TrackIt

2023

WHITE PAPER

CloudFront Functions vs. Lambda@Edge Which One Should You Choose?

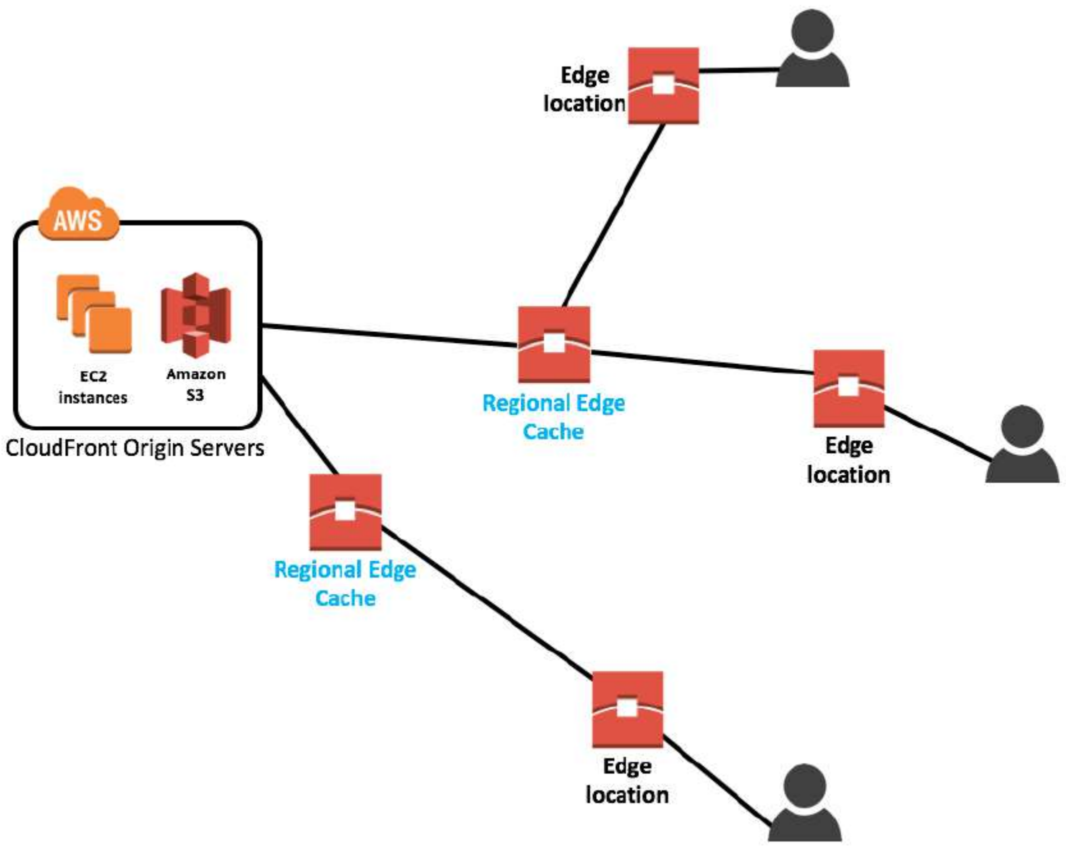
CloudFront Functions vs. Lambda@Edge - Which One Should You Choose?

TrackIt recently published an article titled ‘Migrating Your CDN to Amazon CloudFront’ in which we shared our experiences in executing CloudFront migrations. As a further exploration of the topic, we will now discuss the key distinctions between the two services that often play a vital role in CloudFront distributions - CloudFront Functions and Lambda@Edge.

CloudFront Functions and Lambda@Edge are used not only to replicate the behaviors of other CDNs, but also to act as HTTP middleware that enables the execution of request-related code.

Despite being used for similar purposes, both of these services have notable differences and should be used in specific scenarios. A detailed listing of the differences between CloudFront Functions and Lambda@Edge will be explained, and use cases will be provided to help make the right choice for CloudFront deployments.

CloudFront Functions



CloudFront Architecture | Source: cloudacademy.com

How CloudFront Works

CloudFront stores a cache in two different layers, the Edge location and Regional Edge Cache. The Regional Edge Cache runs on the 13 AWS Regions available across the globe. Due to the complexity involved in opening and maintaining regional data centers, AWS has also established smaller data centers called Edge locations that are used to help reduce latency.

Edge locations are easier to maintain and only support a few AWS services at a time (ex: CloudFront, Route53, and AWS Shield). As of April 2022, there were 225 Edge locations spread across 47 countries. Edge locations are the actual data centers that users access when requesting content cached in CloudFront.

CloudFront Functions enable access to requests as they arrive on the Edge Location through a lightweight Javascript runtime. CloudFront Function code is executed directly on an Edge location and runs at the physical location closest to users. Since CloudFront Functions are executed before the request hits the cache and invoked for each request, the latency they incur must be kept to a minimum. To address these needs for low latency, the following limitations have been placed on CloudFront Functions:

- CloudFront Functions cannot access the body of a request
- The maximum package size for code is limited to 10kB
- CloudFront Functions cannot perform dynamic code evaluation
- CloudFront Functions cannot get Internet access
- CloudFront Functions cannot use the await/async pattern
- CloudFront Functions cannot directly access filesystems
- The date function always returns the same time (the time at which the CloudFront Function started)
- CloudFront Functions have a very limited execution time (less than 1ms). This limit is not clearly defined and is provided to the developer as a number between 0 and 100. This number represents the maximum execution time allowed as a percentage, 100 being the maximum allowed.

Lambda@Edge Functions

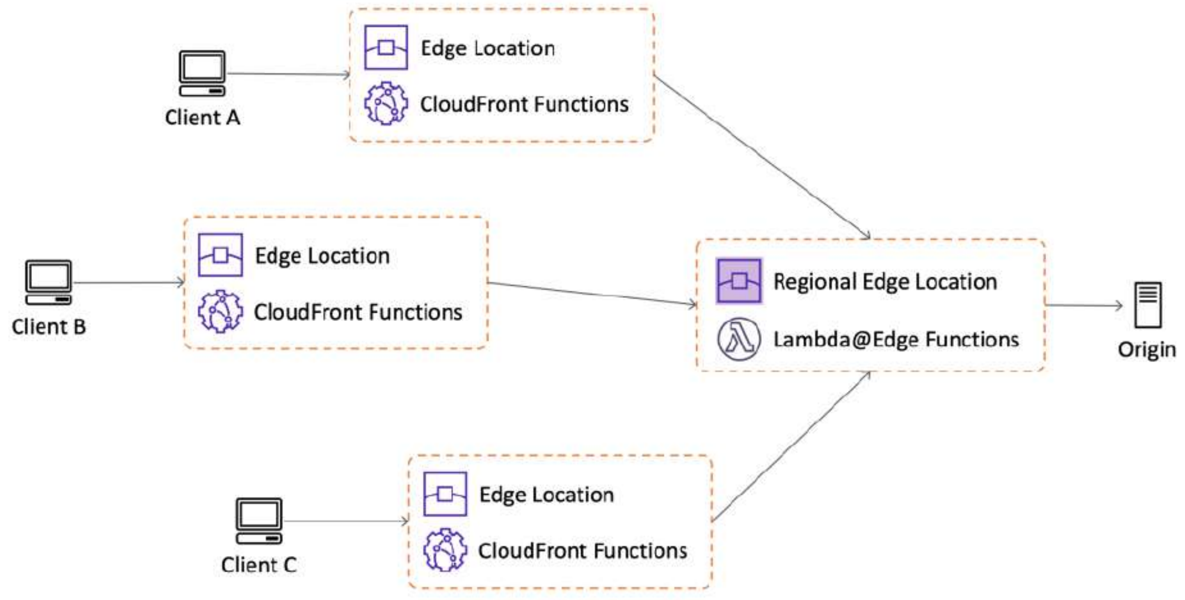
Lambda@Edge's role is similar to Cloudfront Functions. It serves as a middleware service that allows developers to run code with both Python & Javascript. The main difference is that Lambda@Edge runs on the Regional Edge Cache. Lambda@Edge Functions have fewer limitations and are very similar to conventional Lambda functions. Lambda@Edge has the following benefits:

- Lamda@Edge can access public internet
- Lambda@Edge can be run before or after your cache
- Lambda@Edge allows developers to view and modify not only the client request/response but also the origin request/response

- Lambda@Edge can access underlying filesystems
- Lambda@Edge can read the body of the request
- Lambda@Edge can use a much bigger package size for code (1MB for a client request/response trigger and 50MB for an origin request/response trigger)
- Lambda@Edge allows up to 5 seconds for a client request/response trigger and up to 30 seconds for an origin request/response trigger.

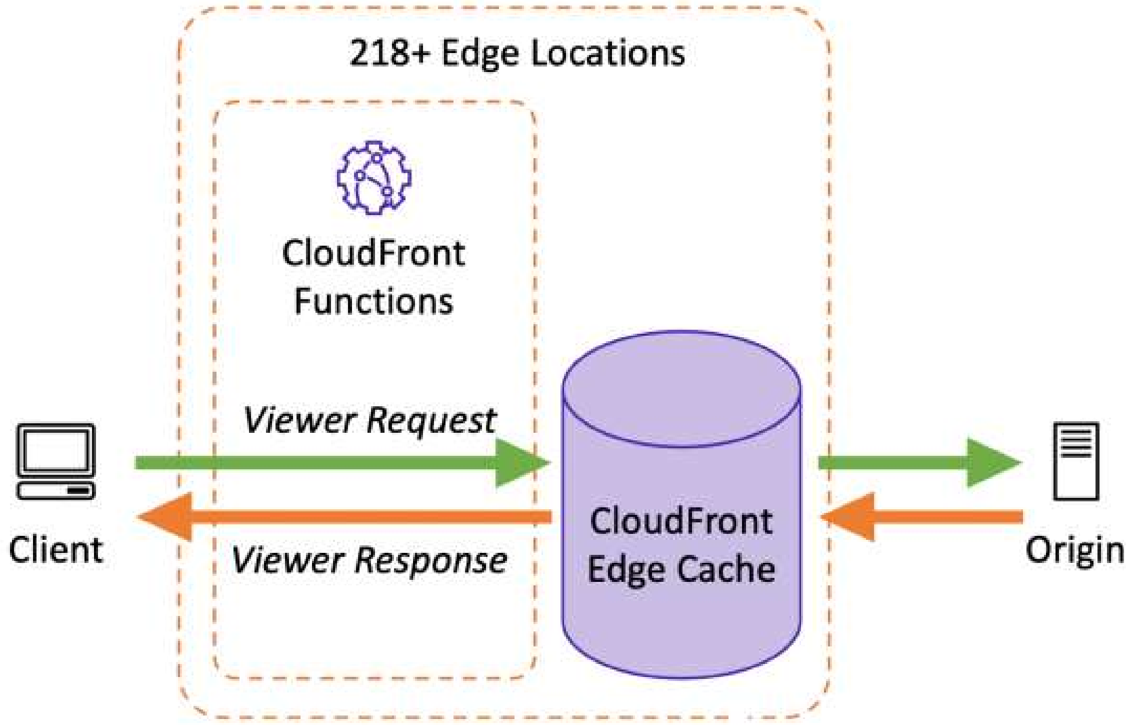
However, Lambda@Edge still differs from standard Lambda functions in the following ways:

- (Con) Lambda@Edge won't allow access to resources inside a VPC
- (Con) Developers cannot use environment variables within Lambda@Edge Functions
- (Con) Lambda@Edge Functions do not have the Lambda dead-letter queue functionality that allows developers to re-execute functions when they crash
- (Con) Lambda@Edge Functions cannot have different layers
- (Con) Lambda@Edge Functions cannot use container and ARM (Advanced RISC Machine) architecture
- (Pro) Since Lambda@Edge Functions are handled by CloudFront, developers do not need to specify reserved concurrency



CloudFront and Lambda@Edge Functions Within an Architecture | Source: doc.aws.amazon.com

In summary, Cloudfront Functions run on Edge locations that are the closest to the user but are also limited in regards to features. Lambda@Edge Functions run on Regional Edge Locations in major AWS Regions and provide more features along with an increased execution time capability.



CloudFront Request Flow

When to Choose CloudFront Functions

Cloudfront Functions are ideal for short tasks that do not require reading the body of the request. The following are a few basic use cases for CloudFront Functions:

CloudFront Use Case #1: Redirecting Traffic Based on Simple Conditions

CloudFront Functions can be useful to redirect traffic based on simple conditions. For instance, a user could choose to redirect traffic coming from a specific country of origin using the HTTP header `CloudFront-Viewer-Country`. In the example below, the HTTP header contains the ISO3166 two-letter code of the country of origin.

```
function handler(event) {
    var request = event.request;
    var supportedCountries = ['de', 'it', 'fr'];

    if (request.uri.substr(3, 1) != '/') {
        var headers = request.headers;
        var nextUri;
        if (headers['cloudfront-viewer-country']) {
            var countryCode =
headers['cloudfront-viewer-country'].value.toLowerCase();
            if (supportedCountries.includes(countryCode)) {
                nextUri = '/' + countryCode + request.uri;
            }
        }

        if (!nextUri) {
            nextUri = '/en' + request.uri;
        }

        return {
            statusCode: 302,
            statusDescription: 'Found',
            headers: {
                location: { value: newUri }
            }
        }
    }

    return request;
}
```

Function Redirecting Traffic Coming from a Specific Country of Origin

The CloudFront function in this example checks (Line #5) to see whether a country code exists in the request URL and whether the account has access to the specific `CloudFront-Viewer-Country` header. For countries that are specified in the `supportedCountries` array, their country codes are added to the beginning of the URL (Line #12) . If the country code is not in the `supportedCountries` array, the code defaults to /en (Line #17). Finally, the function redirects the user to the new URL (Lines #20-26) or if the country code was already in the `supportedCountries` array, the request is forwarded as it is (Line #29).

CloudFront Use Case #2: Modifying the CloudFront Cache Key

CloudFront Functions have been specifically designed to assist in modifying headers, cookies, and query strings of requests. For instance, CloudFront functions can be used to easily script the creation of an HTTP header for a cache policy.

CloudFront Functions can be used to modify the CloudFront cache key for a username of an authenticated request using a basic authentication mechanism.

The basic authentication mechanism works as follows:

- To authenticate a request, the header Authorization must be added containing the word `Basic` followed by a space and then a base64 encoding of the username and password separated by *:* (For example: `Basic base64(username:password)`, resulting in the following: `Basic dXNlcm5hbWU6cGFzc3dvcmQ=`).

```
const crypto = require('crypto');

function hashText(input) {
    return crypto.createHash('md5').update(input).digest('hex');
}

function handler(event) {
    var request = event.request;
    var headers = request.headers;

    delete headers['username-cache-key'];

    if (headers['authorization']) {
        var authorizations = headers['authorization'].split(' ');
        if (authorizations.length === 2 && authorizations[0].toLowerCase()
=== 'basic') {
            var credentials = atob(authorizations[1]).split(':');

            headers['username-cache-key'] = { value:
hashText(credentials[0]) };
        }
    }

    return request;
}
```

Function Creating a Cache Key for a Username

The CloudFront function in this example begins by ensuring that the received request does not contain the cache key (Line #11). If the request contains an Authorization header, the content of this header is split with a spacebar (Lines #13-14). The function checks if the result of the split has a length of 2. For the the header provided above containing the word `Basic` and the base64 encoded credentials (`dXNlcm5hbWU6cGFzc3dvcmQ=`), the first element of the split array is the word `Basic` (Line #15) and the second element is (`dXNlcm5hbWU6cGFzc3dvcmQ=`). The second element is decoded and split using a *:* (Line #16). This results in the creation of a new array that contains two cells, the username and the password. The username is hashed using the md5 algorithm and added to the request as an HTTP header (Line #18). The request is then forwarded to the next step (Line #22).

After setting up this CloudFront function, a custom cache policy needs to be configured within the AWS CloudFront console using the `Username-Cache-Key` HTTP header as a custom cache key.

Use Case #3: Managing CORS, CSP, X-Frame-Options, and other Security HTTP Headers

Since CloudFront Functions allow you to easily manipulate HTTP headers, they can be used to automatically add standard HTTP security headers to a request.

```
function handler(event) {
    var response = event.response;
    var headers = response.headers;

    headers['strict-transport-security'] = { value: 'max-age=63072000;
includeSubdomains; preload'};
    headers['content-security-policy'] = { value: "default-src 'none';
img-src 'self'; script-src 'self'; style-src 'self'; object-src 'none'"};
    headers['x-content-type-options'] = { value: 'nosniff'};
    headers['x-frame-options'] = {value: 'DENY'};
    headers['x-xss-protection'] = {value: '1; mode=block'};
    headers['access-control-allow-origin'] = {value: "*"};

    return response;
}
```

Function Adding Static HTTP Security Headers

CloudFront Functions help add standard W3C (World Wide Web Consortium) defined HTTP headers to the response. The different security headers are listed below:

- Strict-Transport-Security: Forces the browser to use HTTPS
- Content-Security-Policy: Controls the sources from which assets can be loaded
- X-Content-Type-Options: Forces the browser to follow the content-type advertised by the server
- X-Frame-Options: Denies the use of the requested page in an iframe
- X-XSS-Protection: Blocks external script load when cross-scripting attacks are detected
- Access-Control-Allow-Origin: Controls the domains from which the page can be loaded

When to Choose Lambda@Edge Functions

It is important to note that Lambda@Edge can run before and after the CloudFront Regional Edge Cache but not before the CloudFront Edge Cache like CloudFront Functions. Lambda@Edge functions are not the ideal choice for short tasks but are useful for accessing networks, filesystems, and request bodies.

Lambda@Edge Use Case #1: Redirecting Request by Country or Latency

Lambda@Edge can redirect traffic to the appropriate Amazon S3 Origin Region so a bucket closer to the viewer can be accessed. This helps to reduce latency when the Region specified is closer to the viewer's country. There are two ways to activate this capability:

1. The simplest process is to look at the `cloudfront-viewer-country`.

   (Note: This can also be done through CloudFront Functions since they don't require access to the body nor the network as discussed in `CloudFront Use Case #1: Redirecting Traffic Based on Simple Condition`.)

2. In a scenario where there are two buckets, one in Europe and the other in the USA, a TXT Latency record can be created in Route53 to enable the testing of lower latency in the Lambda@Edge Function:

```python
# Need to install DNS package
import dns.resolver

def lambda_handler(event, context):
   request = event['Records'][0]['cf']['request']
   res = dns.resolver.query("latency-cross-region.example.com", "txt")
   buff = res[0].to_text().replace('"', '').split(";")
   bucket_name = buff[0]
   region = buff[1]
   domainName = f"{bucket_name}.s3.{region}.amazonaws.com"
   request['origin']['s3']['domainName'] = domainName
   request['origin']['s3']['path'] = ''
   request['origin']['s3']['region'] = region
   request['headers']['host'] = [{'key':'Host', 'value': domainName}]
   return request
```

Function Testing S3 origin latency

The Lambda@Edge Function in this example queries the DNS records that have been created (Line #6) and retrieves the region and bucket name with lower latency (Lines #7-10). The function updates the request origin S3 domain name, region, and path (Lines #11-14) and then returns the request.

Lambda@Edge Use Case #2: Authenticating with JSON Web Tokens (JWT)

To verify whether a user is authenticated to access origin content, developers can use Lambda@Edge with external libraries such as JWT to perform the verification. The code snippet below is an example of Lambda@Edge functions with Amazon Cognito authentication by the awslab:

(Note: A verification process can also be implemented using CloudFront Functions. However, developers will not be able to use external libraries such as JWT.)

```
'use strict';
var jwt = require('jsonwebtoken');
var jwkToPem = require('jwk-to-pem');

var USERPOOLID = '##USERPOOLID##';
var JWKS = '##JWKS##';
var COGNITOREGION = '##COGNITOREGION##';

var iss = 'https://cognito-idp.' + COGNITOREGION + '.amazonaws.com/' + USERPOOLID;
var pems;

pems = {};
var keys = JSON.parse(JWKS).keys;
for(var i = 0; i < keys.length; i++) {
   //Convert each key to PEM
   var key_id = keys[i].kid;
   var modulus = keys[i].n;
   var exponent = keys[i].e;
   var key_type = keys[i].kty;
   var jwk = { kty: key_type, n: modulus, e: exponent};
   var pem = jwkToPem(jwk);
   pems[key_id] = pem;
}

const response401 = {
   status: '401',
   statusDescription: 'Unauthorized'
};

exports.handler = (event, context, callback) => {
   const cfrequest = event.Records[0].cf.request;
   const headers = cfrequest.headers;
   console.log('getting started');
   console.log('pems=' + pems);

   //Fail if no authorization header found
   if(!headers.authorization) {
       console.log("no auth header");
       callback(null, response401);
```

```
        return false;
    }

    //strip out "Bearer " to extract JWT token only
    var jwtToken = headers.authorization[0].value.slice(7);
    console.log('jwtToken=' + jwtToken);

    //Fail if the token is not jwt
    var decodedJwt = jwt.decode(jwtToken, {complete: true});
    if (!decodedJwt) {
        console.log("Not a valid JWT token");
        callback(null, response401);
        return false;
    }

    //Fail if token is not from your UserPool
    if (decodedJwt.payload.iss != iss) {
        console.log("invalid issuer");
        callback(null, response401);
        return false;
    }

    //Reject the jwt if it's not an 'Access Token'
    if (decodedJwt.payload.token_use != 'access') {
        console.log("Not an access token");
        callback(null, response401);
        return false;
    }

    //Get the kid from the token and retrieve corresponding PEM
    var kid = decodedJwt.header.kid;
    var pem = pems[kid];
    if (!pem) {
        console.log('Invalid access token');
        callback(null, response401);
        return false;
    }

    console.log('Start verify token');

    //Verify the signature of the JWT token to ensure it's really coming from your User Pool
    jwt.verify(jwtToken, pem, { issuer: iss }, function(err, payload) {
      if(err) {
        console.log('Token failed verification');
        callback(null, response401);
        return false;
      } else {
        //Valid token.
        console.log('Successful verification');
        //remove authorization header
        delete cfrequest.headers.authorization;
        //CloudFront can proceed to fetch the content from origin
        callback(null, cfrequest);
        return true;
      }
    });
};
```
Lambda@Edge for authentication with Cognito | Source: awslab

The Lambda@Edge function in this example verifies whether an authorization header has been provided (Lines #37-41). It extracts the token from the header (Line #44) and then decodes the token (Line #48). The function then verifies that the decoded token is valid for the Cognito User Pool (Lines #56-76) and checks whether the token is really coming from the User Pool (Lines #81-96). If there is an error during the verification, the function returns a response with the status code 401. If no errors occur, the function simply goes through the content of the origin.

Conclusion

CloudFront Functions and Lambda@Edge have notable differences and should be deployed based on application requirements, as each service has its own benefits and limitations. CloudFront Functions are useful when running small scripts that are closest to users. Lambda@Edge Functions are better suited for more complex code that requires access to external services or the Internet.

ABOUT TRACKIT

TrackIt is an Amazon Web Services Advanced Consulting Partner specializing in cloud management, consulting, and software development solutions based in Marina del Rey, CA. TrackIt specializes in Modern Software Development, DevOps, Infrastructure-As-Code, Serverless, CI/CD, and Containerization with specialized expertise in Media & Entertainment workflows, High-Performance Computing environments, and data storage.

TrackIt's forté is cutting-edge software design with deep expertise in containerization, serverless architectures, and innovative pipeline development. The TrackIt team can help you architect, design, build and deploy a customized solution tailored to your exact requirements.

In addition to providing cloud management, consulting, and modern software development services, TrackIt also provides an open-source AWS cost management tool that allows users to optimize their costs and resources on AWS.