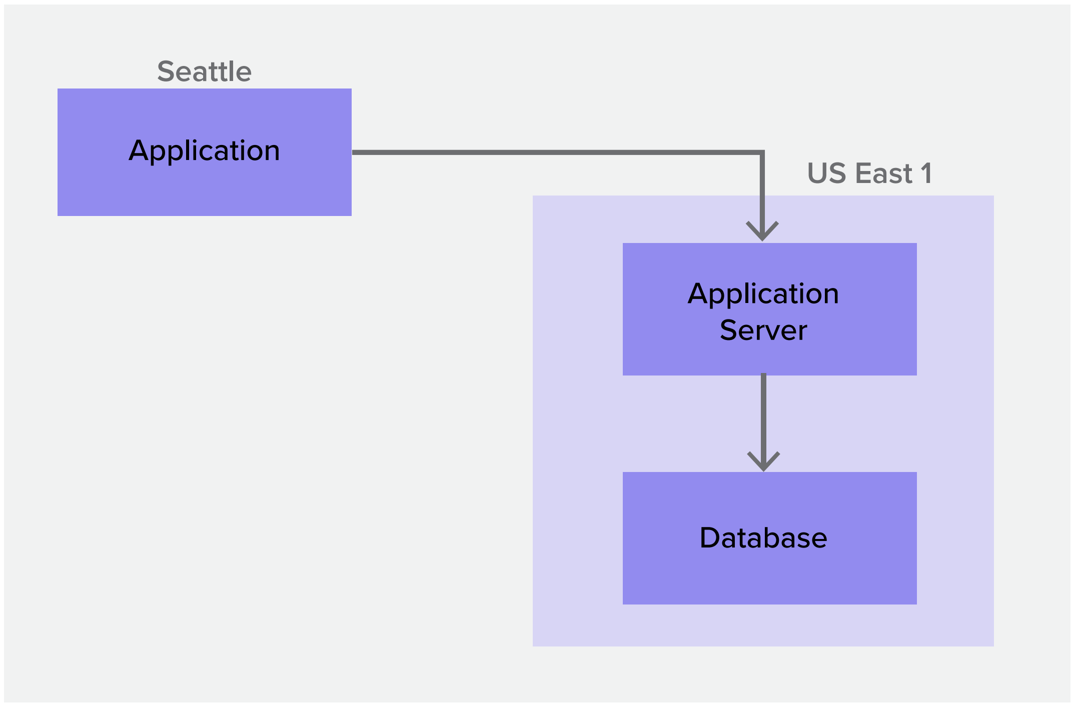


Figure 1: Legacy three-tier architecture.

This design has several issues with performance, UX, regional redundancy, compute on ingress, etc. Let's take a closer look at these challenges.

Performance Issues

From a performance standpoint, the design often results in high latency where the best round trip times are typically at the rate of three-digit milliseconds. Latency is further compounded when multiple request-response round trips are required to complete an operation (e.g., when the client application is responsible for executing many parts of the business logic).

Inconsistent UX

Another major issue is inconsistent UX. Clients from different regions may experience different levels of latency based on their distance from the server as shown in Figure 2 below:

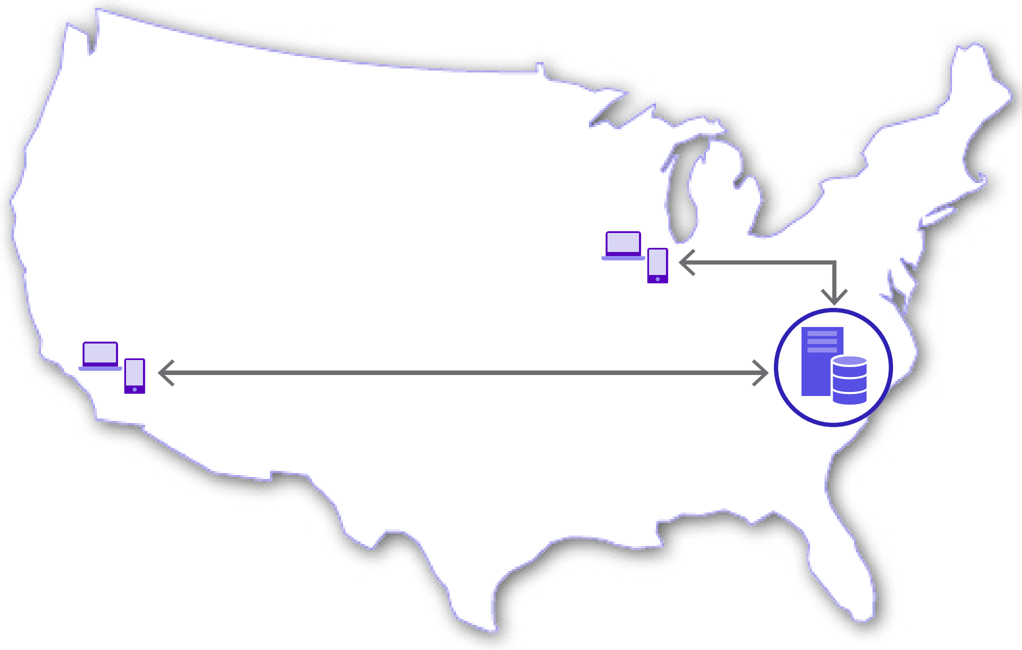


Figure 2: Two clients accessing a single server in the legacy three-tier model.

Here, the server and database are located on the US east coast, while customers are located in the Midwest and west coast. Taking an e-commerce application as an example, the customer whose client machine is closer to the server may have a better chance of adding that last in-stock product to their cart due to lower latencies.

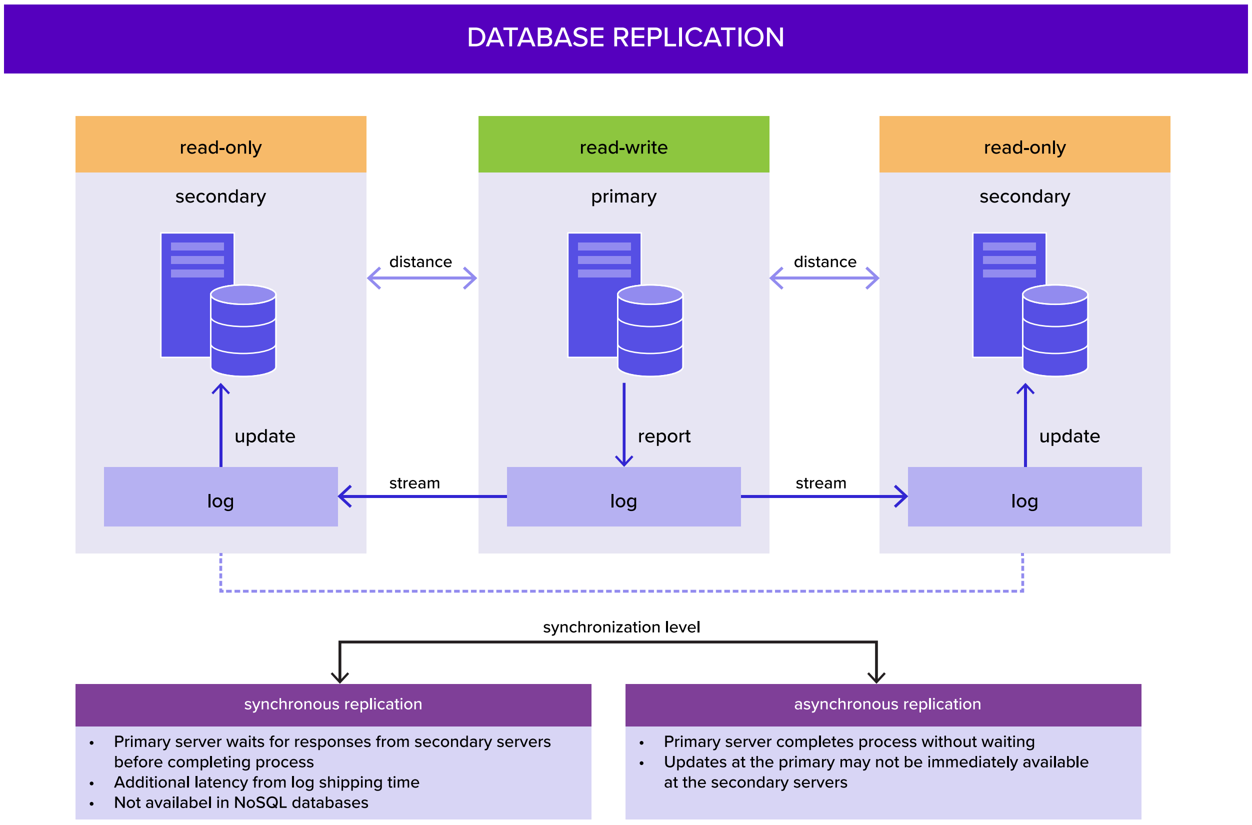
Lack of Redundancy Against Regional Failures

Another major problem with the legacy three-tier architecture is redundancy against regional failures. If an infrastructure outage or application issue occurs, there are no built-in mechanisms for failover to keep the platform available. This design is limited to vertical scaling since there are no replicas to segregate data across. Additionally, the lack of replicas means there are few opportunities, if any, to introduce load balancing into the architecture.

Many of today's database systems put the onus on the client application to store and specify the endpoint(s) for making requests via a client-side configuration. Such endpoints are stateful and connection-based (e.g., TCP) in many legacy architectures, which means they require maintenance of long-lived sessions. If the architecture were expanded to handle multiple regions, the client application might have to store additional configurations, which can add to the client application's complexity.

Replication Challenges

Database replication (e.g., distributed databases) is a common approach to address the architectural issues outlined above. In terms of latency and performance, replicas can bring data closer to the user, while allowing for redundancy and horizontal scaling. However, replicas require sophisticated synchronization mechanisms to ensure they remain in sync.
See Figure 3 for a typical legacy replication design:



synchronous replication

- Primary server waits for responses from secondary servers before completing process
- Additional latency from log shipping time
- Not availabel in NoSQL databases

asynchronous replication

- Primary server completes process without waiting
- Updates at the primary may not be immediately available at the secondary servers

Figure 3: A typical legacy replication design with a primary server and one or more secondary servers.

This architecture consists of a primary storage location and secondary setups at different locations. The primary and secondary setups must remain in sync. However, different synchronization methods present unique problems for transactionality and performance.

With **synchronous** approaches, the primary server waits for responses from secondary servers before completing a process. This causes latency for the client application during the waiting period. For example, a user who requests that a product be added to their cart must wait for the entire process to complete before the application can notify the user that the product has been added.

With **asynchronous** approaches, the primary server completes the process immediately and sends the updates to the secondary server, which takes time to complete. While this enables the user who claims the final product to receive a confirmation fairly quickly, a second user hoping to claim the same product will continue to see it as available, until the update is propagated to the secondary server. This often requires complex logic to ensure consistency and accuracy.

Physical Distance Latency

This architecture's cross-server synchronization is bound by the physical distances between servers, which may cause latency or stale data. In addition, replication architectures face operational and infrastructure expenses, especially when scaling horizontally across global regions like the US, EU, and APAC. See Figure 4 for a sharding architecture spread across the US and EU regions:

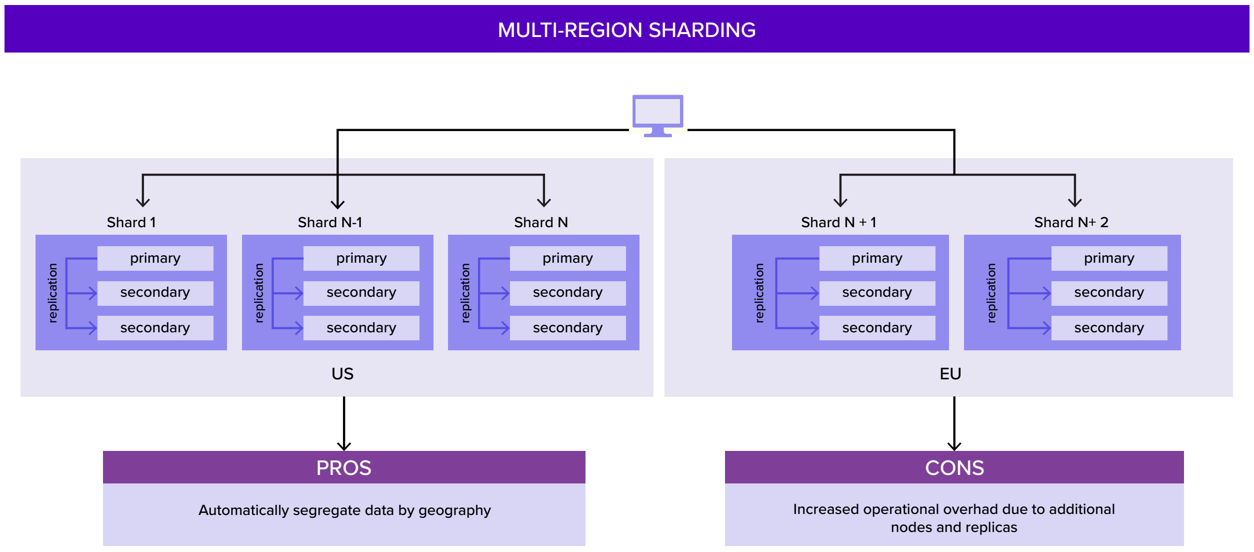


Figure 4: A typical multi-region sharding architecture.

This architecture nicely segregates data by geography but incurs additional implementation and maintenance costs which increase with the number of nodes and replicas. As the number of shoppers on an e-commerce platform grows, the sharding strategy will need to be adjusted. Plus, if data needs to be moved or redistributed across a new infrastructure, this can result in downtime. For a global e-commerce application, this can mean missed sales opportunities and low customer satisfaction. This migration process is also prone to errors which can cause additional downtime and implementation challenges.

Routing and Load Balancing Issues

If you want to take advantage of the performance benefits of replicas, ensure you carefully consider routing and load balancing. The system must route client requests to the closest replica so their requests and actions update as quickly as possible. For e-commerce, client applications for shoppers on both the west coast and the Midwest region must route requests through nearby server replicas.

The system must also take into account large numbers of requests which occur as the user base grows, or during peak periods such as holiday shopping. Here, a load balancer plays an important role by distributing traffic and workloads across replicas. Load balancing can be performed using dedicated hardware (which is often expensive), software, or via DNS. Oftentimes, developers choose to use software to load balance. Various load-balancing algorithms exist, ranging from the simple round-robin approach where servers are chosen sequentially from a list to sourcing algorithms that match source IP addresses to specific servers. While they're generally effective, they often lack the real intelligence necessary to fully optimize load balancing.

Stateful connection-based sessions can further complicate load balancing because of the need to maintain long-lived connections. This can result in balancing connection loads rather than the amount of work being performed by the connections. Solutions such as statistics scaling or reshaping traffic through reconnections often fail to solve latency issues and tend to bring additional challenges and risks.

How Fauna Meets these Challenges

Overcoming the aforementioned challenges requires a global, low-latency architecture with a new approach to replication. Fauna is a distributed document-relational database delivered as a cloud API and backed by a native serverless architecture. Let's take a closer look at how Fauna's design and features meet these challenges.

Inherently distributed

Fauna is built on global cloud services which are inherently distributed, so there is no infrastructure to build out or maintain. This gives customers flexibility over how to distribute resources based on requirements (e.g., across regions). Implementing a similar solution from scratch would require deployment to a global network of systems like EC2 or clusters, and stitching them together, which requires a lot of complex work and can be costly.

With Fauna, data is distributed amongst public region groups which currently include the US, EU, and Classic groups (i.e., a hybrid of both the US and EU region groups), as well as private region groups which you can create with your choice of geo footprint using Virtual Private Fauna. Fauna is also highly consistent.

Note: Additional region groups will be added in the future.
Figures 5 and 6 show the region and classic groups currently available:

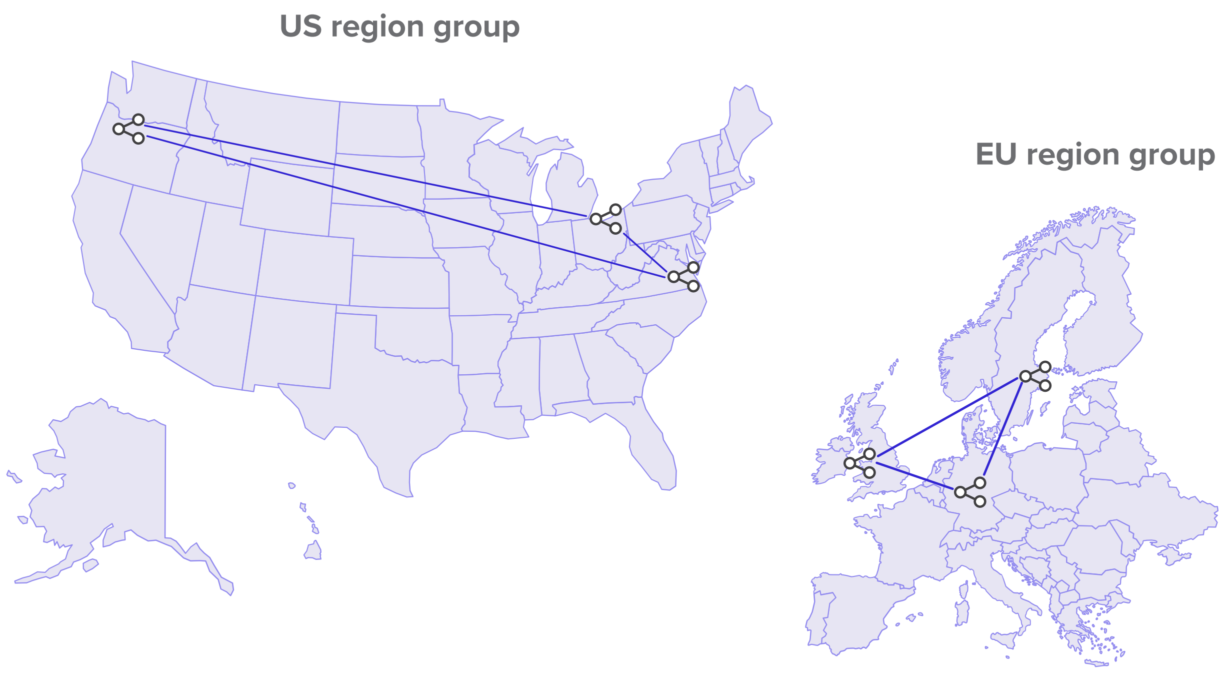


Figure 5: US and EU region groups.

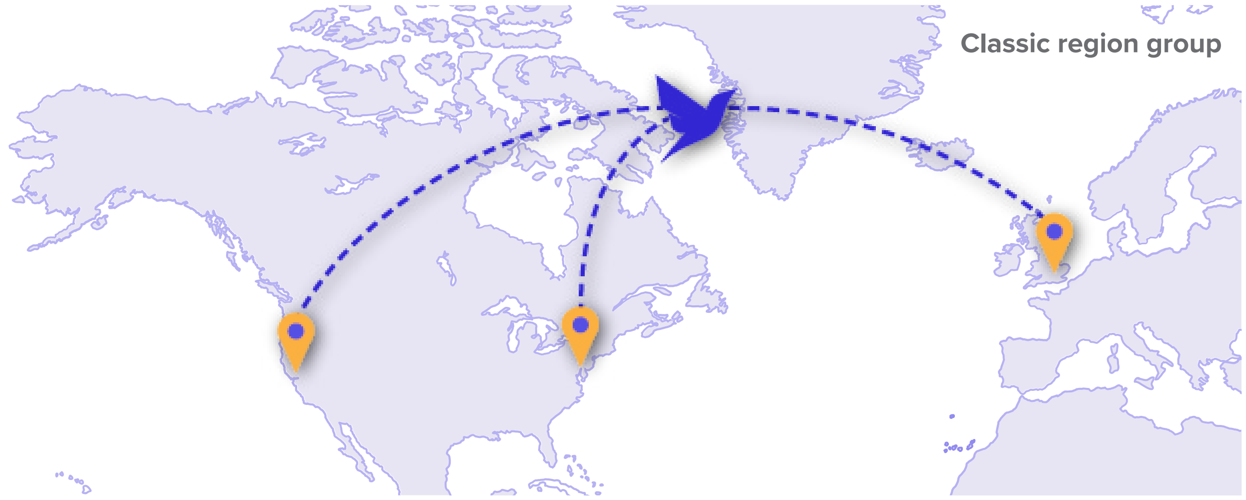


Figure 6: US and EU classic group.

Each public region group consists of multiple deployments in different geographic regions, strategically located to bring data closer to users while replicating across availability zones.

By exposing data through a stateless HTTP-based API, Fauna abstracts away the need to store configuration details (i.e., which endpoint to hit) on the client and eliminates the need for developers to manage hardware or perform database upgrades.

Intelligent Routing and Load Balancing

Fauna has an intelligent DNS routing layer located in every region outside of the Fauna-deployed region groups. This layer directs requests sent via the API to the correct region group where the data lives, using the fastest compute node that can serve the request. This is depicted in Figure 7:

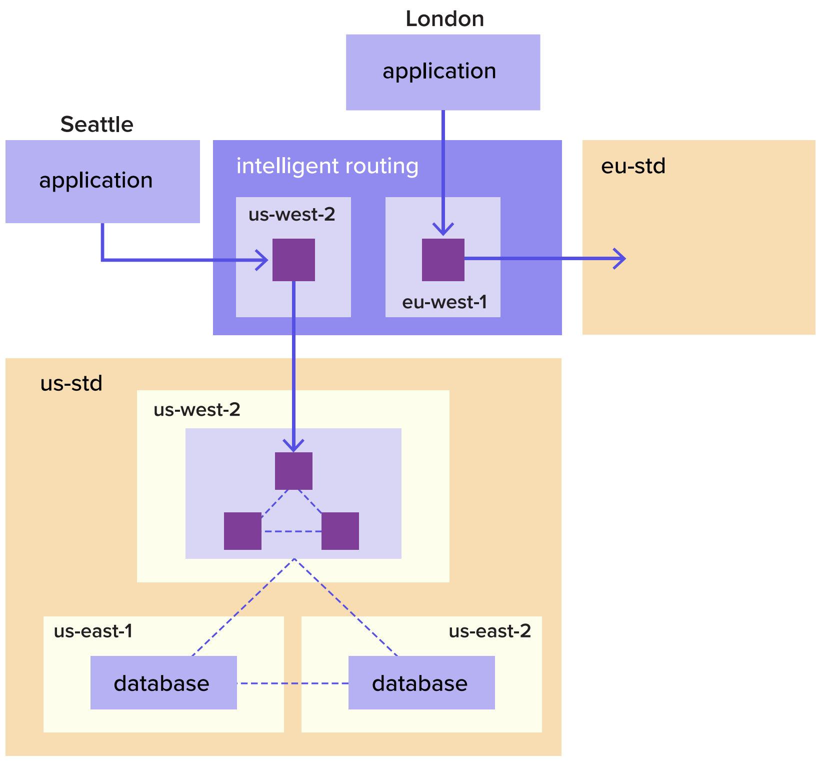


Figure 7: Fauna's load balancing architecture.

Here, client applications run in both Seattle and London. Fauna's intelligent DNS routing system routes requests to the fastest available node for the correct database. This design allows for low latency across the geographical spread served by an application.

When a request arrives, the routing service introspects the key used with the request. This allows applications to run requests at ingress by routing them to the correct region group, region, and host. Fauna first routes requests to the correct region group based on the key, then routes to the region (within the region group) based on latency. Fauna can also throttle requests based on IP addresses or request attributes (e.g., data in request headers), and set provisioned throughput limits on an account, database, or key basis.

Fauna provides true redundancy against regional failures by running custom, application-specific routing algorithms, thanks to its ability to push state information to routing nodes for intelligent routing decisions (e.g., based on the overall health of the system, iteratively failing over a certain percentage of traffic based on database health assessments, etc.).

Durable Transactionality

Fauna's architecture was inspired by the Calvin transaction scheduling and data replication layer protocol. This provides three main benefits:

- Lower write latency since there is only one global round trip on write
- Ideal read latency by communicating with the closest replica
- Strong consistency

Calvin's design uses a deterministic ordering guarantee to significantly reduce contention costs associated with distributed transactions. This is leveraged to eliminate all global communication except for the single global round trip on write.

This ensures every replica sees the same log of transactions. Calvin's design guarantees a final state, which is equivalent to executing transactions in this log one-by-one, and to every other replica.

Calvin's design also removes the need for per-transaction locks by leveraging a pre-computation step that computes the inputs and effects of each transaction ahead of time. Fauna handles this precomputation on stateless query coordinators which scale horizontally.

Behind the scenes, the protocol involves the setup shown below in Figure 8:

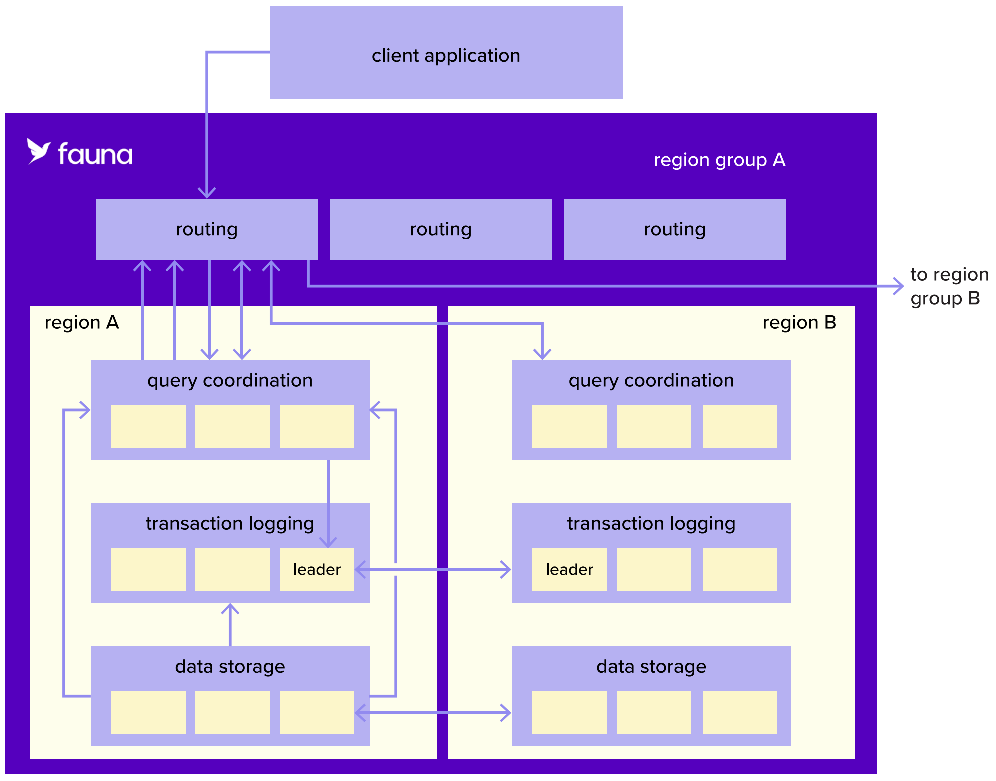


Figure 8: Fauna's implementation of the Calvin protocol.

The protocol works as follows:

1. A client application sends a request, which Fauna routes to the correct region.

2. A **query coordinator** receives the request and assigns a snapshot time. It optimistically executes the transaction into a flattened set of reads and writes and commits them into the transaction log.

3. The **transaction log** is a globally-distributed write-ahead log. The transaction log assembles the flattened transactions into batches for processing by the data storage layer. Fauna's execution engine ensures that the final result of processing this batch of transactions is equivalent to processing them one by one in the order they appeared in this pre-generated log.

   The transaction log is the only part of the protocol that requires consensus across replicas, making it a distinctive feature of Fauna – other geo-replicated systems require at least two rounds of global consensus. This means transactions (writes) are faster with Fauna as they require fewer global round trips.

   Global synchronization of clocks across servers is not required for correctness. Each log node uses its local clock to drive the epoch interval, and clock synchronization across log nodes ensures that each segment is generating epochs (e.g., e1, e2, e3, etc.) at about the same time.

4. The **data** nodes listen for transactions and ensure no values read during the transaction have been changed between the snapshot and transaction commit time. The data node updates its values and informs the query coordinator of the transaction's success or any failures due to conflicts. The query coordinator then notifies clients about the transaction commit result.

How Fauna Compares to other Solutions

While there are several great database solutions, many of them struggle to fulfill the needs of today's global, cloud-native applications.

Fauna guarantees strict serializability while other systems only guarantee serializability. Other systems can offer strict serializability at the expense of latency, while Fauna provides both consistency and good performance.

For use cases where consistency, distribution, and scalability are top-level considerations, Fauna delivers requisite performance criteria without the operational burden that often comes with a managed service offering. This includes transactionality in a distributed system and unfolding downstream effects of duplicating data to ensure performance.

Fauna removes the need to consider the residual effects of how a data model might evolve over time, and how the application's incumbent performance, availability, and consistency objectives will be maintained.

Similarly, this eliminates the need to think through capacity planning, sharding strategy, multi-region expansion challenges, data model flexibility, or some combination thereof, should requirements or access patterns ever change. Fauna delivers elasticity to scale and adapts on demand, without the mental or financial overhead associated with planning for those changes. Fauna's operational and data model flexibility, combined with its global consistency and distribution, lends itself to

- Net-new application development.
- Modernizing an existing application.
- Complementing a legacy application where the underlying database must handle heavy global transactional loads while maintaining flexibility to adapt as an application evolves.

Satisfying Data Sovereignty Requirements

For global applications to be truly globally accessible, they must comply with regulatory laws for PII such as GDPR. GDPR defines several rules for businesses or organizations that collect users' data. Here are a few examples:

- Ensure data stays within the EU while acquiring consent to transfer personal data outside the EU.
- Transparency around how data is processed and for what purpose.
- Freedom for users to transmit personal data between businesses and to correct errors.
- Users have the right to be forgotten (e.g., the right to have their data erased, its distribution ceased, etc.).
- Use of state-of-the-art tools to protect users' data and notify them of data compromises.

From a development standpoint, this introduces several new challenges and complexities. Let's briefly explore these issues and how Fauna addresses them.

Data Locality and Replication

With GDPR, data cannot be replicated to locations outside of the EU. With conventional databases, this means spending more time and attention on maintaining two distinct stack instances without any intermingling (i.e., the system must ensure EU users don't get routed to US services). The problem becomes even more complex with multi-tenant applications, multi-region users (i.e., users who travel across regions and need access to their data), partitioning, and sharding.

Fauna's architecture naturally aligns with GDPR, so you won't need to worry about your application's compliance. Our US and EU public region groups store data in their respective locations, while Virtual Private Fauna instances can also be deployed with the choice of both geo and cloud footprints. Region group selection exists at the database level (i.e., each database, its storage, and its compute services exist in a specific geographic region), and any number of databases can be created in any region group. Each database is also fully isolated so replication never occurs across them.

An e-commerce application, for example, must handle a global user base that spans region groups while managing user account records with PII in accordance with GDPR. The application must avoid erroneously creating duplicate account records for a given user. This means no individual user can have more than one account record, and that record must exist within only one region group.

Fauna offers developers the flexibility to determine which region group the client should send specific user account requests. One approach is to use a Fauna look-up collection, which maps users to region groups. Another approach is to leverage the use of a third-party identity provider (IdP), rather than building a look-up table. In the IdP, a user's profile stores the specific region-group endpoint where requests should be made. This information can then be inserted in the JSON web token (JWT) that's generated during authentication via the IdP, and used to route requests to the proper Fauna region-group. There is also the option to store non-PII in yet another database that's common to all users from all regions.

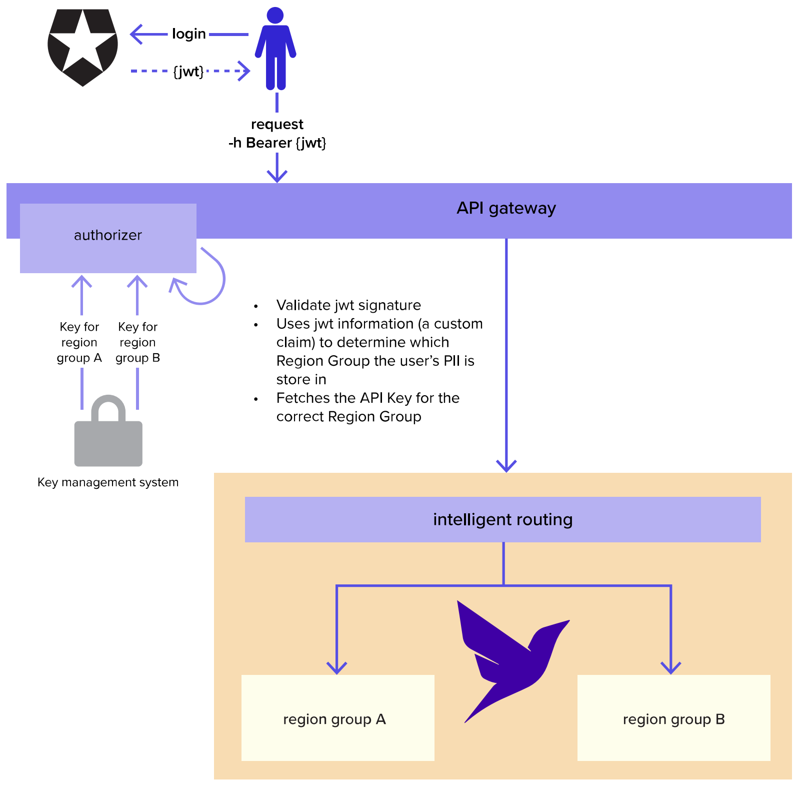


Figure 9: Augmenting a token from a third-party identity provider with a custom claim for routing requests.

Public regions and region groups are multi-tenant. This means many customers store and access data on shared hardware with multiple layers of protection in place to ensure tenants cannot access data that belongs to another tenant. Virtual Private Fauna deployments can be created within a single cloud provider region or a region group. They are single-tenant and provide VM-level isolation of data for a specific customer, allowing the customer to specify the regional footprint and set of cloud providers where they want their data to reside.

Other solutions provide single-tenant solutions for this problem but not a viable multi-tenant serverless solution that fulfills all of these requirements (e.g., some competitors are single-tenant solutions with the choice of geo and cloud provider).
Fauna also supports attribute-based access control (ABAC), where logic can be added to enforce business rules around requests and where their data should live. Custom logic can be written to use the identity fields (e.g., the users' email, role, tenant, etc.) presented to it via identity-based tokens (either from JWTs from an IdP integration, or from Fauna's built-in authentication functionality) to decide whether or not a particular query returns data.

Operational Expenses

Siloed replicated clusters for data sovereignty zones can involve operational expenses such as infrastructure and hardware maintenance, database upgrades, etc. Every sovereignty zone needs its own copy of your data serving infrastructure, and every additional cluster adds to operational costs, complexity, etc. There is increased complexity in integrating your application layer with the database – whether it's multiple connection pools or dedicated application layers per sovereignty zone.

With Fauna's serverless design, this cost is reduced and there is no operations burden to developers. Sovereignty zone infrastructure issues are abstracted away, while intelligent routing exposes all of the region groups (shared or dedicated) behind one endpoint.

Summary

Today, applications must be globally accessible, provide great UX, and satisfy data sovereignty requirements. Traditionally, fulfilling all three of these requirements would involve creating a solution with multiple different database tools, or building costly internal apps to provide optimal routing and consistency. Fauna provides an inherently global, cloud-first approach, that not only addresses these challenges but reduces the operational burden on developers while enabling them to scale.

In addition, Fauna was designed as a public-facing utility with fine-grained ABAC. This provides a higher level of compatibility with modern applications delivered to edge devices, through ephemeral functions or with serverless frontends, rather than the coarse-grained RBAC approach used in many systems.

About Fauna

Fauna is a distributed document-relational database delivered as a cloud API. It combines support for semi-structured data with powerful relational features such as foreign keys, views, and joins. A native serverless architecture means having to worry less about operations. Developers choose Fauna to build new applications faster and confidently scale existing ones across regions and the globe. Builders and scalers like Lexmark, Cloaked, Insights GG, Hannon Hill, Everly Health, Connexin, DLTR, Santander, and Azion trust Fauna to accelerate development and solve mission-critical challenges.

Visit https://fauna.com/global-applications to learn more.

Resources/References

Abadi, Daniel, and Matt Freels. "Consistency without Clocks: The Fauna Distributed Transaction Protocol." Fauna, 2018, https://fauna.com/blog/consistency-without-clocks-faunadb-transaction-protocol.

Atchison, Lee. "The Problem with Sharding." InfoWorld, InfoWorld, 5 July 2021, https://www.infoworld.com/article/3623388/the-problem-with-sharding.html.

"Comparing Fauna and MongoDB." Fauna, 2022, https://fauna.com/blog/comparing-fauna-and-mongodb.

Fauna, Inc. "ABAC Best Practices." Fauna Documentation, Fauna, https://docs.fauna.com/fauna/current/security/best_practices/.

Fauna, Inc. "Attribute-Based Access Control (ABAC)." Fauna Documentation, Fauna, https://docs.fauna.com/fauna/current/security/abac.

Limited, ServerAdminz. "How to Implement Load Balancing in Database Servers?" Medium, Medium, 12 Feb. 2018, https://medium.com/@serveradmns.seo/how-to-implement-load-balancing-in-database-servers-7ad2b35cc943.

Lutkevich, Ben. "What Is Database Replication and How Does It Work?" SearchDataManagement, TechTarget, 24 Feb. 2020, https://www.techtarget.com/searchdatamanagement/definition/database-replication.

"The Raft Consensus Algorithm." Raft Consensus Algorithm, https://raft.github.io/.
"RBAC vs. ABAC: Definitions & When to Use." Okta, https://www.okta.com/identity-101/role-based-access-control-vs-attribute-based-access-control/.

Weaver, Evan. "Comparing Fauna and DynamoDB Pricing." Fauna, 9 Dec. 2020, https://fauna.com/blog/comparing-fauna-and-dynamodb-pricing-features.

Weaver, Evan. "Postgres vs Fauna: Terminology and Features." Fauna, 20 Jan. 2021, https://fauna.com/blog/compare-fauna-vs-postgres.

"What Is Sharding? Database Sharding, Scaling, and You." DNSstuff, 12 Jan. 2021, https://www.dnsstuff.com/what-is-sharding-database-sharding-scaling-and-you.

Wu, Rain. "The Advanced Challenge of Load Balancing." Medium, Geek Culture, 16 June 2021, https://medium.com/geekculture/the-advanced-challenge-of-load-balancing-6f6ef5f36ec4.