Going Global: achieving resiliency and high availability with Multi-Region Deployment

3 July 2024 - 11 min. read

Amazon Cognito | High Availability (HA) | Multi-region deployment

Introduction

In the digital age, data has become a highly valuable asset. When developing an application, the data layer forms the foundation of the entire infrastructure, serving as the bedrock upon which all functionalities are built. Ensuring that this data is not only available but also secure is paramount, as it directly impacts the reliability and trustworthiness of the application.

Multi-region deployments in cloud computing are a strategic approach to achieving high availability and robust disaster recovery. By distributing data across different geographical locations, applications can withstand regional outages and ensure continuous operation, which is essential for maintaining user confidence and service integrity. However, the benefits of multi-region deployments extend beyond disaster recovery. They also support global scalability, allowing applications to serve a worldwide user base with lower latency and higher performance.

Implementing a multi-region strategy requires careful planning and consideration of various factors; in this article, we will provide a full overview of our experience on multi-region deployments, highlighting the pros and cons with particular considerations about databases and, in general, on the data plane.

Without any further ado, let's deep dive into our experience.

High availability: Isn't using managed services enough?

Typically, when we start a new project, we try to use only serverless or managed services (like AWS Lambda, Amazon APIGateway, Amazon DynamoDB, Amazon Cognito, Amazon SQS, and all the infinite AWS Managed Services), thinking that we are automatically achieving fault tolerance and high availability.

And that's true; using Serverless services allows us to take advantage of all the availability zones in a region, making our application resistant to data-center outages.

But what happens if an entire region, or more realistically, one service in the entire region, stops working?

As this site registers all the **AWS outages** in the past years, **region outages**, even if very rare, could happen.

As Solutions Architects, our job is to design highly available solutions in terms of Availability Zones and to create infrastructures that **resist** the region's failures.

This paradigm is not only for creating highly available solutions but also for reaching users worldwide by bringing content closer to their location. In this discussion, we will focus on shifting the paradigm to enhance the resilience and high availability of our applications.

However, even if we are using only serverless services, it's not always as simple as it sounds.

Changing the paradigm from mono-region to multi-region comes with some challenges. You have to think differently and choose the appropriate service that best fits your needs and the multi-region paradigm. Even if you pay attention to all of these problems, sometimes you will find that there is not an easy-to-use and ready solution for them.

The data layer is one of the first aspects you must address: you must design for cross-region data replication and synchronization. To make things worse, each service stores and manages data with its own complexity and limitations.

Also, let's consider the **authentication layer**, for example. In AWS, the only service that allows you to authenticate your APIs is Amazon Cognito.

Unfortunately, this service cannot currently **be deployed in a multi-region solution**, and it's uncertain if it ever will be. So, how can we achieve that?

And what about **logging and monitoring**? In a mono-region solution, logs and metrics are in the same region of the application, and you know for sure that if a user has problems, you can simply watch the logs in AWS Cloudwatch (for example) and understand why your

users are experiencing errors. Instead, you have logs and metrics scattered across all regions in a multi-region solution. Searching for a specific user error is more difficult as you don't know (in an easy way) where the user is and which region is contacting.

In this article, we will focus on Database, object storage, Route53, and Cache System solutions to transform our simple mono-region setup into a multi-region active-active solution.

But first, let's see some advantages and disadvantages of choosing a **multi-region** deployment.

Multi-Region Deployment: PROs and CONs

As with every solution, some pros and cons must be considered before choosing what is appropriate for our application. So let's not waste time and start analyzing all the advantages and disadvantages of the Multi-Region - Active / Active solution.

PROs:

- As we already said, if there are problems in one region, our service continues to **work perfectly without our intervention**.
- Traffic to our application is **distributed** across all the configured regions, allowing us to **better configure** all the compute parts as the income demand of the specific region. This could lead to better cost optimization and better performance.
- We can bring content closer to our users' location giving them better performances.
- For some AWS services, there are **no adjustable regional limits** (such as the AWS Secrets Manager GetSecretValue API call, which has a hard limit of 10,000 requests per second). Having the same resource deployed in different regions increases the service's limit, allowing us better scalability.

CONs:

- **Logging** and **monitoring** are more **difficult** as they are spread across all regions.
- Based on the services used, **costs** can be **higher**.
- **Design and implementation** are more **difficult**.
- Infrastructure as code (**IaC**) **works differently**.

Now that we have seen some pros and cons of the multi-region solution let's deep dive into the core of this article.

A single-region application

Let's imagine a simple application where we have to manage CRUD REST API's. We will have a database with some tables and an API layer that allows us to create, update, delete, and get all the data from those tables.

A possible infrastructure for this type of application is the classic API Gateway / Lambda Function to expose the APIs and DynamoDB or RDS Aurora for the database part.

Now that we have defined our basic mono-region infrastructure, we can start thinking about how to shift it to the multi-region paradigm. There are no special considerations for the APIGateway and Lambda components to take into account. We can simply deploy it in all the chosen regions.

So, the first object that deserves some consideration is our **database**.

To avoid annoying system impairments like split-brain scenarios, we have to make it **globally distributed** across all the regions without compromising the data structure. For this, the AWS Cloud comes to our aid, making different solutions based on the service that we are using available to us.

Database

One of the best solutions is Amazon DynamoDB. AWS created a perfect configuration of DynamoDB that allows us to deploy the same table in different regions in a multi-master way called **DynamoDB Global Tables**. Thanks to this, the application can read and write data to its regional table that will be replicated **automatically across all the configured regions**. The weighty thing to remember is that if different instances write the same record in different regions, AWS uses a **last-writer-wins** reconciliation mechanism to save and replicate **concurrent updates**. Another thing to know is that **transactional writes** works within the region rather than globally. The record is replicated to other regions only when it is committed regionally. So be aware of these limits during the development process of your application.

But what if we need traditional **relational databases** or other type of databases?

Fortunately, AWS created Global configurations for some database services, including AWS RDS Aurora and AWS ElastiCache.

However, they are a bit different from the DynamoDB solution. Unfortunately, they are not in a multi-master configuration. Instead, they consist of a master instance in one chosen region and different **read-replica** instances across all the other regions. In case of region

failures, AWS automatically promotes one read-replica instance to master within some minutes.

For multi-region, active-active relational solutions require a more complex design, but today, we will suffice with active-passive configurations.

Object storage

What about data that is not stored in the Database?

One of the most common data storage methods in the cloud is S3, which can be used to store different kinds of data as objects.

Fortunately, replicating and synchronizing data on S3 is simple thanks to a feature called Amazon S3 Cross Region Replication (CRR).

CRR asynchronously copies each object across one or more destination buckets in different AWS regions. The replication process includes not only the object itself but also its metadata and access control lists, preserving the original object's properties.

Similar to the relation database approach, this one is also a primary standby replication system, so there is a primary bucket that is replicated across regions. Unfortunately, updating replicas does not propagate the changes to the other buckets.

With this solution, we have transformed our single-region database into a globally distributed database replicating data in our object storage, allowing us to save and read our data in different regions.

Now, we only need to route traffic to our multi-region infrastructure; for this, some further considerations have to be taken into account.

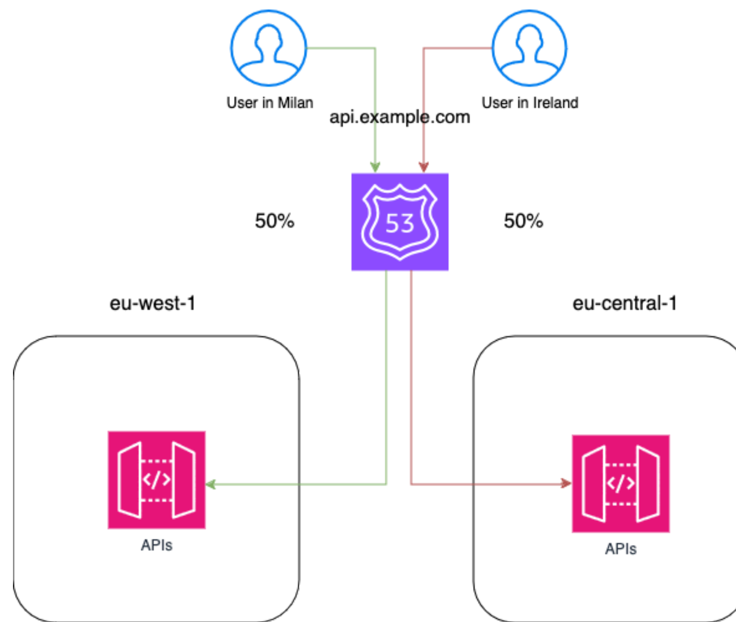
DNS Configuration

We can use different solutions. The choice depends on our goals. The first question we have to answer is: How do we want to split the traffic into the configured regions? Do we want to split the traffic based on volume? User proximity? User latency? Service availability? Let's deep-dive into some of these solutions.

Weighted Records

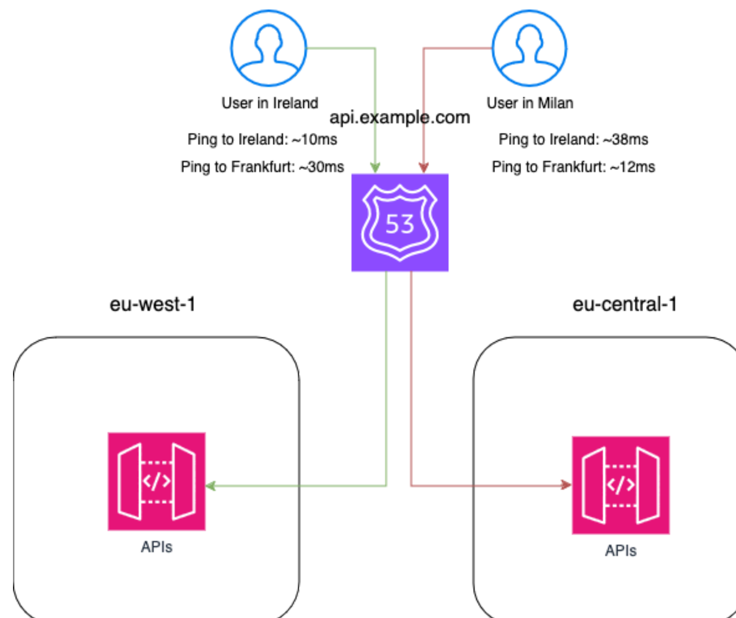
The simplest solution is to divide the traffic balanced on all the available regions using **weighted** records. This will result in all regions having the **same incoming traffic**. Let's suppose that we have configured three regions for our application. We can create three

different weighted records (one for each region) with an 85 weight value (255 as the max weight divided by 3 regions). With this configuration, Route53 will automatically **balance** the incoming traffic into three different regions.



Latency-Based Records

Another way to configure it is to use **latency-based** records. This allows us to route traffic in the configured region that is **nearest** to the requesting user. If you do not have application constraints that disallow this type of configuration, this is the best solution for DNS routing because it improves performance for the final user.

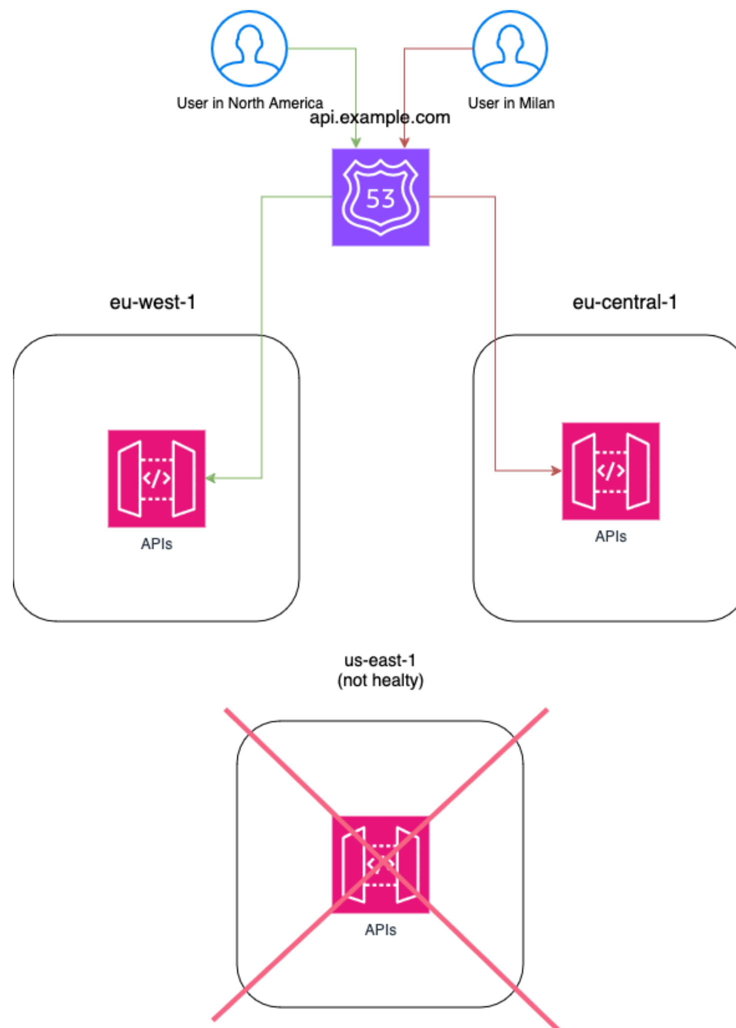


Health Check

But what if one region has **outages** and becomes **unhealthy**?

To solve this problem, we have another important thing to configure: the **health check**. This allows Route53 to route traffic only to healthy regions. In our case, as we are using API

Gateway, we can use **alias** records that allow us to enable the "Evaluate Target Health" feature. With this flag, Route53 automatically does a health check on our resources and understands whether the target is healthy. The automatic health check does not check for HTTP status codes or other metrics but simply verifies that the resource is reachable within some seconds. If we need to use specific metrics or we are not using resources compatible with alias records, we can create our custom Route53 Health Checks. **Custom health checks** can check for the HTTP status code of a specified API path or check a specific Cloudwatch metric, allowing us to **customize** how Route53 declares a resource healthy.



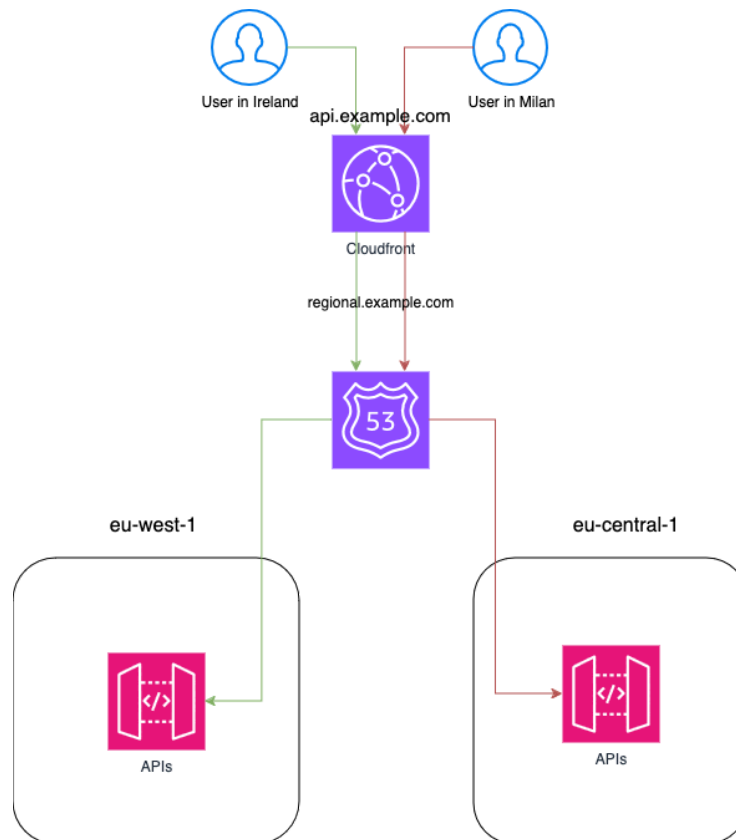
At this point, we have our initial infrastructure successfully configured as a multi-region / active-active solution.

Everything is working as expected, but users' complaints are just around the corner. Performance is crucial: we need to improve users' experience caching our read API results.

Global Cache

Typically, with API Gateway, you can configure a cache that uses instances to store data of GET APIs automatically. This configuration type has additional hourly costs based on the instance size selected, which means higher costs in a multi-region solution.

One simple solution to this problem is to use AWS Cloudfront as our cache system in front of our multi-region solution.



Let's imagine that we want our APIs to be exposed using the domain "api.example.com". We can create our Cloudfront distribution and configure it to cache all the GET endpoints.

Based on our application constraints, we can configure the Time To Live for our cache to expire:

- after some time: the data will be automatically deleted from the cache after the configured time is reached.
- never: we can change our backend code to create a CloudFront Invalidation Request on the appropriate path only when data changes.

Then, as the origin, we will use HTTP integration using the domain "regional.example.com," which is attached to our ApiGateways with a Latency-Based or Weighted configuration. And that's it. Now, we have a cache system on top of our infrastructure.

Conclusion

In this article, we have seen which steps are needed to transform a simple solution deployed in a single region into a multi-region / active-active solution.

The scope was not only to show you how we can transform our mono-region application into a globally distributed one but also to make you think about how many aspects must be

considered before considering this type of approach. One suggestion is to think about it before the actual implementation, changing the paradigm "from mono-region to multi-region" into "Go Global" directly.

We saw how, in a data-driven world, being sure our data will always be updated and accessible, also in case of regional failure, is a priority, both for business continuity and for competitive advantages.

But there is much more to say when talking about the "Go Global" paradigm. As mentioned in the introduction, different problems can happen during the implementation:

- How do we solve the Authentication problem?
- How do we monitor our global solution?
- How do we centralize logs?

... and much more, still needs to be mentioned and deepened. We will definitely describe them in new articles.

Which topic would you like to deal with first? Let us know in the comments!

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Mario Agati

DevOps Engineer and Backend developer @ beSharp. Passionate about music, football, and motorsports. In my free time, I like to annoy my neighbors by playing drums.

Copyright © 2011-2024 by beSharp spa - P.IVA IT02415160189