Anisha Implement - This is a assigment related to AWS

Managing cloud solutions (Lovely Professional University)

Accelerated Image Processing with AWS Lambda & Amazon CloudFront

Name: Anisha Kumari

Registration No: 12312596

Roll no: 15

Sec: K23QC

Description:

This solution leverages AWS Lambda for serverless image processing and Amazon CloudFront for global content delivery, creating a highly scalable and cost-efficient architecture for dynamic image transformations. By processing images on-demand (resizing, format conversion, compression, and optimization) and caching results at edge locations, the system ensures low-latency delivery worldwide. The serverless approach eliminates the need for managing dedicated servers, automatically scales with traffic, and reduces costs by charging only for actual compute usage. This architecture is ideal for applications requiring real-time image processing, such as e-commerce platforms, media websites, and mobile apps, where fast load times and adaptive image delivery are critical for user experience.

Scenario:

An e-commerce company experiences slow page load times due to unoptimized product images, leading to higher bounce rates and lost sales. Their current system relies on pre-generated image variants stored on origin servers, which increases storage costs and complicates management. During peak traffic (e.g., flash sales), their servers struggle to handle the load, causing delays and degraded performance. Additionally, they need to support multiple device resolutions (desktop, mobile, tablet) and modern formats (WebP, AVIF) without manually generating thousands of image versions. A serverless, CDN-backed solution would ensure fast delivery, automatic scaling, and reduced infrastructure overhead.

Problem Statement:

The company requires a scalable, low-latency image processing system that can dynamically resize, compress, and convert images on-the-fly based on URL parameters (e.g., image.jpg?width=800&format=webp), eliminating the need to store thousands of pre-generated variants and significantly reducing storage costs. The solution must leverage a global CDN (CloudFront) to serve optimized images from edge locations, ensuring sub-second latency for users worldwide while automatically scaling to handle unpredictable traffic spikes without manual intervention. High availability and fault tolerance are critical, requiring a distributed architecture with no single point of failure, capable of processing millions of requests during peak loads. Additionally, the system must support modern image formats like WebP and AVIF for superior compression (30-50% smaller than JPEG/PNG), enabling faster page loads and reduced bandwidth consumption. Security

measures, such as hotlink protection and access control, must be implemented to prevent unauthorized usage, while real-time monitoring ensures optimal performance and cost efficiency. This approach not only enhances user experience but also aligns with Core Web Vitals for better SEO rankings, making it a comprehensive solution for media-heavy applications.

Objectives:

1. **Serverless Processing**: Use **AWS Lambda** to process images on-demand, eliminating the need for dedicated servers.
2. **Global Caching**: Leverage **Amazon CloudFront** to cache processed images at edge locations, reducing origin load and improving delivery speed.
3. **Automated Optimization**: Apply dynamic compression, resizing, and format conversion based on device type and network conditions.
4. **Cost Efficiency**: Minimize expenses by only processing images when needed and optimizing cache TTLs.
5. **Scalability**: Ensure the system can handle sudden traffic surges (e.g., viral content or sales events) without performance degradation.
6. **Security**: Protect against hotlinking and unauthorized access using **CloudFront signed URLs** and **S3 bucket policies**.

Outcomes:

- Faster Load Times: Images are delivered from the nearest CloudFront edge location, reducing latency by 50-70%.
- Reduced Costs: Pay-per-use Lambda pricing and efficient caching lower infrastructure expenses compared to traditional servers.
- Simplified Management: No need to pre-generate or store multiple image variants—processing happens dynamically.
- Automatic Scaling: Handles traffic spikes seamlessly without manual scaling.
- Improved SEO & UX: Faster image loading enhances page performance, improving search rankings and user engagement.

Proposed AWS Components:

- AWS Lambda: Executes serverless image transformations (using libraries like Sharp or Pillow).
- Amazon CloudFront: Caches processed images globally and serves them at edge locations.

- Amazon S3: Stores original high-resolution images as the source for processing.
- Lambda@Edge: Processes images closer to users for even lower latency.
- Amazon CloudWatch: Monitors performance, logs errors, and triggers alerts for failures.
- AWS IAM: Manages permissions securely to restrict access to S3 and Lambda functions.

Solution:

Implementation 1: Accelerated Image Processing with AWS Lambda & CloudFront
The plan is to build a serverless, globally distributed image processing system that dynamically transforms and delivers optimized images with minimal latency using AWS services

Steps:

1. Image Storage & Origin Setup

- Store original high-resolution images in Amazon S3 with versioning enabled for backup.
- Configure S3 bucket policies to restrict access, allowing only CloudFront and Lambda to retrieve images.
- Use S3 Lifecycle Policies to automatically archive unused originals to S3 Glacier Deep Archive for cost savings.

2. Dynamic Image Processing with AWS Lambda

- Develop a Lambda function (Node.js/Python) using Sharp (Node) or Pillow (Python) for fast resizing, compression, and format conversion.
- Configure the function to accept parameters (e.g., width=500&format=webp) via URL query strings.
- Optimize Lambda memory settings (e.g., 1024MB) for faster execution and lower costs.

3. CloudFront CDN Configuration

- Create a CloudFront distribution with the S3 bucket as the origin.
- Enable query string forwarding to pass transformation parameters (e.g.,? width=800&quality=70) to Lambda.
- Set optimal TTLs (e.g., 30 days) for cached images to balance freshness and cache efficiency.
- Configure Lambda@Edge (optional) for processing at edge locations, reducing latency further.

4. Automated Request Handling

- Use CloudFront Behaviors to route image requests:
- Cache Hit: Serve the processed image directly from the nearest edge location.

- Cache Miss: Trigger the Lambda function to generate and cache the new variant.
- Implement URL signing to prevent hotlinking and unauthorized access.

5. Optimization & Performance Tuning

- Enable WebP/AVIF auto-detection for supported browsers (reduces file size by 30-50%).
- Apply lazy loading for offscreen images to improve page load performance.
- Use CloudFront compression (Gzip/Brotli) to minimize bandwidth usage.

6. Monitoring & Cost Management

- Set up CloudWatch Alarms to track Lambda errors, high latency, or throttling.
- Monitor CloudFront cache hit ratio (target >90%) to ensure efficient caching.
- Use AWS Cost Explorer to analyze Lambda invocation costs and optimize memory settings.

7. Testing & Validation

- Load test with tools like Apache JMeter to simulate 10,000+ RPS and verify auto-scaling.
- Validate RPO (near-zero data loss) and RTO (sub-second delivery for cached images).
- Conduct real-user monitoring (RUM) to measure improvements in Core Web Vitals (LCP, CLS).

Evaluation Criteria:

Criteria	Description
Latency	Measure time from request to delivery (should be <200ms for cached images).
Cache Hit Ratio	High ratio (e.g., >90%) indicates effective caching.
Cost Efficiency	Compare Lambda costs vs. traditional server expenses.
Scalability	Test under heavy load (e.g., 10,000+ requests/sec).
Image Quality	Ensure visual integrity after compression.
Global Performance	Verify fast delivery across regions (US, EU, Asia).

How we can deploy this in AWS Platform here is the steps :

- **Create S3 bucket**

  AWS Console → S3 → Create bucket

Enable **Versioning**

Management → Lifecycle rules → Add rule → After 90 days → Transition to Glacier Deep Archive

- **Upload original images**

  S3 bucket → Upload → select your high-res images under folder products/

- **Create IAM role for Lambda**

  AWS Console → IAM → Roles → Create role → Lambda

  Attach **AWSLambdaBasicExecutionRole**

- Add inline policy:

```
{
    "Effect":"Allow",
"Action":"s3:GetObject",
    "Resource":"arn:aws:s3:::<your-bucket-name>/*"
    }
```

- **Create Lambda function**

  AWS Console → Lambda → Create function → Author from scratch

  Name: ImageProcessor

  Runtime: Node.js 18.x

  Role: your IAM role

  Memory: 1024 MB, Timeout: 10 s

  Code → Upload ZIP containing index.js + node_modules/sharp

  Configuration → Environment variables → IMAGE_BUCKET=<your-bucket-name>

- **Test Lambda**

  Lambda → Test → Configure test event:

```
{
"queryStringParameters": {
"key":"products/example.jpg",
"width":"800",
"format":"webp"
}
}
```

  Run → confirm 200 response with base64 image

- **Create CloudFront distribution**

  AWS Console → CloudFront → Create distribution → Web

  Origin domain: your S3 bucket

  Origin access: Create new OAI → Update bucket policy

  Default cache behavior → Viewer protocol: Redirect HTTP to HTTPS

  Cache key and origin requests → Forward all query strings

  Lambda Function Associations → Origin Request → select your function's latest version ARN

- **Lock down S3 bucket**

  S3 → Permissions → Bucket policy → allow only your CloudFront OAI and Lambda role to s3:GetObject

- **Deploy & wait**

CloudFront distribution → Save → wait for **Deployed** status

- **Validate**
  Browser →
  https://<CLOUDFRONT_DOMAIN>/products/example.jpg?key=products/example.jpg&width=800&format=webp
  Confirm resized, converted image loads successfully.