# Cache Key and Origin Requests

When working with Amazon CloudFront, optimizing caching and controlling how content is fetched from the origin are critical for performance and cost efficiency. CloudFront uses **cache keys** to determine how requests are handled at the edge locations, and the way origin requests are managed influences what data is fetched from your origin.

---

## 1. Cache Key Overview

The **cache key** is a combination of parameters that CloudFront uses to identify a specific version of an object to store at its edge locations. When a request is made, CloudFront checks the cache key associated with the request to determine whether the content is already cached or if it needs to fetch the object from the origin.

The key components of a cache key include:

- **Request URL path** (the path to the file being requested)
- **Query string parameters**
- **Request headers** (e.g., cookies, user-agent, authorization headers)
- **Protocol** (HTTP vs. HTTPS)
- **Request body** (for POST requests)

---

## 2. Components of a Cache Key

### 2.1 URL Path

- The primary element of the cache key is the **URL path** (e.g., `/images/logo.png`). CloudFront distinguishes content based on different URL paths, and each unique path is treated as a separate cache key.

### 2.2 Query String Parameters

- Some applications use **query strings** to pass additional data, such as user identifiers, search queries, or filters (e.g., `/images/logo.png?size=large`).
- You can configure CloudFront to:
    - **Ignore Query Strings**: Use the URL path as the sole cache key, ignoring any query parameters.
    - **Include All Query Strings**: Include all query parameters in the cache key (this is the default behavior).

    - **Whitelist Specific Query Strings**: Cache only the specific query strings that are important for your application, reducing the number of unique cache entries.

### 2.3 Request Headers

- **Headers** provide additional information about the request (e.g., browser type, language preferences). Including request headers in the cache key allows you to cache different versions of the same object based on these headers.
- Some common headers include:
    - **Accept-Encoding**: Used to cache different versions based on compression (e.g., Gzip or Brotli).
    - **User-Agent**: Caching based on the type of device (e.g., mobile vs. desktop).
    - **Authorization**: If your application requires authenticated access, including this header can result in multiple versions of cached content, which might not be desirable.
- You can configure CloudFront to:
    - **Ignore Headers**: Do not include any request headers in the cache key.
    - **Include All Headers**: Cache different versions for every possible header combination.
    - **Whitelist Specific Headers**: Cache only based on a few select headers (e.g., `Accept-Encoding`).

### 2.4 Cookies

- Cookies can be used to store session data or user-specific preferences, and they can also influence what content is delivered.
- You can configure CloudFront to:
    - **Ignore Cookies**: Serve the same cached content regardless of cookies.
    - **Include All Cookies**: Different cache entries are created for each unique cookie.
    - **Whitelist Specific Cookies**: Cache only based on specific cookies (e.g., session IDs or user preferences).

### 2.5 Protocol (HTTP vs. HTTPS)

- By default, CloudFront distinguishes between HTTP and HTTPS requests. This means content requested via HTTP will be cached separately from HTTPS requests, even if the URL path is the same.

### 2.6 Request Body

- For **POST requests**, you can include the **request body** in the cache key if the content varies based on the request's payload. However, in most cases, POST requests aren't cached by default.

---

## 3. Cache Behavior Settings

Cache behavior defines how CloudFront handles requests and which parts of a request (path, query, headers, etc.) it uses to create the cache key.

### 3.1 Cache Policy

CloudFront allows you to create custom cache policies that define which request components are included in the cache key.

- **Default Cache Policy**: CloudFront uses a default caching policy that includes query strings, headers, and cookies.
- **Custom Cache Policy**: You can create custom cache policies to optimize caching by specifying which query strings, headers, and cookies should be part of the cache key.

### 3.2 Origin Request Policy

The **origin request policy** defines which parts of a request (query strings, headers, and cookies) are forwarded to the origin. By fine-tuning this policy, you can control how often CloudFront fetches content from the origin.

Key considerations:

- **Forwarding unnecessary headers, query strings, or cookies** to the origin can result in more requests to the origin, increased costs, and lower cache efficiency.
- By **restricting the number of request components forwarded**, you can reduce the load on your origin and improve cache hit ratios.

---

## 4. Origin Requests

An **origin request** happens when CloudFront needs to retrieve content from the origin because there is either:

- **No cached copy** of the requested content at the edge location, or
- The **cache has expired**, and CloudFront needs to refresh it by making a new request to the origin.

### 4.1 When CloudFront Sends an Origin Request

CloudFront sends an origin request when:

1. **A cache miss occurs**: The requested object is not found in the edge location's cache.
2. **Cache expiration**: The cached object has reached its time-to-live (TTL) limit, and a new version must be fetched from the origin.

3. **Cache invalidation**: If you manually invalidate a cached object, CloudFront fetches a new copy from the origin.
4. **Dynamic content**: If you're serving dynamic content, CloudFront fetches it directly from the origin.

### 4.2 Reducing Origin Requests

To minimize the number of origin requests and optimize performance, consider the following strategies:

- **Increase TTL**: By configuring a longer time-to-live (TTL) in your cache policy, CloudFront can cache content for a longer period before making another origin request.
- **Fine-tune Cache Keys**: By reducing the number of query strings, headers, or cookies included in the cache key, you decrease the number of cache variations, which results in higher cache hit rates and fewer origin requests.
- **Use Stale Content**: CloudFront can serve stale content while it fetches updated content from the origin in the background. This reduces the latency experienced by end users.

### 4.3 Origin Shield

Origin Shield is an additional layer of caching between CloudFront and your origin. It reduces the number of origin requests by consolidating requests from multiple edge locations and ensuring that the origin is hit less frequently.

---

## 5. Caching and Origin Behavior: Key Strategies

### 5.1 Optimizing for Static Content

- For **static assets** (e.g., images, CSS, JS), configure a cache key that ignores query strings, headers, and cookies. This ensures that CloudFront can cache as many variations as possible, resulting in fewer origin requests and faster load times.
- Use **long TTLs** for static content, especially if the content rarely changes.

### 5.2 Optimizing for Dynamic Content

- For **dynamic content** (e.g., personalized pages, APIs), you may need to include specific query strings, headers, or cookies in the cache key to ensure that personalized versions of content are cached appropriately.
- Consider using **shorter TTLs** for dynamic content, as it may change more frequently.

### 5.3 Cache Invalidation

- You can invalidate specific objects from the cache if they are updated before their TTL expires. This forces CloudFront to fetch a fresh copy from the origin. However, cache invalidations may come with extra costs, so use them judiciously.

---

## 6. Origin Request Policy vs. Cache Policy

CloudFront has two distinct but related policies to manage cache behavior and origin requests:

### 6.1 Cache Policy

- Controls what parts of the request (path, query strings, headers, cookies) are included in the **cache key**.
- Optimizes caching at edge locations by reducing unnecessary cache variations.

### 6.2 Origin Request Policy

- Controls which parts of the request are forwarded to the **origin** when CloudFront makes an origin request.
- Minimizes the amount of data sent to the origin to reduce origin load and improve performance.

### 6.3 Balancing the Two Policies

- To maximize performance, you should configure the cache policy to minimize the number of cache key variations, while configuring the origin request policy to forward only the necessary components to the origin.

---

## 7. Monitoring and Troubleshooting Cache and Origin Requests

### 7.1 Amazon CloudWatch Metrics

CloudFront integrates with CloudWatch to provide insights into:

- **Cache hit/miss ratios**: Helps identify how often requests are served from the cache vs. the origin.
- **Origin request counts**: Shows how many times CloudFront is making requests to the origin.

### 7.2 Real-time Logging

You can enable real-time logging to track how requests are processed and to understand how cache keys and origin requests are being handled. This is especially useful for debugging cache issues or high origin load.

### 7.3 AWS WAF and Security

To protect your origin from malicious requests, you can integrate **AWS WAF** (Web Application Firewall) with CloudFront to block unwanted traffic at the edge, reducing the number of origin requests.