Measurement Classification System

High Level Architecture Document

Aharon Garada

21/08/2024

# General Overview

This High-Level Architecture Document describe a system that processes measurement data from a device associated with a labelled sample, The system should integrate with external services for quality control and classification, handle user feedback, and store relevant data outcomes.

## System Purpose:

* Ingest and processes measurement unlabelled data from external device.
* Integrates with external services for applying quality control and classification.
* Stores relevant data outcomes
* Provide user interface for displaying and modifying tge data

## Architecture:

* Serverless architecture on AWS.
* Components: Backend (Django), Data processing (Django in Lambda), Database (DynamoDB), Frontend (React).

## Data Flow:

* Scheduled extraction of unlabelled data from an external system.
* Apply quality control and classification on data using Lambda functions and external services.
* Data storage in DynamoDB.
* User interface for viewing, editing, and commenting on labelled data.

# Architecture

A diagram of software development

Description automatically generated

The application will be implemented on these main components for a modular and scalable architecture:

**Backend** – (Django): Exposes the application programming interfaces (APIs) for fetching and managing labelled data.

**Data process** (Django in Lambda): Fetches measurement data from a external data, processes it by applying a quality control and classification, and stores it in the AWS DynamoDB database.

**Database** (DynamoDB): Stores the classified data.

**Frontend** (React): Provides a user interface with options to display and edit classified data and add comments accordingly, using RESTful API for interaction with backend.

# Data Flow and Process Stages

## Data Extraction:

* + **Scheduler:** A scheduled task (e.g., CloudWatch Events rule) invokes a Lambda function.
  + **Lambda Function:** Extracts unlabeled measurement data from the external system and stores it in an S3 bucket.

## Quality Control:

* + **S3 Event:** An S3 event triggers a Lambda function.
  + **Lambda Function:** Sends the data to an external quality control service.
  + **External Quality Control Service:** Processes the data and returns a cleaned result.
  + **Lambda Function:** Saves the cleaned data to a DynamoDB table.

## Classification:

* + **DynamoDB Stream:** A DynamoDB stream triggers a Lambda function.
  + **Lambda Function:** Sends the cleaned data to an external machine learning service.
  + **External Machine Learning Service:** Classifies the data and returns a labelled result.
  + **Lambda Function:** Updates the DynamoDB table with the labelled data.

## User Interface:

* + **Frontend Application:** A web application allows users to view labelled data, change labels, and add comments.
  + **API Gateway:** Provides an API for the frontend application to interact with the backend

# Security Aspects

## Security on cloud level:

* **IAM Roles:** We assign IAM roles to Lambda functions and API Gateway with appropriate permissions to access S3, DynamoDB, and other services.
* **VPC:** We create a VPC with private subnets to isolate your resources from the public internet.
* **Security Groups and FW:** We use security groups to control inbound and outbound traffic to your resources.
* **Encryption:** We may choose to encrypt data at rest in S3 and DynamoDB.
* **Secret Manager:** Probable we will have sensitive information, in this case we will store sensitive information (e.g., API keys, passwords) securely in AWS Secrets Manager.

## Security on application level:

* We should implement Access Control Layer
* The system should serve over HTTPS protocol
* WE should consider Implement 2FA
* We will develop the application by implementing Secure Software Development Life Cycle methodology

# Component Descriptions

* **S3:** Object storage for storing measurement data.
* **DynamoDB:** Database for storing processed data and classification results.
* **Lambda:** Serverless compute service for executing functions triggered by events.
* **SQS:** Message queue service for asynchronous communication between components (\*).

An SQS queue can be used between the Lambda functions that extract data and perform quality control/classification to decouple components and improve scalability. This allows for asynchronous processing and better handling of varying workloads.

* **IAM:** Identity and Access Management service for managing user permissions.
* **VPC:** Virtual Private Cloud for isolating your resources from the public internet.
* **API Gateway:** RESTful API gateway for exposing your application's functionality.

**Additional Components:**

* **CloudWatch Events:** A scheduler for triggering Lambda functions based on events.

# Advantages of Using AWS Components

* Serverless architecture for scalability and cost-efficiency.
* Managed services for reduced operational overhead.
* Built-in security features.

# Handling Failures

To mitigate failure risks and ensure system resilience, our architecture incorporates several strategies:

**1. Retries:**

* **Lambda Function Configuration:** Configure Lambda functions to automatically retry failed invocations with exponential backoff. This helps to avoid overwhelming the external service during temporary outages.
* **Custom Retry Logic:** Implement custom retry logic within Lambda functions to handle specific error conditions or retry limits.

**2. Error Handling and Logging:**

* **Comprehensive Logging:** Implement robust logging to capture errors and track the health of external services. This helps in identifying root causes and troubleshooting issues.
* **Error Handling Mechanisms:** Handle errors gracefully within Lambda functions and Backend, providing informative messages to users or triggering alerts.

**3. Asynchronous Processing:**

* **SQS Queues:** As we mentioned before, we can use SQS queues to decouple components and allow for asynchronous processing. If an external service fails, messages can be retried later without affecting the overall system's performance.

**4. Monitoring and Alerts:**

* **CloudWatch Alarms:** Set up CloudWatch alarms to monitor the health of external and internal services and trigger alerts when performance metrics degrade.

# Microservices vs. Monolith

While this architecture leans towards a microservices approach due to the separation of concerns and scalability, a monolith could also be considered for smaller projects or when tight coupling is acceptable. However, microservices generally offer better maintainability, flexibility, and fault isolation.

**Generally, we have here a microservices architecture.**

Here's a breakdown of why:

* **Decentralized Services:** Each component (e.g., data extraction, quality control, classification, and all relevant Lambda functions) is a separate, independent service.
* **Loose Coupling:** Services communicate through APIs or message queues, reducing dependencies and improving maintainability.
* **Scalability:** Individual services can be scaled independently based on demand.
* **Fault Isolation:** Failures in one service do not necessarily affect others.

While some components might be tightly coupled within themselves (e.g., the Lambda functions handling data processing), the overall architecture adheres to the principles of microservices due to the separation of concerns and independent deployment.