NYCSBUS Real-Time Bus Delay and Breakdown Alerting Pipeline

The goal is to create a robust, automated, and observable ETLA (Extract, Transform, Load, Alert) pipeline using AWS serverless technologies.

Dataset Reference: [NYC Open Data: Bus Breakdown and Delays & attached Data Dictionary]()

Simulated Incoming Data Payload:

Create an API endpoint that receives a JSON payload that follows the official data schema:

```
{
    "School_Year": "2024-2025",
    "Busbreakdown_ID": 781099,
    "Run_Type": "Special Ed AM Run",
    "Bus_No": "5518",
    "Route_Number": "X231",
    "Reason": "Mechanical Problem",
    "Occurred_On": "2025-10-22T07:55:00Z",
    "Boro": "Bronx",
    "Bus_Company_Name": "PIONEER TRANSPORTATION",
    "How_Long_Delayed": "25-35 Mins",
    "Number_Of_Students_On_The_Bus": 18,
    "Has_Contractor_Notified_Schools": "Yes",
    "Has_Contractor_Notified_Parents": "Yes",
    "Have_You_Alerted_OPT": "Yes"
}
```

Deliverables

Please prepare to present working code.

1. Architecture and Infrastructure as Code (IaC)

Use Terraform to provision the necessary AWS infrastructure. Your architecture must be serverless and event-driven.

1. API Gateway: An HTTP endpoint to receive the incoming bus event payloads.
2. AWS Lambda (Data Ingestion & Transformation): A Lambda function written in Python that is triggered by the API Gateway.
3. Amazon S3: Store the raw, unmodified JSON payloads for auditing and reprocessing.
4. Amazon DynamoDB: Store the transformed and enriched data. The table should be designed for efficient querying (e.g., using Route_Number as a partition key and Occurred_On as a sort key).
5. IAM Roles: Create appropriate IAM roles with the least-privilege principle for all services.

Deliverable #1: All Terraform code (.tf files) required to deploy the entire infrastructure. The code should be well-structured and commented.

2. Data Processing and Logic

Implement the core transformation logic within the AWS Lambda function.
The function must process the incoming JSON and perform the following actions:

1. Validate the incoming payload to ensure it contains required fields like Busbreakdown_ID, Route_Number, and Reason.
2. Create a new field named alert_priority. Set its value based on the official Reason categories:
    - "high": for "Mechanical Problem", "Flat Tire", "Won't Start", or "Accident".
    - "medium": for "Heavy Traffic" or "Weather Conditions".
    - "low": for "Delayed by School", "Other", or "Problem Run".
3. create a new integer field average_delay_minutes by parsing the How_Long_Delayed string . The function should handle formats like "30 Min", "25-35 Mins", or "1 Hour".

Deliverable #2: The complete, well-commented Python code for the Lambda function. Include any dependencies in a requirements.txt file.

3. Monitoring and Alerting

Robust monitoring and alerting are critical for operational awareness.

- CloudWatch Metrics:
    - Create a custom CloudWatch metric named HighPriorityAlerts that

    increments by 1 every time an event with an "high" alert_priority is processed.

- CloudWatch Alarms & SNS:
    - Configure a CloudWatch Alarm that enters the ALARM state if the HighPriorityAlerts metric is greater than or equal to 3 within a 5-minute period.
    - This alarm should publish a message to an SNS topic.
    - Configure a second alarm for any Lambda invocation errors (Errors metric > 0).

- Logging: Ensure the Lambda function produces structured JSON logs that include the Busbreakdown_ID and Route_Number for easier debugging and filtering in CloudWatch Logs.

Deliverable #3: Terraform code for CloudWatch metrics, Alarms and SNS

4. Authentication and Authorization with Amazon Cognito

Secure the API Gateway endpoint to ensure that only authorized clients can submit data.

- Cognito User Pool: Provision an Amazon Cognito User Pool to manage identities.
- Cognito Authorizer: Configure the API Gateway's POST method to use a Cognito User Pool Authorizer. This authorizer must validate the JSON Web Token (JWT) provided in the Authorization header of the incoming request. Access should be granted only to successfully authenticated users.

Deliverable #4:

- Terraform Code: The .tf files containing the resource definitions for the Cognito User Pool and the API Gateway Authorizer configuration.
- Client Authentication Example: Provide a cURL command or a short client script (e.g., Python) showing how to authenticate a test user, retrieve an identity token, and use that token to make an authorized POST request to the protected API endpoint.

5. Documentation and CI/CD Strategy

Clear documentation and a well-defined deployment strategy are essential for team collaboration and system reliability.

Deliverable #5:
A README.md file that includes:
1. Architecture Diagram: A simple diagram or textual description of the pipeline's architecture and data flow.
2. Setup Instructions: Clear, step-by-step instructions on how to deploy your Terraform stack.
3. Design Decisions: Briefly explain your choice of services (e.g., why DynamoDB for this use case?) and any assumptions you made.
4. CI/CD Plan: Based on your experience with GitHub Actions or Jenkins, describe how you would build a CI/CD pipeline to automatically build, test, and deploy this serverless application. What stages would your pipeline have (e.g., linting, unit tests, IaC plan, deployment)? How would you manage different environments (dev, prod)?

Evaluation Criteria

Submission based on the following:
1. Correctness & Functionality: Does the pipeline work as specified?
2. Architectural Design: Is the architecture truly serverless, scalable, and cost-effective?
3. IaC Quality: Is the Terraform code clean, modular, and reusable?
4. Code Quality: Is the Python code clean, efficient, and well-documented?
5. Monitoring & Observability: Is the monitoring and alerting setup comprehensive and meaningful?
6. Documentation & Communication: Is the documentation clear, and are the design choices well-reasoned?
7. DevOps Best Practices: Does the overall solution show a strong understanding of automation, reliability, and security?