Cloud Gallery

PhotoVault

Technical Architecture Documentation
AWS Serverless Image Processing System

Services Used:

Amazon S3 • AWS Lambda • Amazon Rekognition
Amazon Bedrock • Amazon DynamoDB • Amazon VPC

November 2025

Table of Contents

Project Overview

The Cloud Gallery is a serverless image management application built on AWS infrastructure. The system offers automated image processing, AI-powered analysis, smart captioning, and metadata storage features. This document describes the technical architecture and implementation of various AWS services used in the project.

Architecture Overview

The architecture follows a serverless, event-driven design pattern. When images are uploaded to S3, Lambda functions are triggered automatically to process the images through various AWS AI services (Rekognition and Bedrock), with results stored in DynamoDB. The entire infrastructure is deployed within a Virtual Private Cloud (VPC) for enhanced security and network isolation.

AWS Services Implementation

1. Amazon S3 (Simple Storage Service)

    Purpose and Role

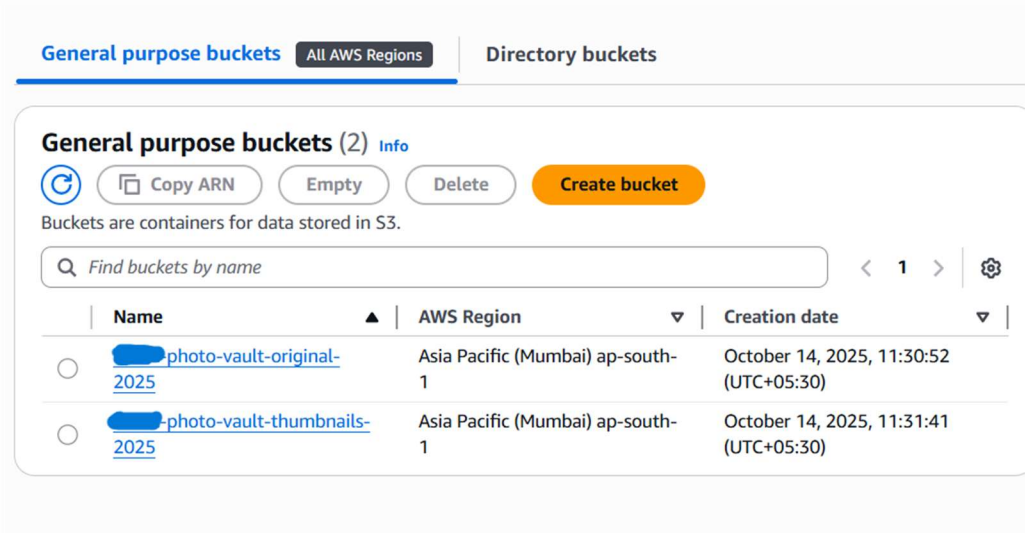
    S3 serves as the primary storage layer for the Cloud Gallery application. It handles image uploads, storage, and acts as the event source that triggers the processing pipeline.

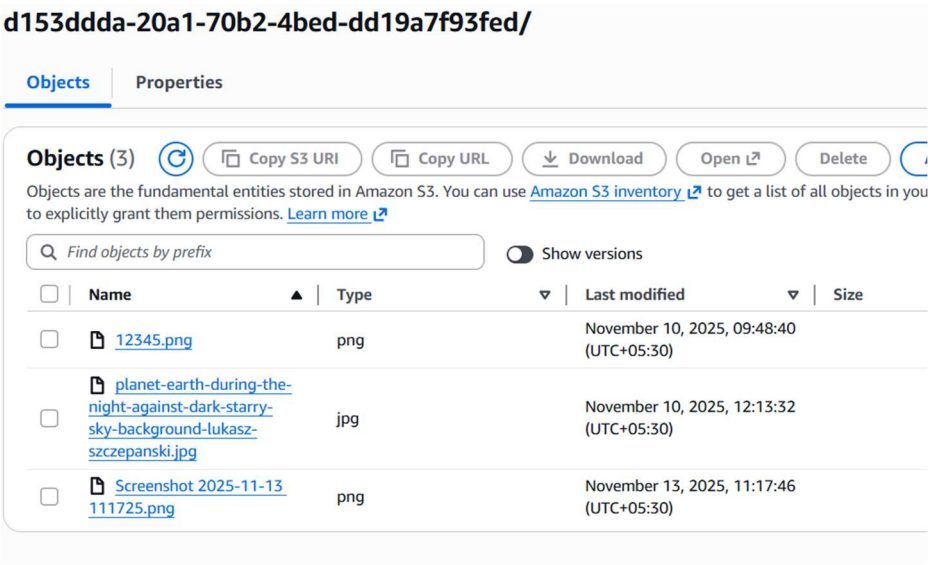
Implementation Details

• Bucket Configuration: Created a dedicated S3 bucket with versioning enabled to maintain image history and allow rollback capabilities
• Event Notifications: Configured S3 event notifications (s3:ObjectCreated:*) to trigger Lambda functions when new images are uploaded
• Storage Classes: Implemented intelligent tiering to optimize storage costs for frequently and infrequently accessed images
• Access Control: Bucket policies and IAM roles configured to restrict access, ensuring only authorized Lambda functions can read/write
• CORS Configuration: Set up Cross-Origin Resource Sharing (CORS) rules for web-based uploads
• Encryption: Server-side encryption (SSE-S3) enabled for all objects at rest

Key Features Implemented

• Pre-signed URLs for secure, temporary upload access
• Lifecycle policies for automatic archival of old images
• S3 Select for efficient metadata queries
• Multi-part upload support for large image files





2. AWS Lambda

Purpose and Role

Lambda functions serve as the compute layer, executing serverless code in response to S3 events. They orchestrate the entire image processing workflow, coordinating between S3, Rekognition, Bedrock, and DynamoDB.

Implementation Details

• Runtime Environment: Python 3.x runtime selected for compatibility with boto3 SDK and AI/ML libraries
• Execution Role: Custom IAM role with policies allowing access to S3 (read), Rekognition (detect), Bedrock (invoke), and DynamoDB (write)
• VPC Configuration: Lambda functions deployed within private subnets of the VPC for secure communication with other services
• Memory and Timeout: Configured with 512MB-1024MB memory and 5-minute timeout to handle image processing workloads

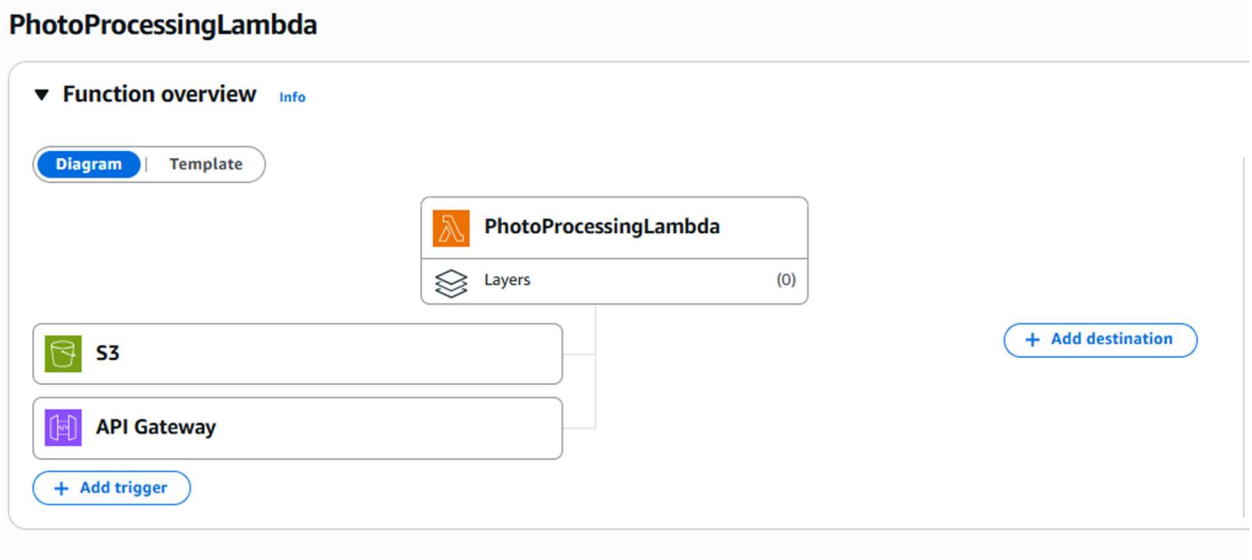
• Environment Variables: Stored configuration like bucket names, DynamoDB table names, and region information
• Layers: Added custom layers for image processing libraries (Pillow, OpenCV) to keep deployment package size manageable

Lambda Function Workflow

1. Triggered by an S3 event when an image is uploaded
2. Extract image metadata (filename, size, upload timestamp)
3. Download image from S3 to /tmp directory for processing
4. Call Rekognition API for image analysis (labels, faces, text detection)
5. Invoke Bedrock API for AI-generated captions
6. Aggregate all results and metadata
7. Store processed data in DynamoDB
8. Clean up temporary files and return a success response

Error Handling and Monitoring

• CloudWatch Logs integration for debugging and monitoring
• Dead Letter Queue (DLQ) configured for failed invocations
• Retry logic with exponential backoff for API calls
• Custom metrics published to CloudWatch for tracking processing time and success rates.



Lambda Function Code

Below is the Python code for the Lambda function that orchestrates the entire image processing pipeline:

```
// lambda_function.js
const { S3Client, GetObjectCommand } = require("@aws-sdk/client-
s3");
const { RekognitionClient, DetectLabelsCommand } =
require("@aws-sdk/client-rekognition");
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient, PutCommand } = require("@aws-
sdk/lib-dynamodb");
const { getSignedUrl } = require("@aws-sdk/s3-request-
presigner");
const { BedrockRuntimeClient, InvokeModelCommand } =
require("@aws-sdk/client-bedrock-runtime");

const s3Client = new S3Client({ region: "ap-south-1" });
const rekognitionClient = new RekognitionClient({ region: "ap-
south-1" });
const dynamoClient = DynamoDBDocumentClient.from(new
DynamoDBClient({ region: "ap-south-1" }));
```

```
const bedrockClient = new BedrockRuntimeClient({ region: "ap-
south-1" });

const TABLE_NAME = "PhotoMetaData";
const BUCKET_NAME = "*****-photo-vault-original-2025";  // ← ADD
THIS

exports.handler = async (event) => {
  console.log("FULL EVENT:", JSON.stringify(event, null, 2));

  // ============ S3 UPLOAD TRIGGER ============
  if (event.Records && event.Records[0]?.s3) {
    try {
      const record = event.Records[0].s3;
      const bucket = record.bucket.name;
      const key =
decodeURIComponent(record.object.key.replace(/\+/g, " "));

      console.log("Raw S3 key:", key);

      // DO NOT STRIP public/ — it's real
      console.log("Using full S3 key:", key);

      // Extract userId from: public/user_data/{userId}/filename
      const parts = key.split("/");
      if (parts.length < 4) throw new Error("Invalid S3 key
structure");
      const userId = parts[2];
      console.log("Extracted userId:", userId);

      // Rekognition — use full key
      console.log("Calling Rekognition on:", { bucket, key });
      const rekognitionResponse = await rekognitionClient.send(
        new DetectLabelsCommand({
          Image: { S3Object: { Bucket: bucket, Name: key } },
          MaxLabels: 10,
          MinConfidence: 70,
        })
      );
```

```
      const labels = (rekognitionResponse.Labels || []).map(l =>
l.Name || "");
      console.log("Rekognition labels:", labels);

      // Signed URL
      const signedUrl = await getSignedUrl(
        s3Client,
        new GetObjectCommand({ Bucket: bucket, Key: key }),
        { expiresIn: 86400 * 7 }
      );

      // Bedrock caption
      let caption = "photo";
      try {
        const prompt = `Write a short, friendly caption using
these tags: ${labels.join(", ")}. Use "you" or "me". Under 12
words.`;
        const command = new InvokeModelCommand({
          modelId: "anthropic.claude-3-haiku-20240307-v1:0",
          contentType: "application/json",
          accept: "application/json",
          body: JSON.stringify({
            anthropic_version: "bedrock-2023-05-31",
            max_tokens: 50,
            messages: [{ role: "user", content: prompt }],
          }),
        });

        const resp = await bedrockClient.send(command);
        const parsed = JSON.parse(new
TextDecoder().decode(resp.body));
        caption = parsed.content[0].text.trim();
        console.log("Bedrock caption:", caption);
      } catch (e) {
        console.error("Bedrock failed:", e);
        caption = labels[0] ? `${labels[0].toLowerCase()}` :
"photo";
      }

      // Save to DynamoDB
      await dynamoClient.send(
```

```
        new PutCommand({
          TableName: TABLE_NAME,
          Item: {
            imageKey: key,
            userId,
            signedUrl,
            labels,
            caption,
            uploadedAt: Date.now(),
          },
        })
      );

      console.log("SUCCESS: Metadata saved for", key);
      return { statusCode: 200, body: "OK" };

    } catch (err) {
      console.error("UPLOAD ERROR:", err);
      return { statusCode: 500, body: JSON.stringify({ error:
err.message }) };
    }
  }

  // ============ SEARCH API (FINAL) ============
  if (event.httpMethod === "GET" && event.path === "/search") {
    try {
      const query = (event.queryStringParameters?.query ||
"").toLowerCase().trim();
      const userId = event.queryStringParameters?.userId;

      if (!userId) {
        return { statusCode: 400, body: JSON.stringify({ error:
"Missing userId" }) };
      }

      // Scan all user items
      const scan = await dynamoClient.send(
        new ScanCommand({
          TableName: TABLE_NAME,
          FilterExpression: "userId = :uid",
          ExpressionAttributeValues: { ":uid": userId },
```

```
        })
      );

      let items = scan.Items || [];

      // Filter by caption or labels
      if (query) {
        items = items.filter(item =>
          (item.caption?.S || item.caption ||
"").toLowerCase().includes(query) ||
          (item.labels?.L || item.labels || []).some(l =>
            (l.S || l || "").toLowerCase().includes(query)
          )
        );
      }

      // REGENERATE FRESH SIGNED URL FOR EACH
      const itemsWithFreshUrl = await Promise.all(
        items.map(async (item) => {
          const key = item.imageKey?.S || item.imageKey;
          let signedUrl = "";
          try {
            signedUrl = await getSignedUrl(
              s3Client,
              new GetObjectCommand({ Bucket: BUCKET_NAME, Key:
key }),
              { expiresIn: 3600 } // 1 hour
            );
          } catch (e) {
            console.error("Failed to sign URL:", key);
          }

          return {
            imageKey: key,
            userId: item.userId?.S || item.userId,
            signedUrl,
            labels: (item.labels?.L || item.labels || []).map(l
=> l.S || l),
            caption: item.caption?.S || item.caption || "",
            uploadedAt: Number(item.uploadedAt?.N ||
Date.now()),
```

```
            };
          })
        );

        // Sort newest first
        itemsWithFreshUrl.sort((a, b) => b.uploadedAt -
a.uploadedAt);

        return {
          statusCode: 200,
          headers: {
            "Content-Type": "application/json",
            "Access-Control-Allow-Origin": "*",
          },
          body: JSON.stringify(itemsWithFreshUrl),
        };
      } catch (err) {
        console.error("SEARCH ERROR:", err);
        return { statusCode: 500, body: JSON.stringify({ error:
err.message }) };
      }
    }

    return { statusCode: 400, body: "Unsupported event" };
};
```

Code Explanation:

- The lambda_handler function is the entry point that processes S3 events
- Helper functions separate concerns: metadata retrieval, Rekognition analysis, and Bedrock caption generation
- Error handling ensures failed images are logged appropriately
- The code uses boto3 SDK for all AWS service interactions
- Environment variables allow configuration without code changes
- Results are structured as JSON for easy DynamoDB storage

Project Screenshots and Visualizations

3. Amazon Rekognition

Purpose and Role

Amazon Rekognition provides deep learning-based computer vision capabilities to analyze images automatically. It extracts meaningful insights such as objects, scenes, faces, text, and activities from uploaded images.

Implementation Details

• API Integration: Lambda functions invoke Rekognition APIs using boto3 client
• Image Input: Images passed as S3 object references (bucket name + key) to avoid data transfer overhead
• Confidence Thresholds: Set minimum confidence levels (e.g., 80%) to filter low-quality detections
• Service Limits: Implemented rate limiting and request throttling to stay within AWS service quotas

Rekognition Features Utilized

• Label Detection: Identifies objects, scenes, and concepts (e.g., "sunset", "beach", "people")
• Face Detection: Detects faces with attributes like age range, emotions, gender estimate
• Text Detection: Extracts text from images using OCR (Optical Character Recognition)
• Image Properties: Analyzes image quality, brightness, sharpness, and dominant colors
• Unsafe Content Detection: Flags images containing inappropriate or explicit content

Data Processing

• Response Processing: Extracted relevant fields from API responses (labels, confidence scores, bounding boxes)
• Data Transformation: Converted Rekognition output to structured format for DynamoDB storage
• Filtering: Applied business logic to filter results based on relevance and confidence thresholds

4. Amazon Bedrock

Purpose and Role

Amazon Bedrock provides access to foundation models (FMs) for generative AI capabilities. In this project, Bedrock is used to generate intelligent, natural language captions for images based on the Rekognition analysis results.

Implementation Details

• Model Selection: Used foundation models like Claude (Anthropic) or Titan (Amazon) for text generation
• API Integration: Invoked Bedrock Runtime API through boto3 with proper authentication
• Prompt Engineering: Crafted structured prompts combining Rekognition labels and context to generate relevant captions
• Input Format: Passed JSON-formatted prompts with detected labels, objects, and scenes as context
• Response Parsing: Extracted generated text from Bedrock API responses and cleaned formatting
• Token Management: Configured max tokens and temperature parameters for optimal caption generation

Caption Generation Process

1. Receive Rekognition analysis results (labels, scenes, objects)
2. Format data into a structured prompt for the foundation model
3. Example prompt: "Generate a descriptive caption for an image containing: [labels]. Create a natural, engaging description."
4. Send request to Bedrock with appropriate model parameters
5. Process response and extract the generated caption text
6. Validate caption quality and length before storing

Cost Optimization

• Implemented caching for common label combinations to reduce API calls
• Set appropriate max_tokens limits to control generation costs
• Batch processing where possible to optimize throughput

5. Amazon DynamoDB

Purpose and Role

DynamoDB serves as the NoSQL database for storing image metadata, Rekognition analysis results, and Bedrock-generated captions. It provides fast, scalable, and low-latency data access for the application.

Implementation Details

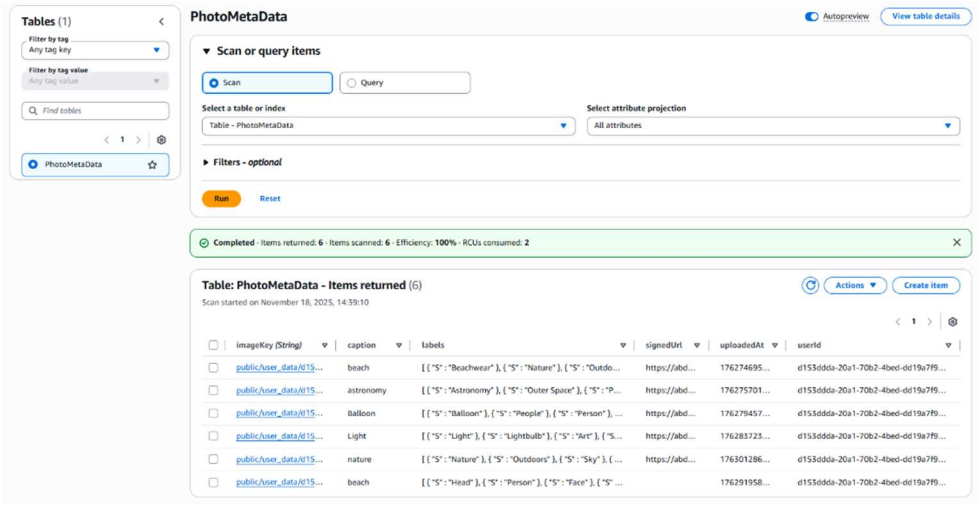
• Table Design: Single table design with composite primary key (Partition Key: ImageID, Sort Key: Timestamp)
• Attributes Stored: Image filename, S3 bucket/key, upload timestamp, file size, Rekognition labels, detected faces, extracted text, Bedrock caption, processing status

• Indexes: Global Secondary Index (GSI) on upload date for time-based queries
• Provisioning: On-demand billing mode for cost efficiency with unpredictable traffic patterns
• Data Format: JSON document structure for flexible schema
• TTL (Time-to-Live): Configured for automatic deletion of old metadata after specified period

Database Operations

• Write Operations: Lambda functions use PutItem to insert processed image metadata
• Read Operations: Query and GetItem operations for retrieving image data
• Update Operations: UpdateItem for modifying processing status or adding new analysis results
• Batch Operations: BatchWriteItem for bulk inserts when processing multiple images
• Conditional Writes: Implemented to prevent data overwrites and ensure idempotency

Performance and Reliability
• Auto-scaling configured for read/write capacity (if using provisioned mode)
• Point-in-time recovery (PITR) enabled for data protection
• DynamoDB Streams enabled for capturing data changes (future use cases like analytics)
• Encryption at rest using AWS KMS



6. Amazon VPC (Virtual Private Cloud)

Purpose and Role

VPC provides a logically isolated network environment for the Cloud Gallery infrastructure. It ensures secure communication between services, controls network access, and provides network-level security through subnets, route tables, and security groups.

Implementation Details

• VPC Architecture: Custom VPC created with CIDR block (e.g., 10.0.0.0/16)
• Subnets: Multiple subnets across availability zones for high availability
  - Public Subnets: For NAT Gateways and bastion hosts (if needed)
  - Private Subnets: Lambda functions deployed here for secure execution
• Internet Gateway: Attached to VPC for internet connectivity
• NAT Gateway: Deployed in public subnets to allow Lambda functions in private subnets to access AWS services and internet

• Route Tables: Configured with routes for public and private subnet traffic

Security Configuration
• Security Groups: Configured to control inbound and outbound traffic
  - Lambda Security Group: Allows outbound HTTPS (443) for AWS API calls
  - Restricts unnecessary inbound traffic (Lambda doesn't accept incoming connections)
• Network ACLs: Additional layer of security at subnet level
• VPC Flow Logs: Enabled for monitoring and troubleshooting network traffic

VPC Endpoints
• Gateway Endpoints: Created for S3 and DynamoDB to enable private connectivity without internet gateway
• Interface Endpoints: Configured for other AWS services (Rekognition, Bedrock) to keep traffic within AWS network
• Benefits: Reduced data transfer costs, improved security, better performance

Network Isolation Benefits
• Lambda functions run in isolated network without direct internet access
• All AWS service communication happens through VPC endpoints or NAT Gateway
• Enhanced security posture with defense-in-depth approach
• Compliance with security best practices for production workloads

Additional AWS Services and Components

IAM (Identity and Access Management)
• Created custom IAM roles for Lambda with least privilege access
• Policies attached for S3, DynamoDB, Rekognition, Bedrock, and CloudWatch access
• Service-linked roles for VPC networking
• Resource-based policies on S3 buckets

CloudWatch

• Logs: Centralized logging for Lambda functions for debugging and auditing
• Metrics: Custom metrics for tracking image processing performance
• Alarms: Configured alarms for Lambda errors, throttling, and DynamoDB capacity
• Dashboards: Created monitoring dashboards for operational visibility

System Data Flow

End-to-End Processing Pipeline

1. User uploads image to S3 bucket through web application or API
2. S3 triggers ObjectCreated event notification
3. Lambda function is invoked with event details (bucket name, object key)
4. Lambda downloads image from S3 to /tmp directory
5. Lambda calls Rekognition APIs in parallel:

- DetectLabels for object/scene detection
- DetectFaces for facial analysis
- DetectText for OCR

6. Lambda processes Rekognition results and formats data
7. Lambda invokes Bedrock with formatted prompt containing Rekognition insights
8. Bedrock generates natural language caption
9. Lambda aggregates all results (metadata, labels, faces, text, caption)
10. Lambda writes complete record to DynamoDB table
11. Processing complete - data available for retrieval through queries
12. CloudWatch logs capture entire process for monitoring

Key Technical Features

Serverless Architecture

• No server management required - fully managed services
• Automatic scaling based on demand
• Pay-per-use pricing model reduces costs
• High availability built-in across multiple availability zones

Event-Driven Processing

• Real-time processing triggered by S3 events
• Asynchronous workflow enables parallel processing
• Decoupled architecture for better maintainability
• Event notifications ensure no image is missed

AI/ML Integration

• Computer vision powered by Rekognition for image understanding
• Generative AI through Bedrock for intelligent captions
• Pre-trained models eliminate training requirements
• Continuous improvements as AWS updates models

Security and Compliance

Security Best Practices Implemented

• Encryption at Rest: All data encrypted using AWS KMS (S3, DynamoDB)
• Encryption in Transit: TLS/SSL for all API communications
• Least Privilege Access: IAM policies follow principle of least privilege
• Network Isolation: VPC provides isolated network environment
• Private Connectivity: VPC endpoints eliminate internet exposure
• Logging and Monitoring: Comprehensive logging through CloudWatch
• Data Protection: S3 versioning and DynamoDB point-in-time recovery

Conclusion

The Cloud Gallery project demonstrates a robust, scalable, and secure implementation of modern cloud architecture using AWS services. By leveraging serverless technologies (Lambda, S3, DynamoDB), AI/ML services (Rekognition, Bedrock), and secure networking (VPC), the system provides automated image processing and intelligent analysis capabilities.

The architecture benefits from:

- Fully managed services reducing operational overhead
- Event-driven design enabling real-time processing
- AI-powered insights without custom model training
- Enterprise-grade security with encryption and network isolation
- Cost-effective pay-per-use pricing model
- High availability and automatic scaling

This implementation provides a solid foundation for future enhancements such as advanced search capabilities using DynamoDB Streams and OpenSearch, front-end web application integration, batch processing workflows, and additional AI features like image similarity search or content moderation.

The project successfully demonstrates cloud-native development best practices, combining multiple AWS services into a cohesive, production-ready solution for intelligent image management.