From On-Premise Monolith to Cloud Native: A Real-World Load Balancer Migration Story

24 September 2025 - 8 min. read

Amazon CloudFront | *AWS Lambda@Edge* | *Cloud Migration*

“We must constantly look at things in a different way.”

Robin Williams (John Keating) - Dead Poets Society

Ahh, break down the monolith! The classic dev exercise that everyone has done once in their lives.

But how about breaking down an infrastructure monolith?

How can we make a big and complex system more agile? We are talking about stuff that has seen generations of system administrators pass by, adding configurations with their personal taste.

Behold the mightiest on-premise load balancer for big corporations! It has survived kernel upgrades, virtualization, breaking changes, and configuration migrations over the years to the point that no one wants to touch it, even with a 5-meter pole stick!

We like challenges, so when we were asked to join this near-impossible migration project, a migration from on-premise legacy systems to AWS cloud-native managed services while maintaining the website operational, we had a little party. We like to celebrate when we are assigned these kinds of challenges.

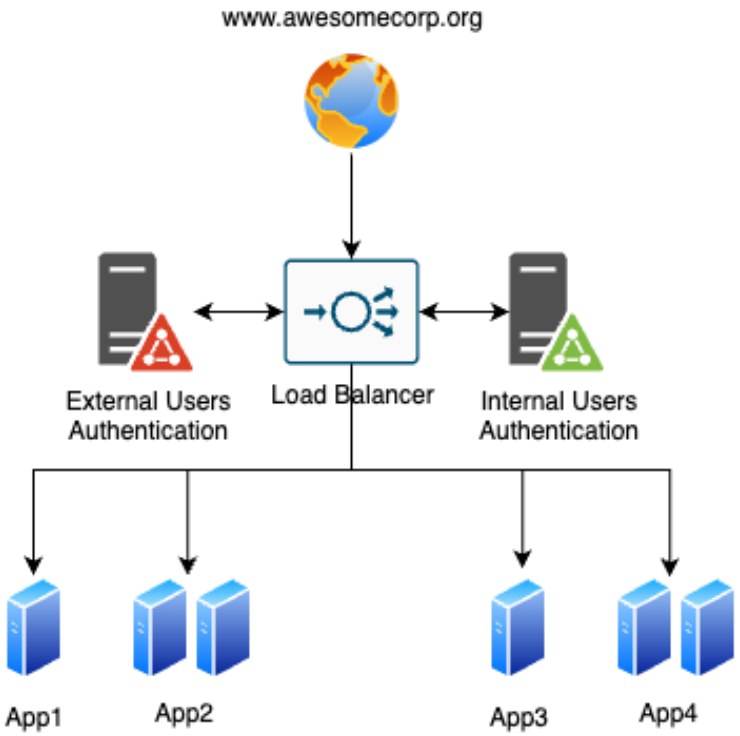
In this article, we will see only a small part of our journey and highlight the approach and pattern used to make this kind of impossible dream come true.

The beast

Imagine this: a corporate website born in the mid-90s that saw the entire Internet transformation: starting with a simple static and cringey website with <blink> tags and animated gifs later to become a design marvel and the key point for everything business related: meetings, calendars, private and public intranet (for contractors and internal staff with different Identity providers and authentication rule), marketing micro-website using multiple installation of the omnipresent WordPress stack.

Everything is (obviously) routed using a pair of highly available load balancer appliances and a single DNS domain, with no shared knowledge of whether something is still alive.

Different identity providers are used, one for internal staff and another for contractors, adding complexity to an already complex scenario. A picture is worth a thousand words, so here is a scaled-down version of the infrastructure.



The Approach

One of my favorite patterns for breaking the monolith is Martin Fowler's "**Strangler Fig Pattern**".

In this approach, you replace an old system incrementally by gradually refactoring components, keeping the old system alive until you need it.

In our case, we have three different main components:

- Backend Load Balancing
- An authentication portal that routes requests to the appropriate authentication servers depending on the user's domain and using different protocols, such as Active Directory's LDAP, OIDC, and SAML.
- Application routing for backend selection, based on subpath

We need something that can route requests, allowing us to work behind the scenes and silently replace components; the same technology should enable us to implement a Proof-of-Concept architecture to validate our hypothesis.

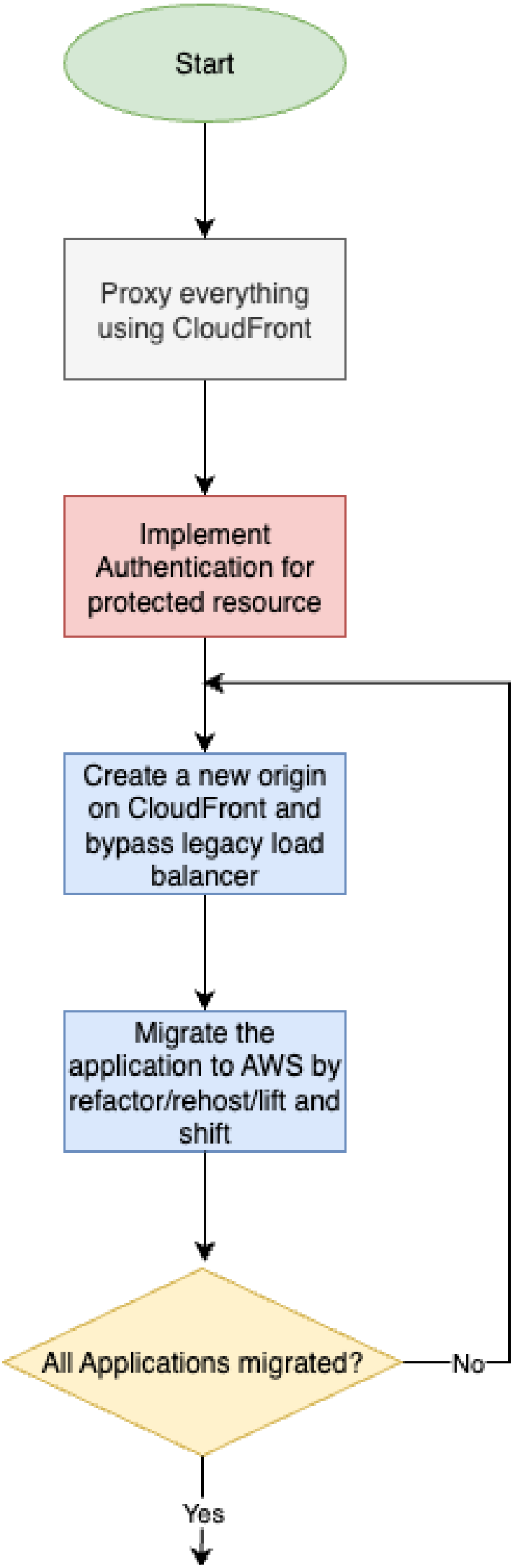
Lucky for us, **Amazon CloudFront** exists.

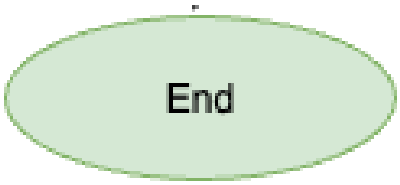
CloudFront as a strangler fig

Amazon CloudFront is a global content delivery network that is easily customizable to accommodate different needs. Using **CloudFront Functions**, we can modify HTTP requests before they reach the server or alter responses before serving a page to the Client. We can define different backends and route requests. With **Lambda@Edge**, we can implement custom application logic and authentication.

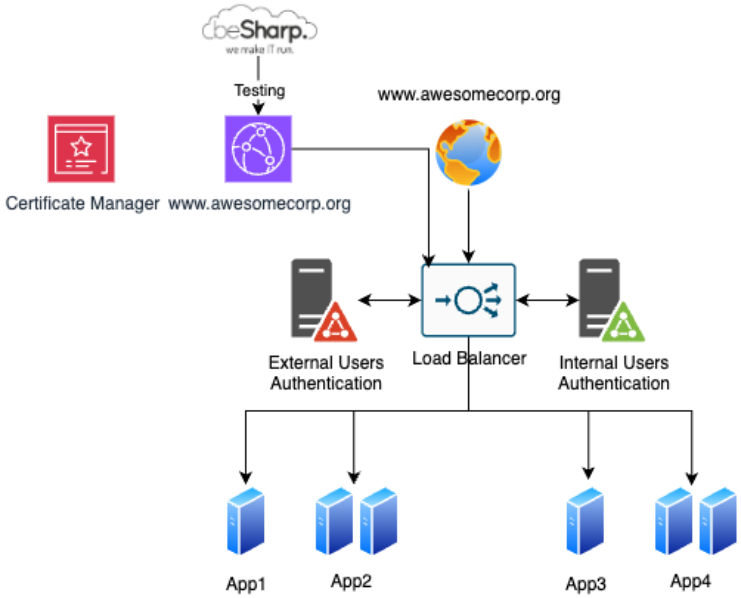
Breaking down the load balancer flow

Here's our strategy: We will replace the load balancer and migrate workloads to the Cloud, resulting in a shiny new infrastructure. Here's our flow:


Start
Proxy everything using CloudFront
Implement Authentication for protected resource
Create a new origin on CloudFront and bypass legacy load balancer
Migrate the application to AWS by refactor/rehost/lift and shift
All Applications migrated?
No
Yes



First, we have to create a “temporary” Amazon CloudFront distribution with the actual domain name to test that everything works as expected. We also have to import the existing certificate into Amazon Certificate Manager (ACM) and configure the legacy load balancer as the origin; since no DNS entry is altered, the website is safe. To test our setup, we only have to modify our hosts file, pointing to the CloudFront distribution's IP addresses.



Once we confirm that everything is okay, we can start our POC for the most critical part that is not available out of the box on Amazon Cloudfront: **authentication**.

By implementing authentication, we can replatform, lift and shift servers to the Cloud, and decompose our application without touching the legacy load balancer.

We will focus only on the authentication part, because it is crucial to move everything else to the Cloud and replace the legacy load balancer. Once the authentication is done, we can leverage Amazon CloudFront origins and behaviors to migrate our website’s subpaths gradually.

Authentication Flow

The legacy Load Balancer has an internal custom backend (auth.awesomecorp.org) that implemented authentication. Every request is intercepted, checking for JWTs, Cookies, and other authentication proof. A load-balancer-hosted page is displayed if a user tries to access a protected resource without authentication. Once the user enters

the credentials, the load balancer establishes the proper user backend (based on the username format) and proceeds with the authentication. After the user is authenticated, it will serve a cookie with different contents, depending on the path of the application.

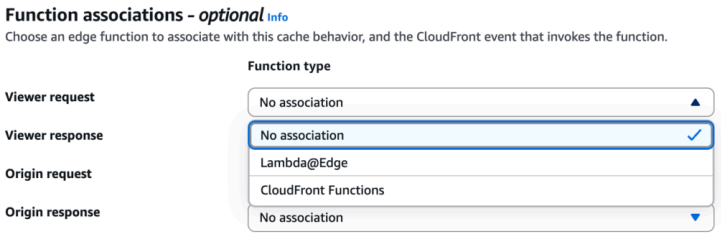
How can we implement this mechanism using only cloud-native services?

As we said, Lambda@Edge and CloudFront Functions can alter the execution flow, but how?

CloudFront intercepts requests and responses at CloudFront edge locations and fires different events, depending on the state of the HTTP Request: viewer request, origin request, origin response, and viewer response.

This table summarizes the events and their typical usage:

Event Type	Trigger Point	Typical Use Case
Viewer Request	When CloudFront receives a viewer request	URL rewrites, authentication
Origin Request	Before CloudFront forwards to the origin	Dynamic origin selection, header changes
Origin Response	After the origin returns the response	Security headers, error handling
Viewer Response	Before CloudFront sends a response to the viewer	Cookie injection, personalization



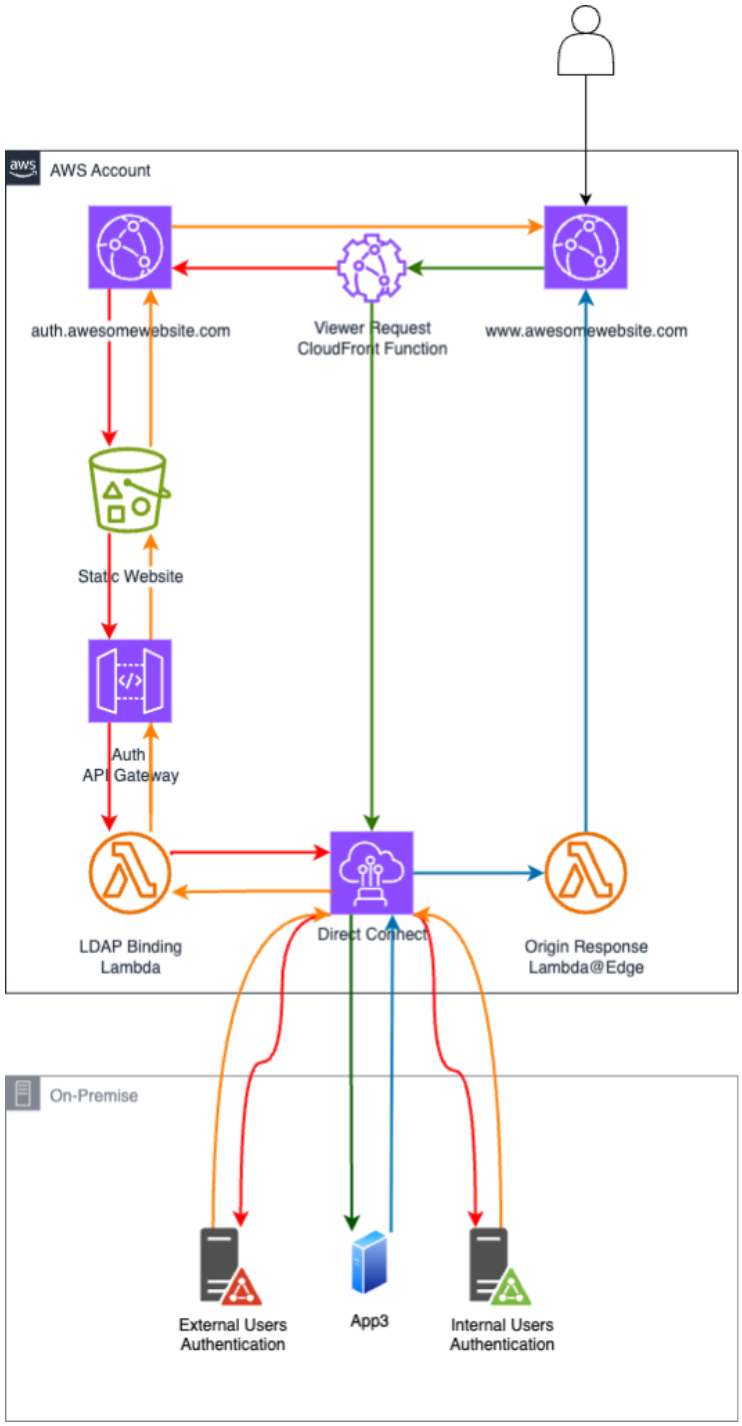
As a rule of thumb, CloudFront functions can be used to alter simple text. When the implementation logic becomes complex and you need to query external services, Lambda@Edge becomes handy.

An extensible serverless authentication mechanism

The authentication flow works by leveraging only serverless components, using a custom Lambda to authenticate users. This enables us to integrate every mechanism used at the customer side. The following flow can also be extended for SAML, OIDC, databases, etc.

When an unauthenticated user initially accesses the CloudFront distribution, a Viewer Request CloudFront Function checks for the presence of a JWT token. If no valid token is found, the function redirects the user to a custom login page hosted on a separate CloudFront distribution backed by an S3 bucket. The user enters their Active Directory credentials, which are then processed through an API Gateway that forwards the authentication request to a Lambda function. Lambda performs the authentication (in this case, an LDAP binding operation against the external Active Directory Domain Controller). The system generates a JWT token upon successful authentication and redirects the user to the distribution with the appropriate authentication cookie set.

The flow becomes streamlined for authenticated users: the Viewer Request CloudFront Function validates the existing JWT token and forwards the request to the backend service while including the authorization header for user verification. Responses flow back through an Origin Response Lambda@Edge function that monitors for authentication failures (401 status codes) and automatically expires the token when necessary, forcing users to re-authenticate if their session becomes invalid.



Here's a description of the flow:

1. Unauthenticated user

1. [Green Arrow] The external user contacts the CloudFront distribution.
2. [Green Arrow] The CloudFront distribution executes the Viewer Request CloudFront Function, which verifies whether the JWT token has been sent in the request.
3. [Red Arrow] Being an unauthenticated user, the Viewer Request CloudFront Function will answer with a redirect to the customized login page.

4. [Red Arrow] The customized login page is served by the auth.awesomecorp.com CloudFront distribution that points to the S3 Bucket hosting the static website.
5. [Red Arrow] The user enters his credentials and clicks the login button. The customized login page sends a request to the API Gateway.
6. [Red Arrow] The API Gateway forwards the login request to the LDAP Binding Lambda.
7. [Red Arrow] The LDAP Binding Lambda performs selects the proper authentication backend and performs an LDAP Binding request to the selected Active Directory Domain Controller
8. [Orange Arrow] The Active Directory Domain Controller answers the request with the LDAP Binding operation result.
9. [Orange Arrow] If the LDAP Binding succeeds, the LDAP Binding Lambda will answer the API Gateway with a status code 200.
10. [Orange Arrow] The API Gateway will forward the response to the customized login page.
11. [Orange Arrow] The customized login page will forward the response to the auth.awesomecorp.com CloudFront distribution, redirecting the user to the www.awesomecorp.com distribution while setting the JWT Token

The user is now authenticated and will follow the flow described in the next section.

2. Authenticated user

1. [Green Arrow] The external user contacts the www.awesomecorp.com CloudFront distribution.
2. [Green Arrow] The CloudFront distribution executes the Viewer Request CloudFront Function, which verifies whether the JWT Token has been sent in the request.
3. [Green Arrow] Being an authenticated user, the Viewer Request CloudFront Function forwards the request to the backend host, which will verify the user,
4. [Blue Arrow] The backend service answers the request, sending it to the CloudFront Distribution
5. The Origin Response Lambda@Edge verifies that the answer has a status code different from 401.

6. [Blue Arrow] The Origin Response Lambda@Edge forwards the answer to the user if the status code is different from 401; otherwise, it will expire the JWT token, making the user unauthenticated.

Next steps

Once implemented, this flow can be extended and modified, allowing us to move and refactor the backend services safely and without impacting the website's availability.

Even for longer migrations, we can put a subsection into “maintenance mode” by hosting a static webpage in an S3 bucket.

Dismantling an infrastructure monolith is not only possible; it can also be done elegantly. By leveraging proven architectural patterns and powerful tools like Amazon CloudFront, CloudFront Functions, and Lambda@Edge, a legacy system can be transformed into a modern, scalable, and maintainable platform.

Sometimes, innovation simply requires looking at things from a different perspective.

Are you dealing with old load balancers? Do you have different migration paths and patterns in mind? 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!

Damiano Giorgi

Ex on-prem systems engineer, lazy and prone to automating boring tasks. In constant search of technological innovations and new exciting things to experience. And that's why I love Cloud Computing! At this moment, the only "hardware" I regularly dedicate myself to is that my bass; if you can't find me in the office or in the band room try at the pub or at some airport, then!
