aws

Developer Guide

AWS IoT Events

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS IoT Events: Developer Guide

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

..... viii
What is AWS IoT Events? 1
Benefits and features 1
Use cases 2
Monitor and maintain remote devices 2
Manage industrial robots 3
Track building automation systems 3
AWS IoT Events end of support 4
Considerations when migrating away from AWS IoT Events 4
Detector models 5
Comparing architectures 5
Step 1: (Optional) export AWS IoT Events detector model configurations 6
Step 2: Create an IAM role 7
Step 3: Create Amazon Kinesis Data Streams 10
Step 4: Create or update the MQTT message routing rule 10
Step 5: Get the endpoint for the destination MQTT topic 12
Step 6: Create an Amazon DynamoDB table 12
Step 7: Create an AWS Lambda function (console) 13
Step 8: Add an Amazon Kinesis Data Streams trigger 21
Step 9: Test data ingestion and output functionality (AWS CLI) 22
Alarms 23
Comparing architectures 23
Step 1: Enable MQTT notifications on the asset property 24
Step 2: Create an AWS Lambda function 24
Step 3: Create AWS IoT Core message routing rule 26
Step 4: View CloudWatch metrics 27
Step 5: Create CloudWatch alarms 27
Step 6: (Optional) import the CloudWatch alarm into AWS IoT SiteWise 28
Setting up 29
Setting up an AWS account 29
Sign up for an AWS account 29
Setting up permissions for AWS IoT Events 29
Action permissions 30
Securing input data 32

Amazon CloudWatch logging role policy ... 33
Amazon SNS messaging role policy ... 35
Getting started ... 37
Prerequisites ... 39
Create an input ... 39
Create a JSON input file ... 40
Create and configure an input ... 40
Create an input within the Detector Model ... 41
Create a detector model ... 41
Test the detector model ... 48
Best practices ... 52
Enable Amazon CloudWatch logging when developing AWS IoT Events detector models ... 52
Publish regularly to save your detector model when working in the AWS IoT Events console ... 53
Tutorials ... 54
Using AWS IoT Events to monitor your IoT devices ... 54
How do you know which states you need in a detector model? ... 56
How do you know if you need one instance of a detector or several? ... 57
Simple step-by-step example ... 58
Create an input to capture device data ... 60
Create a detector model to represent device states ... 61
Send messages as inputs to a detector ... 65
Detector model restrictions and limitations ... 68
A commented example: HVAC temperature control ... 71
Input definitions for detector models ... 72
Create a detector model definition ... 76
Use BatchUpdateDetector ... 96
Use BatchPutMessage for inputs ... 98
Ingest MQTT messages ... 100
Generate Amazon SNS messages ... 102
Configure the DescribeDetector API ... 103
Use the AWS IoT Core rules engine ... 105
Supported actions ... 109
Use built-in actions ... 110
Set timer action ... 110
Reset timer action ... 110
Clear timer action ... 111

Set variable action ... 111
Work with other AWS services ... 112
AWS IoT Core ... 113
AWS IoT Events ... 114
AWS IoT SiteWise ... 115
Amazon DynamoDB ... 117
Amazon DynamoDB(v2) ... 120
Amazon Data Firehose ... 121
AWS Lambda ... 122
Amazon Simple Notification Service ... 123
Amazon Simple Queue Service ... 124
Expressions ... 126
Syntax to filter device data ... 126
Literals ... 126
Operators ... 126
Functions for expressions ... 128
Reference for inputs and variables in expressions ... 132
Substitution templates ... 135
Usage ... 136
Writing AWS IoT Events expressions ... 136
Detector model examples ... 138
HVAC temperature control ... 138
Background story ... 138
Input definitions ... 139
Detector model definition ... 141
BatchPutMessage examples ... 159
BatchUpdateDetector example ... 165
AWS IoT Core rules engine ... 167
Cranes ... 170
Send commands ... 170
Detector models ... 172
Inputs ... 179
Messages ... 179
Example: Event detection with sensors ... 181
Device HeartBeat ... 183
ISA alarm ... 185

Simple alarm ... 195
Monitoring with alarms ... 200
Working with AWS IoT SiteWise ... 200
Acknowledge flow ... 200
Creating an alarm model ... 201
Requirements ... 201
Creating an alarm model (console) ... 202
Responding to alarms ... 205
Managing alarm notifications ... 206
Creating a Lambda function ... 207
Using the Lambda function ... 216
Manage alarm recipients ... 217
Security ... 218
Identity and access management ... 218
Audience ... 219
Authenticating with identities ... 219
Managing access using policies ... 220
More about identity and access management ... 222
How AWS IoT Events works with IAM ... 222
Identity-based policy examples ... 226
Cross-service confused deputy prevention for AWS IoT Events ... 232
Troubleshooting ... 236
Monitoring ... 238
Available tools to monitor AWS IoT Events ... 239
Monitoring AWS IoT Events with Amazon CloudWatch ... 240
Logging AWS IoT Events API calls with AWS CloudTrail ... 242
Compliance validation ... 261
Resilience ... 262
Infrastructure security ... 262
Quotas ... 263
Tagging ... 264
Tag basics ... 264
Tag restrictions and limitations ... 265
Using tags with IAM policies ... 265
Troubleshooting ... 269
Common AWS IoT Events issues and solutions ... 269

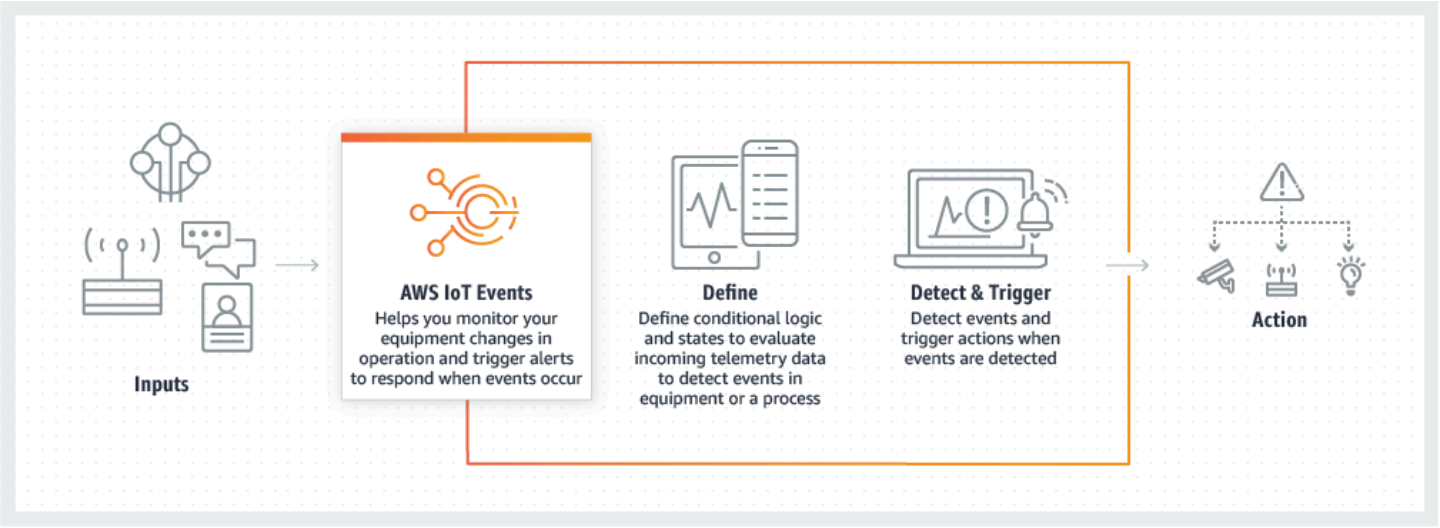
Detector model creation errors 270
Updates from a deleted detector model 270
Action trigger failure (when meeting a condition) 270
Action trigger failure (when breeching a threshold) 271
Incorrect state usage 271
Connection message 271
InvalidRequestException message 271
Amazon CloudWatch Logs `action.setTimer` errors 272
Amazon CloudWatch payload errors 273
Incompatible data types 274
Failed to send message to AWS IoT Events 275
Troubleshooting a detector model 275
Diagnostic information 276
Analyze a detector model (Console) 289
Analyze a detector model (AWS CLI) 291
Commands 296
AWS IoT Events actions 296
AWS IoT Events data 296
Document history 297
Earlier updates 298

End of support notice: On May 20, 2026, AWS will end support for AWS IoT Events. After May 20, 2026, you will no longer be able to access the AWS IoT Events console or AWS IoT Events resources. For more information, see AWS IoT Events end of support.

What is AWS IoT Events?

AWS IoT Events enables you to monitor your equipment or device fleets for failures or changes in operation, and to trigger actions when such events occur. AWS IoT Events continuously watches IoT sensor data from devices, processes, applications, and other AWS services to identify significant events so you can take action.

Use AWS IoT Events to build complex event monitoring applications in the AWS Cloud that you can access through the AWS IoT Events console or APIs.



Topics

- Benefits and features
- Use cases

Benefits and features

Accept inputs from multiple sources

AWS IoT Events accepts inputs from many IoT telemetry data sources. These include sensor devices, management applications, and other AWS IoT services, such as AWS IoT Core and AWS IoT Analytics. You can push any telemetry data input to AWS IoT Events by using a standard API interface (`BatchPutMessage` API) or the AWS IoT Events console.

For more information on getting started with AWS IoT Events, see Getting started with the AWS IoT Events console.

Use simple logical expressions to recognize complex patterns of events

AWS IoT Events can recognize patterns of events that involve multiple inputs from a single IoT device or application, or from diverse equipment and many independent sensors. This is especially useful because each sensor and application provides important information. But only by combining diverse sensor and application data can you get a complete picture of the performance and quality of operations. You can configure AWS IoT Events detectors to recognize these events using simple logical expressions instead of complex code.

For more information on logical expressions, see Expressions to filter, transform, and process event data.

Trigger actions based on events

AWS IoT Events enables you to directly trigger actions in Amazon Simple Notification Service (Amazon SNS), AWS IoT Core, Lambda, Amazon SQS and Amazon Kinesis Firehose. You can also trigger an AWS Lambda function using the AWS IoT rules engine which makes it possible to take actions using other services, such as Connect Customer, or your own enterprise resource planning (ERP) applications.

AWS IoT Events includes a prebuilt library of actions you can take, and also enables you to define your own.

To learn more about triggering actions based on events, see Supported actions to receive data and trigger actions in AWS IoT Events.

Automatically scale to meet the demands of your fleet

AWS IoT Events scales automatically when you are connecting homogeneous devices. You can define a detector once for a specific type of device, and the service will automatically scale and manage all instances of that device that connect to AWS IoT Events.

To explore examples of detector models, see AWS IoT Events detector model examples.

Use cases

AWS IoT Events has many uses. Here are a few example use cases.

Monitor and maintain remote devices

Monitoring a fleet of remotely deployed machines can be challenging, especially when a malfunction occurs without clear context. If one machine stops functioning, this might mean

replacing the entire processing unit or machine. But this isn't sustainable. With AWS IoT Events you can receive messages from multiple sensors on each machine to help you diagnose specific issues over time. Instead of replacing the whole unit, you now have the necessary information to send a technician with the exact part that needs replacement. With millions of machines, savings can add up to millions of dollars, lowering your total cost of owning or maintaining each machine.

Manage industrial robots

Deploying robots in your facilities to automate package movement can greatly enhance efficiency. To minimize costs, robots can be equipped with simple, low-cost sensors that report data to the cloud. However, with dozens of sensors and hundreds of operating modes, detecting issues in real time can be challenging. Using AWS IoT Events, you can build an expert system that processes this sensor data in the cloud, creating alerts to automatically notify technical staff if a failure is imminent.

Track building automation systems

In data centers, monitoring for high temperatures and low humidity helps to prevent equipment failures. Sensors are often purchased from many manufacturers and each type comes with its own management software. However, management software from different vendors sometimes isn't compatible, making it difficult to detect problems. Using AWS IoT Events, you can set up alerts to notify your operations analysts of issues with your heating and cooling systems well in advance of failures. In this way, you can prevent an unscheduled data center shutdown that would cost thousands of dollars in equipment replacement and potential lost revenue.

AWS IoT Events end of support

After careful consideration, we decided to end support for the AWS IoT Events service, effective May 20, 2026. AWS IoT Events will no longer accept new customers beginning May 20, 2025. As an existing customer with an account signed up for the service before May 20, 2025, you can continue to use AWS IoT Events features. After May 20, 2026, you will no longer be able to use AWS IoT Events.

This page provides instructions and considerations for AWS IoT Events customers to transition to an alternate solution to meet your business needs.

> **Note**
>
> The solutions presented in these guides are meant to serve as an illustrative examples, not as a production-ready replacements for AWS IoT Events functionality. Customize the code, workflow, and related AWS resources to your business needs.

Topics

- Considerations when migrating away from AWS IoT Events
- Migration procedure for detector models in AWS IoT Events
- Migration procedure for AWS IoT SiteWise alarms in AWS IoT Events

Considerations when migrating away from AWS IoT Events

- Implement security best practices, including using IAM roles with least privilege for each component and encrypting data at rest and in transit. For more information, see Security best practices in IAM in the *IAM User Guide*.
- Consider the number of shards for the Kinesis stream based on your data ingestion requirements. For more information on Kinesis shards, see Amazon Kinesis Data Streams terminology and concepts in the *Amazon Kinesis Data Streams Developer Guide*.
- Set up comprehensive monitoring and debugging using CloudWatch for metrics and logs. For more information, see What is CloudWatch? in the *Amazon CloudWatch User Guide*.

- Consider the structure of your error handling, including how to manage messages that fail processing repeatedly, implementing retry policies, and setting up a process to isolate and analyze problematic messages.
- Use the AWS Pricing Calculator to estimate costs for your specific use case.

Migration procedure for detector models in AWS IoT Events

This section describes alternative solutions that deliver similar detector model functionality as you migrate away from AWS IoT Events.

You can migrate data ingestion through AWS IoT Core rules to a combination of other AWS services. Instead of data ingestion through the BatchPutMessage API, the data can be routed to the AWS IoT Core MQTT topic.

This migration approach leverages AWS IoT Core MQTT topics as the entry point for your IoT data, replacing the direct input to AWS IoT Events. MQTT topics are chosen for several key reasons. They offer broad compatibility with IoT devices due to MQTT's widespread use in the industry. These topics can handle high volumes of messages from numerous devices, ensuring scalability. They also provide flexibility in routing and filtering messages based on content or device type. Additionally, AWS IoT Core MQTT topics integrate seamlessly with other AWS services, facilitating the migration process.

Data flows from MQTT topics into an architecture combining Amazon Kinesis Data Streams, a AWS Lambda function, a Amazon DynamoDB table, and Amazon EventBridge schedules. This combination of services replicates and enhances the functionality previously provided by AWS IoT Events, offering you more flexibility and control over your IoT data processing pipeline.

Comparing architectures

The current AWS IoT Events architecture ingests data through an AWS IoT Core rule and the `BatchPutMessage` API. This architecture uses AWS IoT Core for data ingestion and event publishing, with messages routed through AWS IoT Events inputs to detector models that define the state logic. An IAM role manages the necessary permissions.

The new solution maintains AWS IoT Core for data ingestion (now with dedicated input and output MQTT topics). It introduces Kinesis Data Streams for data partitioning and an evaluator Lambda function for state logic. Device states are now stored in a DynamoDB table, and an enhanced IAM role manages permissions across these services.

Purpose	Solution	Differences
Data ingestion – Receives data from IoT devices	AWS IoT Core	Now requires two distinct MQTT topics: one for ingesting device data and another for publishing output events
Message direction – Routes incoming messages to appropriate services	AWS IoT Core message routing rule	Maintains same routing functionality but now directs messages to Kinesis Data Streams instead of AWS IoT Events
Data processing – Handles and organizes incoming data streams	Kinesis Data Streams	Replaces AWS IoT Events input functionality, providing data ingestion with device ID partitioning for message processing
Logic evaluation – Processes state changes and triggers actions	Evaluator Lambda	Replaces AWS IoT Events detector model, providing customizable state logic evaluation through code instead of visual workflow
State management – Maintains device states	DynamoDB table	New component that provides persistent storage of device states, replacing internal AWS IoT Events state management
Security – Manages service permissions	IAM role	Updated permissions now include access to Kinesis Data Streams, DynamoDB, and EventBridge in addition to existing AWS IoT Core permissions

Step 1: (Optional) export AWS IoT Events detector model configurations

Before creating new resources, export your AWS IoT Events detector model definitions. These contain your event processing logic and can serve as a historical reference for implementing your new solution.

Console

Using the AWS IoT Events AWS Management Console, perform the following steps to export your detector model configurations:

To export detector models using the AWS Management Console

1. Log into the AWS IoT Events console .
2. In the left navigation pane, choose **Detector models**.
3. Select the detector model to export.
4. Choose **Export**. Read the information message regarding the output and then choose **Export** again.
5. Repeat the process for each detector model that you want to export.

A file containing a JSON output of your detector model is added to your browser's download folder. You can optionally save each detector model configuration to preserve historical data.

AWS CLI

Using the AWS CLI, run the following commands to export your detector model configurations:

To export detector models using AWS CLI

1. List all detector models in your account:

```
aws iotevents list-detector-models
```

2. For each detector model, export its configuration by running:

```
aws iotevents describe-detector-model \
   --detector-model-name your-detector-model-name
```

3. Save the output for each detector model.

Step 2: Create an IAM role

Create an IAM role to provide permissions to replicate the functionality of AWS IoT Events. The role in this example grants access to DynamoDB for state management, EventBridge for scheduling,

Kinesis Data Streams for data ingestion, AWS IoT Core for publishing messages, and CloudWatch for logging. Together, these services to work as a replacement for AWS IoT Events.

1. Create an IAM role with the following permissions. For more detailed instructions on creating an IAM role, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAccess",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/
EventsStateTable"
        },
        {
            "Sid": "SchedulerAccess",
            "Effect": "Allow",
            "Action": [
                "scheduler:CreateSchedule",
                "scheduler:DeleteSchedule"
            ],
            "Resource": "arn:aws:scheduler:us-east-1:123456789012:schedule/*"
        },
        {
            "Sid": "KinesisAccess",
            "Effect": "Allow",
            "Action": [
                "kinesis:GetRecords",
                "kinesis:GetShardIterator",
                "kinesis:DescribeStream",
                "kinesis:ListStreams"
```

```
            ],
            "Resource": "arn:aws:kinesis:us-east-1:123456789012:stream/*"
        },
        {
            "Sid": "IoTPublishAccess",
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": "arn:aws:iot:us-east-1:123456789012:topic/*"
        },
        {
            "Effect": "Allow",
            "Action": "logs:CreateLogGroup",
            "Resource": "arn:aws:logs:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
            "arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/your-
lambda:*"
            ]
        }
    ]
}
```

2. Add the following IAM role trust policy. A trust policy allows the specified AWS services to assume the IAM role so that they can to perform necessary actions. For more detailed instructions on creating an IAM trust policy, see Create a role using custom trust policies in the *IAM User Guide*.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
```

```
                    "scheduler.amazonaws.com",
                    "lambda.amazonaws.com",
                    "iot.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Step 3: Create Amazon Kinesis Data Streams

Create Amazon Kinesis Data Streams using the AWS Management Console or AWS CLI.

Console

To create a Kinesis data stream using the AWS Management Console, follow the procedure found on the Create a data stream page in the *Amazon Kinesis Data Streams Developer Guide*.

Adjust the shard count based on your device count and message payload size.

AWS CLI

Using AWS CLI, create Amazon Kinesis Data Streams to ingest and partition the data from your devices.

Kinesis Data Streams are used in this migration to replace the data ingestion functionality of AWS IoT Events. It provides a scalable and efficient way to collect, process, and analyze real-time streaming data from your IoT devices, while providing flexible data handling and integration with other AWS services.

```
aws kinesis create-stream --stream-name your-kinesis-stream-name --shard-count 4 --
region your-region
```

Adjust the shard count based on your device count and message payload size.

Step 4: Create or update the MQTT message routing rule

You can create a new MQTT message routing rule or update an existing rule.

Console

1. Determine if you need a new MQTT message routing rule or if you can update an existing rule.
2. Open the AWS IoT Core console.
3. In the navigation pane, choose **Message Routing**, and then choose **Rules**.
4. In the **Manage** section, choose **Message routing**, and then **Rules**.
5. Choose **Create rule**.
6. On the **Specify rule properties** page, enter the AWS IoT Core rule name for **Rule name**. For **Rule Description - *optional*, enter a description to identify that you're processing events and forwarding them to Kinesis Data Streams.**
7. On the **Configure SQL statement** page, enter the following for the **SQL statement**: **`SELECT * FROM 'your-database'`**, then choose **Next**.
8. On the **Attach rules actions** page, and under **Rule actions**, choose **kinesis**.
9. Choose your Kinesis stream for the stream. For the partition key, enter **`your-instance-id`**. Select the appropriate role for the IAM role, and then choose **Add rule action**.

For more information, see Creating AWS IoT rules to route device data to other services.

AWS CLI

1. Create a JSON file with the following contents. This JSON configuration file defines an AWS IoT Core rule that selects all messages from a topic and forwards them to the specified Kinesis stream, using the instance ID as the partition key.

```
{
    "sql": "SELECT * FROM 'your-config-file'",
    "description": "Rule to process events and forward to Kinesis Data Streams",
    "actions": [
        {
            "kinesis": {
                "streamName": "your-kinesis-stream-name",
                "roleArn": "arn:aws:iam::your-account-id:role/service-role/your-
iam-role",
                "partitionKey": "${your-instance-id}"
            }
        }
    ],
```

```
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23"
}
```

2. Create the MQTT topic rule using the AWS CLI. This step uses the AWS CLI to create an AWS IoT Core topic rule using the configuration defined in the `events_rule.json` file.

```
aws iot create-topic-rule \
    --rule-name "your-iot-core-rule" \
    --topic-rule-payload file://your-file-name.json
```

Step 5: Get the endpoint for the destination MQTT topic

Use the destination MQTT topic to configure where your topics publish outgoing messages, replacing the functionality previously handled by AWS IoT Events. The endpoint is unique to your AWS account and region.

Console

1. Open the AWS IoT Core console.
2. In the **Connect** section on the left navigation panel, choose **Domain configuration**.
3. Choose the **iot:Data-ATS** domain configuration to open the configuration's detail page.
4. Copy the **Domain name** value. This value is the endpoint. Save the endpoint value because you'll need it in later steps.

AWS CLI

Run the following command to get the AWS IoT Core endpoint for publishing outgoing messages for your account.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS --region your-region
```

Step 6: Create an Amazon DynamoDB table

A Amazon DynamoDB table replaces the state management functionality of AWS IoT Events, providing a scalable and flexible way to persist and manage the state of your devices and the detector model logic in your new solution architecture.

Console

Create a Amazon DynamoDB table to persist the state of the detector models. For more information, see Create a table in DynamoDB in the *Amazon DynamoDB Developer Guide*.

Use the following for the table details:

- For **Table name**, enter a table name of your choosing.
- For **Partition key**, enter your own instance ID.
- You can use the **Default settings** for the **Table settings**

AWS CLI

Run the following command to create a DynamoDB table.

```
aws dynamodb create-table \
                        --table-name your-table-name \
                        --attribute-definitions AttributeName=your-instance-
id,AttributeType=S \
                        --key-schema AttributeName=your-instance-id,KeyType=HASH \
```

Step 7: Create an AWS Lambda function (console)

The Lambda function serves as the core processing engine, replacing the detector model evaluation logic of AWS IoT Events. In the example, we integrate with other AWS services to handle incoming data, manage state, and trigger actions based on your defined rules.

Create a Lambda function with NodeJS runtime. Use the following code snippet, replacing the hard-coded constants:

1. Open the AWS Lambda console.
2. Choose **Create function**.
3. Enter a name for the **Function name**.
4. Select **NodeJS 22.x** as the **Runtime**.
5. In the **Change default execution role** dropdown, choose **Use existing role**, and then select the IAM role that you created in earlier steps.
6. Choose **Create function**.

7. Paste in the following code snippet after replacing the hard coded constants.
8. After your function creates, under the **Code** tab, paste the following code example, replacing the **`your-destination-endpoint`** endpoint with your own.

```
import { DynamoDBClient, GetItemCommand } from '@aws-sdk/client-dynamodb';
import { PutItemCommand } from '@aws-sdk/client-dynamodb';
import { IoTDataPlaneClient, PublishCommand } from "@aws-sdk/client-iot-data-plane";
import { SchedulerClient, CreateScheduleCommand, DeleteScheduleCommand } from "@aws-
sdk/client-scheduler"; // ES Modules import


//// External Clients and Constants
const scheduler = new SchedulerClient({});
const iot = new IoTDataPlaneClient({
    endpoint: 'https://your-destination-endpoint-ats.iot.your-region.amazonaws.com/'
});
const ddb = new DynamoDBClient({});


//// Lambda Handler function
export const handler = async (event) => {
    console.log('Incoming event:', JSON.stringify(event, null, 2));

    if (!event.Records) {
        throw new Error('No records found in event');
    }

    const processedRecords = [];

    for (const record of event.Records) {
        try {
            if (record.eventSource !== 'aws:kinesis') {
                console.log(`Skipping non-Kinesis record from ${record.eventSource}`);
                continue;
            }

            // Assumes that we are processing records from Kinesis
            const payload = record.kinesis.data;
            const decodedData = Buffer.from(payload, 'base64').toString();
            console.log("decoded payload is ", decodedData);
```

```
            const output = await handleDecodedData(decodedData);

            // Add additional processing logic here
            const processedData = {
                output,
                sequenceNumber: record.kinesis.sequenceNumber,
                partitionKey: record.kinesis.partitionKey,
                timestamp: record.kinesis.approximateArrivalTimestamp
            };

            processedRecords.push(processedData);

        } catch (error) {
            console.error('Error processing record:', error);
            console.error('Failed record:', record);
            // Decide whether to throw error or continue processing other records
            // throw error; // Uncomment to stop processing on first error
        }
    }

    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'Processing complete',
            processedCount: processedRecords.length,
            records: processedRecords
        })
    };
};

// Helper function to handle decoded data
async function handleDecodedData(payload) {
    try {
        // Parse the decoded data
        const parsedData = JSON.parse(payload);

        // Extract instanceId
        const instanceId = parsedData.instanceId;
        // Parse the input field
        const inputData = JSON.parse(parsedData.payload);
        const temperature = inputData.temperature;
        console.log('For InstanceId: ', instanceId, ' the temperature is:',
 temperature);
```

```
        await iotEvents.process(instanceId, inputData)

        return {
            instanceId,
            temperature,
            // Add any other fields you want to return
            rawInput: inputData
        };
    } catch (error) {
        console.error('Error handling decoded data:', error);
        throw error;
    }
}


//// Classes for declaring/defining the state machine
class CurrentState {
    constructor(instanceId, stateName, variables, inputs) {
        this.stateName = stateName;
        this.variables = variables;
        this.inputs = inputs;
        this.instanceId = instanceId
    }

    static async load(instanceId) {
        console.log(`Loading state for id ${instanceId}`);
        try {
            const { Item: { state: { S: stateContent } } } = await ddb.send(new
 GetItemCommand({
                TableName: 'EventsStateTable',
                Key: {
                    'InstanceId': { S: `${instanceId}` }
                }
            }));

            const { stateName, variables, inputs } = JSON.parse(stateContent);

            return new CurrentState(instanceId, stateName, variables, inputs);
        } catch (e) {
            console.log(`No state for id ${instanceId}: ${e}`);
            return undefined;
        }
    }
```

```
    static async save(instanceId, state) {
        console.log(`Saving state for id ${instanceId}`);
        await ddb.send(new PutItemCommand({
            TableName: 'your-events-state-table-name',
            Item: {
                'InstanceId': { S: `${instanceId}` },
                'state': { S: state }
            }
        }));
    }

    setVariable(name, value) {
        this.variables[name] = value;
    }

    changeState(stateName) {
        console.log(`Changing state from ${this.stateName} to ${stateName}`);
        this.stateName = stateName;
    }

    async setTimer(instanceId, frequencyInMinutes, payload) {
        console.log(`Setting timer ${instanceId} for frequency of ${frequencyInMinutes}
 minutes`);

        const base64Payload = Buffer.from(JSON.stringify(payload)).toString();
        console.log(base64Payload);

        const scheduleName = `your-schedule-name-${instanceId}-schedule`;
        const scheduleParams = {
            Name: scheduleName,
            FlexibleTimeWindow: {
                Mode: 'OFF'
            },
            ScheduleExpression: `rate(${frequencyInMinutes} minutes)`,
            Target: {
                Arn: "arn:aws::kinesis:your-region:your-account-id:stream/your-kinesis-
stream-name",
                RoleArn: "arn:aws::iam::your-account-id:role/service-role/your-iam-
role",
                Input: base64Payload,
                KinesisParameters: {
                    PartitionKey: instanceId,
                },
                RetryPolicy: {
```

```
                    MaximumRetryAttempts: 3
                }
            },

        };

        const command = new CreateScheduleCommand(scheduleParams);
        console.log(`Sending command to set timer ${JSON.stringify(command)}`);
        await scheduler.send(command);
    }

    async clearTimer(instanceId) {
        console.log(`Cleaning timer ${instanceId}`);

        const scheduleName = `your-schedule-name-${instanceId}-schedule`;
        const command = new DeleteScheduleCommand({
            Name: scheduleName
        });
        await scheduler.send(command);
    }

    async executeAction(actionType, actionPayload) {
        console.log(`Will execute the ${actionType} with payload ${actionPayload}`);
        await iot.send(new PublishCommand({
            topic: `${this.instanceId}`,
            payload: actionPayload,
            qos: 0
        }));
    }

    setInput(value) {
        this.inputs = { ...this.inputs, ...value };
    }

    input(name) {
        return this.inputs[name];
    }
}


class IoTEvents {

    constructor(initialState) {
        this.initialState = initialState;
```

```
        this.states = {};
    }

    state(name) {
        const state = new IoTEventsState();
        this.states[name] = state;
        return state;
    }

    async process(instanceId, input) {
        let currentState = await CurrentState.load(instanceId) || new
 CurrentState(instanceId, this.initialState, {}, {});
        currentState.setInput(input);

        console.log(`With inputs as: ${JSON.stringify(currentState)}`);
        const state = this.states[currentState.stateName];

        currentState = await state.evaluate(currentState);
        console.log(`With output as: ${JSON.stringify(currentState)}`);

        await CurrentState.save(instanceId, JSON.stringify(currentState));
    }
}

class Event {
    constructor(condition, action) {
        this.condition = condition;
        this.action = action;
    }
}

class IoTEventsState {
    constructor() {
        this.eventsList = []
    }

    events(eventListArg) {
        this.eventsList.push(...eventListArg);
        return this;
    }

    async evaluate(currentState) {
        for (const e of this.eventsList) {
            console.log(`Evaluating event ${e.condition}`);
```

```
            if (e.condition(currentState)) {
                console.log(`Event condition met`);
                // Execute any action as defined in iotEvents DM Definition
                await e.action(currentState);
            }
        }

        return currentState;
    }
}

////// DetectorModel Definitions - replace with your own defintions
let processAlarmStateEvent = new Event(
    (currentState) => {
        const source = currentState.input('source');
        return (
            currentState.input('temperature') < 70
        );
    },
    async (currentState) => {
        currentState.changeState('normal');
        await currentState.clearTimer(currentState.instanceId)
        await currentState.executeAction('MQTT', `{"state": "alarm cleared, timer
 deleted" }`);
    }
);

let processTimerEvent = new Event(
    (currentState) => {
        const source = currentState.input('source');
        console.log(`Evaluating timer event with source ${source}`);
        const booleanOutput = (source !== undefined && source !== null &&
            typeof source === 'string' &&
            source.toLowerCase() === 'timer' &&
            // check if the currentState == state from the timer payload
            currentState.input('currentState') !== undefined &&
            currentState.input('currentState') !== null &&
            currentState.input('currentState').toLowerCase !== 'normal');
        console.log(`Timer event evaluated as ${booleanOutput}`);
        return booleanOutput;
    },
    async (currentState) => {
        await currentState.executeAction('MQTT', `{"state": "timer timed out in
 Alarming state" }`);
```

```
    }
);

let processNormalEvent = new Event(
    (currentState) => currentState.input('temperature') > 70,
    async (currentState) => {
        currentState.changeState('alarm');
        await currentState.executeAction('MQTT', `{"state": "alarm detected, timer
 started" }`);
        await currentState.setTimer(currentState.instanceId, 5, {
            "instanceId": currentState.instanceId,
            "payload":"{\"currentState\": \"alarm\", \"source\": \"timer\"}"
        });
    }
);
const iotEvents = new IoTEvents('normal');
iotEvents
    .state('normal')
    .events(
        [
            processNormalEvent
        ]);
iotEvents
    .state('alarm')
    .events([
            processAlarmStateEvent,
            processTimerEvent
        ]
    );
```

Step 8: Add an Amazon Kinesis Data Streams trigger

Add a Kinesis Data Streams trigger to the Lambda function using the AWS Management Console or AWS CLI.

Adding a Kinesis Data Streams trigger to your Lambda function establishes the connection between your data ingestion pipeline and your processing logic, letting it automatically evaluate incoming IoT data streams and react to events in real-time, similar to how AWS IoT Events processes inputs.

Console

For more information, see Create an event source mapping to invoke a Lambda function in the *AWS Lambda Developer Guide*.

Use the following for the event source mapping details:

- For **Function name**, enter the lambda name used in Step 7: Create an AWS Lambda function (console).
- For **Consumer - optional**, enter the ARN for the your Kinesis stream.
- For **Batch size**, enter **10**.

AWS CLI

Run the following command to create the Lambda function trigger.

```
aws lambda create-event-source-mapping \
    --function-name your-lambda-name \
    --event-source arn:aws:kinesis:your-region:your-account-id:stream/your-kinesis-stream-name \
    --batch-size 10 \
    --starting-position LATEST \
    --region your-region
```

Step 9: Test data ingestion and output functionality (AWS CLI)

Publish a payload to the MQTT topic based on what you defined in your detector model. The following is an example payload to the MQTT topic `your-topic-name` to test an implementation.

```
{
  "instanceId": "your-instance-id",
  "payload": "{\"temperature\":78}"
}
```

You should see an MQTT message published to a topic with the following (or similar) content:

```
{
    "state": "alarm detected, timer started"
```

```
}
```

Migration procedure for AWS IoT SiteWise alarms in AWS IoT Events

This section describes alternative solutions that deliver similar alarm functionality as you migrate away from AWS IoT Events.

For AWS IoT SiteWise properties that use AWS IoT Events alarms, you can migrate to a solution using CloudWatch alarms. This approach provides robust monitoring capabilities with established SLAs and additional features like anomaly detection and grouped alarms.

Comparing architectures

The current AWS IoT Events alarm configuration for AWS IoT SiteWise properties requires creating `AssetModelCompositeModels` in the asset model, as described in Define external alarms in AWS IoT SiteWise in the *AWS IoT SiteWise User Guide*. Modifications to the new solution are typically managed through the AWS IoT Events console.

The new solution provides alarm management by leveraging CloudWatch alarms. This approach uses AWS IoT SiteWise notifications to publish property data points to AWS IoT Core MQTT topics, which are then processed by a Lambda function. The function transforms these notifications into CloudWatch metrics, enabling alarm monitoring through CloudWatch's alarming framework.

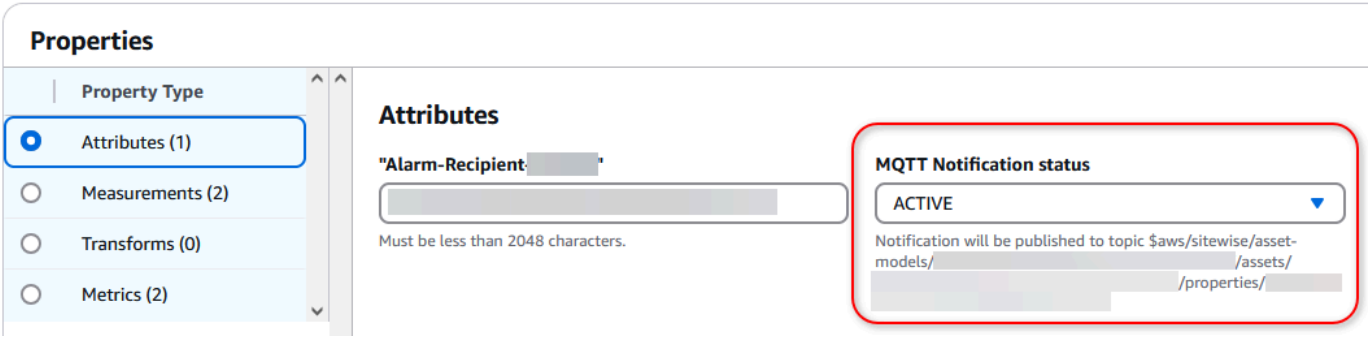
Purpose	Solution	Differences
Data source – Property data from AWS IoT SiteWise	AWS IoT SiteWise MQTT notifications	Replaces direct IoT Events integration with MQTT notifications from AWS IoT SiteWise properties
Data processing – Transforms property data	Lambda function	Processes AWS IoT SiteWise property notifications and converts them to CloudWatch metrics
Alarm evaluation – Monitors metrics and triggers alarms	Amazon CloudWatch alarms	Replaces AWS IoT Events alarms with CloudWatch alarms, offering additional features like anomaly detection

Purpose	Solution	Differences
Integration – Connection with AWS IoT SiteWise	AWS IoT SiteWise external alarms	Optional capability to import CloudWatch alarms back into AWS IoT SiteWise as external alarms

Step 1: Enable MQTT notifications on the asset property

If you are using AWS IoT Events integrations for AWS IoT SiteWise alarms, you can turn on MQTT notifications for each property to monitor.

1. Follow the Configure alarms on assets in AWS IoT SiteWise procedure until you each the step to **Edit** the asset model's properties.
2. For each property to migrate, change the **MQTT Notification status** to **ACTIVE**.



3. Note the topic path to which the alarm publishes for each modified alarm attribute.

For more information, see the following documentation resources:

- Understand asset properties in MQTT topics in the *AWS IoT SiteWise User Guide*.
- MQTT topics in the *AWS IoT Developer Guide*.

Step 2: Create an AWS Lambda function

Create an Lambda function for reading the TQV array published by the MQTT topic and publish individual values to CloudWatch. We'll use this Lambda function as a destination action to trigger in AWS IoT Core Message Rules.

1. Open the AWS Lambda console.

2. Choose **Create function**.
3. Enter a name for the **Function name**.
4. Select **NodeJS 22.x** as the **Runtime**.
5. In the **Change default execution role** dropdown, choose **Use existing role**, and then select the IAM role that you created in earlier steps.

> **Note**
>
> This procedure assumes that you've already migrated your detector model. If you don't have an IAM role, see Step 2: Create an IAM role.

6. Choose **Create function**.
7. Paste in the following code snippet after replacing the hard coded constants.

```
import json
import boto3
from datetime import datetime

# Initialize CloudWatch client
cloudwatch = boto3.client('cloudwatch')

def lambda_handler(message, context):
    try:
        # Parse the incoming IoT message
        # Extract relevant information
        asset_id = message['payload']['assetId']
        property_id = message['payload']['propertyId']

        # Process each value in the values array
        for value in message['payload']['values']:
            # Extract timestamp and value
            timestamp = datetime.fromtimestamp(value['timestamp']['timeInSeconds'])
            metric_value = value['value']['doubleValue']
            quality = value.get('quality', 'UNKNOWN')

            # Publish to CloudWatch
            response = cloudwatch.put_metric_data(
                Namespace='IoTSiteWise/AssetMetrics',
                MetricData=[
                    {
                        'MetricName': f'Property_your-property-id',
```

```
                        'Value': metric_value,
                        'Timestamp': timestamp,
                        'Dimensions': [
                            {
                                'Name': 'AssetId',
                                'Value': 'your-asset-id'
                            },
                            {
                                'Name': 'Quality',
                                'Value': quality
                            }
                        ]
                    }
                ]
            )

        return {
            'statusCode': 200,
            'body': json.dumps('Successfully published metrics to CloudWatch')
        }

    except Exception as e:
        print(f'Error processing message: {str(e)}')
        return {
            'statusCode': 500,
            'body': json.dumps(f'Error: {str(e)}')
        }
```

Step 3: Create AWS IoT Core message routing rule

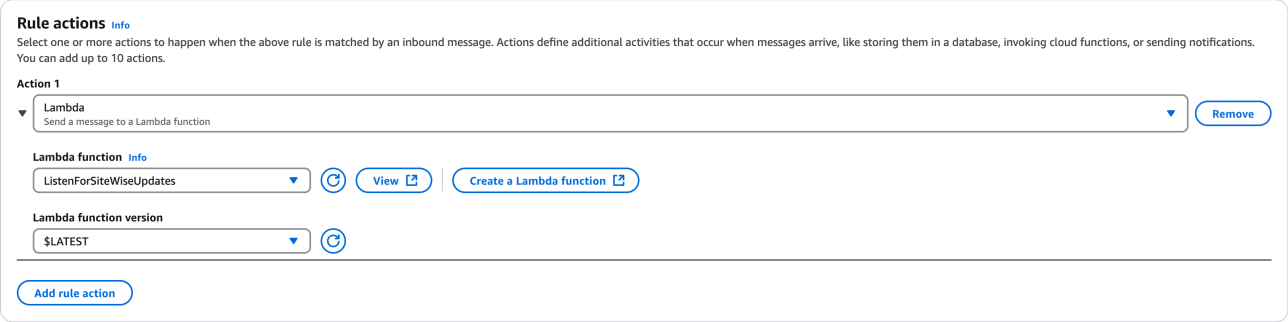
- Follow the Tutorial: Republishing an MQTT message procedure entering the following information when prompted:

  a. Name message routing rule `SiteWiseToCloudwatchAlarms`.

  b. For the query, you can use the following:

  ```
  SELECT * FROM '$aws/sitewise/asset-models/your-asset-model-id/assets/your-asset-id/properties/your-property-id'
  ```

  c. In **Rule actions**, select the **Lambda** action to send the data generated from AWS IoT SiteWise to CloudWatch. For example:



Step 4: View CloudWatch metrics

As you ingest data to AWS IoT SiteWise, the property selected earlier in Step 1: Enable MQTT notifications on the asset property, the routes data to the Lambda function we created in Step 2: Create an AWS Lambda function. In this step, you can check to see the Lambda sending your metrics to CloudWatch.

1. Open the CloudWatch AWS Management Console.
2. In the left navigation, choose **Metrics**, then **All metrics**.
3. Choose an alarm's URL to open it.
4. Under the **Source** tab, the CloudWatch output looks similar to this example. This source information confirms that the metric data is feeding into CloudWatch.

```
{
    "view": "timeSeries",
    "stacked": false,
    "metrics": [
        [ "IoTSiteWise/AssetMetrics", "Property_your-property-id-hash", "Quality",
 "GOOD", "AssetId", "your-asset-id-hash", { "id": "m1" } ]
    ],
    "region": "your-region"
}
```

Step 5: Create CloudWatch alarms

Follow the Create a CloudWatch alarm based on a static threshold procedure in the *Amazon CloudWatch User Guide* to create alarms for each relevant metric.

> **Note**
>
> There are many options for alarm configuration in Amazon CloudWatch For more information on CloudWatch alarms, see Using Amazon CloudWatch alarms in the *Amazon CloudWatch User Guide*.

Step 6: (Optional) import the CloudWatch alarm into AWS IoT SiteWise

You can configure CloudWatch alarms to send data back to AWS IoT SiteWise using CloudWatch alarm actions and Lambda. This integration enables you to view alarm states and properties in the SiteWise Monitor portal.

1. Configure the external alarm as a property in an asset model. For more information, see Define external alarms in AWS IoT SiteWise in the *AWS IoT SiteWise User Guide*.
2. Create a Lambda function that uses the BatchPutAssetPropertyValue API found in the *AWS IoT SiteWise User Guide* to send alarm data to AWS IoT SiteWise.
3. Set up CloudWatch alarm actions to invoke your Lambda function when alarm states change. For more information, see the Alarm actions section in the *Amazon CloudWatch User Guide*.

Setting up AWS IoT Events

This section provides a guide to setting up AWS IoT Events, including creating an AWS account, configuring necessary permissions, and establishing roles for managing access to resources.

Topics

- Setting up an AWS account
- Setting up permissions for AWS IoT Events

Setting up an AWS account

Sign up for an AWS account

To get started with AWS, you need an AWS account. For information about creating an AWS account, see Getting started with an AWS account in the *AWS Account Management Reference Guide*.

Setting up permissions for AWS IoT Events

Implementing proper permissions is important for secure and effective use of AWS IoT Events. This section describes the permissions that are required to use some features of AWS IoT Events. You can use AWS CLI commands or the AWS Identity and Access Management (IAM) console to create roles and associated permission policies to access resources or perform certain functions in AWS IoT Events.

The IAM User Guide has more detailed information about securely controlling permissions to access AWS resources. For information specific to AWS IoT Events, see Actions, resources, and condition keys for AWS IoT Events.

To use the IAM console to create and manage roles and permissions, see IAM tutorial: Delegate access across AWS accounts using IAM roles.

> **Note**
>
> Keys can be 1-128 characters and can include:

- uppercase or lowercase letters a-z
- numbers 0-9
- special characters **-**, **_**, or **:**.

Action permissions for AWS IoT Events

AWS IoT Events enables you to trigger actions which use other AWS services. To do so, you must grant AWS IoT Events permission to perform these actions on your behalf. This section contains a list of the actions and an example policy which grants permission to perform all these actions on your resources. Change the *region* and *account-id* references as required. When possible, you should also change the wildcards (*) to refer to specific resources that will be accessed. You can use the IAM console to grant permission to AWS IoT Events to send an Amazon SNS alert that you have defined. .

AWS IoT Events supports the following actions that let you use a timer or set a variable:

- `setTimer` to create a timer.
- `resetTimer` to reset the timer.
- `clearTimer` to delete the timer.
- `setVariable` to create a variable.

AWS IoT Events supports the following actions that let you work with AWS services:

- `iotTopicPublish` to publish a message on an MQTT topic.
- `iotEvents` to send data to AWS IoT Events as an input value.
- `iotSiteWise` to send data to an asset property in AWS IoT SiteWise.
- `dynamoDB` to send data to an Amazon DynamoDB table.
- `dynamoDBv2` to send data to an Amazon DynamoDB table.
- `firehose` to send data to an Amazon Data Firehose stream.
- `lambda` to invoke an AWS Lambda function.
- `sns` to send data as a push notification.
- `sqs` to send data to an Amazon SQS queue.

Example Policy

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "iot:Publish",
            "Resource": "arn:aws:iot:us-east-1:123456789012:topic/*"
        },
        {
            "Effect": "Allow",
            "Action": "iotevents:BatchPutMessage",
            "Resource": "arn:aws:iotevents:us-east-1:123456789012:input/*"
        },
        {
            "Effect": "Allow",
            "Action": "iotsitewise:BatchPutAssetPropertyValue",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "dynamodb:PutItem",
            "Resource": "arn:aws:dynamodb:us-east-1:123456789012:table/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "firehose:PutRecord",
                "firehose:PutRecordBatch"
            ],
            "Resource": "arn:aws:firehose:us-east-1:123456789012:deliverystream/
*"
        },
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*"
        },
        {
            "Effect": "Allow",
```

```
            "Action": "sns:Publish",
            "Resource": "arn:aws:sns:us-east-1:123456789012:*"
        },
        {
            "Effect": "Allow",
            "Action": "sqs:SendMessage",
            "Resource": "arn:aws:sqs:us-east-1:123456789012:*"
        }
    ]
}
```

Securing input data in AWS IoT Events

It's important to consider who can grant access to input data for use in a detector model. If you have a user or entity whose overall permissions you want to restrict, but that is permitted to create or update a detector model, you must also grant permission for that user or entity to update input routing. This means that in addition to granting permission for `iotevents:CreateDetectorModel` and `iotevents:UpdateDetectorModel`, you must also grant permission for `iotevents:UpdateInputRouting`.

Example

The following policy adds permission for `iotevents:UpdateInputRouting`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "updateRoutingPolicy",
            "Effect": "Allow",
            "Action": [
                "iotevents:UpdateInputRouting"
            ],
            "Resource": "*"
        }
    ]
}
```

You can specify a list of input Amazon Resource Names (ARNs) instead of the wildcard "*" for the "Resource" to limit this permission to specific inputs. This enables you to restrict access to the input data that is consumed by detector models created or updated by the user or entity.

Amazon CloudWatch logging role policy for AWS IoT Events

The following policy documents provide the role policy and trust policy that allow AWS IoT Events to submit logs to CloudWatch on your behalf.

Role policy:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:PutMetricFilter",
                "logs:PutRetentionPolicy",
                "logs:GetLogEvents",
                "logs:DeleteLogStream"
            ],
            "Resource": [
                "arn:aws:logs:*:*:*"
            ]
        }
    ]
}
```

Trust policy:

JSON

```
{
  "Version":"2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [

          "iotevents.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

You also need an IAM permissions policy attached to the user that allows the user to pass roles, as follows. For more information, see Granting a user permissions to pass a role to an AWS service in the *IAM User Guide*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "",
            "Effect": "Allow",
            "Action": [
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::123456789012:role/Role_To_Pass"
        }
    ]
}
```

You can use the following command to put the resource policy for CloudWatch logs. This allows AWS IoT Events to put log events into CloudWatch streams.

```
aws logs put-resource-policy --policy-name ioteventsLoggingPolicy --policy-
document "{ \"Version\": \"2012-10-17\",        \"Statement\": [ { \"Sid\":
 \"IoTEventsToCloudWatchLogs\", \"Effect\": \"Allow\", \"Principal\": { \"Service\":
 [ \"iotevents.amazonaws.com\" ] }, \"Action\":\"logs:PutLogEvents\", \"Resource\": \"*
\" } ] }"
```

Use the following command to put logging options. Replace the `roleArn` with the logging role that you created.

```
aws iotevents put-logging-options --cli-input-json "{ \"loggingOptions\": {\"roleArn\":
 \"arn:aws:iam::123456789012:role/testLoggingRole\", \"level\": \"INFO\", \"enabled\":
 true } }"
```

Amazon SNS messaging role policy for AWS IoT Events

Integrating AWS IoT Events with Amazon SNS requires careful permission management for secure and efficient notification delivery. This guide walks you through the process of configuring IAM roles and policies to allow AWS IoT Events to publish messages to Amazon SNS topics.

The following policy documents provide the role policy and trust policy that allow AWS IoT Events to send SNS messages.

Role policy:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "sns:*"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:sns:us-east-1:123456789012:testAction"
        }
    ]
}
```

Trust policy:

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "iotevents.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Getting started with the AWS IoT Events console

This section shows you how to create an input and a detector model using the AWS IoT Events console. You model two states of an engine: a normal state and an over-pressure condition. When the measured pressure in the engine exceeds a certain threshold, the model transitions from the normal state to the over-pressure state. Then it sends an Amazon SNS message to alert a technician about the condition. When the pressure again drops below the threshold for three consecutive pressure readings, the model returns to the normal state and sends another Amazon SNS message as a confirmation.

We check for three consecutive readings below the pressure threshold to eliminate possible stuttering of over-pressure or normal messages, in case of a nonlinear recovery phase or an anomalous pressure reading.

On the console, you can also find several pre-made detector model templates which you can customize. You can also use the console to import detector models that others have written or export your detector models and use them in different AWS Regions. If you import a detector model, make sure that you create the required inputs or recreate them for the new Region, and update any role ARNs used.

Use the AWS IoT Events console to learn about the following.

Define inputs

To monitor your devices and processes, they must have a way to get telemetry data into AWS IoT Events. This is done by sending messages as *inputs* to AWS IoT Events. You can do this in several ways:

- Use the BatchPutMessage operation.
- In AWS IoT Core, write an AWS IoT Events action rule for the AWS IoT rules engine that forwards your message data into AWS IoT Events. You must identify the input by name.
- In AWS IoT Analytics, use the CreateDataset operation to create a data set with `contentDeliveryRules`. These rules specify the AWS IoT Events input where data set contents are sent automatically.

Before your devices can send data in this way, you must define one or more inputs. To do so, give each input a name and specify which fields in the incoming message data the input monitors.

Create a detector model

Define a *detector model* (a model of your equipment or process) using *states*. For each state, define conditional (Boolean) logic that evaluates the incoming inputs to detect significant events. When the detector model detects an event, it can change the state or initiate custom-built or predefined actions using other AWS services. You can define additional events that initiate actions when entering or exiting a state and, optionally, when a condition is met.

In this tutorial, you send an Amazon SNS message as the action when the model enters or exits a certain state.

Monitor a device or process

If you monitor several devices or processes, specify a field in each input that identifies the particular device or process from which the input comes. See the `key` field in `CreateDetectorModel`. When the input field identified by the `key` recognizes a new value, a new device is identified and a detector is created. Each detector is an instance of the detector model. The new detector continues responding to inputs coming from that device until its detector model is updated or deleted.

If you monitor a single process (even if several devices or subprocesses are sending inputs), you don't specify a unique identifying `key` field. In this case, the model creates a single detector (instance) when the first input arrives.

Send messages as inputs to your detector model

There are several ways to send a message from a device or process as an input into an AWS IoT Events detector that don't require you to perform additional formatting on the message. In this tutorial, you use the AWS IoT console to write an AWS IoT Events action rule for the AWS IoT rules engine that forwards your message data into AWS IoT Events.

To do this, identify the input by name and continue to use the AWS IoT console to generate messages that are forwarded as inputs to AWS IoT Events.

> **Note**
>
> This tutorial uses the console to create the same `input` and `detector model` shown in the example at Tutorials for AWS IoT Events uses cases. You can use the this JSON example to help you follow the tutorial.

Topics

- Prerequisites to get started with AWS IoT Events
- Create an input for models in AWS IoT Events
- Create a detector model in AWS IoT Events
- Send inputs to test the detector model in AWS IoT Events

Prerequisites to get started with AWS IoT Events

If you don't have an AWS account, create one.

1. Follow the step in Setting up AWS IoT Events to ensure proper account setup and permissions.
2. Create two Amazon Simple Notification Service (Amazon SNS) topics.

   This tutorial (and the corresponding example) assume that you created two Amazon SNS topics. The ARNs of these topics are shown as: `arn:aws:sns:us-east-1:123456789012:underPressureAction` and `arn:aws:sns:us-east-1:123456789012:pressureClearedAction`. Replace these values with the ARNs of Amazon SNS topics that you create. For more information, see the Amazon Simple Notification Service Developer Guide.

   As an alternative to publishing alerts to Amazon SNS topics, you can have the detectors send MQTT messages with a topic that you specify. With this option, you can verify that your detector model is creating instances and that those instances are sending alerts by using the AWS IoT Core console to subscribe to and monitor messages sent to those MQTT topics. You can also define the MQTT topic name dynamically at runtime by using an input or variable created in the detector model.
3. Choose an AWS Region that supports AWS IoT Events. For more information, see AWS IoT Events in the *AWS General Reference*. For help, see Getting started with a service in the AWS Management Console in the *Getting Started with the AWS Management Console*.

Create an input for models in AWS IoT Events

When you construct the inputs for your models, we recommend gathering files that contain sample message payloads that your devices or processes send to report their health status. Having these files helps you define the inputs that are required.

You can create an input through multiple methods that are described in this section.

Create a JSON input file

1. To get started, create a file named `input.json` on your local file system with the following contents:

   ```
   {
     "motorid": "Fulton-A32",
     "sensorData": {
       "pressure": 23,
       "temperature": 47
     }
   }
   ```

2. Now that you have this starter `input.json` file, you can create an input. There are two ways to create an input. You can create an input by using the navigation pane in the AWS IoT Events console . Or, you can create an input within the detector model after it's created.

Create and configure an input

Learn how to create an *input*, for an alarm model or a detector model.

1. Log into the AWS IoT Events console or select the option to Create a new AWS IoT Events account.
2. In the AWS IoT Events console, in the upper left corner, select and expand the navigation pane.
3. In the left navigation pane, select **Inputs**.
4. In the right corner of the console, choose **Create input**.
5. Provide a unique**InputName**.
6. *Optional* – enter a **Description** for your input.
7. To **Upload a JSON file**, select the `input.json` file that you created in the overview for Create a JSON input file. **Choose input attributes** appears with a list of your entered attributes.
8. For **Choose input attributes**, select the attributes to use, and choose **Create**. In this example, we select **motorid** and **sensorData.pressure**.
9. *Optional* – Add relevant **Tags** to the input.

> **Note**
>
> You can also create additional inputs within the detector model in the AWS IoT Events console . For more information, see Create an input within the Detector Model in AWS IoT Events.

Create an input within the Detector Model in AWS IoT Events

Detector inputs in AWS IoT Events serve as the bridge between your data sources and detector models. Detector inputs provide the raw data that powers the event detection and automation capabilities of AWS IoT Events. Learn to configure detector inputs to help your models respond accurately to real-world events and conditions in your IoT ecosystem.

This section shows how to define an *input* for a detector model to receive telemetry data, or messages.

To define an input for a detector model

1. Open the AWS IoT Events console.
2. In the AWS IoT Events console, choose **Create detector model**.
3. Choose **Create new**.
4. Choose **Create input**.
5. For the input, enter an **InputName**, an optional **Description**, and choose **Upload file**. In the dialog box that displays, select the `input.json` file that you created in the overview for Create a JSON input file.
6. For **Choose input attributes**, select the attributes to use, and choose **Create**. In this example, we select **motorId** and **sensorData.pressure**.

Create a detector model in AWS IoT Events

In this topic, you define a *detector model* (a model of your equipment or process) using *states*.

For each state, you define conditional (Boolean) logic that evaluates the incoming inputs to detect a significant event. When an event is detected, it changes the state and can initiate additional actions. These events are known as transition events.

In your states, you also define events that can run actions whenever the detector enters or exits that state or when an input is received (these are known as `OnEnter`, `OnExit` and `OnInput` events). The actions run only if the event's conditional logic evaluates to `true`.

To create a detector model

1. The first detector state has been created for you. To modify it, select the circle with label **State_1** in the main editing space.
2. In the **State** pane, enter the **State name** and **OnEnter**, choose **Add event**.
3. On the **Add OnEnter event** page, enter an **Event name** and the **Event condition**. In this example, enter `true` to indicate the event is always initiated when the state is entered.
4. Under **Event actions**, choose **Add action**.
5. Under **Event actions**, do the following:

   a. Select **Set variable**

   b. For **Variable operation**, choose **Assign value**.

   c. For **Variable name**, enter the name of the variable to set.

   d. For **Variable value**, enter the value **`0`** (zero).
6. Choose **Save**.

   A variable, like the one you defined, can be set (given a value) in any event in the detector model. The variable's value can only be referenced (for example, in an event's conditional logic) after the detector has reached a state and run an action where it is defined or set.
7. In the **State** pane, choose the **X** next to **State** to return to the **Detector model palette**.
8. To create a second detector state, in the **Detector model palette**, choose **State** and drag it into the main editing space. This creates a state titled `untitled_state_1`.
9. Pause on the first state (**Normal**). An arrow appears on the circumference of the state.
10. Click and drag the arrow from the first state to the second state. A directed line from the first state to the second state (labeled **Untitled**) appears.
11. Select the **Untitled** line. In the **Transition event** pane, enter an **Event name** and **Event trigger logic**.
12. In the **Transition event** pane, choose **Add action**.
13. On the **Add transition event actions** pane, choose **Add action**.
14. For **Choose an action**, choose **Set variable**.

a. For **Variable operation**, choose **Assign value**.

b. For **Variable name**, enter the name of the variable.

c. For **Assign value**, enter the value such as: `$variable.pressureThresholdBreached + 3`

d. Choose **Save**.

15. Select the second state **untitled_state_1**.

16. In the **State** pane, enter the **State name** and for **On Enter**, choose **Add event**.

17. On the **Add OnEnter event** page, enter the **Event name** and **Event condition**. Choose **Add action**.

18. For **Choose an action**, choose **Send SNS message**.

a. For **SNS topic**, enter the target ARN of your Amazon SNS topic.

b. Choose **Save**.

19. Continue to add the events in the example.

a. For **OnInput**, choose **Add event**, and enter and save the following event information.

```
Event name: Overpressurized
Event condition: $input.PressureInput.sensorData.pressure > 70
Event actions:
  Set variable:
    Variable operation: Assign value
    Variable name: pressureThresholdBreached
    Assign value: 3
```

b. For **OnInput**, choose **Add event**, and enter and save the following event information.

```
Event name: Pressure Okay
Event condition: $input.PressureInput.sensorData.pressure <= 70
Event actions:
  Set variable:
    Variable operation: Decrement
    Variable name: pressureThresholdBreached
```

c. For **OnExit**, choose **Add event**, and enter and save the following event information using the ARN of the Amazon SNS topic that you created.

```
Event name: Normal Pressure Restored
Event condition: true
Event actions:
  Send SNS message:
    Target arn: arn:aws:sns:us-east-1:123456789012:pressureClearedAction
```

20. Pause on the second state (**Dangerous**). An arrow appears on the circumference of the state
21. Click and drag the arrow from the second state to the first state. A directed line with label **Untitled** appears.
22. Choose the **Untitled** line and in the **Transition event** pane, enter an **Event name** and **Event trigger logic** using the following information.

```
{
  Event name: BackToNormal
  Event trigger logic: $input.PressureInput.sensorData.pressure <= 70 &&
 $variable.pressureThresholdBreached <= 0
}
```

    For more information about why we test for the `$input` value and the `$variable` value in the trigger logic, see the entry for availability of variable values in AWS IoT Events detector model restrictions and limitations.
23. Select the **Start** state. By default, this state was created when you created a detector model). In the **Start** pane, choose the **Destination state** (for example, **Normal**).
24. Next, configure your detector model to listen for inputs. In the upper-right corner, choose **Publish**.
25. On the **Publish detector model** page, do the following.

    a. Enter a **Detector model name**, a **Description**, and the name of a **Role**. This role is created for you.

    b. Choose **Create a detector for each unique key value**. To create and use your own **Role**, follow the steps in Setting up permissions for AWS IoT Events and enter it as the **Role** here.
26. For **Detector creation key**, choose the name of one of the attributes of the input you defined earlier. The attribute that you choose as the detector creation key must be present in each

message input, and must be unique to each device that sends messages. This example uses the **motorid** attribute.

27. Choose **Save and publish**.

> **Note**
>
> The number of unique detectors created for a given detector model is based on the input messages sent. When a detector model is created, a key is selected from the input attributes. This key determines which detector instance to use. If the key hasn't been seen before (for this detector model), a new detector instance is created. If the key has been seen before, we use the existing detector instance corresponding to this key value.

You can make a backup copy of your detector model definition (in JSON) recreate or update the detector model or use as a template to create another detector model.

You can do this from the console or by using the following CLI command. If necessary, change the name of the detector model to match what you used when you published it in the previous step.

```
aws iotevents describe-detector-model  --detector-model-name motorDetectorModel >
 motorDetectorModel.json
```

This creates a file (`motorDetectorModel.json`) that has contents similar to the following.

```
{
    "detectorModel": {
        "detectorModelConfiguration": {
            "status": "ACTIVE",
            "lastUpdateTime": 1552072424.212,
            "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole",
            "creationTime": 1552072424.212,
            "detectorModelArn": "arn:aws:iotevents:us-
west-2:123456789012:detectorModel/motorDetectorModel",
            "key": "motorid",
            "detectorModelName": "motorDetectorModel",
            "detectorModelVersion": "1"
        },
        "detectorModelDefinition": {
            "states": [
                {
```

```
                    "onInput": {
                        "transitionEvents": [
                            {
                                "eventName": "Overpressurized",
                                "actions": [
                                    {
                                        "setVariable": {
                                            "variableName":
"pressureThresholdBreached",
                                            "value":
"$variable.pressureThresholdBreached + 3"
                                        }
                                    }
                                ],
                                "condition": "$input.PressureInput.sensorData.pressure
> 70",
                                "nextState": "Dangerous"
                            }
                        ],
                        "events": []
                    },
                    "stateName": "Normal",
                    "onEnter": {
                        "events": [
                            {
                                "eventName": "init",
                                "actions": [
                                    {
                                        "setVariable": {
                                            "variableName":
"pressureThresholdBreached",
                                            "value": "0"
                                        }
                                    }
                                ],
                                "condition": "true"
                            }
                        ]
                    },
                    "onExit": {
                        "events": []
                    }
                },
                {
```

```
                    "onInput": {
                        "transitionEvents": [
                            {
                                "eventName": "Back to Normal",
                                "actions": [],
                                "condition": "$variable.pressureThresholdBreached <= 1
&& $input.PressureInput.sensorData.pressure <= 70",
                                "nextState": "Normal"
                            }
                        ],
                        "events": [
                            {
                                "eventName": "Overpressurized",
                                "actions": [
                                    {
                                        "setVariable": {
                                            "variableName":
"pressureThresholdBreached",
                                            "value": "3"
                                        }
                                    }
                                ],
                                "condition": "$input.PressureInput.sensorData.pressure
> 70"
                            },
                            {
                                "eventName": "Pressure Okay",
                                "actions": [
                                    {
                                        "setVariable": {
                                            "variableName":
"pressureThresholdBreached",
                                            "value":
"$variable.pressureThresholdBreached - 1"
                                        }
                                    }
                                ],
                                "condition": "$input.PressureInput.sensorData.pressure
<= 70"
                            }
                        ]
                    },
                    "stateName": "Dangerous",
                    "onEnter": {
```

```
                        "events": [
                            {
                                "eventName": "Pressure Threshold Breached",
                                "actions": [
                                    {
                                        "sns": {
                                            "targetArn": "arn:aws:sns:us-
west-2:123456789012:MyIoTButtonSNSTopic"
                                        }
                                    }
                                ],
                                "condition": "$variable.pressureThresholdBreached > 1"
                            }
                        ]
                    },
                    "onExit": {
                        "events": [
                            {
                                "eventName": "Normal Pressure Restored",
                                "actions": [
                                    {
                                        "sns": {
                                            "targetArn": "arn:aws:sns:us-
west-2:123456789012:IoTVirtualButtonTopic"
                                        }
                                    }
                                ],
                                "condition": "true"
                            }
                        ]
                    }
                }
            ],
            "initialStateName": "Normal"
        }
    }
}
```

Send inputs to test the detector model in AWS IoT Events

There are several ways to receive telemetry data in AWS IoT Events (see Supported actions to receive data and trigger actions in AWS IoT Events). This topic shows you how to create an AWS IoT

rule in the AWS IoT console that forwards messages as inputs to your AWS IoT Events detector. You can use the AWS IoT console's MQTT client to send test messages. You can use this method to get telemetry data into AWS IoT Events when your devices are able to send MQTT messages using the AWS IoT message broker.

To send inputs to test the detector model

1. Open the AWS IoT Core console. In the left navigation pane, under **Manage**, choose **Message routing**, then choose **Rules**.
2. Choose **Create rule** in the upper right.
3. On the **Create a rule** page, complete the following steps:

   1. **Step 1. Specify rule properties**. Complete the following fields:
      - **Rule name.** Enter a name for your rule, such as `MyIoTEventsRule`.

        > **Note**
        >
        > Do not use spaces.

      - **Rule description**. This is optional.
      - Choose **Next**.
   2. **Step 2. Configure SQL statement**. Complete the following fields:
      - **SQL version**. Select the appropriate option from the list.
      - **SQL statement**. Enter **`SELECT *, topic(2) as motorid FROM 'motors/+/status'`**.

      Choose **Next**.
   3. **Step 3. Attach rule actions**. In the **Rule actions** section, complete the following:
      - **Action 1. Select IoT Events.** The following fields appear:
        a. **Input name**. Select the appropriate option from the list. If your input doesn't appear, choose **Refresh**.

        To create a new input, choose **Create IoT Events input**. Complete the following fields:
        - **Input name**. Enter `PressureInput`.
        - **Description**. This is optional.

- **Upload a JSON file**. Upload a copy of your JSON file. There is a link to a sample file on this screen, if you don't have a file. The code includes:

```
{
  "motorid": "Fulton-A32",
  "sensorData": {
    "pressure": 23,
    "temperature": 47
  }
}
```

- **Choose input attributes**. Select the appropriate option(s).
- **Tags**. This is optional.

Choose **Create**.

Return to the **Create rule** screen and refresh the **Input name** field. Select the input you just created.

b. **Batch mode**. This is optional. If the payload is an array of messages, select this option.

c. **Message ID**. This is optional, but recommended.

d. **IAM role**. Select the appropriate role from the list. If the role isn't listed, choose **Create new role**.

Type a **Role name** and choose **Create**.

To add another rule, choose **Add rule action**

- **Error action**. This section is optional. To add an action, choose **Add error action** and select the appropriate action from the list.

Complete the fields that appear.

- Choose **Next**.

4. **Step 4. Review and create.** Review the information on the screen and choose **Create**.

4. In the left navigation pane, under **Test**, choose **MQTT test client**.
5. Choose **Publish to a topic**. Complete the following fields:

- **Topic name**. Enter a name to identify the message, such as `motors/Fulton-A32/status`.
- **Message payload**. Enter the following:

```
{
  "messageId": 100,
  "sensorData": {
    "pressure": 39
  }
}
```

> **Note**
>
> Change the `messageId` each time you publish a new message.

6. For **Publish**, keep the topic the same, but change the `"pressure"` in the payload to a value greater than the threshold value that you specified in the detector model (such as **85**).
7. Choose **Publish**.

The detector instance that you created generates and sends you an Amazon SNS message. Continue to send messages with pressure readings above or below the pressure threshold (70 for this example) to see the detector in operation.

In this example, you must send three messages with pressure readings below the threshold to transition back to the **Normal** state and receive an Amazon SNS message that indicates the overpressure condition has cleared. Once back in the **Normal** state, one message with a pressure reading above the limit causes the detector to enter the **Dangerous** state and send an Amazon SNS message indicating that condition.

Now that you have created a simple input and detector model, try the following.

- See more detector model examples (templates) on the console.
- Follow the steps in Create an AWS IoT Events detector for two states using CLI to create an input and detector model using the AWS CLI
- Learn details of the Expressions to filter, transform, and process event data used in events.
- Learn about Supported actions to receive data and trigger actions in AWS IoT Events.
- If something isn't working, see Troubleshooting AWS IoT Events.

Best practices for AWS IoT Events

Follow these best practices to get the maximum benefit from AWS IoT Events.

Topics

- Enable Amazon CloudWatch logging when developing AWS IoT Events detector models
- Publish regularly to save your detector model when working in the AWS IoT Events console

Enable Amazon CloudWatch logging when developing AWS IoT Events detector models

Amazon CloudWatch monitors your AWS resources and the applications that you run on AWS in real time. With CloudWatch, you gain system-wide visibility into resource use, application performance, and operational health. When you develop or debug an AWS IoT Events detector model, CloudWatch helps you know what AWS IoT Events is doing, and any errors that it encounters.

To enable CloudWatch

1. If you haven't already, follow the steps in Setting up permissions for AWS IoT Events to create a role with an attached policy that grants permission to create and manage CloudWatch logs for AWS IoT Events.
2. Go to the AWS IoT Events console.
3. In the navigation pane, choose **Settings**.
4. On the **Settings** page, choose **Edit**.
5. On the **Edit logging options** page, in the **Logging options** section, do the following:
    a. For **Level of verbosity**, select an option.
    b. For **Select role**, select a role with sufficient permissions to perform the logging actions that you chose.
    c. (Optional) If you chose **Debug** for the **Level of verbosity**, you can add Debug targets by doing the following:
        i. Under **Debug targets**, choose **Add Model Option**.

ii. Enter a **Detector Model Name** and (optional) **KeyValue** to specify the detector models and specific detectors (instances) to log.

6. Choose **Update**.

Your logging options are successfully updated.

Publish regularly to save your detector model when working in the AWS IoT Events console

When you use the AWS IoT Events console, your work in progress is saved locally in your browser. However, you must choose **Publish** to save your detector model to AWS IoT Events. After you publish a detector model, your published work will become available in any browser that you use to access your account.

> **Note**
>
> If you don't publish your work, it will not be saved. After you publish a detector model, you can't change its name. However, you can continue modifying its definition.

Tutorials for AWS IoT Events uses cases

AWS IoT Events tutorials provide a collection of procedures covering various aspects of AWS IoT Events, from basic setup to more specific use cases. Each tutorial shows examples of practical scenarios, helping you build real-world skills in creating detector models, configuring inputs, setting up actions, and integrating with other AWS services to create powerful IoT solutions.

This chapter shows you how to:

- Get help to decide which states to include in your detector model, and determine whether you need one detector instance or several.
- Follow an example that uses the AWS CLI.
- Create an input to receive telemetry data from a device and a detector model to monitor and report on the state of the device that sends that data.
- Review restrictions and limitations on inputs, detector models, and the AWS IoT Events service.
- See a more complex example of a detector model, with comments included.

Topics

- Using AWS IoT Events to monitor your IoT devices
- Create an AWS IoT Events detector for two states using CLI
- AWS IoT Events detector model restrictions and limitations
- A commented example: HVAC temperature control with AWS IoT Events

Using AWS IoT Events to monitor your IoT devices

You can use AWS IoT Events to monitor your devices or processes, and take action based on significant events. To do so, follow these basic steps:

Create inputs

You must have a way for your devices and processes to get telemetry data into AWS IoT Events. You do this by sending messages as *inputs* to AWS IoT Events. You can send messages as inputs in several ways:

- Use the BatchPutMessage operation.

- Define an `iotEvents` rule-action for the AWS IoT Core rules engine. The rule-action forwards message data from your input into AWS IoT Events.
- In AWS IoT Analytics, use the CreateDataset operation to create a data set with `contentDeliveryRules`. These rules specify the AWS IoT Events input where data set contents are sent automatically.
- Define an iotEvents action in an AWS IoT Events detector model's `onInput`, `onExit` or `transitionEvents` event. Information about the detector model instance and the event that initiated the action are fed back into the system as an input with the name that you specify.

Before your devices start sending data in this way, you must define one or more inputs. To do so, give each input a name and specify which fields in the incoming message data the input monitors. AWS IoT Events receives its input, in the form of JSON payload, from many sources. Each input can be acted on by itself, or combined with other inputs to detect more complex events.

Create a detector model

Define a *detector model* (a model of your equipment or process) using *states*. For each state, you define conditional (Boolean) logic that evaluates the incoming inputs to detect significant events. When an event is detected, it can change the state or initiate custom-built or predefined actions using other AWS services. You can define additional events that initiate actions when entering or exiting a state and, optionally, when a condition is met.

In this tutorial, you send an Amazon SNS message as the action when the model enters or exits a certain state.

Monitor a device or process

If you're monitoring several devices or processes, you specify a field in each input that identifies the particular device or process the input comes from. (See the `key` field in `CreateDetectorModel`.) When a new device is identified (a new value is seen in the input field identified by the `key`), a detector is created. (Each detector is an instance of the detector model.) Then the new detector continues responding to inputs coming from that device until its detector model is updated or deleted.

If you're monitoring a single process (even if several devices or subprocesses are sending inputs), you don't specify a unique identifying `key` field. In this case, a single detector (instance) is created when the first input arrives.

Send messages as inputs to your detector model

There are several ways to send a message from a device or process as an input into an AWS IoT Events detector that don't require you to perform additional formatting on the message. In this tutorial, you use the AWS IoT console to write an AWS IoT Events action rule for the AWS IoT Core rules engine that forwards your message data into AWS IoT Events. To do this, you identify the input by name. Then you continue to use the AWS IoT console to generate some messages that are forwarded as inputs to AWS IoT Events.

How do you know which states you need in a detector model?

To determine what states your detector model should have, first decide what actions you can take. For example, if your automobile runs on gasoline, you look at the fuel gauge when you start a trip to see if you need to refuel. Here you have one action: tell the driver to "go get gas". Your detector model needs two states: "car doesn't need fuel", and "car does need fuel". In general, you want to define one state for each possible action, plus one more for when no action is required. This works even if the action itself is more complicated. For example, you might want to look up and include information on where to find the closest gas station, or the cheapest price, but you do this when you send the message to "go get gas".

To decide which state to enter next, you look at inputs. Inputs contain the information that you need to decide what state you should be in. To create an input, you select one or more fields in a message sent by your device or process that help you decide. In this example, you need one input that tells you the current fuel level ("percent full"). Maybe your car is sending you several different messages, each with several different fields. To create this input, you must select the message and the field that reports the current gas gauge level. The length of the trip you are about to take ("distance to destination") can be hardcoded to keep things simple; you can use your average trip length. You'll do some calculations based on the input (how many gallons does that percent full translate to? is the average trip length greater than the miles you can travel, given the gallons you have and your average "miles per gallon"). You perform these calculations and send messages in *events*.

So far you have two states and one input. You need an event in the first state that performs the calculations based on the input and decides whether to go to the second state. That is a transition event. (`transitionEvents` are in a state's `onInput` event list. *On* receiving an *input* in this first state, the *event* performs a *transition* to the second state, if the event's `condition` is met.) When you reach the second state, you send the message as soon as you enter the state. (You use an `onEnter` event. On entering the second state, this event sends the message. No need to wait

for another input to arrive.) There are other types of events, but that's all you need for a simple example.

The other types of events are `onExit` and `onInput`. As soon as an input is received, and the condition is met, an `onInput` event performs the specified actions. When an operation exits its current state, and the condition is met, the `onExit` event performs the specified actions.

Are you missing anything? Yes, how do you get back to the first "car doesn't need fuel" state? After you fill your gas tank, the input shows a full tank. In your second state you need a transition event back to the first state that happens when the input is received (in the second state's `onInput:` events). It should transition back to the first state if its calculations show you now have enough gas to get you where you want to go.

That's the basics. Some detector models get more complex by adding states that reflect important inputs, not just possible actions. For example, you might have three states in a detector model that keeps track of the temperature: a "normal" state, a "too hot" state, and a "potential problem" state. You transition to the potential problem state when the temperature rises above a certain level, but hasn't become too hot yet. You don't want to send an alarm unless it stays at this temperature for more than 15 minutes. If the temperature returns to normal before then, the detector transitions back to the normal state. If the timer expires, the detector transitions to the too hot state and sends an alarm, just to be cautious. You could do the same thing using variables and a more complex set of event conditions. But often it is easier to use another state to, in effect, store the results of your calculations.

How do you know if you need one instance of a detector or several?

To decide how many instances you need, ask yourself "What are you interested in knowing?" Let's say you want to know what the weather is like today. Is it raining (state)? Do you need to take an umbrella (action)? You can have a sensor that reports the temperature, another that reports the humidity, and others that report the barometric pressure, wind speed and direction, and precipitation. But you must monitor all these sensors together to determine the state of the weather (rain, snow, overcast, sunny) and the appropriate action to take (grab an umbrella or apply sunscreen). In spite of the number of sensors, you want one detector instance to monitor the state of the weather and inform you which action to take.

But if you're the weather forecaster for your region, you might have multiple instances of such sensor arrays, situated at different locations throughout the region. People at each location need to know what the weather is like in that location. In this case, you need multiple instances of your detector. The data reported by each sensor in each location must include a field that you have

designated as the `key` field. This field enables AWS IoT Events to create a detector instance for the area, and then to continue to route this information to that detector instance as it continues to arrive. No more ruined hair or sunburned noses!

Essentially, you need one detector instance if you have one situation (one process or one location) to monitor. If you have many situations (locations, processes) to monitor, you need multiple detector instances.

Create an AWS IoT Events detector for two states using CLI

In this example, we call the AWS IoT Events APIs using AWS CLI commands to create a detector that models two states of an engine: a normal state and an over-pressure condition.

When the measured pressure in the engine exceeds a certain threshold, the model transitions to the over-pressure state and sends an Amazon Simple Notification Service (Amazon SNS) message to alert a technician to the condition. When the pressure drops below the threshold for three consecutive pressure readings, the model returns to the normal state and sends another Amazon SNS message as a confirmation that the condition has cleared. We require three consecutive readings below the pressure threshold to eliminate possible stuttering of over-pressure/normal messages in case of a nonlinear recovery phase or a one-off anomalous recovery reading.

The following is an overview of the steps to create the detector.

Create *inputs*.

To monitor your devices and processes, they must have a way to get telemetry data into AWS IoT Events. This is done by sending messages as *inputs* to AWS IoT Events. You can do this in several ways:

- Use the BatchPutMessage operation. This method is easy but requires that your devices or processes are able to access the AWS IoT Events API through an SDK or the AWS CLI.
- In AWS IoT Core, write an AWS IoT Events action rule for the AWS IoT Core rules engine that forwards your message data into AWS IoT Events. This identifies the input by name. Use this method if your devices or processes can, or already are, sending messages through AWS IoT Core. This method generally requires less computing power from a device.
- In AWS IoT Analytics, use the CreateDataset operation to create a data set with `contentDeliveryRules` that specify the AWS IoT Events input, where data set contents are sent automatically. Use this method if you want to control your devices or processes based on data aggregated or analyzed in AWS IoT Analytics.

Before your devices can send data in this way, you must define one or more inputs. To do so, give each input a name and specify which fields in the incoming message data that the input monitors.

Create a detector model

Create a *detector model* (a model of your equipment or process) using *states*. For each state, define conditional (Boolean) logic that evaluates the incoming inputs to detect significant events. When an event is detected, it can change the state or initiate custom-built or predefined actions using other AWS services. You can define additional events that initiate actions when entering or exiting a state and, optionally, when a condition is met.

Monitor several devices or processes

If you're monitoring several devices or processes and you want to keep track of each of them separately, specify a field in each input that identifies the particular device or process the input comes from. See the `key` field in `CreateDetectorModel`. When a new device is identified (a new value is seen in the input field identified by the `key`), a detector instance is created. The new detector instance continues to respond to inputs coming from that particular device until its detector model is updated or deleted. You have as many unique detectors (instances) as there are unique values in input `key` fields.

Monitor a single device or process

If you're monitoring a single process (even if several devices or subprocesses are sending inputs), you don't specify a unique identifying `key` field. In this case, a single detector (instance) is created when the first input arrives. For example, you might have temperature sensors in each room of a house, but only one HVAC unit to heat or cool the entire house. So you can only control this as a single process, even if each room occupant wants their vote (input) to prevail.

Send messages from your devices or processes as inputs to your detector model

We described the several ways to send a message from a device or process as an input into an AWS IoT Events detector in *inputs*. After you created the inputs and build the detector model, you're ready to start sending data.

> **Note**
>
> When you create a detector model, or update an existing one, it takes several minutes before the new or updated detector model begins receiving messages and creating

detectors (instances). If the detector model is updated, during this time you might continue to see behavior based on the previous version.

Topics

- Create an AWS IoT Events input to capture device data
- Create a detector model to represent device states in AWS IoT Events
- Send messages as inputs to a detector in AWS IoT Events

Create an AWS IoT Events input to capture device data

When setting up inputs for AWS IoT Events, you can leverage the AWS CLI to define how your devices communicate sensor data. For example, if your devices send JSON-formatted messages with motor identifiers and sensor readings, you can capture this data by creating an input that maps specific attributes from the messages, such as the pressure and the motor ID. The process starts by defining an input in a JSON file, specifying the relevant data points, and using the AWS CLI to register the input for AWS IoT Events. This enables AWS IoT to monitor and respond to critical conditions based on real-time sensor data.

As an example, suppose your devices send messages with the following format.

```
{
  "motorid": "Fulton-A32",
  "sensorData": {
    "pressure": 23,
    "temperature": 47
  }
}
```

You can create an input to capture the `pressure` data and the `motorid` (that identifies the specific device that sent the message) using the following AWS CLI command.

```
aws iotevents create-input  --cli-input-json file://pressureInput.json
```

The file `pressureInput.json` contains the following.

```
{
```

```
  "inputName": "PressureInput",
  "inputDescription": "Pressure readings from a motor",
  "inputDefinition": {
    "attributes": [
      { "jsonPath": "sensorData.pressure" },
      { "jsonPath": "motorid" }
    ]
  }
}
```

When you create your own inputs, remember to first collect example messages as JSON files from your devices or processes. You can use them to create an input from the console or the CLI.

Create a detector model to represent device states in AWS IoT Events

In Create an AWS IoT Events input to capture device data, you created an `input` based on a message that reports pressure data from a motor. To continue with the example, here is a detector model that responds to an over-pressure event in a motor.

You create two states: "`Normal`", and "`Dangerous`". Each detector (instance) enters the "`Normal`" state when it's created. The instance is created when an input with a unique value for the `key` "`motorid`" arrives.

If the detector instance receives a pressure reading of 70 or greater, it enters the "`Dangerous`" state and sends an Amazon SNS message as a warning. If the pressure readings return to normal (less than 70) for three consecutive inputs, the detector returns to the "`Normal`" state and sends another Amazon SNS message as an all clear.

This example detector model assumes you have created two Amazon SNS topics whose Amazon Resource Names (ARNs) are shown in the definition as `"targetArn": "arn:aws:sns:us-east-1:123456789012:underPressureAction"` and `"targetArn": "arn:aws:sns:us-east-1:123456789012:pressureClearedAction"`.

For more information, see the Amazon Simple Notification Service Developer Guide and, more specifically, the documentation of the CreateTopic operation in the *Amazon Simple Notification Service API Reference*.

This example also assumes you have created an AWS Identity and Access Management (IAM) role with appropriate permissions. The ARN of this role is shown in the detector model definition as `"roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole"`. Follow the steps in

Setting up permissions for AWS IoT Events to create this role and copy the ARN of the role in the appropriate place in the detector model definition.

You can create the detector model using the following AWS CLI command.

```
aws iotevents create-detector-model  --cli-input-json file://motorDetectorModel.json
```

The file "motorDetectorModel.json" contains the following.

```
{
  "detectorModelName": "motorDetectorModel",
  "detectorModelDefinition": {
    "states": [
      {
        "stateName": "Normal",
        "onEnter": {
          "events": [
            {
              "eventName": "init",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "pressureThresholdBreached",
                    "value": "0"
                  }
                }
              ]
            }
          ]
        },
        "onInput": {
          "transitionEvents": [
            {
              "eventName": "Overpressurized",
              "condition": "$input.PressureInput.sensorData.pressure > 70",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "pressureThresholdBreached",
                    "value": "$variable.pressureThresholdBreached + 3"
                  }
```

```
                }
              ],
              "nextState": "Dangerous"
            }
          ]
        }
      },
      {
        "stateName": "Dangerous",
        "onEnter": {
          "events": [
            {
              "eventName": "Pressure Threshold Breached",
              "condition": "$variable.pressureThresholdBreached > 1",
              "actions": [
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-
east-1:123456789012:underPressureAction"
                  }
                }
              ]
            }
          ]
        },
        "onInput": {
          "events": [
            {
              "eventName": "Overpressurized",
              "condition": "$input.PressureInput.sensorData.pressure > 70",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "pressureThresholdBreached",
                    "value": "3"
                  }
                }
              ]
            },
            {
              "eventName": "Pressure Okay",
              "condition": "$input.PressureInput.sensorData.pressure <= 70",
              "actions": [
                {
```

```
                  "setVariable": {
                    "variableName": "pressureThresholdBreached",
                    "value": "$variable.pressureThresholdBreached - 1"
                  }
                }
              ]
            }
          ],
          "transitionEvents": [
            {
              "eventName": "BackToNormal",
              "condition": "$input.PressureInput.sensorData.pressure <= 70 &&
 $variable.pressureThresholdBreached <= 1",
              "nextState": "Normal"
            }
          ]
        },
        "onExit": {
          "events": [
            {
              "eventName": "Normal Pressure Restored",
              "condition": "true",
              "actions": [
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-
east-1:123456789012:pressureClearedAction"
                  }
                }
              ]
            }
          ]
        }
      }
    ],
    "initialStateName": "Normal"
  },
  "key" : "motorid",
  "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole"
}
```

Send messages as inputs to a detector in AWS IoT Events

You have now defined an input that identifies the important fields in messages sent from a device (see Create an AWS IoT Events input to capture device data). In the previous section, you created a `detector model` that responds to an over-pressure event in a motor (see Create a detector model to represent device states in AWS IoT Events).

To complete the example, send messages from a device (in this case a computer with the AWS CLI installed) as inputs to the detector.

> **Note**
>
> When you create a detector model or update an existing one, it takes several minutes before the new or updated detector model begins to receive messages and create detectors (instances). If you update the detector model, during this time you might continue to see behavior based on the previous version.

Use the following AWS CLI command to send a message with data that breaches the threshold.

```
aws iotevents-data batch-put-message --cli-input-json file://highPressureMessage.json
 --cli-binary-format raw-in-base64-out
```

The file "`highPressureMessage.json`" contains the following.

```
{
  "messages": [
    {
      "messageId": "00001",
      "inputName": "PressureInput",
      "payload": "{\"motorid\": \"Fulton-A32\", \"sensorData\": {\"pressure\": 80,
 \"temperature\": 39} }"
    }
  ]
}
```

You must change the `messageId` in each message sent. If you don't change it, the AWS IoT Events system deduplicates the messages. AWS IoT Events ignores a message if it has the same `messageID` as another message that was sent within the last five minutes.

At this point, a detector (instance) is created to monitor events for the motor "Fulton-A32". This detector enters the "Normal" state when it's created. But because we sent a pressure value above the threshold, it immediately transitions to the "Dangerous" state. As it does so, the detector sends a message to the Amazon SNS endpoint whose ARN is arn:aws:sns:us-east-1:123456789012:underPressureAction.

Run the following AWS CLI command to send a message with data that is beneath the pressure threshold.

```
aws iotevents-data batch-put-message --cli-input-json file://normalPressureMessage.json
 --cli-binary-format raw-in-base64-out
```

The file normalPressureMessage.json contains the following.

```
{
  "messages": [
    {
      "messageId": "00002",
      "inputName": "PressureInput",
      "payload": "{\"motorid\": \"Fulton-A32\", \"sensorData\": {\"pressure\": 60,
 \"temperature\": 29} }"
    }
  ]
}
```

You must change the messageId in the file each time you invoke the BatchPutMessage command within a five minute period. Send the message two more times. After the message is sent three times, the detector (instance) for the motor "Fulton-A32" sends a message to the Amazon SNS endpoint "arn:aws:sns:us-east-1:123456789012:pressureClearedAction" and reenters the "Normal" state.

Note

You can send multiple messages at one time with BatchPutMessage. However, the order in which these messages are processed isn't guaranteed. To guarantee messages (inputs) are processed in order, send them one at a time and wait for a successful response each time the API is called.

The following are example SNS message payloads created by the detector model example described in this section.

on event "Pressure Threshold Breached"


```
IoT> {
  "eventTime":1558129816420,
  "payload":{
    "actionExecutionId":"5d7444df-a655-3587-a609-dbd7a0f55267",
    "detector":{
      "detectorModelName":"motorDetectorModel",
      "keyValue":"Fulton-A32",
      "detectorModelVersion":"1"
    },
    "eventTriggerDetails":{
      "inputName":"PressureInput",
      "messageId":"00001",
      "triggerType":"Message"
    },
    "state":{
      "stateName":"Dangerous",
      "variables":{
        "pressureThresholdBreached":3
      },
      "timers":{}
    }
  },
  "eventName":"Pressure Threshold Breached"
}
```


on event "Normal Pressure Restored"


```
IoT> {
  "eventTime":1558129925568,
  "payload":{
    "actionExecutionId":"7e25fd38-2533-303d-899f-c979792a12cb",
    "detector":{
      "detectorModelName":"motorDetectorModel",
      "keyValue":"Fulton-A32",
      "detectorModelVersion":"1"
    },
    "eventTriggerDetails":{
      "inputName":"PressureInput",
```

```
      "messageId":"00004",
      "triggerType":"Message"
    },
    "state":{
      "stateName":"Dangerous",
      "variables":{
        "pressureThresholdBreached":0
      },
      "timers":{}
    }
  },
  "eventName":"Normal Pressure Restored"
}
```

If you have defined any timers, their current state is also shown in the SNS message payloads.

The message payloads contain information about the state of the detector (instance) at the time the message was sent (that is, at the time the SNS action was run). You can use the https://docs.aws.amazon.com/iotevents/latest/apireference/API_iotevents-data_DescribeDetector.html operation to get similar information about the state of the detector.

AWS IoT Events detector model restrictions and limitations

The following things are important to consider when creating a detector model.

How to use the `actions` field

The `actions` field is a list of objects. You can have more than one object, but only one action is allowed in each object.

Example

```
            "actions": [
              {
                "setVariable": {
                  "variableName": "pressureThresholdBreached",
                  "value": "$variable.pressureThresholdBreached - 1"
                }
              }
              {
                "setVariable": {
```

```
            "variableName": "temperatureIsTooHigh",
            "value": "$variable.temperatureIsTooHigh - 1"
          }
        }
      ]
```

How to use the `condition` field

The `condition` is required for `transitionEvents` and is optional in other cases.

If the `condition` field isn't present, it's equivalent to `"condition": true`.

The result of the evaluation of a condition expression should be a Boolean value. If the result isn't a Boolean value, it's equivalent to `false` and won't initiate the `actions` or transition to the `nextState` specified in the event.

Availability of variable values

By default, if the value of a variable is set in an event, its new value isn't available or used to evaluate conditions in other events in the same group. The new value isn't available or used in an event condition in the same `onInput`, `onEnter` or `onExit` field.

Set the `evaluationMethod` parameter in the detector model definition to change this behavior. When the `evaluationMethod` is set to `SERIAL`, variables are updated and event conditions are evaluated in the order that the events are defined. Otherwise, when the `evaluationMethod` is set to `BATCH` or defaults to it, variables within a state are updated and events within a state are performed only after all event conditions are evaluated.

In the `"Dangerous"` state, in the `onInput` field, `"$variable.pressureThresholdBreached"` is decremented by one in the `"Pressure Okay"` event when the condition is met (when the current input has pressure less than or equal to 70).

```
{
  "eventName": "Pressure Okay",
  "condition": "$input.PressureInput.sensorData.pressure <= 70",
  "actions": [
    {
      "setVariable": {
        "variableName": "pressureThresholdBreached",
        "value": "$variable.pressureThresholdBreached - 1"
      }
```

```
            }
          ]
        }
```

The detector should transition back to the `"Normal"` state when `"$variable.pressureThresholdBreached"` reaches 0 (that is, when the detector has received three contiguous pressure readings less than or equal to 70). The `"BackToNormal"` event in `transitionEvents` must test that `"$variable.pressureThresholdBreached"` is less than or equal to 1 (not 0), and also verify again that the current value given by `"$input.PressureInput.sensorData.pressure"` is less than or equal to 70.

```
        "transitionEvents": [
          {
            "eventName": "BackToNormal",
            "condition": "$input.PressureInput.sensorData.pressure <= 70 &&
$variable.pressureThresholdBreached <= 1",
            "nextState": "Normal"
          }
        ]
```

Otherwise, if the condition tests for only the value of the variable, two normal readings followed by an over-pressure reading would fulfill the condition and transition back to the `"Normal"` state. The condition is looking at the value that `"$variable.pressureThresholdBreached"` was given during the previous time an input was processed. The value of the variable is reset to 3 in the `"Overpressurized"` event, but remember that this new value is not yet available to any `condition`.

By default, every time a control enters the `onInput` field, a `condition` can only see the value of a variable as it was at the start of processing the input, before it's changed by any actions specified in `onInput`. The same is true for `onEnter` and `onExit`. Any change made to a variable when we enter or exit the state isn't available to other conditions specified in the same `onEnter` or `onExit` fields.

Latency when updating a detector model

If you update, delete, and recreate a detector model (see UpdateDetectorModel), there is some delay before all spawned detectors (instances) are deleted and the new model is used to recreate the detectors. They are recreated after the new detector model takes effect and new inputs arrive. During this time inputs might continue to be processed by the detectors spawned

by the previous version of the detector model. During this period, you might continue to receive alerts defined by the previous detector model.

Spaces in input keys

Spaces are allowed in input keys, but references to the key must be enclosed in backticks, both in the definition of the input attribute and when the value of the key is referenced in an expression. For example, given a message payload like the following:

```
{
  "motor id": "A32",
  "sensorData" {
    "motor pressure": 56,
    "motor temperature": 39
  }
}
```

Use the following to define the input.

```
{
  "inputName": "PressureInput",
  "inputDescription": "Pressure readings from a motor",
  "inputDefinition": {
    "attributes": [
      { "jsonPath": "sensorData.`motor pressure`" },
      { "jsonPath": "`motor id`" }
    ]
  }
}
```

In a conditional expression, you must refer to the value of any such key using backticks also.

```
$input.PressureInput.sensorData.`motor pressure`
```

A commented example: HVAC temperature control with AWS IoT Events

Some of the following example JSON files have comments inline, which makes them invalid JSON. Complete versions of these examples, without comments, are available at Example: Using HVAC temperature control with AWS IoT Events.

This example implements a thermostat control model that gives you the ability to do the following.

- Define just one detector model that can be used to monitor and control multiple areas. A detector instance is created for each area.
- Ingest temperature data from multiple sensors in each control area.
- Change the temperature set point for an area.
- Set operational parameters for each area and reset these parameters while the instance is in use.
- Dynamically add or delete sensors from an area.
- Specify a minimum runtime to protect heating and cooling units.
- Reject anomalous sensor readings.
- Define emergency set points that immediately engage heating or cooling if any one sensor reports a temperature above or below a given threshold.
- Report anomalous readings and temperature spikes.

Topics

- Input definitions for detector models in AWS IoT Events
- Create an AWS IoT Events detector model definition
- Use BatchUpdateDetector to update an AWS IoT Events detector model
- Use BatchPutMessage for inputs in AWS IoT Events
- Ingest MQTT messages in AWS IoT Events
- Generate Amazon SNS messages in AWS IoT Events
- Configure the DescribeDetector API in AWS IoT Events
- Use the AWS IoT Core rules engine for AWS IoT Events

Input definitions for detector models in AWS IoT Events

We want to create one detector model that we can use to monitor and control the temperature in several different areas. Each area can have several sensors that report the temperature. We assume each area is served by one heating unit and one cooling unit that can be turned on or off to control the temperature in the area. Each area is controlled by one detector instance.

Because the different areas we monitor and control might have different characteristics that demand different control parameters, we define the `'seedTemperatureInput'` to provide those

parameters for each area. When we send one of these input messages to AWS IoT Events, a new detector model instance is created that has the parameters we want to use in that area. Here's the definition of that input.

CLI command:

```
aws iotevents create-input --cli-input-json file://seedInput.json
```

File: `seedInput.json`

```
{
  "inputName": "seedTemperatureInput",
  "inputDescription": "Temperature seed values.",
  "inputDefinition": {
    "attributes": [
      { "jsonPath": "areaId" },
      { "jsonPath": "desiredTemperature" },
      { "jsonPath": "allowedError" },
      { "jsonPath": "rangeHigh" },
      { "jsonPath": "rangeLow" },
      { "jsonPath": "anomalousHigh" },
      { "jsonPath": "anomalousLow" },
      { "jsonPath": "sensorCount" },
      { "jsonPath": "noDelay" }
    ]
  }
}
```

Response:

```
{
    "inputConfiguration": {
        "status": "ACTIVE",
        "inputArn": "arn:aws:iotevents:us-west-2:123456789012:input/
seedTemperatureInput",
        "lastUpdateTime": 1557519620.736,
        "creationTime": 1557519620.736,
        "inputName": "seedTemperatureInput",
        "inputDescription": "Temperature seed values."
    }
}
```

Notes

- A new detector instance is created for each unique `'areaId'` received in any message. See the `'key'` field in the `'areaDetectorModel'` definition.
- The average temperature can vary from the `'desiredTemperature'` by the `'allowedError'` before the heating or cooling units are activated for the area.
- If any sensor reports a temperature above the `'rangeHigh'`, the detector reports a spike and immediately starts the cooling unit.
- If any sensor reports a temperature below the `'rangeLow'`, the detector reports a spike and immediately starts the heating unit.
- If any sensor reports a temperature above the `'anomalousHigh'` or below the `'anomalousLow'`, the detector reports an anomalous sensor reading, but ignores the reported temperature reading.
- The `'sensorCount'` tells the detector how many sensors are reporting for the area. The detector calculates the average temperature in the area by giving the appropriate weight factor to each temperature reading it receives. Because of this, the detector won't have to keep track of what each sensor reports, and the number of sensors can be changed dynamically, as needed. However, if an individual sensor goes offline, the detector won't know this or make allowances for it. We recommend that you create another detector model specifically for monitoring the connection status of each sensor. Having two complementary detector models simplifies the design of both.
- The `'noDelay'` value can be `true` or `false`. After a heating or cooling unit is turned on, it should remain on for a certain minimum time to protect the integrity of the unit and lengthen its operating life. If `'noDelay'` is set to `false`, the detector instance enforces a delay before it turns off the cooling and heating units, to ensure that they are run for the minimum time. The number of seconds of delay has been hardcoded in the detector model definition because we are unable to use a variable value to set a timer.

The `'temperatureInput'` is used to transmit sensor data to a detector instance.

CLI command:

```
aws iotevents create-input --cli-input-json file://temperatureInput.json
```

File: `temperatureInput.json`

```
{
  "inputName": "temperatureInput",
  "inputDescription": "Temperature sensor unit data.",
  "inputDefinition": {
    "attributes": [
      { "jsonPath": "sensorId" },
      { "jsonPath": "areaId" },
      { "jsonPath": "sensorData.temperature" }
    ]
  }
}
```

Response:

```
{
    "inputConfiguration": {
        "status": "ACTIVE",
        "inputArn": "arn:aws:iotevents:us-west-2:123456789012:input/temperatureInput",
        "lastUpdateTime": 1557519707.399,
        "creationTime": 1557519707.399,
        "inputName": "temperatureInput",
        "inputDescription": "Temperature sensor unit data."
    }
}
```

Notes

- The `'sensorId'` isn't used by an example detector instance to control or monitor a sensor directly. It's automatically passed into notifications sent by the detector instance. From there, it can be used to identify the sensors that are failing (for example, a sensor that regularly sends anomalous readings might be about to fail), or that have gone offline (when it's used as an input to an additional detector model that monitors the device's heartbeat). The `'sensorId'` can also help identify warm or cold zones in an area if its readings regularly differ from the average.
- The `'areaId'` is used to route the sensor's data to the appropriate detector instance. A detector instance is created for each unique `'areaId'` received in any message. See the `'key'` field in the `'areaDetectorModel'` definition.

Create an AWS IoT Events detector model definition

The `'areaDetectorModel'` example has comments inline.

CLI command:

```
aws iotevents create-detector-model --cli-input-json file://areaDetectorModel.json
```

File: `areaDetectorModel.json`

```
{
  "detectorModelName": "areaDetectorModel",
  "detectorModelDefinition": {
    "states": [
      {
        "stateName": "start",
        // In the 'start' state we set up the operation parameters of the new detector
 instance.
        //   We get here when the first input message arrives. If that is a
 'seedTemperatureInput'
        //   message, we save the operation parameters, then transition to the 'idle'
 state. If
        //   the first message is a 'temperatureInput', we wait here until we get a
        //   'seedTemperatureInput' input to ensure our operation parameters are set.
 We can
        //   also reenter this state using the 'BatchUpdateDetector' API. This enables
 us to
        //   reset the operation parameters without needing to delete the detector
 instance.
        "onEnter": {
          "events": [
            {
              "eventName": "prepare",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    // initialize 'sensorId' to an invalid value (0) until an actual
 sensor reading
                    //   arrives
                    "variableName": "sensorId",
                    "value": "0"
```

```
                }
              },
              {
                "setVariable": {
                  // initialize 'reportedTemperature' to an invalid value (0.1) until
an actual
                  //   sensor reading arrives
                  "variableName": "reportedTemperature",
                  "value": "0.1"
                }
              },
              {
                "setVariable": {
                  // When using 'BatchUpdateDetector' to re-enter this state, this
variable should
                  //   be set to true.
                  "variableName": "resetMe",
                  "value": "false"
                }
              }
            ]
          }
        ]
      },
      "onInput": {
        "transitionEvents": [
          {
            "eventName": "initialize",
            "condition": "$input.seedTemperatureInput.sensorCount > 0",
            // When a 'seedTemperatureInput' message with a valid 'sensorCount' is
received,
            //   we use it to set the operational parameters for the area to be
monitored.
            "actions": [
              {
                "setVariable": {
                  "variableName": "rangeHigh",
                  "value": "$input.seedTemperatureInput.rangeHigh"
                }
              },
              {
                "setVariable": {
                  "variableName": "rangeLow",
                  "value": "$input.seedTemperatureInput.rangeLow"
```

```
        }
      },
      {
        "setVariable": {
          "variableName": "desiredTemperature",
          "value": "$input.seedTemperatureInput.desiredTemperature"
        }
      },
      {
        "setVariable": {
          // Assume we're at the desired temperature when we start.
          "variableName": "averageTemperature",
          "value": "$input.seedTemperatureInput.desiredTemperature"
        }
      },
      {
        "setVariable": {
          "variableName": "allowedError",
          "value": "$input.seedTemperatureInput.allowedError"
        }
      },
      {
        "setVariable": {
          "variableName": "anomalousHigh",
          "value": "$input.seedTemperatureInput.anomalousHigh"
        }
      },
      {
        "setVariable": {
          "variableName": "anomalousLow",
          "value": "$input.seedTemperatureInput.anomalousLow"
        }
      },
      {
        "setVariable": {
          "variableName": "sensorCount",
          "value": "$input.seedTemperatureInput.sensorCount"
        }
      },
      {
        "setVariable": {
          "variableName": "noDelay",
          "value": "$input.seedTemperatureInput.noDelay == true"
        }
```

```
              }
            ],
            "nextState": "idle"
          },
          {
            "eventName": "reset",
            "condition": "($variable.resetMe == true) &&
($input.temperatureInput.sensorData.temperature < $variable.anomalousHigh &&
$input.temperatureInput.sensorData.temperature > $variable.anomalousLow)",
            // This event is triggered if we have reentered the 'start' state using
the
            //   'BatchUpdateDetector' API with 'resetMe' set to true. When we
reenter using
            //   'BatchUpdateDetector' we do not automatically continue to the 'idle'
state, but
            //   wait in 'start' until the next input message arrives. This event
enables us to
            //   transition to 'idle' on the next valid 'temperatureInput' message
that arrives.
            "actions": [
              {
                "setVariable": {
                  "variableName": "averageTemperature",
                  "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                }
              }
            ],
            "nextState": "idle"
          }
        ]
      },
      "onExit": {
        "events": [
          {
            "eventName": "resetHeatCool",
            "condition": "true",
            // Make sure the heating and cooling units are off before entering
'idle'.
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOff"
                }
```

```
                },
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOff"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Heating/Off"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Cooling/Off"
                  }
                }
              ]
            }
          ]
        }
      },


      {
        "stateName": "idle",
        "onInput": {
          "events": [
            {
              "eventName": "whatWasInput",
              "condition": "true",
              // By storing the 'sensorId' and the 'temperature' in variables, we make
them
              //   available in any messages we send out to report anomalies, spikes,
or just
              //   if needed for debugging.
              "actions": [
                {
                  "setVariable": {
                    "variableName": "sensorId",
                    "value": "$input.temperatureInput.sensorId"
                  }
                },
                {
                  "setVariable": {
```

```
                    "variableName": "reportedTemperature",
                    "value": "$input.temperatureInput.sensorData.temperature"
                  }
                }
              ]
            },
            {
              "eventName": "changeDesired",
              "condition": "$input.seedTemperatureInput.desiredTemperature !=
$variable.desiredTemperature",
              // This event enables us to change the desired temperature at any time by
sending a
              //   'seedTemperatureInput' message. But note that other operational
parameters are not
              //   read or changed.
              "actions": [
                {
                  "setVariable": {
                    "variableName": "desiredTemperature",
                    "value": "$input.seedTemperatureInput.desiredTemperature"
                  }
                }
              ]
            },
            {
              "eventName": "calculateAverage",
              "condition": "$input.temperatureInput.sensorData.temperature <
$variable.anomalousHigh && $input.temperatureInput.sensorData.temperature >
$variable.anomalousLow",
              // If a valid temperature reading arrives, we use it to update the
average temperature.
              //   For simplicity, we assume our sensors will be sending updates at
about the same rate,
              //   so we can calculate an approximate average by giving equal weight to
each reading we receive.
              "actions": [
                {
                  "setVariable": {
                    "variableName": "averageTemperature",
                    "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                  }
                }
              ]
```

```
            }
          ],
          "transitionEvents": [
            {
              "eventName": "anomalousInputArrived",
              "condition": "$input.temperatureInput.sensorData.temperature >=
$variable.anomalousHigh || $input.temperatureInput.sensorData.temperature <=
$variable.anomalousLow",
              // When an anomalous reading arrives, send an MQTT message, but stay in
the current state.
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/anomaly"
                  }
                }
              ],
              "nextState": "idle"
            },

            {
              "eventName": "highTemperatureSpike",
              "condition": "$input.temperatureInput.sensorData.temperature >
$variable.rangeHigh",
              // When even a single temperature reading arrives that is above the
'rangeHigh', take
              //   emergency action to begin cooling, and report a high temperature
spike.
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/spike"
                  }
                },
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOn"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Cooling/On"
                  }
                },
```

```
                {
                  "setVariable": {
                    // This is necessary because we want to set a timer to delay the
shutoff
                    //   of a cooling/heating unit, but we only want to set the timer
when we
                    //   enter that new state initially.
                    "variableName": "enteringNewState",
                    "value": "true"
                  }
                }
              ],
              "nextState": "cooling"
            },

            {
              "eventName": "lowTemperatureSpike",
              "condition": "$input.temperatureInput.sensorData.temperature <
$variable.rangeLow",
              // When even a single temperature reading arrives that is below the
'rangeLow', take
              //   emergency action to begin heating, and report a low-temperature
spike.
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/spike"
                  }
                },
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOn"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Heating/On"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "enteringNewState",
                    "value": "true"
                  }
```

```
            }
          ],
          "nextState": "heating"
        },

        {
          "eventName": "highTemperatureThreshold",
          "condition": "((((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) >
($variable.desiredTemperature + $variable.allowedError))",
          // When the average temperature is above the desired temperature plus the
allowed error factor,
          //   it is time to start cooling. Note that we calculate the average
temperature here again
          //   because the value stored in the 'averageTemperature' variable is not
yet available for use
          //   in our condition.
          "actions": [
            {
              "sns": {
                "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOn"
              }
            },
            {
              "iotTopicPublish": {
                "mqttTopic": "hvac/Cooling/On"
              }
            },
            {
              "setVariable": {
                "variableName": "enteringNewState",
                "value": "true"
              }
            }
          ],
          "nextState": "cooling"
        },

        {
          "eventName": "lowTemperatureThreshold",
          "condition": "((((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) <
($variable.desiredTemperature - $variable.allowedError))",
```

```
            // When the average temperature is below the desired temperature minus
the allowed error factor,
            //   it is time to start heating. Note that we calculate the average
temperature here again
            //   because the value stored in the 'averageTemperature' variable is not
yet available for use
            //   in our condition.
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOn"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Heating/On"
                }
              },
              {
                "setVariable": {
                  "variableName": "enteringNewState",
                  "value": "true"
                }
              }
            ],
            "nextState": "heating"
          }
        ]
      }
    },


    {
      "stateName": "cooling",
      "onEnter": {
        "events": [
          {
            "eventName": "delay",
            "condition": "!$variable.noDelay && $variable.enteringNewState",
            // If the operational parameters specify that there should be a minimum
time that the
            //   heating and cooling units should be run before being shut off again,
we set
            //   a timer to ensure the proper operation here.
```

```
              "actions": [
                {
                  "setTimer": {
                    "timerName": "coolingTimer",
                    "seconds": 180
                  }
                },
                {
                  "setVariable": {
                    // We use this 'goodToGo' variable to store the status of the timer
expiration
                    //   for use in conditions that also use input variable values. If
                    //   'timeout()' is used in such mixed conditionals, its value is
lost.
                    "variableName": "goodToGo",
                    "value": "false"
                  }
                }
              ]
            },
            {
              "eventName": "dontDelay",
              "condition": "$variable.noDelay == true",
              // If the heating/cooling unit shutoff delay is not used, no need to
wait.
              "actions": [
                {
                  "setVariable": {
                    "variableName": "goodToGo",
                    "value": "true"
                  }
                }
              ]
            },
            {
              "eventName": "beenHere",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "enteringNewState",
                    "value": "false"
                  }
                }
```

```
            ]
          }
        ]
      },

      "onInput": {
        "events": [
          // These are events that occur when an input is received (if the condition
is
          //   satisfied), but don't cause a transition to another state.
          {
            "eventName": "whatWasInput",
            "condition": "true",
            "actions": [
              {
                "setVariable": {
                  "variableName": "sensorId",
                  "value": "$input.temperatureInput.sensorId"
                }
              },
              {
                "setVariable": {
                  "variableName": "reportedTemperature",
                  "value": "$input.temperatureInput.sensorData.temperature"
                }
              }
            ]
          },
          {
            "eventName": "changeDesired",
            "condition": "$input.seedTemperatureInput.desiredTemperature !=
$variable.desiredTemperature",
            "actions": [
              {
                "setVariable": {
                  "variableName": "desiredTemperature",
                  "value": "$input.seedTemperatureInput.desiredTemperature"
                }
              }
            ]
          },
          {
            "eventName": "calculateAverage",
```

```
            "condition": "$input.temperatureInput.sensorData.temperature <
$variable.anomalousHigh && $input.temperatureInput.sensorData.temperature >
$variable.anomalousLow",
            "actions": [
              {
                "setVariable": {
                  "variableName": "averageTemperature",
                  "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                }
              }
            ]
          },
          {
            "eventName": "areWeThereYet",
            "condition": "(timeout(\"coolingTimer\"))",
            "actions": [
              {
                "setVariable": {
                  "variableName": "goodToGo",
                  "value": "true"
                }
              }
            ]
          }
        ],
        "transitionEvents": [
          // Note that some tests of temperature values (for example, the test for an
anomalous value)
          //   must be placed here in the 'transitionEvents' because they work
together with the tests
          //   in the other conditions to ensure that we implement the proper
"if..elseif..else" logic.
          //   But each transition event must have a destination state ('nextState'),
and even if that
          //   is actually the current state, the "onEnter" events for this state
will be executed again.
          //   This is the reason for the 'enteringNewState' variable and related.
          {
            "eventName": "anomalousInputArrived",
            "condition": "$input.temperatureInput.sensorData.temperature >=
$variable.anomalousHigh || $input.temperatureInput.sensorData.temperature <=
$variable.anomalousLow",
            "actions": [
```

```
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/anomaly"
                  }
                }
              ],
              "nextState": "cooling"
            },

            {
              "eventName": "highTemperatureSpike",
              "condition": "$input.temperatureInput.sensorData.temperature >
$variable.rangeHigh",
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/spike"
                  }
                }
              ],
              "nextState": "cooling"
            },

            {
              "eventName": "lowTemperatureSpike",
              "condition": "$input.temperatureInput.sensorData.temperature <
$variable.rangeLow",
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/spike"
                  }
                },
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOff"
                  }
                },
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOn"
                  }
                },
                {
```

```
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Cooling/Off"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Heating/On"
                }
              },
              {
                "setVariable": {
                  "variableName": "enteringNewState",
                  "value": "true"
                }
              }
            ],
            "nextState": "heating"
          },

          {
            "eventName": "desiredTemperature",
            "condition": "((((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) <=
($variable.desiredTemperature - $variable.allowedError)) && $variable.goodToGo ==
true",
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOff"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Cooling/Off"
                }
              }
            ],
            "nextState": "idle"
          }
        ]
      }
    },
```

```
      {
        "stateName": "heating",
        "onEnter": {
          "events": [
            {
              "eventName": "delay",
              "condition": "!$variable.noDelay && $variable.enteringNewState",
              "actions": [
                {
                  "setTimer": {
                    "timerName": "heatingTimer",
                    "seconds": 120
                  }
                },
                {
                  "setVariable": {
                    "variableName": "goodToGo",
                    "value": "false"
                  }
                }
              ]
            },
            {
              "eventName": "dontDelay",
              "condition": "$variable.noDelay == true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "goodToGo",
                    "value": "true"
                  }
                }
              ]
            },
            {
              "eventName": "beenHere",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "enteringNewState",
                    "value": "false"
                  }
                }
```

```
            ]
          }
        ]
      },

      "onInput": {
        "events": [
          {
            "eventName": "whatWasInput",
            "condition": "true",
            "actions": [
              {
                "setVariable": {
                  "variableName": "sensorId",
                  "value": "$input.temperatureInput.sensorId"
                }
              },
              {
                "setVariable": {
                  "variableName": "reportedTemperature",
                  "value": "$input.temperatureInput.sensorData.temperature"
                }
              }
            ]
          },
          {
            "eventName": "changeDesired",
            "condition": "$input.seedTemperatureInput.desiredTemperature !=
$variable.desiredTemperature",
            "actions": [
              {
                "setVariable": {
                  "variableName": "desiredTemperature",
                  "value": "$input.seedTemperatureInput.desiredTemperature"
                }
              }
            ]
          },
          {
            "eventName": "calculateAverage",
            "condition": "$input.temperatureInput.sensorData.temperature <
$variable.anomalousHigh && $input.temperatureInput.sensorData.temperature >
$variable.anomalousLow",
            "actions": [
```

```
              {
                "setVariable": {
                  "variableName": "averageTemperature",
                  "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                }
              }
            ]
          },
          {
            "eventName": "areWeThereYet",
            "condition": "(timeout(\"heatingTimer\"))",
            "actions": [
              {
                "setVariable": {
                  "variableName": "goodToGo",
                  "value": "true"
                }
              }
            ]
          }
        ],
        "transitionEvents": [
          {
            "eventName": "anomalousInputArrived",
            "condition": "$input.temperatureInput.sensorData.temperature >=
$variable.anomalousHigh || $input.temperatureInput.sensorData.temperature <=
$variable.anomalousLow",
            "actions": [
              {
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/anomaly"
                }
              }
            ],
            "nextState": "heating"
          },

          {
            "eventName": "highTemperatureSpike",
            "condition": "$input.temperatureInput.sensorData.temperature >
$variable.rangeHigh",
            "actions": [
              {
```

```
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/spike"
                }
              },
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOff"
                }
              },
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOn"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Heating/Off"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Cooling/On"
                }
              },
              {
                "setVariable": {
                  "variableName": "enteringNewState",
                  "value": "true"
                }
              }
            ],
            "nextState": "cooling"
          },

          {
            "eventName": "lowTemperatureSpike",
            "condition": "$input.temperatureInput.sensorData.temperature <
$variable.rangeLow",
            "actions": [
              {
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/spike"
                }
              }
```

```
              ],
              "nextState": "heating"
            },

            {
              "eventName": "desiredTemperature",
              "condition": "((((($variable.averageTemperature * ($variable.sensorCount
 - 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) >=
 ($variable.desiredTemperature + $variable.allowedError)) && $variable.goodToGo ==
 true",
              "actions": [
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOff"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Heating/Off"
                  }
                }
              ],
              "nextState": "idle"
            }
          ]
        }
      }

    ],

    "initialStateName": "start"
  },
  "key": "areaId",
  "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole"
}
```

Response:

```
{
    "detectorModelConfiguration": {
        "status": "ACTIVATING",
        "lastUpdateTime": 1557523491.168,
        "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole",
```

```
        "creationTime": 1557523491.168,
        "detectorModelArn": "arn:aws:iotevents:us-west-2:123456789012:detectorModel/
areaDetectorModel",
        "key": "areaId",
        "detectorModelName": "areaDetectorModel",
        "detectorModelVersion": "1"
    }
}
```

Use BatchUpdateDetector to update an AWS IoT Events detector model

You can use the `BatchUpdateDetector` operation to put a detector instance into a known state, including timer and variable values. In the following example, the `BatchUpdateDetector` operation resets operational parameters for an area that is under temperature monitoring and control. This operation enables you to do this without having to delete, and recreate, or update the detector model.

CLI command:

```
aws iotevents-data batch-update-detector --cli-input-json file://areaDM.BUD.json
```

File: `areaDM.BUD.json`

```
{
  "detectors": [
    {
      "messageId": "0001",
      "detectorModelName": "areaDetectorModel",
      "keyValue": "Area51",
      "state": {
        "stateName": "start",
        "variables": [
          {
            "name": "desiredTemperature",
            "value": "22"
          },
          {
            "name": "averageTemperature",
            "value": "22"
          },
          {
```

```
            "name": "allowedError",
            "value": "1.0"
          },
          {
            "name": "rangeHigh",
            "value": "30.0"
          },
          {
            "name": "rangeLow",
            "value": "15.0"
          },
          {
            "name": "anomalousHigh",
            "value": "60.0"
          },
          {
            "name": "anomalousLow",
            "value": "0.0"
          },
          {
            "name": "sensorCount",
            "value": "12"
          },
          {
            "name": "noDelay",
            "value": "true"
          },
          {
            "name": "goodToGo",
            "value": "true"
          },
          {
            "name": "sensorId",
            "value": "0"
          },
          {
            "name": "reportedTemperature",
            "value": "0.1"
          },
          {
            "name": "resetMe",
            // When 'resetMe' is true, our detector model knows that we have reentered
 the 'start' state
```

```
            //   to reset operational parameters, and will allow the next valid
 temperature sensor
            //   reading to cause the transition to the 'idle' state.
            "value": "true"
          }
        ],
        "timers": [
        ]
      }
    }
  ]
}
```

Response:

```
{
    "batchUpdateDetectorErrorEntries": []
}
```

Use BatchPutMessage for inputs in AWS IoT Events

Example 1

Use the `BatchPutMessage` operation to send a `"seedTemperatureInput"` message that sets the operational parameters for a given area under temperature control and monitoring. Any message received by AWS IoT Events that has a new `"areaId"` causes a new detector instance to be created. But the new detector instance won't change state to `"idle"` and begin monitoring the temperature and controlling heating or cooling units until a `"seedTemperatureInput"` message is received for the new area.

CLI command:

```
aws iotevents-data batch-put-message --cli-input-json file://seedExample.json --cli-
binary-format raw-in-base64-out
```

File: `seedExample.json`

```
{
  "messages": [
```

```
    {
      "messageId": "00001",
      "inputName": "seedTemperatureInput",
      "payload": "{\"areaId\": \"Area51\", \"desiredTemperature\": 20.0, \"allowedError
\": 0.7, \"rangeHigh\": 30.0, \"rangeLow\": 15.0, \"anomalousHigh\": 60.0,
 \"anomalousLow\": 0.0, \"sensorCount\": 10, \"noDelay\": false}"
    }
  ]
}
```

Response:

```
{
    "BatchPutMessageErrorEntries": []
}
```

Example

2

Use the `BatchPutMessage` operation to send a `"temperatureInput"` message to report temperature sensor data for a sensor in a given control and monitoring area.

CLI command:

```
aws iotevents-data batch-put-message --cli-input-json file://temperatureExample.json --
cli-binary-format raw-in-base64-out
```

File: `temperatureExample.json`

```
{
  "messages": [
    {
      "messageId": "00005",
      "inputName": "temperatureInput",
      "payload": "{\"sensorId\": \"05\", \"areaId\": \"Area51\", \"sensorData\":
 {\"temperature\": 23.12} }"
    }
  ]
}
```

Response:

```
{
    "BatchPutMessageErrorEntries": []
}
```

Example 3

Use the `BatchPutMessage` operation to send a `"seedTemperatureInput"` message to change the value of the desired temperature for a given area.

CLI command:

```
aws iotevents-data batch-put-message --cli-input-json file://seedSetDesiredTemp.json --
cli-binary-format raw-in-base64-out
```

File: `seedSetDesiredTemp.json`

```
{
  "messages": [
    {
      "messageId": "00001",
      "inputName": "seedTemperatureInput",
      "payload": "{\"areaId\": \"Area51\", \"desiredTemperature\": 23.0}"
    }
  ]
}
```

Response:

```
{
    "BatchPutMessageErrorEntries": []
}
```

Ingest MQTT messages in AWS IoT Events

If your sensor computing resources can't use the `"BatchPutMessage"` API, but can send their data to the AWS IoT Core message broker using a lightweight MQTT client, you can create an AWS IoT Core topic rule to redirect message data to an AWS IoT Events input. The following is a

definition of an AWS IoT Events topic rule that takes the "areaId" and "sensorId" input fields from the MQTT topic, and the "sensorData.temperature" field from the message payload "temp" field, and ingests this data into our AWS IoT Events "temperatureInput".

CLI command:

```
aws iot create-topic-rule --cli-input-json file://temperatureTopicRule.json
```

File: seedSetDesiredTemp.json

```
{
  "ruleName": "temperatureTopicRule",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as areaId, topic(4) as sensorId, temp as
 sensorData.temperature FROM 'update/temperature/#'",
    "description": "Ingest temperature sensor messages into IoT Events",
    "actions": [
      {
        "iotEvents": {
          "inputName": "temperatureInput",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/anotheRole"
        }
      }
    ],
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23"
  }
}
```

Response: [none]

If the sensor sends a message on the topic "update/temperature/Area51/03" with the following payload.

```
{ "temp": 24.5 }
```

This results in data being ingested into AWS IoT Events as if the following "BatchPutMessage" API call had been made.

```
aws iotevents-data batch-put-message --cli-input-json file://spoofExample.json --cli-
binary-format raw-in-base64-out
```

File: `spoofExample.json`

```
{
  "messages": [
    {
      "messageId": "54321",
      "inputName": "temperatureInput",
      "payload": "{\"sensorId\": \"03\", \"areaId\": \"Area51\", \"sensorData\":
 {\"temperature\": 24.5} }"
    }
  ]
}
```

Generate Amazon SNS messages in AWS IoT Events

The following are examples of SNS messages generated by the `"Area51"` detector instance.

AWS IoT Events can integrate with Amazon SNS to generate and publish notifications based on detected events. This section demonstrates how an AWS IoT Events detector instance, specifically the "Area51" detector, generates Amazon SNS messages. These examples showcase the structure and content of Amazon SNS notifications triggered by various states and events within the AWS IoT Events detector, illustrating the power of combining AWS IoT Events with Amazon SNS for real-time alerting and communication.

```
Heating system off command> {
  "eventTime":1557520274729,
  "payload":{
    "actionExecutionId":"f3159081-bac3-38a4-96f7-74af0940d0a4",
    "detector":{

 "detectorModelName":"areaDetectorModel","keyValue":"Area51","detectorModelVersion":"1"},"event
{"inputName":"seedTemperatureInput","messageId":"00001","triggerType":"Message"},"state":
{"stateName":"start","variables":
{"sensorCount":10,"rangeHigh":30.0,"resetMe":false,"enteringNewState":true,"averageTemperature"
{}}},"eventName":"resetHeatCool"}
```

```
Cooling system off command> {"eventTime":1557520274729,"payload":
{"actionExecutionId":"98f6a1b5-8f40-3cdb-9256-93afd4d66192","detector":
{"detectorModelName":"areaDetectorModel","keyValue":"Area51","detectorModelVersion":"1"},"event
{"inputName":"seedTemperatureInput","messageId":"00001","triggerType":"Message"},"state":
{"stateName":"start","variables":
{"sensorCount":10,"rangeHigh":30.0,"resetMe":false,"enteringNewState":true,"averageTemperature"
{}}},"eventName":"resetHeatCool"}
```

Configure the DescribeDetector API in AWS IoT Events

The `DescribeDetector` API in AWS IoT Events lets you to retrieve detailed information about a specific detector instance. This operation provides insights into the current state, variable values, and active timers of a detector. By using this API, you can monitor the real-time status of your AWS IoT Events detectors, facilitating debugging, analysis, and management of your IoT event processing workflows.

CLI command:

```
aws iotevents-data describe-detector --detector-model-name areaDetectorModel --key-
value Area51
```

Response:

```
{
    "detector": {
        "lastUpdateTime": 1557521572.216,
        "creationTime": 1557520274.405,
        "state": {
            "variables": [
                {
                    "name": "resetMe",
                    "value": "false"
                },
                {
                    "name": "rangeLow",
                    "value": "15.0"
                },
                {
                    "name": "noDelay",
                    "value": "false"
                },
```

```
                {
                    "name": "desiredTemperature",
                    "value": "20.0"
                },
                {
                    "name": "anomalousLow",
                    "value": "0.0"
                },
                {
                    "name": "sensorId",
                    "value": "\"01\""
                },
                {
                    "name": "sensorCount",
                    "value": "10"
                },
                {
                    "name": "rangeHigh",
                    "value": "30.0"
                },
                {
                    "name": "enteringNewState",
                    "value": "false"
                },
                {
                    "name": "averageTemperature",
                    "value": "19.572"
                },
                {
                    "name": "allowedError",
                    "value": "0.7"
                },
                {
                    "name": "anomalousHigh",
                    "value": "60.0"
                },
                {
                    "name": "reportedTemperature",
                    "value": "15.72"
                },
                {
                    "name": "goodToGo",
                    "value": "false"
                }
```

```
            ],
            "stateName": "idle",
            "timers": [
                {
                    "timestamp": 1557520454.0,
                    "name": "idleTimer"
                }
            ]
        },
        "keyValue": "Area51",
        "detectorModelName": "areaDetectorModel",
        "detectorModelVersion": "1"
    }
}
```

Use the AWS IoT Core rules engine for AWS IoT Events

The following rules republish AWS IoT Core MQTT messages as shadow update request messages. We assume that AWS IoT Core things are defined for a heating unit and a cooling unit for each area that is controlled by the detector model. In this example, we have defined things named `"Area51HeatingUnit"` and `"Area51CoolingUnit"`.

CLI command:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowCoolOffRule.json
```

File: `ADMShadowCoolOffRule.json`

```
{
  "ruleName": "ADMShadowCoolOff",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Cooling/Off'",
    "description": "areaDetectorModel mqtt topic publish to cooling unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
          "topic": "$$aws/things/${payload.detector.keyValue}CoolingUnit/shadow/
update",
```

```
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
}
```

Response: [empty]

CLI command:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowCoolOnRule.json
```

File: ADMShadowCoolOnRule.json

```
{
  "ruleName": "ADMShadowCoolOn",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Cooling/On'",
    "description": "areaDetectorModel mqtt topic publish to cooling unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
          "topic": "$$aws/things/${payload.detector.keyValue}CoolingUnit/shadow/
update",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
}
```

Response: [empty]

CLI command:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowHeatOffRule.json
```

File: ADMShadowHeatOffRule.json

```
{
  "ruleName": "ADMShadowHeatOff",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Heating/Off'",
    "description": "areaDetectorModel mqtt topic publish to heating unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
          "topic": "$$aws/things/${payload.detector.keyValue}HeatingUnit/shadow/
update",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
}
```

Response: [empty]

CLI command:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowHeatOnRule.json
```

File: ADMShadowHeatOnRule.json

```
{
  "ruleName": "ADMShadowHeatOn",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Heating/On'",
    "description": "areaDetectorModel mqtt topic publish to heating unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
```

```
          "topic": "$$aws/things/${payload.detector.keyValue}HeatingUnit/shadow/
update",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
}
```

Response: [empty]

Supported actions to receive data and trigger actions in AWS IoT Events

AWS IoT Events can trigger actions when it detects a specified event or transition event. You can define built-in actions to use a timer or set a variable, or send data to other AWS resources. Learn how to configure and customize these actions to create automated responses to your various IoT events.

> **Note**
>
> When you define an action in a detector model, you can use expressions for parameters that are string data type. For more information, see Expressions.

AWS IoT Events supports the following actions that let you use a timer or set a variable:

- `setTimer` to create a timer.
- `resetTimer` to reset the timer.
- `clearTimer` to delete the timer.
- `setVariable` to create a variable.

AWS IoT Events supports the following actions that let you work with AWS services:

- `iotTopicPublish` to publish a message on an MQTT topic.
- `iotEvents` to send data to AWS IoT Events as an input value.
- `iotSiteWise` to send data to an asset property in AWS IoT SiteWise.
- `dynamoDB` to send data to an Amazon DynamoDB table.
- `dynamoDBv2` to send data to an Amazon DynamoDB table.
- `firehose` to send data to an Amazon Data Firehose stream.
- `lambda` to invoke an AWS Lambda function.
- `sns` to send data as a push notification.
- `sqs` to send data to an Amazon SQS queue.

Use the AWS IoT Events built-in timer and variable actions

AWS IoT Events supports the following actions that let you use a timer or set a variable:

- `setTimer` to create a timer.
- `resetTimer` to reset the timer.
- `clearTimer` to delete the timer.
- `setVariable` to create a variable.

Set timer action

Set timer action

The `setTimer` action lets you create a timer with duration in seconds.

More information (2)

When you create a timer, you must specify the following parameters.

`timerName`

The name of the timer.

`durationExpression`

(Optional) The duration of the timer, in seconds.

The evaluated result of a duration expression is rounded down to the nearest whole number. For example, if you set the timer to 60.99 seconds, the evaluated result of the duration expression is 60 seconds.

For more information, see SetTimerAction in the *AWS IoT Events API Reference*.

Reset timer action

Reset timer action

The `resetTimer` action lets you set the timer to the previously evaluated result of the duration expression.

More information (1)

When you reset a timer, you must specify the following parameter.

`timerName`

The name of the timer.

AWS IoT Events doesn't reevaluate the duration expression when you reset the timer.

For more information, see ResetTimerAction in the *AWS IoT Events API Reference*.

Clear timer action

Clear timer action

The `clearTimer` action lets you delete an existing timer.

More information (1)

When you delete a timer, you must specify the following parameter.

`timerName`

The name of the timer.

For more information, see ClearTimerAction in the *AWS IoT Events API Reference*.

Set variable action

Set variable action

The `setVariable` action lets you create a variable with a specified value.

More information (2)

When you create a variable, you must specify the following parameters.

`variableName`

The name of the variable.

`value`

The new value of the variable.

For more information, see SetVariableAction in the *AWS IoT Events API Reference*.

AWS IoT Events working with other AWS services

AWS IoT Events supports the following actions that let you work with AWS services:

- `iotTopicPublish` to publish a message on an MQTT topic.
- `iotEvents` to send data to AWS IoT Events as an input value.
- `iotSiteWise` to send data to an asset property in AWS IoT SiteWise.
- `dynamoDB` to send data to an Amazon DynamoDB table.
- `dynamoDBv2` to send data to an Amazon DynamoDB table.
- `firehose` to send data to an Amazon Data Firehose stream.
- `lambda` to invoke an AWS Lambda function.
- `sns` to send data as a push notification.
- `sqs` to send data to an Amazon SQS queue.

Important

- You must choose the same AWS Region for both AWS IoT Events and the AWS services to work with. For the list of supported Regions, see AWS IoT Events endpoints and quotas in the *Amazon Web Services General Reference*.
- You must use the same AWS Region when you create other AWS resources for the AWS IoT Events actions. If you switch AWS Regions, you might have issues accessing the AWS resources.

By default, AWS IoT Events generates a standard payload in JSON for any action. This action payload contains all attribute-value pairs that have the information about the detector model instance and the event that triggered the action. To configure the action payload, you can use a

content expression. For more information, see Expressions to filter, transform, and process event data and the Payload data type in the *AWS IoT Events API Reference*.

AWS IoT Core

IoT topic publish action

The AWS IoT Core action lets you publish an MQTT message through the AWS IoT message broker. For the list of supported Regions, see AWS IoT Core endpoints and quotas in the *Amazon Web Services General Reference*.

The AWS IoT message broker connects AWS IoT clients by sending messages from publishing clients to subscribing clients. For more information, see Device communication protocols in the *AWS IoT Developer Guide*.

More information (2)

When you publish an MQTT message, you must specify the following parameters.

`mqttTopic`

The MQTT topic that receives the message.

You can define an MQTT topic name dynamically at runtime using variables or input values created in the detector model.

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `iot:Publish` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see IotTopicPublishAction in the *AWS IoT Events API Reference*.

AWS IoT Events

IoT Events action

The AWS IoT Events action lets you send data to AWS IoT Events as input. For the list of supported Regions, see AWS IoT Events endpoints and quotas in the *Amazon Web Services General Reference*.

AWS IoT Events lets you to monitor your equipment or device fleets for failures or changes in operation, and to trigger actions when such events occur. For more information, see What is AWS IoT Events? in the *AWS IoT Events Developer Guide*.

More information (2)

When you send data to AWS IoT Events, you must specify the following parameters.

`inputName`

The name of the AWS IoT Events input that receives the data.

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `iotevents:BatchPutMessage` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see IotEventsAction in the *AWS IoT Events API Reference*.

AWS IoT SiteWise

IoT SiteWise action

The AWS IoT SiteWise action lets you send data to an asset property in AWS IoT SiteWise. For the list of supported Regions, see AWS IoT SiteWise endpoints and quotas in the *Amazon Web Services General Reference*.

AWS IoT SiteWise is a managed service that lets you collect, organize, and analyze data from industrial equipment at scale. For more information, see What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.

More information (11)

When you send data to an asset property in AWS IoT SiteWise, you must specify the following parameters.

> **Important**
>
> To receive the data, you must use an existing asset property in AWS IoT SiteWise.
>
> - If you use the AWS IoT Events console, you must specify `propertyAlias` to identify the target asset property.
> - If you use the AWS CLI, you must specify either `propertyAlias` or both `assetId` and `propertyId` to identify the target asset property.
>
> For more information, see Mapping industrial data streams to asset properties in the *AWS IoT SiteWise User Guide*.

`propertyAlias`

(Optional) The alias of the asset property. You can also specify an expression.

`assetId`

(Optional) The ID of the asset that has the specified property. You can also specify an expression.

`propertyId`

(Optional) The ID of the asset property. You can also specify an expression.

`entryId`

(Optional) A unique identifier for this entry. You can use the entry ID to track which data entry causes an error in case of failure. The default is a new unique identifier. You can also specify an expression.

`propertyValue`

A structure that contains details about the property value.

`quality`

(Optional) The quality of the asset property value. The value must be `GOOD`, `BAD`, or `UNCERTAIN`. You can also specify an expression.

`timestamp`

(Optional) A structure that contains timestamp information. If you don't specify this value, the default is the event time.

`timeInSeconds`

The timestamp, in seconds, in the Unix epoch format. The valid range is between 1-31556889864403199. You can also specify an expression.

`offsetInNanos`

(Optional) The nanosecond offset converted from `timeInSeconds`. The valid range is between 0-999999999. You can also specify an expression.

`value`

A structure that contains an asset property value.

> **Important**
>
> You must specify one of the following value types, depending on the `dataType` of the specified asset property. For more information, see AssetProperty in the *AWS IoT SiteWise API Reference*.

`booleanValue`

(Optional) The asset property value is a Boolean value that must be TRUE or FALSE. You can also specify an expression. If you use an expression, the evaluated result should be a Boolean value.

`doubleValue`

(Optional) The asset property value is a double. You can also specify an expression. If you use an expression, the evaluated result should be a double.

`integerValue`

(Optional) The asset property value is an integer. You can also specify an expression. If you use an expression, the evaluated result should be an integer.

`stringValue`

(Optional) The asset property value is a string. You can also specify an expression. If you use an expression, the evaluated result should be a string.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `iotsitewise:BatchPutAssetPropertyValue` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see IotSiteWiseAction in the *AWS IoT Events API Reference*.

Amazon DynamoDB

DynamoDB action

The Amazon DynamoDB action lets you send data to a DynamoDB table. One column of the DynamoDB table receives all attribute-value pairs in the action payload that you specify. For the list of supported Regions, see Amazon DynamoDB endpoints and quotas in the *Amazon Web Services General Reference*.

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. For more information, see What is DynamoDB? in the *Amazon DynamoDB Developer Guide*.

More information (10)

When you send data to one column of a DynamoDB table, you must specify the following parameters.

`tableName`

The name of the DynamoDB table that receives the data. The `tableName` value must match the table name of the DynamoDB table. You can also specify an expression.

`hashKeyField`

The name of the hash key (also called partition key). The `hashKeyField` value must match the partition key of the DynamoDB table. You can also specify an expression.

`hashKeyType`

(Optional) The data type of the hash key. The value of the hash key type must be `STRING` or `NUMBER`. The default is `STRING`. You can also specify an expression.

`hashKeyValue`

The value of the hash key. The `hashKeyValue` uses substitution templates. These templates provide data at runtime. You can also specify an expression.

`rangeKeyField`

(Optional) The name of the range key (also called the sort key). The `rangeKeyField` value must match the sort key of the DynamoDB table. You can also specify an expression.

`rangeKeyType`

(Optional) The data type of the range key. The value of the hash key type must be `STRING` or `NUMBER`. The default is `STRING`. You can also specify an expression.

`rangeKeyValue`

(Optional) The value of the range key. The `rangeKeyValue` uses substitution templates. These templates provide data at runtime. You can also specify an expression.

operation

(Optional) The type of operation to perform. You can also specify an expression. The operation value must be one of the following values:

- `INSERT` - Insert data as a new item into the DynamoDB table. This is the default value.
- `UPDATE` - Update an existing item of the DynamoDB table with new data.
- `DELETE` - Delete an existing item from the DynamoDB table.

`payloadField`

(Optional) The name of the DynamoDB column that receives the action payload. The default name is `payload`. You can also specify an expression.

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

If the specified payload type is a string, `DynamoDBAction` sends non-JSON data to the DynamoDB table as binary data. The DynamoDB console displays the data as Base64-encoded text. The `payloadField` value is *`payload-field`*`_raw`. You can also specify an expression.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `dynamodb:PutItem` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see DynamoDBAction in the *AWS IoT Events API Reference*.

Amazon DynamoDB(v2)

DynamoDBv2 action

The Amazon DynamoDB(v2) action lets you write data to a DynamoDB table. A separate column of the DynamoDB table receives one attribute-value pair in the action payload that you specify. For the list of supported Regions, see Amazon DynamoDB endpoints and quotas in the *Amazon Web Services General Reference*.

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. For more information, see What is DynamoDB? in the *Amazon DynamoDB Developer Guide*.

More information (2)

When you send data to multiple columns of a DynamoDB table, you must specify the following parameters.

`tableName`

The name of the DynamoDB table that receives the data. You can also specify an expression.

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

> **Important**
>
> The payload type must be JSON. You can also specify an expression.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `dynamodb:PutItem` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see DynamoDBv2Action in the *AWS IoT Events API Reference*.

Amazon Data Firehose

Firehose action

The Amazon Data Firehose action lets you send data to an Firehose delivery stream. For the list of supported Regions, see Amazon Data Firehose endpoints and quotas in the *Amazon Web Services General Reference*.

Amazon Data Firehose is a fully managed service for delivering real-time streaming data to destinations such as Amazon Simple Storage Service (Amazon Simple Storage Service), Amazon Redshift, Amazon OpenSearch Service (OpenSearch Service), and Splunk. For more information, see What is Amazon Data Firehose? in the *Amazon Data Firehose Developer Guide*.

More information (3)

When you send data to an Firehose delivery stream, you must specify the following parameters.

`deliveryStreamName`

The name of the Firehose delivery stream that receives the data.

`separator`

(Optional) You can use a character separator to separate continuous data sent to the Firehose delivery stream. The separator value must be `'\n'`(newline), `'\t'` (tab), `'\r\n'` (Windows new line), or `','` (comma).

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `firehose:PutRecord` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see FirehoseAction in the *AWS IoT Events API Reference*.

AWS Lambda

Lambda action

The AWS Lambda action lets you call a Lambda function. For the list of supported Regions, see AWS Lambda endpoints and quotas in the *Amazon Web Services General Reference*.

AWS Lambda is a compute service that lets you run code without provisioning or managing servers. For more information, see What is AWS Lambda? in the *AWS Lambda Developer Guide*.

More information (2)

When you call a Lambda function, you must specify the following parameters.

`functionArn`

The ARN of the Lambda function to call.

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `lambda:InvokeFunction` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see LambdaAction in the *AWS IoT Events API Reference*.

Amazon Simple Notification Service

SNS action

The Amazon SNS topic publish action lets you publish an Amazon SNS message. For the list of supported Regions, see Amazon Simple Notification Service endpoints and quotas in the *Amazon Web Services General Reference*.

Amazon Simple Notification Service (Amazon Simple Notification Service) is a web service that coordinates and manages the delivery or sending of messages to subscribing endpoints or clients. For more information, see What is Amazon SNS? in the *Amazon Simple Notification Service Developer Guide*.

> **Note**
>
> The Amazon SNS topic publish action doesn't support Amazon SNS FIFO (first in, first out) topics. Because the rules engine is a fully distributed service, the messages may not display in a specified order when the Amazon SNS action is initiated.

More information (2)

When you publish an Amazon SNS message, you must specify the following parameters.

`targetArn`

The ARN of the Amazon SNS target that receives the message.

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `sns:Publish` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see SNSTopicPublishAction in the *AWS IoT Events API Reference*.

Amazon Simple Queue Service

SQS action

The Amazon SQS action lets you send data to an Amazon SQS queue. For the list of supported Regions, see Amazon Simple Queue Service endpoints and quotas in the *Amazon Web Services General Reference*.

Amazon Simple Queue Service (Amazon SQS) offers a secure, durable, and available hosted queue that lets you integrate and decouple distributed software systems and components. For more information, see What is Amazon Simple Queue Service> in the *Amazon Simple Queue Service Developer Guide*.

> **Note**
>
> The Amazon SQS action doesn't support >Amazon SQS FIFO (first in, first out) topics. Because the rules engine is a fully distributed service, the messages may not display in a specified order when the Amazon SQS action is initiated.

More information (3)

When you send data to an Amazon SQS queue, you must specify the following parameters.

`queueUrl`

The URL of the Amazon SQS queue that receives the data.

`useBase64`

(Optional) AWS IoT Events encodes the data into Base64 text, if you specify TRUE. The default is FALSE.

`payload`

(Optional) The default payload contains all attribute-value pairs that have the information about the detector model instance and the event triggered the action. You can also customize the payload. For more information, see Payload in the *AWS IoT Events API Reference*.

> **Note**
>
> Make sure that the policy attached to your AWS IoT Events service role grants the `sqs:SendMessage` permission. For more information, see Identity and access management for AWS IoT Events.

For more information, see SNSTopicPublishAction in the *AWS IoT Events API Reference*.

You can also use Amazon SNS and the AWS IoT Core rules engine to trigger an AWS Lambda function. This makes it possible to take actions using other services, such as Connect Customer, or even a company enterprise resource planning (ERP) application.

> **Note**
>
> To collect and process large streams of data records in real time, you can use other AWS services, such as Amazon Kinesis. From there, you can complete an initial analysis and then send the results to AWS IoT Events as an input to a detector.

Expressions to filter, transform, and process event data

Expressions are used to evaluate incoming data, perform calculations, and determine the conditions under which specific actions or state transitions should occur. AWS IoT Events provides several ways to specify values when you create and update detector models. You can use expressions to specify literal values, or AWS IoT Events can evaluate the expressions before you specify particular values.

Topics

- Syntax to filter device data and define actions in AWS IoT Events
- Expression examples and usage for AWS IoT Events

Syntax to filter device data and define actions in AWS IoT Events

Expressions offer syntax for filtering device data and defining actions. You can use literals, operators, functions, references, and substitution templates in the AWS IoT Events expressions. By combining these components, you can create powerful and flexible expressions to process IoT data, perform calculations, manipulate strings, and make logical decisions within your detector models.

Literals

- Integer
- Decimal
- String
- Boolean

Operators

Unary

- Not (Boolean): !
- Not (bitwise): ~
- Minus (arithmetic): -

String

- Concatenation: **+**

  Both operands must be strings. String literals must be enclosed in single quotes (').

  For example: `'my' + 'string'` -> `'mystring'`

Arithmetic

- Addition: **+**

  Both operands must be numeric.

- Subtraction: **-**
- Division: **/**

  The result of the division is a rounded integer value unless at least one of the operands (divisor or dividend) is a decimal value.

- Multiplication: *****

Bitwise (Integer)

- OR: **|**

  For example: `13 | 5` -> `13`

- AND: **&**

  For example: `13 & 5` -> `5`

- XOR: **^**

  For example: `13 ^ 5` -> `8`

- NOT: **~**

  For example: `~13` -> `-14`

Boolean

- Less Than: **<**
- Less Than Or Equal To: **<=**
- Equal To: **==**
- Not Equal To: **!=**
- Greater Than Or Equal To: **>=**

- Greater Than: **>**
- AND: **&&**
- OR: **||**

> **Note**
>
> When a subexpression of `||` contains undefined data, that subexpression is treated as `false`.

Parentheses

You can use parentheses to group terms within an expression.

Functions to use in AWS IoT Events expressions

AWS IoT Events provides a set of built-in functions to enhance the capabilities of your detector model expressions. These functions enable timer management, type conversion, null checking, trigger type identification, input verification, string manipulation, and bitwise operations. By leveraging these functions, you can create a responsive AWS IoT Events processing logic, improving the overall effectiveness of your IoT applications.

Built-in Functions

`timeout``("`*`timer-name`*`")`

Evaluates to `true` if the specified timer has elapsed. Replace "*`timer-name`*" with the name of a timer that you defined, in quotation marks. In an event action, you can define a timer and then start the timer, reset it, or clear one that you previously defined. See the field `detectorModelDefinition.states.onInput|onEnter|onExit.events.actions.setTimer.timerName`.

A timer set in one state can be referenced in a different state. You must visit the state in which you created the timer before you enter the state in which the timer is referenced.

For example, a detector model has two states, `TemperatureChecked` and `RecordUpdated`. You created a timer in the `TemperatureChecked` state. You must visit the `TemperatureChecked` state first before you can use the timer in the `RecordUpdated` state.

To ensure accuracy, the minimum time that a timer should be set is 60 seconds.

> **Note**
>
> `timeout()` returns `true` only the first time it's checked following the actual timer expiration and returns `false` thereafter.

`convert`(*`type`*, *`expression`*)

Evaluates to the value of the expression converted to the specified type. The *`type`* value must be `String`, `Boolean`, or `Decimal`. Use one of these keywords or an expression that evaluates to a string containing the keyword. Only the following conversions succeed and return a valid value:

- Boolean -> string

  Returns the string `"true"` or `"false"`.
- Decimal -> string
- String -> Boolean
- String -> decimal

  The string specified must be a valid representation of a decimal number, or `convert()` fails.

If `convert()` doesn't return a valid value, the expression that it's a part of is also invalid. This result is equivalent to `false` and won't trigger the `actions` or transition to the `nextState` specified as part of the event in which the expression occurs.

`isNull`(*`expression`*)

Evaluates to `true` if the expression returns null. For example, if the input `MyInput` receives the message `{ "a": null }`, then the following evaluates to `true`, but `isUndefined($input.MyInput.a)` evaluates to `false`.

```
isNull($input.MyInput.a)
```

`isUndefined`(*`expression`*)

Evaluates to `true` if the expression is undefined. For example, if the input `MyInput` receives the message `{ "a": null }`, then the following evaluates to `false`, but `isNull($input.MyInput.a)` evaluates to `true`.

```
isUndefined($input.MyInput.a)
```

triggerType("*type*")

The *type* value can be "Message" or "Timer". Evaluates to true if the event condition in which it appears is being evaluated because a timer has expired like in the following example.

```
triggerType("Timer")
```

Or an input message was received.

```
triggerType("Message")
```

currentInput("*input*")

Evaluates to true if the event condition in which it appears is being evaluated because the specified input message was received. For example, if the input Command receives the message { "value": "Abort" }, then the following evaluates to true.

```
currentInput("Command")
```

Use this function to verify that the condition is being evaluated because a particular input has been received and a timer hasn't expired, as in the following expression.

```
currentInput("Command") && $input.Command.value == "Abort"
```

String Matching Functions

startsWith(*expression1*, *expression2*)

Evaluates to true if the first string expression starts with the second string expression. For example, if input MyInput receives the message { "status": "offline"}, then the following evaluates to true.

```
startsWith($input.MyInput.status, "off")
```

Both expressions must evaluate to a string value. If either expression does not evaluate to a string value, then the result of the function is undefined. No conversions are performed.

endsWith(*expression1*, *expression2*)

Evaluates to `true` if the first string expression ends with the second string expression. For example, if input `MyInput` receives the message `{ "status": "offline" }`, then the following evaluates to `true`.

```
endsWith($input.MyInput.status, "line")
```

Both expressions must evaluate to a string value. If either expression does not evaluate to a string value, then the result of the function is undefined. No conversions are performed.

contains(*expression1*, *expression2*)

Evaluates to `true` if the first string expression contains the second string expression. For example, if input `MyInput` receives the message `{ "status": "offline" }`, then the following evaluates to `true`.

```
contains($input.MyInput.value, "fli")
```

Both expressions must evaluate to a string value. If either expression does not evaluate to a string value, then the result of the function is undefined. No conversions are performed.

Bitwise Integer Manipulation Functions

bitor(*expression1*, *expression2*)

Evaluates the bitwise OR of the integer expressions (the binary OR operation is performed on the corresponding bits of the integers). For example, if input `MyInput` receives the message `{ "value1": 13, "value2": 5 }`, then the following evaluates to 13.

```
bitor($input.MyInput.value1, $input.MyInput.value2)
```

Both expressions must evaluate to an integer value. If either expression does not evaluate to an integer value, then the result of the function is undefined. No conversions are performed.

bitand(*expression1*, *expression2*)

Evaluates the bitwise AND of the integer expressions (the binary AND operation is performed on the corresponding bits of the integers). For example, if input `MyInput` receives the message `{ "value1": 13, "value2": 5 }`, then the following evaluates to 5.

```
bitand($input.MyInput.value1, $input.MyInput.value2)
```

Both expressions must evaluate to an integer value. If either expression does not evaluate to an integer value, then the result of the function is undefined. No conversions are performed.

bitxor(*expression1*, *expression2*)

Evaluates the bitwise XOR of the integer expressions (the binary XOR operation is performed on the corresponding bits of the integers). For example, if input `MyInput` receives the message `{ "value1": 13, "value2": 5 }`, then the following evaluates to 8.

```
bitxor($input.MyInput.value1, $input.MyInput.value2)
```

Both expressions must evaluate to an integer value. If either expression does not evaluate to an integer value, then the result of the function is undefined. No conversions are performed.

bitnot(*expression*)

Evaluates the bitwise NOT of the integer expression (the binary NOT operation is performed on the bits of the integer). For example, if input `MyInput` receives the message `{ "value": 13 }`, then the following evaluates to `-14`.

```
bitnot($input.MyInput.value)
```

Both expressions must evaluate to an integer value. If either expression does not evaluate to an integer value, then the result of the function is undefined. No conversions are performed.

AWS IoT Events reference for inputs and variables in expressions

Inputs

`$input.`*input-name*`.`*path-to-data*

`input-name` is an input that you create using the CreateInput action.

For example, if you have an input named `TemperatureInput` for which you defined `inputDefinition.attributes.jsonPath` entries, the values might appear in the following available fields.

```
{
```

```
    "temperature": 78.5,
    "date": "2018-10-03T16:09:09Z"
  }
```

To reference the value of the `temperature` field, use the following command.

```
$input.TemperatureInput.temperature
```

For fields whose values are arrays, you can reference members of the array using `[`*n*`]`. For example, given the following values:

```
{
    "temperatures": [
      78.4,
      77.9,
      78.8
    ],
    "date": "2018-10-03T16:09:09Z"
  }
```

The value `78.8` can be referenced with the following command.

```
$input.TemperatureInput.temperatures[2]
```

Variables

`$variable.`*`variable-name`*

The *`variable-name`* is a variable that you defined using the CreateDetectorModel action.

For example, if you have a variable named `TechnicianID` that you defined using `detectorDefinition.states.onInputEvents.actions.setVariable.variableName`, you can reference the (string) value most recently given to the variable with the following command.

```
$variable.TechnicianID
```

You can set the values of variables only using the `setVariable` action. You can't assign values for variables in an expression. A variable can't be unset. For example, you can't assign it the value `null`.

> **Note**
>
> In references that use identifiers that don't follow the (regular expression) pattern `[a-zA-Z][a-zA-Z0-9_]*`, you must enclose those identifiers in backticks (`). For example, a reference to an input named `MyInput` with a field named `_value` must specify this field as `$input.MyInput.`_value`.`.

When you use references in expressions, check the following:

- When you use a reference as an operand with one or more operators, make sure that all data types that you reference are compatible.

  For example, in the following expression, integer 2 is an operand of both the == and && operators. To ensure that the operands are compatible, `$variable.testVariable + 1` and `$variable.testVariable` must reference an integer or decimal.

  In addition, integer 1 is an operand of the + operator. Therefore, `$variable.testVariable` must reference an integer or decimal.

  ```
  '$variable.testVariable + 1 == 2 && $variable.testVariable'
  ```

- When you use a reference as an argument passed to a function, make sure that the function supports the data types that you reference.

  For example, the following `timeout("time-name")` function requires a string with double quotes as the argument. If you use a reference for the *`timer-name`* value, you must reference a string with double quotes.

  ```
  timeout("timer-name")
  ```

  > **Note**
  >
  > For the `convert(type, expression)` function, if you use a reference for the *`type`* value, the evaluated result of your reference must be `String`, `Decimal`, or `Boolean`.

AWS IoT Events expressions support integer, decimal, string, and Boolean data types. The following table provides a list of incompatible pairs of types.

Incompatible pairs of types
Integer, string
Integer, Boolean
Decimal, string
Decimal, Boolean
String, Boolean

Substitution templates for AWS IoT Events expressions

```
'${expression}'
```

The `${}` identifies the string as an interpolated string. The `expression` can be any AWS IoT Events expression. This includes operators, functions, and references.

For example, you used the SetVariableAction action to define a variable. The `variableName` is `SensorID`, and the `value` is `10`. You can create the following substitution templates.

Substitution template	Result string
`'${'Sensor ' + $variable.SensorID}'`	`"Sensor 10"`
`'Sensor ' + '${$variable.SensorID + 1}'`	`"Sensor 11"`
`'Sensor 10: ${$variable.SensorID == 10}'`	`"Sensor 10: true"`

Substitution template	Result string
`'{\"sensor\":\"${$variable.SensorID + 1}\"}'`	`"{\"sensor\":\"11\"}"`
`'{\"sensor\":${$variable.SensorID + 1}}'`	`"{\"sensor\":11}"`

Expression examples and usage for AWS IoT Events

You can specify values in a detector model in the following ways:

- Enter supported expressions in the AWS IoT Events console.
- Pass the expressions to the AWS IoT Events APIs as parameters.

Expressions support literals, operators, functions, references, and substitution templates.

> **Important**
>
> Your expressions must reference a integer, decimal, string, or Boolean value.

Writing AWS IoT Events expressions

See the following examples to help you write your AWS IoT Events expressions:

Literal

For literal values, the expressions must contain single quotes. A Boolean value must be either `true` or `false`.

```
'123'        # Integer
'123.12'     # Decimal
'hello'      # String
'true'       # Boolean
```

Reference

For references, you must specify either variables or input values.

- The following input references a decimal number, `10.01`.

```
$input.GreenhouseInput.temperature
```

- The following variable references a string, `Greenhouse Temperature Table`.

```
$variable.TableName
```

Substitution template

For a substitution template, you must use `${}`, and the template must be in single quotes. A substitution template can also contain a combination of literals, operators, functions, references, and substitution templates.

- The evaluated result of the following expression is a string, `50.018 in Fahrenheit`.

```
'${$input.GreenhouseInput.temperature * 9 / 5 + 32} in Fahrenheit'
```

- The evaluated result of the following expression is a string, `{\"sensor_id\": \"Sensor_1\",\"temperature\":\"50.018\"}`.

```
'{\"sensor_id\":\"${$input.GreenhouseInput.sensors[0].sensor1}\",\"temperature\":
\"${$input.GreenhouseInput.temperature*9/5+32}\"}'
```

String concatenation

For a string concatenation, you must use +. A string concatenation can also contain a combination of literals, operators, functions, references, and substitution templates.

- The evaluated result of the following expression is a string, `Greenhouse Temperature Table 2000-01-01`.

```
'Greenhouse Temperature Table ' + $input.GreenhouseInput.date
```

AWS IoT Events detector model examples

This page provides a list of example use cases that demonstrate how to configure various AWS IoT Events features. The examples range from basic detections like temperature thresholds to more advanced anomaly detection and machine learning scenarios. Each example includes procedures and code snippets to help you set up AWS IoT Events detections, actions, and integrations. These examples showcase the flexibility of the AWS IoT Events service and how it can be customized for diverse IoT applications and use cases. Refer to this page when exploring AWS IoT Events capabilities or if you need guidance implementing a specific detection or automation workflow.

Topics

- Example: Using HVAC temperature control with AWS IoT Events
- Example: A crane detecting conditions using AWS IoT Events
- Send commands in response to detected conditions in AWS IoT Events
- An AWS IoT Events detector model for crane monitoring
- AWS IoT Events inputs for crane monitoring
- Send alarm and operational messages with AWS IoT Events
- Example: AWS IoT Events event detection with sensors and applications
- Example: Device HeartBeat to monitor device connections with AWS IoT Events
- Example: An ISA alarm in AWS IoT Events
- Example: Build a simple alarm with AWS IoT Events

Example: Using HVAC temperature control with AWS IoT Events

Background story

This example implements a temperature control model (a thermostat) with these features:

- One detector model you define that can monitor and control multiple areas. (A detector instance will be created for each area.)
- Each detector instance receives temperature data from multiple sensors placed in each control area.
- You can change the desired temperature (the set point) for each area at any time.

- You can define the operational parameters for each area and change these parameters at any time.
- You can add sensors to or delete sensors from an area at any time.
- You can enable a minimum run for time heating and cooling units to protect them from damage.
- The detectors will reject, and report, anomalous sensor readings.
- You can define emergency temperature set points. If any one sensor reports a temperature above or below the set points you have defined, heating or cooling units will be engaged immediately, and the detector will report that temperature spike.

This example demonstrates the following functional capabilities:

- Create event detector models.
- Create inputs.
- Ingest inputs into a detector model.
- Evaluate trigger conditions.
- Refer to state variables in conditions and set the values of variables depending on conditions.
- Refer to timers in conditions and set timers depending on conditions.
- Take actions that send Amazon SNS and MQTT messages.

Input definitions for an HVAC system in AWS IoT Events

A `seedTemperatureInput` is used to create a detector instance for an area and define its operational parameters.

Configuring inputs for HVAC systems in AWS IoT Events is important for effective climate control. This example shows how to set up inputs that capture parameters such as, temperature, humidity, occupancy, and energy consumption data. Learn to define input attributes, configure data sources, and set up preprocessing rules to help your detector models receive accurate and timely information for optimal management and efficiency.

CLI command used:

```
aws iotevents create-input --cli-input-json file://seedInput.json
```

File: `seedInput.json`

```
{
  "inputName": "seedTemperatureInput",
  "inputDescription": "Temperature seed values.",
  "inputDefinition": {
    "attributes": [
      { "jsonPath": "areaId" },
      { "jsonPath": "desiredTemperature" },
      { "jsonPath": "allowedError" },
      { "jsonPath": "rangeHigh" },
      { "jsonPath": "rangeLow" },
      { "jsonPath": "anomalousHigh" },
      { "jsonPath": "anomalousLow" },
      { "jsonPath": "sensorCount" },
      { "jsonPath": "noDelay" }
    ]
  }
}
```

Response:

```
{
    "inputConfiguration": {
        "status": "ACTIVE",
        "inputArn": "arn:aws:iotevents:us-west-2:123456789012:input/
seedTemperatureInput",
        "lastUpdateTime": 1557519620.736,
        "creationTime": 1557519620.736,
        "inputName": "seedTemperatureInput",
        "inputDescription": "Temperature seed values."
    }
}
```

A `temperatureInput` should be sent by each sensor in each area, as necessary.

CLI command used:

```
aws iotevents create-input --cli-input-json file://temperatureInput.json
```

File: `temperatureInput.json`

```
{
```

```
  "inputName": "temperatureInput",
  "inputDescription": "Temperature sensor unit data.",
  "inputDefinition": {
    "attributes": [
      { "jsonPath": "sensorId" },
      { "jsonPath": "areaId" },
      { "jsonPath": "sensorData.temperature" }
    ]
  }
}
```

Response:

```
{
    "inputConfiguration": {
        "status": "ACTIVE",
        "inputArn": "arn:aws:iotevents:us-west-2:123456789012:input/temperatureInput",
        "lastUpdateTime": 1557519707.399,
        "creationTime": 1557519707.399,
        "inputName": "temperatureInput",
        "inputDescription": "Temperature sensor unit data."
    }
}
```

Detector model definition for an HVAC system using AWS IoT Events

The `areaDetectorModel` defines how each detector instance works. Each `state machine` instance will ingest temperature sensor readings, then change state and send control messages depending on these readings.

CLI command used:

```
aws iotevents create-detector-model --cli-input-json file://areaDetectorModel.json
```

File: `areaDetectorModel.json`

```
{
  "detectorModelName": "areaDetectorModel",
  "detectorModelDefinition": {
    "states": [
      {
```

```
        "stateName": "start",
        "onEnter": {
          "events": [
            {
              "eventName": "prepare",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "sensorId",
                    "value": "0"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "reportedTemperature",
                    "value": "0.1"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "resetMe",
                    "value": "false"
                  }
                }
              ]
            }
          ]
        },
        "onInput": {
          "transitionEvents": [
            {
              "eventName": "initialize",
              "condition": "$input.seedTemperatureInput.sensorCount > 0",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "rangeHigh",
                    "value": "$input.seedTemperatureInput.rangeHigh"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "rangeLow",
```

```
            "value": "$input.seedTemperatureInput.rangeLow"
          }
        },
        {
          "setVariable": {
            "variableName": "desiredTemperature",
            "value": "$input.seedTemperatureInput.desiredTemperature"
          }
        },
        {
          "setVariable": {
            "variableName": "averageTemperature",
            "value": "$input.seedTemperatureInput.desiredTemperature"
          }
        },
        {
          "setVariable": {
            "variableName": "allowedError",
            "value": "$input.seedTemperatureInput.allowedError"
          }
        },
        {
          "setVariable": {
            "variableName": "anomalousHigh",
            "value": "$input.seedTemperatureInput.anomalousHigh"
          }
        },
        {
          "setVariable": {
            "variableName": "anomalousLow",
            "value": "$input.seedTemperatureInput.anomalousLow"
          }
        },
        {
          "setVariable": {
            "variableName": "sensorCount",
            "value": "$input.seedTemperatureInput.sensorCount"
          }
        },
        {
          "setVariable": {
            "variableName": "noDelay",
            "value": "$input.seedTemperatureInput.noDelay == true"
          }
```

```
              }
            ],
            "nextState": "idle"
          },
          {
            "eventName": "reset",
            "condition": "($variable.resetMe == true) &&
($input.temperatureInput.sensorData.temperature < $variable.anomalousHigh &&
$input.temperatureInput.sensorData.temperature > $variable.anomalousLow)",
            "actions": [
              {
                "setVariable": {
                  "variableName": "averageTemperature",
                  "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                }
              }
            ],
            "nextState": "idle"
          }
        ]
      },
      "onExit": {
        "events": [
          {
            "eventName": "resetHeatCool",
            "condition": "true",
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOff"
                }
              },
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOff"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Heating/Off"
                }
              },
              {
```

```
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Cooling/Off"
                }
              }
            ]
          }
        ]
      }
    },


    {
      "stateName": "idle",
      "onInput": {
        "events": [
          {
            "eventName": "whatWasInput",
            "condition": "true",
            "actions": [
              {
                "setVariable": {
                  "variableName": "sensorId",
                  "value": "$input.temperatureInput.sensorId"
                }
              },
              {
                "setVariable": {
                  "variableName": "reportedTemperature",
                  "value": "$input.temperatureInput.sensorData.temperature"
                }
              }
            ]
          },
          {
            "eventName": "changeDesired",
            "condition": "$input.seedTemperatureInput.desiredTemperature !=
$variable.desiredTemperature",
            "actions": [
              {
                "setVariable": {
                  "variableName": "desiredTemperature",
                  "value": "$input.seedTemperatureInput.desiredTemperature"
                }
              }
```

```
            ]
          },
          {
            "eventName": "calculateAverage",
            "condition": "$input.temperatureInput.sensorData.temperature <
$variable.anomalousHigh && $input.temperatureInput.sensorData.temperature >
$variable.anomalousLow",
            "actions": [
              {
                "setVariable": {
                  "variableName": "averageTemperature",
                  "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                }
              }
            ]
          }
        ],
        "transitionEvents": [
          {
            "eventName": "anomalousInputArrived",
            "condition": "$input.temperatureInput.sensorData.temperature >=
$variable.anomalousHigh || $input.temperatureInput.sensorData.temperature <=
$variable.anomalousLow",
            "actions": [
              {
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/anomaly"
                }
              }
            ],
            "nextState": "idle"
          },

          {
            "eventName": "highTemperatureSpike",
            "condition": "$input.temperatureInput.sensorData.temperature >
$variable.rangeHigh",
            "actions": [
              {
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/spike"
                }
              },
```

```
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOn"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Cooling/On"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "enteringNewState",
                    "value": "true"
                  }
                }
              ],
              "nextState": "cooling"
            },

            {
              "eventName": "lowTemperatureSpike",
              "condition": "$input.temperatureInput.sensorData.temperature <
$variable.rangeLow",
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/spike"
                  }
                },
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOn"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Heating/On"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "enteringNewState",
                    "value": "true"
```

```
                }
              }
            ],
            "nextState": "heating"
          },

          {
            "eventName": "highTemperatureThreshold",
            "condition": "((((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) >
($variable.desiredTemperature + $variable.allowedError))",
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOn"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Cooling/On"
                }
              },
              {
                "setVariable": {
                  "variableName": "enteringNewState",
                  "value": "true"
                }
              }
            ],
            "nextState": "cooling"
          },

          {
            "eventName": "lowTemperatureThreshold",
            "condition": "((((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) <
($variable.desiredTemperature - $variable.allowedError))",
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOn"
                }
              },
              {
```

```
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Heating/On"
                }
              },
              {
                "setVariable": {
                  "variableName": "enteringNewState",
                  "value": "true"
                }
              }
            ],
            "nextState": "heating"
          }
        ]
      }
    },


    {
      "stateName": "cooling",
      "onEnter": {
        "events": [
          {
            "eventName": "delay",
            "condition": "!$variable.noDelay && $variable.enteringNewState",
            "actions": [
              {
                "setTimer": {
                  "timerName": "coolingTimer",
                  "seconds": 180
                }
              },
              {
                "setVariable": {
                  "variableName": "goodToGo",
                  "value": "false"
                }
              }
            ]
          },
          {
            "eventName": "dontDelay",
            "condition": "$variable.noDelay == true",
            "actions": [
```

```
            {
              "setVariable": {
                "variableName": "goodToGo",
                "value": "true"
              }
            }
          ]
        },
        {
          "eventName": "beenHere",
          "condition": "true",
          "actions": [
            {
              "setVariable": {
                "variableName": "enteringNewState",
                "value": "false"
              }
            }
          ]
        }
      ]
    },

    "onInput": {
      "events": [
        {
          "eventName": "whatWasInput",
          "condition": "true",
          "actions": [
            {
              "setVariable": {
                "variableName": "sensorId",
                "value": "$input.temperatureInput.sensorId"
              }
            },
            {
              "setVariable": {
                "variableName": "reportedTemperature",
                "value": "$input.temperatureInput.sensorData.temperature"
              }
            }
          ]
        },
        {
```

```
            "eventName": "changeDesired",
            "condition": "$input.seedTemperatureInput.desiredTemperature !=
$variable.desiredTemperature",
            "actions": [
              {
                "setVariable": {
                  "variableName": "desiredTemperature",
                  "value": "$input.seedTemperatureInput.desiredTemperature"
                }
              }
            ]
          },
          {
            "eventName": "calculateAverage",
            "condition": "$input.temperatureInput.sensorData.temperature <
$variable.anomalousHigh && $input.temperatureInput.sensorData.temperature >
$variable.anomalousLow",
            "actions": [
              {
                "setVariable": {
                  "variableName": "averageTemperature",
                  "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                }
              }
            ]
          },
          {
            "eventName": "areWeThereYet",
            "condition": "(timeout(\"coolingTimer\"))",
            "actions": [
              {
                "setVariable": {
                  "variableName": "goodToGo",
                  "value": "true"
                }
              }
            ]
          }
        ],
        "transitionEvents": [
          {
            "eventName": "anomalousInputArrived",
```

```
              "condition": "$input.temperatureInput.sensorData.temperature >=
$variable.anomalousHigh || $input.temperatureInput.sensorData.temperature <=
$variable.anomalousLow",
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/anomaly"
                  }
                }
              ],
              "nextState": "cooling"
            },

            {
              "eventName": "highTemperatureSpike",
              "condition": "$input.temperatureInput.sensorData.temperature >
$variable.rangeHigh",
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/spike"
                  }
                }
              ],
              "nextState": "cooling"
            },

            {
              "eventName": "lowTemperatureSpike",
              "condition": "$input.temperatureInput.sensorData.temperature <
$variable.rangeLow",
              "actions": [
                {
                  "iotTopicPublish": {
                    "mqttTopic": "temperatureSensor/spike"
                  }
                },
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOff"
                  }
                },
                {
                  "sns": {
```

```
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOn"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Cooling/Off"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Heating/On"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "enteringNewState",
                    "value": "true"
                  }
                }
              ],
              "nextState": "heating"
            },

            {
              "eventName": "desiredTemperature",
              "condition": "((((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) <=
($variable.desiredTemperature - $variable.allowedError)) && $variable.goodToGo ==
true",
              "actions": [
                {
                  "sns": {
                    "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOff"
                  }
                },
                {
                  "iotTopicPublish": {
                    "mqttTopic": "hvac/Cooling/Off"
                  }
                }
              ],
              "nextState": "idle"
            }
          ]
```

```
        }
      },


      {
        "stateName": "heating",
        "onEnter": {
          "events": [
            {
              "eventName": "delay",
              "condition": "!$variable.noDelay && $variable.enteringNewState",
              "actions": [
                {
                  "setTimer": {
                    "timerName": "heatingTimer",
                    "seconds": 120
                  }
                },
                {
                  "setVariable": {
                    "variableName": "goodToGo",
                    "value": "false"
                  }
                }
              ]
            },
            {
              "eventName": "dontDelay",
              "condition": "$variable.noDelay == true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "goodToGo",
                    "value": "true"
                  }
                }
              ]
            },
            {
              "eventName": "beenHere",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
```

```
                    "variableName": "enteringNewState",
                    "value": "false"
                  }
                }
              ]
            }
          ]
        },

        "onInput": {
          "events": [
            {
              "eventName": "whatWasInput",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "sensorId",
                    "value": "$input.temperatureInput.sensorId"
                  }
                },
                {
                  "setVariable": {
                    "variableName": "reportedTemperature",
                    "value": "$input.temperatureInput.sensorData.temperature"
                  }
                }
              ]
            },
            {
              "eventName": "changeDesired",
              "condition": "$input.seedTemperatureInput.desiredTemperature !=
$variable.desiredTemperature",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "desiredTemperature",
                    "value": "$input.seedTemperatureInput.desiredTemperature"
                  }
                }
              ]
            },
            {
              "eventName": "calculateAverage",
```

```
            "condition": "$input.temperatureInput.sensorData.temperature <
$variable.anomalousHigh && $input.temperatureInput.sensorData.temperature >
$variable.anomalousLow",
            "actions": [
              {
                "setVariable": {
                  "variableName": "averageTemperature",
                  "value": "((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount)"
                }
              }
            ]
          },
          {
            "eventName": "areWeThereYet",
            "condition": "(timeout(\"heatingTimer\"))",
            "actions": [
              {
                "setVariable": {
                  "variableName": "goodToGo",
                  "value": "true"
                }
              }
            ]
          }
        ],
        "transitionEvents": [
          {
            "eventName": "anomalousInputArrived",
            "condition": "$input.temperatureInput.sensorData.temperature >=
$variable.anomalousHigh || $input.temperatureInput.sensorData.temperature <=
$variable.anomalousLow",
            "actions": [
              {
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/anomaly"
                }
              }
            ],
            "nextState": "heating"
          },

          {
            "eventName": "highTemperatureSpike",
```

```
            "condition": "$input.temperatureInput.sensorData.temperature >
$variable.rangeHigh",
            "actions": [
              {
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/spike"
                }
              },
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOff"
                }
              },
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:coolOn"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Heating/Off"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Cooling/On"
                }
              },
              {
                "setVariable": {
                  "variableName": "enteringNewState",
                  "value": "true"
                }
              }
            ],
            "nextState": "cooling"
          },

          {
            "eventName": "lowTemperatureSpike",
            "condition": "$input.temperatureInput.sensorData.temperature <
$variable.rangeLow",
            "actions": [
              {
```

```
                "iotTopicPublish": {
                  "mqttTopic": "temperatureSensor/spike"
                }
              }
            ],
            "nextState": "heating"
          },

          {
            "eventName": "desiredTemperature",
            "condition": "((((($variable.averageTemperature * ($variable.sensorCount
- 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) >=
($variable.desiredTemperature + $variable.allowedError)) && $variable.goodToGo ==
true",
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:heatOff"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Heating/Off"
                }
              }
            ],
            "nextState": "idle"
          }
        ]
      }
    }

  ],

  "initialStateName": "start"
 },
 "key": "areaId",
 "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole"
}
```

Response:

```
{
```

```
    "detectorModelConfiguration": {
        "status": "ACTIVATING",
        "lastUpdateTime": 1557523491.168,
        "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole",
        "creationTime": 1557523491.168,
        "detectorModelArn": "arn:aws:iotevents:us-west-2:123456789012:detectorModel/
areaDetectorModel",
        "key": "areaId",
        "detectorModelName": "areaDetectorModel",
        "detectorModelVersion": "1"
    }
}
```

BatchPutMessage examples for an HVAC system in AWS IoT Events

In this example, `BatchPutMessage` is used to create a detector instance for an area and define the initial operating parameters.

CLI command used:

```
aws iotevents-data batch-put-message --cli-input-json file://seedExample.json --cli-
binary-format raw-in-base64-out
```

File: `seedExample.json`

```
{
  "messages": [
    {
      "messageId": "00001",
      "inputName": "seedTemperatureInput",
      "payload": "{\"areaId\": \"Area51\", \"desiredTemperature\": 20.0, \"allowedError
\": 0.7, \"rangeHigh\": 30.0, \"rangeLow\": 15.0, \"anomalousHigh\": 60.0,
 \"anomalousLow\": 0.0, \"sensorCount\": 10, \"noDelay\": false}"
    }
  ]
}
```

Response:

```
{
    "BatchPutMessageErrorEntries": []
}
```

In this example, `BatchPutMessage` is used to report temperature sensor readings for a single sensor in an area.

CLI command used:

```
aws iotevents-data batch-put-message --cli-input-json file://temperatureExample.json --
cli-binary-format raw-in-base64-out
```

File: `temperatureExample.json`

```
{
  "messages": [
    {
      "messageId": "00005",
      "inputName": "temperatureInput",
      "payload": "{\"sensorId\": \"05\", \"areaId\": \"Area51\", \"sensorData\":
 {\"temperature\": 23.12} }"
    }
  ]
}
```

Response:

```
{
    "BatchPutMessageErrorEntries": []
}
```

In this example, `BatchPutMessage` is used to change the desired temperature for an area.

CLI command used:

```
aws iotevents-data batch-put-message --cli-input-json file://seedSetDesiredTemp.json --
cli-binary-format raw-in-base64-out
```

File: `seedSetDesiredTemp.json`

```
{
  "messages": [
    {
      "messageId": "00001",
      "inputName": "seedTemperatureInput",
      "payload": "{\"areaId\": \"Area51\", \"desiredTemperature\": 23.0}"
```

```
    }
  ]
}
```

Response:

```
{
    "BatchPutMessageErrorEntries": []
}
```

Examples of Amazon SNS messages generated by the `Area51` detector instance:

```
Heating system off command> {
  "eventTime":1557520274729,
  "payload":{
    "actionExecutionId":"f3159081-bac3-38a4-96f7-74af0940d0a4",
    "detector":{
      "detectorModelName":"areaDetectorModel",
      "keyValue":"Area51",
      "detectorModelVersion":"1"
    },
    "eventTriggerDetails":{
      "inputName":"seedTemperatureInput",
      "messageId":"00001",
      "triggerType":"Message"
    },
    "state":{
      "stateName":"start",
      "variables":{
        "sensorCount":10,
        "rangeHigh":30.0,
        "resetMe":false,
        "enteringNewState":true,
        "averageTemperature":20.0,
        "rangeLow":15.0,
        "noDelay":false,
        "allowedError":0.7,
        "desiredTemperature":20.0,
        "anomalousHigh":60.0,
        "reportedTemperature":0.1,
        "anomalousLow":0.0,
```

```
        "sensorId":0
      },
      "timers":{}
    }
  },
  "eventName":"resetHeatCool"
}
```

```
Cooling system off command> {
  "eventTime":1557520274729,
  "payload":{
    "actionExecutionId":"98f6a1b5-8f40-3cdb-9256-93afd4d66192",
    "detector":{
      "detectorModelName":"areaDetectorModel",
      "keyValue":"Area51",
      "detectorModelVersion":"1"
    },
    "eventTriggerDetails":{
      "inputName":"seedTemperatureInput",
      "messageId":"00001",
      "triggerType":"Message"
    },
    "state":{
      "stateName":"start",
      "variables":{
        "sensorCount":10,
        "rangeHigh":30.0,
        "resetMe":false,
        "enteringNewState":true,
        "averageTemperature":20.0,
        "rangeLow":15.0,
        "noDelay":false,
        "allowedError":0.7,
        "desiredTemperature":20.0,
        "anomalousHigh":60.0,
        "reportedTemperature":0.1,
        "anomalousLow":0.0,
        "sensorId":0
      },
      "timers":{}
    }
  },
```

```
  "eventName":"resetHeatCool"
}
```

In this example, we use the `DescribeDetector` API to get information about the current state of a detector instance.

```
aws iotevents-data describe-detector --detector-model-name areaDetectorModel --key-
value Area51
```

Response:

```
{
    "detector": {
        "lastUpdateTime": 1557521572.216,
        "creationTime": 1557520274.405,
        "state": {
            "variables": [
                {
                    "name": "resetMe",
                    "value": "false"
                },
                {
                    "name": "rangeLow",
                    "value": "15.0"
                },
                {
                    "name": "noDelay",
                    "value": "false"
                },
                {
                    "name": "desiredTemperature",
                    "value": "20.0"
                },
                {
                    "name": "anomalousLow",
                    "value": "0.0"
                },
                {
                    "name": "sensorId",
                    "value": "\"01\""
                },
                {
                    "name": "sensorCount",
```

```
                    "value": "10"
                },
                {
                    "name": "rangeHigh",
                    "value": "30.0"
                },
                {
                    "name": "enteringNewState",
                    "value": "false"
                },
                {
                    "name": "averageTemperature",
                    "value": "19.572"
                },
                {
                    "name": "allowedError",
                    "value": "0.7"
                },
                {
                    "name": "anomalousHigh",
                    "value": "60.0"
                },
                {
                    "name": "reportedTemperature",
                    "value": "15.72"
                },
                {
                    "name": "goodToGo",
                    "value": "false"
                }
            ],
            "stateName": "idle",
            "timers": [
                {
                    "timestamp": 1557520454.0,
                    "name": "idleTimer"
                }
            ]
        },
        "keyValue": "Area51",
        "detectorModelName": "areaDetectorModel",
        "detectorModelVersion": "1"
    }
```

```
}
```

BatchUpdateDetector example for an HVAC system in AWS IoT Events

In this example, `BatchUpdateDetector` is used to change operational parameters for a working detector instance.

Efficient HVAC system management often requires batch updates to multiple detectors. This section demonstrates how to use AWS IoT Events's batch update feature for detectors. Learn to simultaneously modify multiple control parameters, update threshold values, so that you can adjust response actions across a fleet of devices, improving your ability to manage large-scale systems effectively.

CLI command used:

```
aws iotevents-data batch-update-detector --cli-input-json file://areaDM.BUD.json
```

File: `areaDM.BUD.json`

```
{
  "detectors": [
    {
      "messageId": "0001",
      "detectorModelName": "areaDetectorModel",
      "keyValue": "Area51",
      "state": {
        "stateName": "start",
        "variables": [
          {
            "name": "desiredTemperature",
            "value": "22"
          },
          {
            "name": "averageTemperature",
            "value": "22"
          },
          {
            "name": "allowedError",
            "value": "1.0"
          },
          {
```

```
            "name": "rangeHigh",
            "value": "30.0"
          },
          {
            "name": "rangeLow",
            "value": "15.0"
          },
          {
            "name": "anomalousHigh",
            "value": "60.0"
          },
          {
            "name": "anomalousLow",
            "value": "0.0"
          },
          {
            "name": "sensorCount",
            "value": "12"
          },
          {
            "name": "noDelay",
            "value": "true"
          },
          {
            "name": "goodToGo",
            "value": "true"
          },
          {
            "name": "sensorId",
            "value": "0"
          },
          {
            "name": "reportedTemperature",
            "value": "0.1"
          },
          {
            "name": "resetMe",
            "value": "true"
          }
        ],
        "timers": [
        ]
      }
    }
```

```
  ]
}
```

Response:

```
{
    An error occurred (InvalidRequestException) when calling the BatchUpdateDetector
 operation: Number of variables in the detector exceeds the limit 10
}
```

The AWS IoT Core rules engine and AWS IoT Events

The following rules republish AWS IoT Events MQTT messages as shadow update request messages. We assume that AWS IoT Core things are defined for a heating unit and a cooling unit for each area that is controlled by the detector model.

In this example, we have defined things named `Area51HeatingUnit` and `Area51CoolingUnit`.

CLI command used:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowCoolOffRule.json
```

File: `ADMShadowCoolOffRule.json`

```
{
  "ruleName": "ADMShadowCoolOff",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Cooling/Off'",
    "description": "areaDetectorModel mqtt topic publish to cooling unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
          "topic": "$$aws/things/${payload.detector.keyValue}CoolingUnit/shadow/
update",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
```

```
}
```

Response: [empty]

CLI command used:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowCoolOnRule.json
```

File: ADMShadowCoolOnRule.json

```
{
  "ruleName": "ADMShadowCoolOn",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Cooling/On'",
    "description": "areaDetectorModel mqtt topic publish to cooling unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
          "topic": "$$aws/things/${payload.detector.keyValue}CoolingUnit/shadow/
update",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
}
```

Response: [empty]

CLI command used:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowHeatOffRule.json
```

File: ADMShadowHeatOffRule.json

```
{
  "ruleName": "ADMShadowHeatOff",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Heating/Off'",
```

```
    "description": "areaDetectorModel mqtt topic publish to heating unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
          "topic": "$$aws/things/${payload.detector.keyValue}HeatingUnit/shadow/
update",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
}
```

Response: [empty]

CLI command used:

```
aws iot create-topic-rule --cli-input-json file://ADMShadowHeatOnRule.json
```

File: ADMShadowHeatOnRule.json

```
{
  "ruleName": "ADMShadowHeatOn",
  "topicRulePayload": {
    "sql": "SELECT topic(3) as state.desired.command FROM 'hvac/Heating/On'",
    "description": "areaDetectorModel mqtt topic publish to heating unit shadow
 request",
    "ruleDisabled": false,
    "awsIotSqlVersion": "2016-03-23",
    "actions": [
      {
        "republish": {
          "topic": "$$aws/things/${payload.detector.keyValue}HeatingUnit/shadow/
update",
          "roleArn": "arn:aws:iam::123456789012:role/service-role/ADMShadowRole"
        }
      }
    ]
  }
}
```

Response: [empty]

Example: A crane detecting conditions using AWS IoT Events

An operator of many cranes wants to detect when the machines need maintenance or replacement and trigger appropriate notifications. Each crane has a motor. A motor emits messages (inputs) with information about pressure and temperature. The operator wants two levels of event detectors:

- A crane-level event detector
- A motor-level event detector

Using messages from the motors (that contain metadata with both the `craneId` and the `motorid`), the operator can execute both levels of event detectors using appropriate routing. When event conditions are met, notifications should be sent to appropriate Amazon SNS topics. The operator can configure the detector models so that duplicate notifications are not raised.

This example demonstrates the following functional capabilities:

- Create, Read, Update, Delete (CRUD) of inputs.
- Create, Read, Update, Delete (CRUD) of event detector models and different versions of event detectors.
- Routing one input to multiple event detectors.
- Ingestion of inputs into a detector model.
- Evaluation of trigger conditions and lifecycle events.
- Ability to refer to state variables in conditions and set their values depending on conditions.
- Runtime orchestration with definition, state, trigger evaluator, and actions executor.
- Execution of actions in `ActionsExecutor` with an SNS target.

Send commands in response to detected conditions in AWS IoT Events

This page provides an example for using AWS IoT Events commands to set up inputs, create detector models, and send simulated sensor data. The examples demonstrate how to leverage AWS IoT Events to monitor industrial equipment like motors and cranes.

```
#Create Pressure Input
aws iotevents create-input  --cli-input-json file://pressureInput.json
aws iotevents describe-input --input-name PressureInput
aws iotevents update-input  --cli-input-json file://pressureInput.json
aws iotevents list-inputs
aws iotevents delete-input --input-name PressureInput

#Create Temperature Input
aws iotevents create-input  --cli-input-json file://temperatureInput.json
aws iotevents describe-input --input-name TemperatureInput
aws iotevents update-input  --cli-input-json file://temperatureInput.json
aws iotevents list-inputs
aws iotevents delete-input --input-name TemperatureInput

#Create Motor Event Detector using pressure and temperature input
aws iotevents create-detector-model  --cli-input-json file://motorDetectorModel.json
aws iotevents describe-detector-model --detector-model-name motorDetectorModel
aws iotevents update-detector-model  --cli-input-json file://
updateMotorDetectorModel.json
aws iotevents list-detector-models
aws iotevents list-detector-model-versions --detector-model-name motorDetectorModel
aws iotevents delete-detector-model --detector-model-name motorDetectorModel

#Create Crane Event Detector using temperature input
aws iotevents create-detector-model  --cli-input-json file://craneDetectorModel.json
aws iotevents describe-detector-model --detector-model-name craneDetectorModel
aws iotevents update-detector-model  --cli-input-json file://
updateCraneDetectorModel.json
aws iotevents list-detector-models
aws iotevents list-detector-model-versions --detector-model-name craneDetectorModel
aws iotevents delete-detector-model --detector-model-name craneDetectorModel

#Replace craneIds
sed -i '' "s/100008/100009/g" messages/*

#Replace motorIds
sed -i '' "s/200008/200009/g" messages/*

#Send HighPressure message
aws iotevents-data batch-put-message --cli-input-json file://messages/
highPressureMessage.json --cli-binary-format raw-in-base64-out
```

```
#Send HighTemperature message
aws iotevents-data batch-put-message --cli-input-json file://messages/
highTemperatureMessage.json --cli-binary-format raw-in-base64-out

#Send LowPressure message
aws iotevents-data batch-put-message --cli-input-json file://messages/
lowPressureMessage.json --cli-binary-format raw-in-base64-out

#Send LowTemperature message
aws iotevents-data batch-put-message --cli-input-json file://messages/
lowTemperatureMessage.json --cli-binary-format raw-in-base64-out
```

An AWS IoT Events detector model for crane monitoring

Monitor your equipment or device fleets for failures or changes in operation, and trigger actions when such events occur. You define detector models in JSON which specify states, rules, and actions. This allows you to monitor inputs like temperature and pressure, track threshold breaches, and send alerts. The examples show detector models for a crane and motor, detecting overheating issues and notifying by Amazon SNS when a threshold is exceeded. You can update models to refine behavior without disrupting monitoring.

File: `craneDetectorModel.json`

```
{
    "detectorModelName": "craneDetectorModel",
    "detectorModelDefinition": {
        "states": [
            {
                "stateName": "Running",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "init",
                            "condition": "true",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "craneThresholdBreached",
                                        "value": "0"
                                    }
                                }
```

```
                            ]
                        }
                    ]
                },
                "onInput": {
                    "events": [
                        {
                            "eventName": "Overheated",
                            "condition": "$input.TemperatureInput.temperature > 35",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "craneThresholdBreached",
                                        "value": "$variable.craneThresholdBreached + 1"
                                    }
                                }
                            ]
                        },
                        {
                            "eventName": "Crane Threshold Breached",
                            "condition": "$variable.craneThresholdBreached > 5",
                            "actions": [
                                {
                                    "sns": {
                                        "targetArn": "arn:aws:sns:us-
east-1:123456789012:CraneSNSTopic"
                                    }
                                }
                            ]
                        },
                        {
                            "eventName": "Underheated",
                            "condition": "$input.TemperatureInput.temperature < 25",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "craneThresholdBreached",
                                        "value": "0"
                                    }
                                }
                            ]
                        }
                    ]
                }
```

```
            }
        ],
        "initialStateName": "Running"
    },
    "key": "craneid",
    "roleArn": "arn:aws:iam::123456789012:role/columboSNSRole"
}
```

To update an existing detector model. File: `updateCraneDetectorModel.json`

```
{
    "detectorModelName": "craneDetectorModel",
    "detectorModelDefinition": {
        "states": [
            {
                "stateName": "Running",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "init",
                            "condition": "true",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "craneThresholdBreached",
                                        "value": "0"
                                    }
                                },
                                {
                                    "setVariable": {
                                        "variableName": "alarmRaised",
                                        "value": "'false'"
                                    }
                                }
                            ]
                        }
                    ]
                },
                "onInput": {
                    "events": [
                        {
                            "eventName": "Overheated",
                            "condition": "$input.TemperatureInput.temperature > 30",
```

```
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "craneThresholdBreached",
                                        "value": "$variable.craneThresholdBreached + 1"
                                    }
                                }
                            ]
                        },
                        {
                            "eventName": "Crane Threshold Breached",
                            "condition": "$variable.craneThresholdBreached > 5 &&
 $variable.alarmRaised == 'false'",
                            "actions": [
                                {
                                    "sns": {
                                        "targetArn": "arn:aws:sns:us-
east-1:123456789012:CraneSNSTopic"
                                    }
                                },
                                {
                                    "setVariable": {
                                        "variableName": "alarmRaised",
                                        "value": "'true'"
                                    }
                                }
                            ]
                        },
                        {
                            "eventName": "Underheated",
                            "condition": "$input.TemperatureInput.temperature < 10",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "craneThresholdBreached",
                                        "value": "0"
                                    }
                                }
                            ]
                        }
                    ]
                }
            }
        ],
```

```
        "initialStateName": "Running"
    },
    "roleArn": "arn:aws:iam::123456789012:role/columboSNSRole"
}
```

File: motorDetectorModel.json

```
{
    "detectorModelName": "motorDetectorModel",
    "detectorModelDefinition": {
        "states": [
            {
                "stateName": "Running",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "init",
                            "condition": "true",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "motorThresholdBreached",
                                        "value": "0"
                                    }
                                }
                            ]
                        }
                    ]
                },
                "onInput": {
                    "events": [
                        {
                            "eventName": "Overheated And Overpressurized",
                            "condition": "$input.PressureInput.pressure > 70 &&
 $input.TemperatureInput.temperature > 30",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "motorThresholdBreached",
                                        "value": "$variable.motorThresholdBreached + 1"
                                    }
                                }
                            ]
```

```
                        },
                        {
                            "eventName": "Motor Threshold Breached",
                            "condition": "$variable.motorThresholdBreached > 5",
                            "actions": [
                                {
                                    "sns": {
                                        "targetArn": "arn:aws:sns:us-
east-1:123456789012:MotorSNSTopic"
                                    }
                                }
                            ]
                        }
                    ]
                }
            }
        ],
        "initialStateName": "Running"
    },
    "key": "motorId",
    "roleArn": "arn:aws:iam::123456789012:role/columboSNSRole"
}
```

To update an existing detector model. File: `updateMotorDetectorModel.json`

```
{
    "detectorModelName": "motorDetectorModel",
    "detectorModelDefinition": {
        "states": [
            {
                "stateName": "Running",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "init",
                            "condition": "true",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "motorThresholdBreached",
                                        "value": "0"
                                    }
                                }
```

```
                            ]
                        }
                    ]
                },
                "onInput": {
                    "events": [
                        {
                            "eventName": "Overheated And Overpressurized",
                            "condition": "$input.PressureInput.pressure > 70 &&
 $input.TemperatureInput.temperature > 30",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "motorThresholdBreached",
                                        "value": "$variable.motorThresholdBreached + 1"
                                    }
                                }
                            ]
                        },
                        {
                            "eventName": "Motor Threshold Breached",
                            "condition": "$variable.motorThresholdBreached > 5",
                            "actions": [
                                {
                                    "sns": {
                                        "targetArn": "arn:aws:sns:us-
east-1:123456789012:MotorSNSTopic"
                                    }
                                }
                            ]
                        }
                    ]
                }
            }
        ],
        "initialStateName": "Running"
    },
    "roleArn": "arn:aws:iam::123456789012:role/columboSNSRole"
}
```

AWS IoT Events inputs for crane monitoring

In this example, we demonstrate how to set up inputs for a crane monitoring system using AWS IoT Events. It captures pressure and temperature inputs to illustrate how to structure inputs for complex industrial equipment monitoring.

File: `pressureInput.json`

```
{
    "inputName": "PressureInput",
    "inputDescription": "this is a pressure input description",
    "inputDefinition": {
        "attributes": [
          {"jsonPath": "pressure"}
        ]
    }
}
```

File: `temperatureInput.json`

```
{
    "inputName": "TemperatureInput",
    "inputDescription": "this is temperature input description",
    "inputDefinition": {
        "attributes": [
            {"jsonPath": "temperature"}
        ]
    }
}
```

Send alarm and operational messages with AWS IoT Events

Effective message handling is important in crane monitoring systems. This section showcases how to configure AWS IoT Events to process and respond to various message types from crane sensors. Setting up alarms based on a particular message can help you parse, filter, and route status updates to trigger appropriate actions.

File: `highPressureMessage.json`

```
{
   "messages": [
```

```
        {
           "messageId": "1",
           "inputName": "PressureInput",
           "payload": "{\"craneid\": \"100009\", \"pressure\": 80, \"motorid\":
 \"200009\"}"

        }
    ]
}
```

File: highTemperatureMessage.json

```
{
   "messages": [
        {
           "messageId": "2",
           "inputName": "TemperatureInput",
           "payload": "{\"craneid\": \"100009\", \"temperature\": 40, \"motorid\":
 \"200009\"}"
        }
    ]
}
```

File: lowPressureMessage.json

```
{
   "messages": [
        {
           "messageId": "1",
           "inputName": "PressureInput",
           "payload": "{\"craneid\": \"100009\", \"pressure\": 20, \"motorid\":
 \"200009\"}"
        }
    ]
}
```

File: lowTemperatureMessage.json

```
{
   "messages": [
        {
           "messageId": "2",
```

```
            "inputName": "TemperatureInput",
            "payload": "{\"craneid\": \"100009\", \"temperature\": 20, \"motorid\":
 \"200009\"}"
        }
    ]
}
```

Example: AWS IoT Events event detection with sensors and applications

This detector model is one of the templates available from the AWS IoT Events console. It's included here for your convenience.

This example demonstrates AWS IoT Events's application event detection using sensor data. It shows how you can create a detector model that monitors specified events so that you can trigger appropriate actions. You can create multiple sensor inputs, define complex event conditions, and set up graduated response mechanisms.

```
{
    "detectorModelName": "EventDetectionSensorsAndApplications",
    "detectorModelDefinition": {
        "states": [
            {
                "onInput": {
                    "transitionEvents": [],
                    "events": []
                },
                "stateName": "Device_exception",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Send_mqtt",
                            "actions": [
                                {
                                    "iotTopicPublish": {
                                        "mqttTopic": "Device_stolen"
                                    }
                                }
                            ],
                            "condition": "true"
                        }
```

```
                    ]
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "To_in_use",
                            "actions": [],
                            "condition": "$variable.position !=
$input.AWS_IoTEvents_Blueprints_Tracking_DeviceInput.gps_position",
                            "nextState": "Device_in_use"
                        }
                    ],
                    "events": []
                },
                "stateName": "Device_idle",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Set_position",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "position",
                                        "value":
"$input.AWS_IoTEvents_Blueprints_Tracking_DeviceInput.gps_position"
                                    }
                                }
                            ],
                            "condition": "true"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
```

```
                        {
                            "eventName": "To_exception",
                            "actions": [],
                            "condition":
 "$input.AWS_IoTEvents_Blueprints_Tracking_UserInput.device_id !=
 $input.AWS_IoTEvents_Blueprints_Tracking_DeviceInput.device_id",
                            "nextState": "Device_exception"
                        }
                    ],
                    "events": []
                },
                "stateName": "Device_in_use",
                "onEnter": {
                    "events": []
                },
                "onExit": {
                    "events": []
                }
            }
        ],
        "initialStateName": "Device_idle"
    }
}
```

Example: Device HeartBeat to monitor device connections with AWS IoT Events

This detector model is one of the templates available from the AWS IoT Events console. It's included here for your convenience.

The Defective Heart Beat (DHB) example illustrates how AWS IoT Events can be used in healthcare monitoring. This example shows how you can create a detector model that analyzes heart rate data, detects irregular patterns, and triggers appropriate responses. Learn to set up inputs, define thresholds, and configure alerts for potential cardiac issues, showcasing AWS IoT Events's versatility in related healthcare applications.

```
{
    "detectorModelDefinition": {
        "states": [
            {
                "onInput": {
```

```
                    "transitionEvents": [
                        {
                            "eventName": "To_normal",
                            "actions": [],
                            "condition":
"currentInput(\"AWS_IoTEvents_Blueprints_Heartbeat_Input\")",
                            "nextState": "Normal"
                        }
                    ],
                    "events": []
                },
                "stateName": "Offline",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Send_notification",
                            "actions": [
                                {
                                    "sns": {
                                        "targetArn": "sns-topic-arn"
                                    }
                                }
                            ],
                            "condition": "true"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "Go_offline",
                            "actions": [],
                            "condition": "timeout(\"awake\")",
                            "nextState": "Offline"
                        }
                    ],
                    "events": [
                        {
                            "eventName": "Reset_timer",
```

```
                            "actions": [
                                {
                                    "resetTimer": {
                                        "timerName": "awake"
                                    }
                                }
                            ],
                            "condition":
 "currentInput(\"AWS_IoTEvents_Blueprints_Heartbeat_Input\")"
                        }
                    ]
                },
                "stateName": "Normal",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Create_timer",
                            "actions": [
                                {
                                    "setTimer": {
                                        "seconds": 300,
                                        "timerName": "awake"
                                    }
                                }
                            ],
                            "condition":
 "$input.AWS_IoTEvents_Blueprints_Heartbeat_Input.value > 0"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            }
        ],
        "initialStateName": "Normal"
    }
}
```

Example: An ISA alarm in AWS IoT Events

This detector model is one of the templates available from the AWS IoT Events console. It's included here for your convenience.

```
{
    "detectorModelName": "AWS_IoTEvents_Blueprints_ISA_Alarm",
    "detectorModelDefinition": {
        "states": [
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "unshelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unshelve\" &&
$variable.state == \"rtnunack\"",
                            "nextState": "RTN_Unacknowledged"
                        },
                        {
                            "eventName": "unshelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unshelve\" &&
$variable.state == \"ack\"",
                            "nextState": "Acknowledged"
                        },
                        {
                            "eventName": "unshelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unshelve\" &&
$variable.state == \"unack\"",
                            "nextState": "Unacknowledged"
                        },
                        {
                            "eventName": "unshelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unshelve\" &&
$variable.state == \"normal\"",
                            "nextState": "Normal"
                        }
                    ],
                    "events": []
                },
                "stateName": "Shelved",
```

```
                "onEnter": {
                    "events": []
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "abnormal_condition",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.value > $variable.higher_threshold ||
$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.value < $variable.lower_threshold",
                            "nextState": "Unacknowledged"
                        },
                        {
                            "eventName": "acknowledge",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"acknowledge\"",
                            "nextState": "Normal"
                        },
                        {
                            "eventName": "shelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"shelve\"",
                            "nextState": "Shelved"
                        },
                        {
                            "eventName": "remove_from_service",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"remove\"",
                            "nextState": "Out_of_service"
                        },
                        {
                            "eventName": "suppression",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"suppressed\"",
```

```
                            "nextState": "Suppressed_by_design"
                        }
                    ],
                    "events": []
                },
                "stateName": "RTN_Unacknowledged",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "State Save",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "state",
                                        "value": "\"rtnunack\""
                                    }
                                }
                            ],
                            "condition": "true"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "abnormal_condition",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.value > $variable.higher_threshold ||
$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.value < $variable.lower_threshold",
                            "nextState": "Unacknowledged"
                        },
                        {
                            "eventName": "shelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"shelve\"",
                            "nextState": "Shelved"
                        },
```

```
                        {
                            "eventName": "remove_from_service",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"remove\"",
                            "nextState": "Out_of_service"
                        },
                        {
                            "eventName": "suppression",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"suppressed\"",
                            "nextState": "Suppressed_by_design"
                        }
                    ],
                    "events": [
                        {
                            "eventName": "Create Config variables",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "lower_threshold",
                                        "value":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.lower_threshold"
                                    }
                                },
                                {
                                    "setVariable": {
                                        "variableName": "higher_threshold",
                                        "value":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.higher_threshold"
                                    }
                                }
                            ],
                            "condition": "$variable.lower_threshold !=
$variable.lower_threshold"
                        }
                    ]
                },
                "stateName": "Normal",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "State Save",
```

```
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "state",
                                        "value": "\"normal\""
                                    }
                                }
                            ],
                            "condition": "true"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "acknowledge",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"acknowledge\"",
                            "nextState": "Acknowledged"
                        },
                        {
                            "eventName": "return_to_normal",
                            "actions": [],
                            "condition":
"($input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.value <= $variable.higher_threshold
&& $input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.value >=
$variable.lower_threshold)",
                            "nextState": "RTN_Unacknowledged"
                        },
                        {
                            "eventName": "shelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"shelve\"",
                            "nextState": "Shelved"
                        },
                        {
                            "eventName": "remove_from_service",
```

```
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"remove\"",
                            "nextState": "Out_of_service"
                        },
                        {
                            "eventName": "suppression",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"suppressed\"",
                            "nextState": "Suppressed_by_design"
                        }
                    ],
                    "events": []
                },
                "stateName": "Unacknowledged",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "State Save",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "state",
                                        "value": "\"unack\""
                                    }
                                }
                            ],
                            "condition": "true"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "unsuppression",
                            "actions": [],
```

```
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unsuppressed\" &&
$variable.state == \"normal\"",
                            "nextState": "Normal"
                        },
                        {
                            "eventName": "unsuppression",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unsuppressed\" &&
$variable.state == \"unack\"",
                            "nextState": "Unacknowledged"
                        },
                        {
                            "eventName": "unsuppression",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unsuppressed\" &&
$variable.state == \"ack\"",
                            "nextState": "Acknowledged"
                        },
                        {
                            "eventName": "unsuppression",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"unsuppressed\" &&
$variable.state == \"rtnunack\"",
                            "nextState": "RTN_Unacknowledged"
                        }
                    ],
                    "events": []
                },
                "stateName": "Suppressed_by_design",
                "onEnter": {
                    "events": []
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
```

```
                            "eventName": "return_to_service",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"add\" && $variable.state
== \"rtnunack\"",
                            "nextState": "RTN_Unacknowledged"
                        },
                        {
                            "eventName": "return_to_service",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"add\" && $variable.state
== \"unack\"",
                            "nextState": "Unacknowledged"
                        },
                        {
                            "eventName": "return_to_service",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"add\" && $variable.state
== \"ack\"",
                            "nextState": "Acknowledged"
                        },
                        {
                            "eventName": "return_to_service",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"add\" && $variable.state
== \"normal\"",
                            "nextState": "Normal"
                        }
                    ],
                    "events": []
                },
                "stateName": "Out_of_service",
                "onEnter": {
                    "events": []
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
```

```
                    "transitionEvents": [
                        {
                            "eventName": "re-alarm",
                            "actions": [],
                            "condition": "timeout(\"snooze\")",
                            "nextState": "Unacknowledged"
                        },
                        {
                            "eventName": "return_to_normal",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"reset\"",
                            "nextState": "Normal"
                        },
                        {
                            "eventName": "shelve",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"shelve\"",
                            "nextState": "Shelved"
                        },
                        {
                            "eventName": "remove_from_service",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"remove\"",
                            "nextState": "Out_of_service"
                        },
                        {
                            "eventName": "suppression",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_ISA_Alarm_Input.command == \"suppressed\"",
                            "nextState": "Suppressed_by_design"
                        }
                    ],
                    "events": []
                },
                "stateName": "Acknowledged",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Create Timer",
                            "actions": [
```

```
                                {
                                    "setTimer": {
                                        "seconds": 60,
                                        "timerName": "snooze"
                                    }
                                }
                            ],
                            "condition": "true"
                        },
                        {
                            "eventName": "State Save",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "state",
                                        "value": "\"ack\""
                                    }
                                }
                            ],
                            "condition": "true"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            }
        ],
        "initialStateName": "Normal"
    },
    "detectorModelDescription": "This detector model is used to detect if a monitored
 device is in an Alarming State in accordance to the ISA 18.2.",
    "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole",
    "key": "alarmId"
}
```

Example: Build a simple alarm with AWS IoT Events

This detector model is one of the templates available from the AWS IoT Events console. It's included here for your convenience.

```
{
```

```
    "detectorModelDefinition": {
        "states": [
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "not_fixed",
                            "actions": [],
                            "condition": "timeout(\"snoozeTime\")",
                            "nextState": "Alarming"
                        },
                        {
                            "eventName": "reset",
                            "actions": [],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_Simple_Alarm_Input.command == \"reset\"",
                            "nextState": "Normal"
                        }
                    ],
                    "events": [
                        {
                            "eventName": "DND",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "dnd_active",
                                        "value": "1"
                                    }
                                }
                            ],
                            "condition":
"$input.AWS_IoTEvents_Blueprints_Simple_Alarm_Input.command == \"dnd\""
                        }
                    ]
                },
                "stateName": "Snooze",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Create Timer",
                            "actions": [
                                {
                                    "setTimer": {
                                        "seconds": 120,
```

```
                                    "timerName": "snoozeTime"
                                }
                            }
                        ],
                        "condition": "true"
                    }
                ]
            },
            "onExit": {
                "events": []
            }
        },
        {
            "onInput": {
                "transitionEvents": [
                    {
                        "eventName": "out_of_range",
                        "actions": [],
                        "condition":
"$input.AWS_IoTEvents_Blueprints_Simple_Alarm_Input.value > $variable.threshold",
                        "nextState": "Alarming"
                    }
                ],
                "events": [
                    {
                        "eventName": "Create Config variables",
                        "actions": [
                            {
                                "setVariable": {
                                    "variableName": "threshold",
                                    "value":
"$input.AWS_IoTEvents_Blueprints_Simple_Alarm_Input.threshold"
                                }
                            }
                        ],
                        "condition": "$variable.threshold != $variable.threshold"
                    }
                ]
            },
            "stateName": "Normal",
            "onEnter": {
                "events": [
                    {
                        "eventName": "Init",
```

```
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "dnd_active",
                                        "value": "0"
                                    }
                                }
                            ],
                            "condition": "true"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            },
            {
                "onInput": {
                    "transitionEvents": [
                        {
                            "eventName": "reset",
                            "actions": [],
                            "condition":
 "$input.AWS_IoTEvents_Blueprints_Simple_Alarm_Input.command == \"reset\"",
                            "nextState": "Normal"
                        },
                        {
                            "eventName": "acknowledge",
                            "actions": [],
                            "condition":
 "$input.AWS_IoTEvents_Blueprints_Simple_Alarm_Input.command == \"acknowledge\"",
                            "nextState": "Snooze"
                        }
                    ],
                    "events": [
                        {
                            "eventName": "Escalated Alarm Notification",
                            "actions": [
                                {
                                    "sns": {
                                        "targetArn": "arn:aws:sns:us-
west-2:123456789012:escalatedAlarmNotification"
                                    }
                                }
```

```
                            ],
                            "condition": "timeout(\"unacknowledgeTIme\")"
                        }
                    ]
                },
                "stateName": "Alarming",
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Alarm Notification",
                            "actions": [
                                {
                                    "sns": {
                                        "targetArn": "arn:aws:sns:us-
west-2:123456789012:alarmNotification"
                                    }
                                },
                                {
                                    "setTimer": {
                                        "seconds": 300,
                                        "timerName": "unacknowledgeTIme"
                                    }
                                }
                            ],
                            "condition": "$variable.dnd_active != 1"
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            }
        ],
        "initialStateName": "Normal"
    },
    "detectorModelDescription": "This detector model is used to detect if a monitored
 device is in an Alarming State.",
    "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole",
    "key": "alarmId"
}
```

Monitoring with alarms in AWS IoT Events

AWS IoT Events alarms help you monitor your data for changes. The data can be metrics that you measure for your equipment and processes. You can create alarms that send notifications when a threshold is breached. Alarms help you detect issues, streamline maintenance, and optimize performance of your equipment and processes.

Alarms are instances of alarm models. The alarm model specifies what to detect, when to send notifications, who gets notified, and more. You can also specify one or more supported actions that occur when the alarm state changes. AWS IoT Events routes input attributes derived from your data to the appropriate alarms. If the data that you're monitoring is outside the specified range, the alarm is invoked. You can also acknowledge the alarms or set them to the snooze mode.

Working with AWS IoT SiteWise

You can use AWS IoT Events alarms to monitor asset properties in AWS IoT SiteWise. AWS IoT SiteWise sends asset property values to AWS IoT Events alarms. AWS IoT Events sends the alarm state to AWS IoT SiteWise.

AWS IoT SiteWise also supports external alarms. You might choose external alarms if you use alarms outside of AWS IoT SiteWise and have a solution that returns alarm state data. The external alarm contains a measurement property that ingests the alarm state data.

AWS IoT SiteWise doesn't evaluate the state of external alarms. Additionally, you can't acknowledge or snooze an external alarm when the alarm state changes.

You can use the SiteWise Monitor feature to view the state of external alarms in SiteWise Monitor portals.

For more information, see Monitoring data with alarms in the *AWS IoT SiteWise User Guide* and Monitoring with alarms in the *SiteWise Monitor Application Guide*.

Acknowledge flow

When you create an alarm model, you choose whether to enable acknowledge flow. If you enable acknowledge flow, your team gets notified when the alarm state changes. Your team can acknowledge the alarm and leave a note. For example, you can include the information of

the alarm and the actions that you're going to take to address the issue. If the data that you're monitoring is outside the specified range, the alarm is invoked.

Alarms have the following states:

`DISABLED`

When the alarm is in the `DISABLED` state, it isn't ready to evaluate data. To enable the alarm, you must change the alarm to the `NORMAL` state.

`NORMAL`

When the alarm is in the `NORMAL` state, it's ready to evaluate data.

`ACTIVE`

If the alarm is in the `ACTIVE` state, the alarm is invoked. The data that you're monitoring is outside the specified range.

`ACKNOWLEDGED`

When the alarm is in the `ACKNOWLEDGED` state, the alarm was invoked and you acknowledged the alarm.

`LATCHED`

The alarm was invoked, but you didn't acknowledge the alarm after a period of time. The alarm automatically changes to the `NORMAL` state.

`SNOOZE_DISABLED`

When the alarm is in the `SNOOZE_DISABLED` state, the alarm is disabled for a specified period of time. After the snooze time, the alarm automatically changes to the `NORMAL` state.

Creating an alarm model in AWS IoT Events

You can use AWS IoT Events alarms to monitor your data and get notified when a threshold is breached. Alarms provide parameters that you use to create or configure an alarm model. You can use the AWS IoT Events console or AWS IoT Events API to create or configure the alarm model. When you configure the alarm model, changes take effect as new data arrives.

Requirements

The following requirements apply when you create an alarm model.

- You can create an alarm model to monitor an input attribute in AWS IoT Events or an asset property in AWS IoT SiteWise.
  - If you choose to monitor an input attribute in AWS IoT Events, Create an input for models in AWS IoT Events before you create the alarm model.
  - If you choose to monitor an asset property, you must create an asset model in AWS IoT SiteWise before you create the alarm model.
- You must have an IAM role that allows your alarm to perform actions and access AWS resources. For more information, see Setting up permissions for AWS IoT Events.
- All the AWS resources that this tutorial uses must be in the same AWS Region.

Creating an alarm model (console)

The following shows you how to create an alarm model to monitor an AWS IoT Events attribute in the AWS IoT Events console.

1. Sign in to the AWS IoT Events console.
2. In the navigation pane, choose **Alarm models**.
3. On the **Alarm models** page, choose **Create alarm model**.
4. In the **Alarm model details** section, do the following:
   a. Enter a unique name.
   b. (Optional) Enter a description.
5. In the **Alarm target** section, do the following:

   > **Important**
   >
   > If you choose **AWS IoT SiteWise asset property**, you must have created an asset model in AWS IoT SiteWise.

   a. Choose **AWS IoT Events input attribute**.
   b. Choose the input.
   c. Choose the input attribute key. This input attribute is used as a key to create the alarm. AWS IoT Events routes inputs associated with this key to the alarm.

> **Important**
>
> If the input message payload does not contain this input attribute key, or if the key is not in the same JSON path specified in the key, then the message will fail the ingestion in AWS IoT Events.

6. In the **Threshold definitions** section, you define the input attribute, threshold value, and comparison operator that AWS IoT Events uses to change the state of the alarm.

   a. For **Input attribute**, choose the attribute that you want to monitor.

      Each time that this input attribute receives new data, it's evaluated to determine the state of the alarm.

   b. For **Operator**, choose the comparison operator. The operator compares your input attribute with the threshold value for your attribute.

      You can choose from these options:

      - **> greater than**
      - **>= greater than or equal to**
      - **< less than**
      - **<= less than or equal to**
      - **= equal to**
      - **!= not equal to**

   c. For threshold **Value**, enter a number or choose an attribute in AWS IoT Events inputs. AWS IoT Events compares this value with the value of the input attribute you choose.

   d. (Optional) For **Severity**, Use a number that your team understands to reflect the severity of this alarm.

7. (Optional) In the **Notification settings** section, configure notification settings for the alarm.

   You can add up to 10 notifications. For **Notification 1**, do the following:

   a. For **Protocol**, choose from the following options:

      - **Email & text** - The alarm sends an SMS notification and an email notification.
      - **Email** - The alarm sends an email notification.

- **Text** - The alarm sends an SMS notification.

b. For **Sender**, specify the email address that can send notifications about this alarm.

To add more email addresses to your sender list, choose **Add sender**.

c. (Optional) For **Recipient**, choose the recipient.

To add more users to your recipient list, choose **Add new user**. You must add new users to your IAM Identity Center store before you can add them to your alarm model. For more information, see Manage IAM Identity Center access of alarm recipients in AWS IoT Events.

d. (Optional) For **Additional custom message**, enter a message that describes what the alarm detects and what actions the recipients should take.

8. In the **Instance** section, you can enable or disable all alarm instances that are created based on this alarm model.

9. In the **Advanced settings** section, do the following:

a. For **Acknowledge flow**, you can enable or disable notifications.

- If you choose **Enabled**, you receive a notification when the alarm state changes. You must acknowledge the notification before the alarm state can return to normal.
- If you choose **Disabled**, no action is required. The alarm automatically changes to the normal state when the measurement returns to the specified range.

For more information, see Acknowledge flow.

b. For **Permissions**, choose one of the following options:

- You can **Create a new role from AWS policy templates** and AWS IoT Events automatically creates an IAM role for you.
- You can **Use an existing IAM role** that allows this alarm model to perform actions and access other AWS resources.

For more information, see Identity and access management for AWS IoT Events.

c. For **Additional notification settings**, you can edit your AWS Lambda function to manage alarm notifications. Choose one of the following options for your AWS Lambda function:

- **Create a new AWS Lambda function** - AWS IoT Events creates a new AWS Lambda function for you.

- **Use an existing AWS Lambda function** - Use an existing AWS Lambda function by choosing an AWS Lambda function name.

For more information about the possible actions, see AWS IoT Events working with other AWS services.

d. (Optional) For **Set state action**, you can add one or more AWS IoT Events actions to take when the alarm state changes.

10. (Optional) You can add **Tags** to manage your alarms. For more information, see Tagging your AWS IoT Events resources.
11. Choose **Create**.

Responding to alarms in AWS IoT Events

Responding to alarms effectively is an important aspect of managing IoT systems with AWS IoT Events. Explore various ways to configure and handle alarms, including: setting up notification channels, defining escalation procedures, and implementing automated response actions. Learn to create nuanced alarm conditions, prioritize alerts, and integrate with other AWS services to build a responsive alarm management system for your IoT applications.

If you enabled acknowledge flow, you receive notifications when the alarm state changes. To respond to the alarm, you can acknowledge, disable, enable, reset, or snooze the alarm.

Console

The following shows you how to respond to an alarm in the AWS IoT Events console.

1. Sign in to the AWS IoT Events console.
2. In the navigation pane, choose **Alarm models**.
3. Choose the target alarm model.
4. In the **List of alarms** section, choose the target alarm.
5. You can choose one of the following options from **Actions**:

   - **Acknowledge** - The alarm changes to the `ACKNOWLEDGED` state.
   - **Disable** - The alarm changes to the `DISABLED` state.
   - **Enable** - The alarm changes to the `NORMAL` state.

- **Reset** - The alarm changes to the `NORMAL` state.
- **Snooze**, and then do the following:
  1. Choose the **Snooze length** or enter a **Custom snooze length**.
  2. Choose **Save**.

  The alarm changes to the `SNOOZE_DISABLED` state

For more information about the alarm states, see Acknowledge flow.

API

To respond to one or more alarms, you can use the following AWS IoT Events API operations:

- BatchAcknowledgeAlarm
- BatchDisableAlarm
- BatchEnableAlarm
- BatchResetAlarm
- BatchSnoozeAlarm

Managing alarm notifications in AWS IoT Events

AWS IoT Events integrates with Lambda, offering custom event processing capabilities. This section explores how to use Lambda functions within your AWS IoT Events detector models, allowing you to execute complex logic, interact with external services, and implement sophisticated event handling.

AWS IoT Events uses a Lambda function to manage alarm notifications. You can use the Lambda function provided by AWS IoT Events or create a new one.

Topics

- Creating a Lambda function in AWS IoT Events
- Using the Lambda function provided by AWS IoT Events
- Manage IAM Identity Center access of alarm recipients in AWS IoT Events

Creating a Lambda function in AWS IoT Events

AWS IoT Events provides a Lambda function that enables alarms to send and receive email and SMS notifications.

Requirements

The following requirements apply when you create a Lambda function for alarms:

- If your alarm sends SMS notifications, ensure Amazon SNS is configured to deliver SMS messages.
  - For more information, see the following documentation:
    - Mobile text messaging with Amazon SNS and Origination identities for Amazon SNS SMS messages in the *Amazon Simple Notification Service Developer Guide*.
    - What is AWS End User Messaging SMS? in the *AWS SMS User Guide*.
- If your alarm sends email or SMS notifications, you must have an IAM role that allows AWS Lambda to work with Amazon SES and Amazon SNS.

  Example policy:

  JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ses:GetIdentityVerificationAttributes",
                "ses:SendEmail",
                "ses:VerifyEmailIdentity"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "sns:Publish",
                "sns:OptInPhoneNumber",
```

```
                "sns:CheckIfPhoneNumberIsOptedOut",
                "sms-voice:DescribeOptedOutNumbers"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Deny",
            "Action": "sns:Publish",
            "Resource": "arn:aws:sns:*:*:*"
        },
        {
          "Effect" : "Allow",
          "Action" : [
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents"
          ],
          "Resource" : "*"
        }
    ]
}
```

- You must choose the same AWS Region for both AWS IoT Events and AWS Lambda. For the list of supported Regions, see AWS IoT Events endpoints and quotas and AWS Lambda endpoints and quotas in the *Amazon Web Services General Reference*.

Deploy a Lambda function for AWS IoT Events using CloudFormation

This tutorial uses an CloudFormation template to deploy a Lambda function. This template automatically creates an IAM role that allows the Lambda function to work with Amazon SES and Amazon SNS.

The following shows you how to use the AWS Command Line Interface (AWS CLI) to create a CloudFormation stack.

1. In your device's terminal, run `aws --version` to check if you installed the AWS CLI. For more information, see Installing or updating to the latest version of the AWS CLI in the *AWS Command Line Interface User Guide*.
2. Run `aws configure list` to check if you configured the AWS CLI in the AWS Region that has all your AWS resources for this tutorial. For more information, see Set and view configuration settings using commands in the *AWS Command Line Interface User Guide*

3. Download the CloudFormation template, notificationLambda.template.yaml.zip.

> **Note**
>
> If you have difficulty downloading the file, the template is also available in the CloudFormation template.

4. Unzip the content and save it locally as `notificationLambda.template.yaml`.
5. Open a terminal on your device and navigate to the directory where you downloaded the `notificationLambda.template.yaml` file.
6. To create a CloudFormation stack, run the following command:

```
aws cloudformation create-stack --stack-name notificationLambda-stack --template-
body file://notificationLambda.template.yaml --capabilities CAPABILITY_IAM
```

You might modify this CloudFormation template to customize the Lambda function and its behavior.

> **Note**
>
> AWS Lambda retries function errors twice. If the function doesn't have enough capacity to handle all incoming requests, events might wait in the queue for hours or days to be sent to the function. You can configure an undelivered-message queue (DLQ) on the function to capture events that weren't successfully processed. For more information, see Asynchronous invocation in the *AWS Lambda Developer Guide*.

You can also create or configure the stack in the CloudFormation console. For more information, see Working with stacks, in the *AWS CloudFormation User Guide*.

Creating a custom Lambda function for AWS IoT Events

You can create a Lambda function or modify the one provided by AWS IoT Events.

The following requirements apply when you create a custom Lambda function.

- Add permissions that allow your Lambda function to perform specified actions and access AWS resources.

- If you use the Lambda function provided by AWS IoT Events, make sure that you choose the Python 3.7 runtime.

Example Lambda function:

```
import boto3
import json
import logging
import datetime
logger = logging.getLogger()
logger.setLevel(logging.INFO)
ses = boto3.client('ses')
sns = boto3.client('sns')
def check_value(target):
  if target:
    return True
  return False

# Check whether email is verified. Only verified emails are allowed to send emails to
 or from.
def check_email(email):
  if not check_value(email):
    return False
  result = ses.get_identity_verification_attributes(Identities=[email])
  attr = result['VerificationAttributes']
  if (email not in attr or attr[email]['VerificationStatus'] != 'Success'):
      logging.info('Verification email for {} sent. You must have all the emails
 verified before sending email.'.format(email))
      ses.verify_email_identity(EmailAddress=email)
      return False
  return True

# Check whether the phone holder has opted out of receiving SMS messages from your
 account
def check_phone_number(phone_number):
  try:
    result = sns.check_if_phone_number_is_opted_out(phoneNumber=phone_number)
    if (result['isOptedOut']):
        logger.info('phoneNumber {} is not opt in of receiving SMS messages. Phone
 number must be opt in first.'.format(phone_number))
        return False
    return True
  except Exception as e:
```

```
    logging.error('Your phone number {} must be in E.164 format in SSO. Exception
 thrown: {}'.format(phone_number, e))
    return False

def check_emails(emails):
  result = True
  for email in emails:
      if not check_email(email):
          result = False
  return result

def lambda_handler(event, context):
  logging.info('Received event: ' + json.dumps(event))
  nep = json.loads(event.get('notificationEventPayload'))
  alarm_state = nep['alarmState']
  default_msg = 'Alarm ' + alarm_state['stateName'] + '\n'
  timestamp =
 datetime.datetime.utcfromtimestamp(float(nep['stateUpdateTime'])/1000).strftime('%Y-
%m-%d %H:%M:%S')
  alarm_msg = "{} {} {} at {} UTC ".format(nep['alarmModelName'], nep.get('keyValue',
 'Singleton'), alarm_state['stateName'], timestamp)
  default_msg += 'Sev: ' + str(nep['severity']) + '\n'
  if (alarm_state['ruleEvaluation']):
    property = alarm_state['ruleEvaluation']['simpleRule']['inputProperty']
    default_msg += 'Current Value: ' + str(property) + '\n'
    operator = alarm_state['ruleEvaluation']['simpleRule']['operator']
    threshold = alarm_state['ruleEvaluation']['simpleRule']['threshold']
    alarm_msg += '({} {} {})'.format(str(property), operator, str(threshold))
  default_msg += alarm_msg + '\n'

  emails = event.get('emailConfigurations', [])
  logger.info('Start Sending Emails')
  for email in emails:
    from_adr = email.get('from')
    to_adrs = email.get('to', [])
    cc_adrs = email.get('cc', [])
    bcc_adrs = email.get('bcc', [])
    msg = default_msg + '\n' + email.get('additionalMessage', '')
    subject = email.get('subject', alarm_msg)
    fa_ver = check_email(from_adr)
    tas_ver = check_emails(to_adrs)
    ccas_ver = check_emails(cc_adrs)
    bccas_ver = check_emails(bcc_adrs)
    if (fa_ver and tas_ver and ccas_ver and bccas_ver):
```

```
      ses.send_email(Source=from_adr,
                     Destination={'ToAddresses': to_adrs, 'CcAddresses': cc_adrs,
'BccAddresses': bcc_adrs},
                     Message={'Subject': {'Data': subject}, 'Body': {'Text': {'Data':
msg}}})
      logger.info('Emails have been sent')

 logger.info('Start Sending SNS message to SMS')
 sns_configs = event.get('smsConfigurations', [])
 for sns_config in sns_configs:
   sns_msg = default_msg + '\n' + sns_config.get('additionalMessage', '')
   phone_numbers = sns_config.get('phoneNumbers', [])
   sender_id = sns_config.get('senderId')
   for phone_number in phone_numbers:
       if check_phone_number(phone_number):
         if check_value(sender_id):
           sns.publish(PhoneNumber=phone_number, Message=sns_msg,
MessageAttributes={'AWS.SNS.SMS.SenderID':{'DataType': 'String','StringValue':
sender_id}})
         else:
           sns.publish(PhoneNumber=phone_number, Message=sns_msg)
         logger.info('SNS messages have been sent')
```

For more information, see What is AWS Lambda? in the *AWS Lambda Developer Guide*.

CloudFormation template

Use the following CloudFormation template to create your Lambda function.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: 'Notification Lambda for Alarm Model'
Resources:
  NotificationLambdaRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      Path: "/"
      ManagedPolicyArns:
        - 'arn:aws:iam::aws:policy/AWSLambdaExecute'
```

```
      Policies:
        - PolicyName: "NotificationLambda"
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              - Effect: "Allow"
                Action:
                  - "ses:GetIdentityVerificationAttributes"
                  - "ses:SendEmail"
                  - "ses:VerifyEmailIdentity"
                Resource: "*"
              - Effect: "Allow"
                Action:
                  - "sns:Publish"
                  - "sns:OptInPhoneNumber"
                  - "sns:CheckIfPhoneNumberIsOptedOut"
                  - "sms-voice:DescribeOptedOutNumbers"
                Resource: "*"
              - Effect: "Deny"
                Action:
                  - "sns:Publish"
                Resource: "arn:aws:sns:*:*:*"
  NotificationLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      Role: !GetAtt NotificationLambdaRole.Arn
      Runtime: python3.7
      Handler: index.lambda_handler
      Timeout: 300
      MemorySize: 3008
      Code:
        ZipFile: |
          import boto3
          import json
          import logging
          import datetime
          logger = logging.getLogger()
          logger.setLevel(logging.INFO)
          ses = boto3.client('ses')
          sns = boto3.client('sns')
          def check_value(target):
            if target:
              return True
            return False
```

```
        # Check whether email is verified. Only verified emails are allowed to send
emails to or from.
        def check_email(email):
          if not check_value(email):
            return False
          result = ses.get_identity_verification_attributes(Identities=[email])
          attr = result['VerificationAttributes']
          if (email not in attr or attr[email]['VerificationStatus'] != 'Success'):
              logging.info('Verification email for {} sent. You must have all the
emails verified before sending email.'.format(email))
              ses.verify_email_identity(EmailAddress=email)
              return False
          return True

        # Check whether the phone holder has opted out of receiving SMS messages from
your account
        def check_phone_number(phone_number):
          try:
            result = sns.check_if_phone_number_is_opted_out(phoneNumber=phone_number)
            if (result['isOptedOut']):
                logger.info('phoneNumber {} is not opt in of receiving SMS messages.
Phone number must be opt in first.'.format(phone_number))
                return False
            return True
          except Exception as e:
            logging.error('Your phone number {} must be in E.164 format in SSO.
Exception thrown: {}'.format(phone_number, e))
            return False

        def check_emails(emails):
          result = True
          for email in emails:
              if not check_email(email):
                  result = False
          return result

        def lambda_handler(event, context):
          logging.info('Received event: ' + json.dumps(event))
          nep = json.loads(event.get('notificationEventPayload'))
          alarm_state = nep['alarmState']
          default_msg = 'Alarm ' + alarm_state['stateName'] + '\n'
```

```
            timestamp =
 datetime.datetime.utcfromtimestamp(float(nep['stateUpdateTime'])/1000).strftime('%Y-
%m-%d %H:%M:%S')
            alarm_msg = "{} {} {} at {} UTC ".format(nep['alarmModelName'],
 nep.get('keyValue', 'Singleton'), alarm_state['stateName'], timestamp)
            default_msg += 'Sev: ' + str(nep['severity']) + '\n'
            if (alarm_state['ruleEvaluation']):
              property = alarm_state['ruleEvaluation']['simpleRule']['inputProperty']
              default_msg += 'Current Value: ' + str(property) + '\n'
              operator = alarm_state['ruleEvaluation']['simpleRule']['operator']
              threshold = alarm_state['ruleEvaluation']['simpleRule']['threshold']
              alarm_msg += '({} {} {})'.format(str(property), operator, str(threshold))
            default_msg += alarm_msg + '\n'

            emails = event.get('emailConfigurations', [])
            logger.info('Start Sending Emails')
            for email in emails:
              from_adr = email.get('from')
              to_adrs = email.get('to', [])
              cc_adrs = email.get('cc', [])
              bcc_adrs = email.get('bcc', [])
              msg = default_msg + '\n' + email.get('additionalMessage', '')
              subject = email.get('subject', alarm_msg)
              fa_ver = check_email(from_adr)
              tas_ver = check_emails(to_adrs)
              ccas_ver = check_emails(cc_adrs)
              bccas_ver = check_emails(bcc_adrs)
              if (fa_ver and tas_ver and ccas_ver and bccas_ver):
                ses.send_email(Source=from_adr,
                               Destination={'ToAddresses': to_adrs, 'CcAddresses':
 cc_adrs, 'BccAddresses': bcc_adrs},
                               Message={'Subject': {'Data': subject}, 'Body': {'Text':
 {'Data': msg}}})
                logger.info('Emails have been sent')

            logger.info('Start Sending SNS message to SMS')
            sns_configs = event.get('smsConfigurations', [])
            for sns_config in sns_configs:
              sns_msg = default_msg + '\n' + sns_config.get('additionalMessage', '')
              phone_numbers = sns_config.get('phoneNumbers', [])
              sender_id = sns_config.get('senderId')
              for phone_number in phone_numbers:
                  if check_phone_number(phone_number):
                    if check_value(sender_id):
```

```
                    sns.publish(PhoneNumber=phone_number, Message=sns_msg,
MessageAttributes={'AWS.SNS.SMS.SenderID':{'DataType': 'String','StringValue':
sender_id}})
                else:
                  sns.publish(PhoneNumber=phone_number, Message=sns_msg)
                logger.info('SNS messages have been sent')
```

Using the Lambda function provided by AWS IoT Events

With alarm notifications, you can use the Lambda function provided by AWS IoT Events for managing alarm notifications.

The following requirements apply when you use the Lambda function provided by AWS IoT Events to manage your alarm notifications:

- You must verify the email address that sends the email notifications in Amazon Simple Email Service (Amazon SES). For more information, see Verifying an email address identity, in the *Amazon Simple Email Service Developer Guide*.

  If you receive a verification link, click the link to verify your email address. You might also check your spam folder for a verification email.
- If your alarm sends SMS notifications, you must use E.164 international phone number formatting for phone numbers. This format contains `+<country-calling-code><area-code><phone-number>`.

  Example phone numbers:

Country	Local phone number	E.164 formatted number
United States	206-555-0100	+12065550100
United Kingdom	020-1234-1234	+442012341234
Lithuania	8+601+12345	+37060112345

  To find a country calling code, go to countrycode.org.

  The Lambda function provided by AWS IoT Events checks if you use E.164 formatted phone numbers. However, it doesn't verify the phone numbers. If you ensure that you entered accurate

phone numbers but didn't receive SMS notifications, you might contact the phone carriers. The carriers may block the messages.

Manage IAM Identity Center access of alarm recipients in AWS IoT Events

AWS IoT Events uses AWS IAM Identity Center to manage the SSO access of alarms recipients. Implementing IAM Identity Center for AWS IoT Events notification recipients can enhance security and user experience. To enable the alarm to send notifications to the recipients, you must enable IAM Identity Center and add recipients to your IAM Identity Center store. For more information, see Add Users in *AWS IAM Identity Center User Guide*.

Important

- You must choose the same AWS Region for AWS IoT Events, AWS Lambda, and IAM Identity Center.
- AWS Organizations only supports one IAM Identity Center Region at a time. If you want to make IAM Identity Center available in a different Region, you must first delete your current IAM Identity Center configuration. For more information, see IAM Identity Center Region Data in *AWS IAM Identity Center User Guide*.

Security in AWS IoT Events

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to AWS IoT Events, see AWS services in scope by compliance program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using AWS IoT Events. The following topics show you how to configure AWS IoT Events to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your AWS IoT Events resources.

Topics

- Identity and access management for AWS IoT Events
- Monitoring AWS IoT Events to maintain reliability, availability, and performance
- Compliance validation for AWS IoT Events
- Resilience in AWS IoT Events
- Infrastructure security in AWS IoT Events

Identity and access management for AWS IoT Events

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in)

and *authorized* (have permissions) to use AWS IoT Events resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- More about identity and access management
- How AWS IoT Events works with IAM
- AWS IoT Events identity-based policy examples
- Cross-service confused deputy prevention for AWS IoT Events
- Troubleshoot AWS IoT Events identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshoot AWS IoT Events identity and access)
- **Service administrator** - determine user access and submit permission requests (see How AWS IoT Events works with IAM)
- **IAM administrator** - write policies to manage access (see AWS IoT Events identity-based policy examples)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

More about identity and access management

For more information about identity and access management for AWS IoT Events, continue to the following pages:

- How AWS IoT Events works with IAM
- Troubleshoot AWS IoT Events identity and access

How AWS IoT Events works with IAM

Before you use IAM to manage access to AWS IoT Events, you should understand what IAM features are available to use with AWS IoT Events. To get a high-level view of how AWS IoT Events and other AWS services work with IAM, see AWS services that work with IAM in the *IAM User Guide*.

Topics

- AWS IoT Events identity-based policies
- AWS IoT Events resource-based policies
- Authorization based on AWS IoT Events tags
- AWS IoT Events IAM roles

AWS IoT Events identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. AWS IoT Events supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Actions

The `Action` element of an IAM identity-based policy describes the specific action or actions that will be allowed or denied by the policy. Policy actions usually have the same name as the associated AWS API operation. The action is used in a policy to grant permissions to perform the associated operation.

Policy actions in AWS IoT Events use the following prefix before the action: `iotevents:`. For example, to grant someone permission to create an AWS IoT Events input with the AWS IoT Events `CreateInput` API operation, you include the `iotevents:CreateInput` action in their policy.

To grant someone permission to send an input with the AWS IoT Events `BatchPutMessage` API operation, you include the `iotevents-data:BatchPutMessage` action in their policy. Policy statements must include either an `Action` or `NotAction` element. AWS IoT Events defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "iotevents:action1",
      "iotevents:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Describe`, include the following action:

```
"Action": "iotevents:Describe*"
```

To see a list of AWS IoT Events actions, see Actions Defined by AWS IoT Events in the *IAM User Guide*.

Resources

The `Resource` element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. You specify a resource using an ARN or using the wildcard (*) to indicate that the statement applies to all resources.

The AWS IoT Events detector model resource has the following ARN:

```
arn:${Partition}:iotevents:${Region}:${Account}:detectorModel/${detectorModelName}
```

For more information about the format of ARNs, see Identify AWS resources with Amazon Resource Names (ARNs).

For example, to specify the `Foobar` detector model in your statement, use the following ARN:

```
"Resource": "arn:aws:iotevents:us-east-1:123456789012:detectorModel/Foobar"
```

To specify all instances that belong to a specific account, use the wildcard (*):

```
"Resource": "arn:aws:iotevents:us-east-1:123456789012:detectorModel/*"
```

Some AWS IoT Events actions, such as those for creating resources, cannot be performed on a specific resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

Some AWS IoT Events API actions involve multiple resources. For example, `CreateDetectorModel` references inputs in its condition statements, so a user must have permissions to use the input and the detector model. To specify multiple resources in a single statement, separate the ARNs with commas.

```
"Resource": [
      "resource1",
      "resource2"
```

To see a list of AWS IoT Events resource types and their ARNs, see Resources Defined by AWS IoT Events in the *IAM User Guide*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by AWS IoT Events.

Condition keys

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can build conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant a user permission to access a resource only if it is tagged with their user name. For more information, see IAM policy elements: Variables and tags in the *IAM User Guide*.

AWS IoT Events does not provide any service-specific condition keys, but it does support using some global condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*."

Examples

To view examples of AWS IoT Events identity-based policies, see AWS IoT Events identity-based policy examples.

AWS IoT Events resource-based policies

AWS IoT Events does not support resource-based policies." To view an example of a detailed resource-based policy page, see https://docs.aws.amazon.com/lambda/latest/dg/access-control-resource-based.html.

Authorization based on AWS IoT Events tags

You can attach tags to AWS IoT Events resources or pass tags in a request to AWS IoT Events. To control access based on tags, you provide tag information in the condition element of a policy using the `iotevents:ResourceTag/`*`key-name`*, `aws:RequestTag/`*`key-name`*, or `aws:TagKeys` condition keys. For more information about tagging AWS IoT Events resources, see Tagging your AWS IoT Events resources.

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see View AWS IoT Events inputs based on tags.

AWS IoT Events IAM roles

An IAM role is an entity within your AWS account that has specific permissions.

Using temporary credentials with AWS IoT Events

You can use temporary credentials to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS Security Token Service (AWS STS) API operations such as AssumeRole or GetFederationToken.

AWS IoT Events does not support using temporary credentials.

Service-linked roles

Service-linked roles allow AWS services to access resources in other services to complete an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view but not edit the permissions for service-linked roles.

AWS IoT Events does not support service-linked roles.

Service roles

This feature allows a service to assume a service role on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

AWS IoT Events supports service roles.

AWS IoT Events identity-based policy examples

By default, users and roles don't have permission to create or modify AWS IoT Events resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating policies on the JSON tab in the *IAM User Guide*.

Topics

- Policy best practices
- Using the AWS IoT Events console
- Allow users to view their own permissions in AWS IoT Events
- Access one AWS IoT Events input
- View AWS IoT Events inputs based on tags

Policy best practices

Identity-based policies are very powerful. They determine whether someone can create, access, or delete AWS IoT Events resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get Started Using AWS Managed Policies** – To start using AWS IoT Events quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get started using permissions with AWS managed policies in the *IAM User Guide*.

- **Grant Least Privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant least privilege in the *IAM User Guide*.
- **Enable MFA for Sensitive Operations** – For extra security, require users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.
- **Use Policy Conditions for Extra Security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.

Using the AWS IoT Events console

To access the AWS IoT Events console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the AWS IoT Events resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

To ensure that those entities can still use the AWS IoT Events console, also attach the following AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iotevents:BatchPutMessage",
                "iotevents:BatchUpdateDetector",
                "iotevents:CreateDetectorModel",
```

```
                "iotevents:CreateInput",
                "iotevents:DeleteDetectorModel",
                "iotevents:DeleteInput",
                "iotevents:DescribeDetector",
                "iotevents:DescribeDetectorModel",
                "iotevents:DescribeInput",
                "iotevents:DescribeLoggingOptions",
                "iotevents:ListDetectorModelVersions",
                "iotevents:ListDetectorModels",
                "iotevents:ListDetectors",
                "iotevents:ListInputs",
                "iotevents:ListTagsForResource",
                "iotevents:PutLoggingOptions",
                "iotevents:TagResource",
                "iotevents:UntagResource",
                "iotevents:UpdateDetectorModel",
                "iotevents:UpdateInput",
                "iotevents:UpdateInputRouting"
            ],
            "Resource": "arn:aws:iotevents:us-
east-1:123456789012:detectorModel/your-detector-model-name",
    "Resource": "arn:aws:iotevents:us-east-1:123456789012:input/your-input-name"
        }
    ]
}
```

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that you're trying to perform.

Allow users to view their own permissions in AWS IoT Events

This example shows how you might create a policy that allows users to view the inline and managed policies that are attached to their user identity. Allowing users to view their own IAM permissions is useful for security awareness and self-service capabilities. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": [
                "arn:aws:iam::*:user/${aws:username}"
            ]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Access one AWS IoT Events input

Granular access control to AWS IoT Events inputs is important for maintaining security in multi-user or multi-team environments. This section shows how to create IAM policies that grant access to specific AWS IoT Events inputs while restricting access to others.

In this example, you can grant a user in your AWS account access to one of your AWS IoT Events inputs, `exampleInput`. You also can allow the user to add, update, and delete inputs.

The policy grants the `iotevents:ListInputs`, `iotevents:DescribeInput`, `iotevents:CreateInput`, `iotevents:DeleteInput`, and `iotevents:UpdateInput` permissions to the user. For an example walkthrough for the Amazon Simple Storage Service (Amazon S3) that grants permissions to users and tests them using the console, see Controlling access to a bucket with user policies.

JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"ListInputsInConsole",
         "Effect":"Allow",
         "Action":[
            "iotevents:ListInputs"
         ],
         "Resource":"arn:aws:iotevents:us-east-2:123456789012:input/*"
      },
      {
         "Sid":"ViewSpecificInputInfo",
         "Effect":"Allow",
         "Action":[
            "iotevents:DescribeInput"
         ],
         "Resource":"arn:aws:iotevents:us-east-1:123456789012:input/inputName"
      },
      {
         "Sid":"ManageInputs",
         "Effect":"Allow",
         "Action":[
            "iotevents:CreateInput",
            "iotevents:DeleteInput",
            "iotevents:DescribeInput",
            "iotevents:ListInputs",
            "iotevents:UpdateInput"
         ],
         "Resource":"arn:aws:iotevents:us-east-1:123456789012:input/*"
      }
   ]
}
```

View AWS IoT Events inputs based on tags

Tags help you organize AWS IoT Events resources. You can use conditions in your identity-based policy to control access to AWS IoT Events resources based on tags. This example shows how you might create a policy that allows viewing an *input*. However, permission is granted only if the *input* tag `Owner` has the value of that user's user name. This policy also grants the permissions necessary to complete this action on the console.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ListInputsInConsole",
            "Effect": "Allow",
            "Action": "iotevents:ListInputs",
            "Resource": "*"
        },
        {
            "Sid": "ViewInputsIfOwner",
            "Effect": "Allow",
            "Action": "iotevents:ListInputs",
            "Resource": "arn:aws:iotevents:*:*:input/*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/Owner": "${aws:username}"}
            }
        }
    ]
}
```

You can attach this policy to the users in your account. If a user named `richard-roe` attempts to view an AWS IoT Events *input*, the *input* must be tagged `Owner=richard-roe` or `owner=richard-roe`. Otherwise he is denied access. The condition tag key `Owner` matches both `Owner` and `owner` because condition key names are not case-sensitive. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.

Cross-service confused deputy prevention for AWS IoT Events

Note

- The AWS IoT Events service only allows you to use roles to start actions in the same account in which a resource was created. This helps prevent a confused deputy attack in AWS IoT Events.
- This page serves as a reference for you to see how the confused deputy issue works and can be prevented in the event that cross account resources were allowed in the AWS IoT Events service.

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem.

Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that AWS IoT Events gives another service to the resource. If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions. If you use both global condition context keys and the `aws:SourceArn` value contains the account ID, the `aws:SourceAccount` value and the account in the `aws:SourceArn` value must use the same account ID when used in the same policy statement.

Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use. The value of `aws:SourceArn` must be the Detector Model or Alarm model associated with the `sts:AssumeRole` request.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn`

global context condition key with wildcards (*) for the unknown portions of the ARN. For example, `arn:aws:iotevents:*:123456789012:*`.

The following examples show how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in AWS IoT Events to prevent the confused deputy problem.

Topics

- Example: Secure access to an AWS IoT Events detector model
- Example: Secure access to an AWS IoT Events alarm model
- Example: Access an AWS IoT Events resource in a specified region
- Example: Configure logging options for AWS IoT Events

Example: Secure access to an AWS IoT Events detector model

This example demonstrates how to create an IAM policy that securely grants access to a specific detector model in AWS IoT Events. The policy uses conditions to ensure that only the specified AWS account and AWS IoT Events service can assume the role, adding an extra layer of security. In this example, the role can only access the detector model named *WindTurbine01*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "iotevents.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                "aws:SourceAccount": "123456789012"
                },
                "ArnEquals": {
                "aws:SourceArn": "arn:aws:iotevents:us-
east-1:123456789012:detectorModel/WindTurbine01"
```

```
                }
            }
        }
    ]
}
```

Example: Secure access to an AWS IoT Events alarm model

This example demonstrates how to create an IAM policy that allows AWS IoT Events to securely access alarm models. The policy uses conditions to ensure that only the specified AWS account and AWS IoT Events service can assume the role.

In this example, the role can access any alarm model within the specified AWS account, as indicated by the * wildcard in the alarm model ARN. The `aws:SourceAccount` and `aws:SourceArn` conditions work together to prevent the confused deputy problem.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "iotevents.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                "aws:SourceAccount": "123456789012"
                },
                "ArnEquals": {
                "aws:SourceArn": "arn:aws:iotevents:us-
east-1:123456789012:alarmModel/*"
                }
            }
        }
    ]
```

```
}
```

Example: Access an AWS IoT Events resource in a specified region

This example demonstrates how to configure an IAM role to access AWS IoT Events resources in a specific AWS region. By using region-specific ARNs in your IAM policies, you can restrict access to AWS IoT Events resources across different geographical areas. This approach can help maintain security and compliance in multi-region deployments. The region in this example is *us-east-1*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "iotevents.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                "aws:SourceAccount": "123456789012"
                },
                "ArnEquals": {
                "aws:SourceArn": "arn:aws:iotevents:us-east-1:123456789012:*"
                }
            }
        }
    ]
}
```

Example: Configure logging options for AWS IoT Events

Proper logging is important for monitoring, debugging, and auditing your AWS IoT Events applications. This section provides an overview of logging options available in AWS IoT Events.

This example demonstrates how to configure an IAM role that allows AWS IoT Events to log data to CloudWatch Logs. The use of wildcards (*) in the resource ARN allows for comprehensive logging across your AWS IoT Events infrastructure.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "iotevents.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                "aws:SourceAccount": "123456789012"
                },
                "ArnEquals": {
                "aws:SourceArn": "arn:aws:iotevents:us-east-1:123456789012:*"
                }
            }
        }
    ]
}
```

Troubleshoot AWS IoT Events identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS IoT Events and IAM.

Topics

- I am not authorized to perform an action in AWS IoT Events
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my AWS IoT Events resources

I am not authorized to perform an action in AWS IoT Events

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a *`input`* but does not have `iotevents:`*`ListInputs`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 iotevents:ListInputs on resource: my-example-input
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *`my-example-input`* resource using the `iotevents:`*`ListInput`* action.

I am not authorized to perform `iam:PassRole`

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AWS IoT Events.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS IoT Events. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my AWS IoT Events resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

Consult the following topics to determine your best options:

- To learn whether AWS IoT Events supports these features, see How AWS IoT Events works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Monitoring AWS IoT Events to maintain reliability, availability, and performance

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT Events and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. Before you start monitoring AWS IoT Events, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- Which resources will you monitor?
- How often will you monitor these resources?
- Which monitoring tools will you use?
- Who will perform the monitoring tasks?

- Who should be notified when something goes wrong?

The next step is to establish a baseline for normal AWS IoT Events performance in your environment, by measuring performance at various times and under different load conditions. As you monitor AWS IoT Events, store historical monitoring data so that you can compare it with current performance data, identify normal performance patterns and performance anomalies, and devise methods to address issues.

For example, if you're using Amazon EC2, you can monitor CPU utilization, disk I/O, and network utilization for your instances. When performance falls outside your established baseline, you might need to reconfigure or optimize the instance to reduce CPU utilization, improve disk I/O, or reduce network traffic.

Topics

- Available tools to monitor AWS IoT Events
- Monitoring AWS IoT Events with Amazon CloudWatch
- Logging AWS IoT Events API calls with AWS CloudTrail

Available tools to monitor AWS IoT Events

AWS provides various tools that you can use to monitor AWS IoT Events. You can configure some of these tools to do the monitoring for you, while some of the tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

Automated monitoring tools

You can use the following automated monitoring tools to watch AWS IoT Events and report when something is wrong:

- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see Using Amazon CloudWatch dashboards in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log-processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see Working with CloudTrail log files in the *AWS CloudTrail User Guide*.

Manual monitoring tools

Another important part of monitoring AWS IoT Events involves manually monitoring those items that the CloudWatch alarms don't cover. The AWS IoT Events, CloudWatch, and other AWS console dashboards provide an at-a-glance view of the state of your AWS environment. We recommend that you also check the log files on AWS IoT Events.

- The AWS IoT Events console shows:
  - Detector models
  - Detectors
  - Inputs
  - Settings
- The CloudWatch home page shows:
  - Current alarms and status
  - Graphs of alarms and resources
  - Service health status

  In addition, you can use CloudWatch to do the following:
  - Create Creating a CloudWatch dashboard to monitor the services you care about
  - Graph metric data to troubleshoot issues and discover trends
  - Search and browse all your AWS resource metrics
  - Create and edit alarms to be notified of problems

Monitoring AWS IoT Events with Amazon CloudWatch

When you develop or debug an AWS IoT Events detector model, you need to know what AWS IoT Events is doing, and any errors it encounters. Amazon CloudWatch monitors your AWS resources and the applications you run on AWS in real time. With CloudWatch, you gain systemwide visibility into resource use, application performance, and operational health. Enable Amazon CloudWatch logging when developing AWS IoT Events detector models has information on how to enable CloudWatch logging for AWS IoT Events. To generate logs like the one shown below you must set the **Level of verbosity** to 'Debug' and provide one or more **Debug Targets** that is a **Detector Model Name** and an optional **KeyValue**.

The following example shows a CloudWatch DEBUG level log entry generated by AWS IoT Events.

```
{
  "timestamp": "2019-03-15T15:56:29.412Z",
  "level": "DEBUG",
  "logMessage": "Summary of message evaluation",
  "context": "MessageEvaluation",
  "status": "Success",
  "messageId": "SensorAggregate_2th846h",
  "keyValue": "boiler_1",
  "detectorModelName": "BoilerAlarmDetector",
  "initialState": "high_temp_alarm",
  "initialVariables": {
    "high_temp_count": 1,
    "high_pressure_count": 1
  },
  "finalState": "no_alarm",
  "finalVariables": {
    "high_temp_count": 0,
    "high_pressure_count": 0
  },
  "message": "{ \"temp\": 34.9, \"pressure\": 84.5}",
  "messageType": "CUSTOMER_MESSAGE",
  "conditionEvaluationResults": [
    {
      "result": "True",
      "eventName": "alarm_cleared",
      "state": "high_temp_alarm",
      "lifeCycle": "OnInput",
      "hasTransition": true
    },
    {
      "result": "Skipped",
      "eventName": "alarm_escalated",
      "state": "high_temp_alarm",
      "lifeCycle": "OnInput",
      "hasTransition": true,
      "resultDetails": "Skipped due to transition from alarm_cleared event"
    },
    {
      "result": "True",
      "eventName": "should_recall_technician",
      "state": "no_alarm",
      "lifeCycle": "OnEnter",
      "hasTransition": true
```

```
    }
  ]
}
```

Logging AWS IoT Events API calls with AWS CloudTrail

AWS IoT Events is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS IoT Events. CloudTrail captures all API calls for AWS IoT Events as events, including calls from the AWS IoT Events console and from code calls to the AWS IoT Events APIs.

If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS IoT Events. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT Events, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

AWS IoT Events information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS IoT Events, that activity is recorded in a CloudTrail event with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Working with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for AWS IoT Events, create a trail. A trail enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act on the event data collected in CloudTrail logs. For more information, see:

- Creating a trail for your AWS account
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element. AWS IoT Events actions are documented in the AWS IoT Events API reference.

Understanding AWS IoT Events log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. AWS CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files are not an ordered stack trace of the public API calls, so they don't appear in any specific order.

When CloudTrail logging is enabled in your AWS account, most API calls made to AWS IoT Events actions are tracked in CloudTrail log files where they are written with other AWS service records. CloudTrail determines when to create and write to a new file based on a time period and file size.

Every log entry contains information about who generated the request. The user identity information in the log entry helps you determine the following:

- Whether the request was made with root or IAM user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

You can store your log files in your Amazon S3 bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted with Amazon S3 server-side encryption (SSE).

To be notified upon log file delivery, you can configure CloudTrail to publish Amazon SNS notifications when new log files are delivered. For more information, see Configuring Amazon SNS notifications for CloudTrail.

You can also aggregate AWS IoT Events log files from multiple AWS Regions and multiple AWS accounts into a single Amazon S3 bucket.

For more information, see Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts.

Example: DescribeDetector action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `DescribeDetector` action.


```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE",
    "arn": "arn:aws:sts::123456789012:assumed-role/Admin/bertholt-brecht",
    "accountId": "123456789012",
    "accessKeyId": "access-key-id",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-08T18:53:58Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/Admin",
        "accountId": "123456789012",
        "userName": "Admin"
      }
    }
  },
  "eventTime": "2019-02-08T19:02:44Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "DescribeDetector",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-cli/1.15.65 Python/3.7.1 Darwin/16.7.0 botocore/1.10.65",
  "requestParameters": {
    "detectorModelName": "pressureThresholdEventDetector-brecht",
    "keyValue": "1"
  },
```

```
  "responseElements": null,
  "requestID": "00f41283-ea0f-4e85-959f-bee37454627a",
  "eventID": "5eb0180d-052b-49d9-a289-0eb8d08d4c27",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: CreateDetectorModel action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `CreateDetectorModel` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-Lambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEvents-RoleForIotEvents-
ABC123DEF456/IotEvents-Lambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABC123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABC123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:54:43Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "CreateDetectorModel",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
```

```
    "detectorModelName": "myDetectorModel",
    "key": "HIDDEN_DUE_TO_SECURITY_REASONS",
    "roleArn": "arn:aws:iam::123456789012:role/events_action_execution_role"
  },
  "responseElements": null,
  "requestID": "cecfbfa1-e452-4fa6-b86b-89a89f392b66",
  "eventID": "8138d46b-50a3-4af0-9c5e-5af5ef75ea55",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: CreateInput action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `CreateInput` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-Lambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABC123DEF456/IotEvents-Lambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABC123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABC123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:54:43Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "CreateInput",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
```

```
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "inputName": "batchputmessagedetectorupdated",
    "inputDescription": "batchputmessagedetectorupdated"
  },
  "responseElements": null,
  "requestID": "fb315af4-39e9-4114-94d1-89c9183394c1",
  "eventID": "6d8cf67b-2a03-46e6-bbff-e113a7bded1e",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: DeleteDetectorModel action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `DeleteDetectorModel` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:54:11Z",
  "eventSource": "iotevents.amazonaws.com",
```

```
  "eventName": "DeleteDetectorModel",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "detectorModelName": "myDetectorModel"
  },
  "responseElements": null,
  "requestID": "149064c1-4e24-4160-a5b2-1065e63ee2e4",
  "eventID": "7669db89-dcc0-4c42-904b-f24b764dd808",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: DeleteInput action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `DeleteInput` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:54:38Z",
  "eventSource": "iotevents.amazonaws.com",
```

```
  "eventName": "DeleteInput",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "errorCode": "ResourceNotFoundException",
  "errorMessage": "Input of name: NoSuchInput not found",
  "requestParameters": {
    "inputName": "NoSuchInput"
  },
  "responseElements": null,
  "requestID": "ce6d28ac-5baf-423d-a5c3-afd009c967e3",
  "eventID": "be0ef01d-1c28-48cd-895e-c3ff3172c08e",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: DescribeDetectorModel action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `DescribeDetectorModel` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AAKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
```

```
    }
  },
  "eventTime": "2019-02-07T23:54:20Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "DescribeDetectorModel",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "detectorModelName": "myDetectorModel"
  },
  "responseElements": null,
  "requestID": "18a11622-8193-49a9-85cb-1fa6d3929394",
  "eventID": "1ad80ff8-3e2b-4073-ac38-9cb3385beb04",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: DescribeInput action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `DescribeInput` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AAKIAI44QH8DHBEXAMPLE",
    "sessionContext": {

      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
```

```
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:56:09Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "DescribeInput",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "inputName": "input_createinput"
  },
  "responseElements": null,
  "requestID": "3af641fa-d8af-41c9-ba77-ac9c6260f8b8",
  "eventID": "bc4e6cc0-55f7-45c1-b597-ec99aa14c81a",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: DescribeLoggingOptions action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `DescribeLoggingOptions` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
```

```
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:53:23Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "DescribeLoggingOptions",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": null,
  "responseElements": null,
  "requestID": "b624b6c5-aa33-41d8-867b-025ec747ee8f",
  "eventID": "9c7ce626-25c8-413a-96e7-92b823d6c850",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: ListDetectorModels action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `ListDetectorModels` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
```

```
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:53:23Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "ListDetectorModels",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "nextToken": "CkZEZXRlY3Rvck1vZGVsMl9saXN0ZGV0ZWN0b3Jtb2RlbHN0ZXN0X2VlOWJkZTk1YT",
    "maxResults": 3
  },
  "responseElements": null,
  "requestID": "6d70f262-da95-4bb5-94b4-c08369df75bb",
  "eventID": "2d01a25c-d5c7-4233-99fe-ce1b8ec05516",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: ListDetectorModelVersions action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `ListDetectorModelVersions` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
```

```
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:53:33Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "ListDetectorModelVersions",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "detectorModelName": "myDetectorModel",
    "maxResults": 2
  },
  "responseElements": null,
  "requestID": "ebecb277-6bd8-44ea-8abd-fbf40ac044ee",
  "eventID": "fc6281a2-3fac-4e1e-98e0-ca6560b8b8be",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: ListDetectors action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `ListDetectors` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
```

```
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:53:54Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "ListDetectors",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "detectorModelName": "batchputmessagedetectorinstancecreated",
    "stateName": "HIDDEN_DUE_TO_SECURITY_REASONS"
  },
  "responseElements": null,
  "requestID": "4783666d-1e87-42a8-85f7-22d43068af94",
  "eventID": "0d2b7e9b-afe6-4aef-afd2-a0bb1e9614a9",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: ListInputs action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `ListInputs` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
```

```
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:53:57Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "ListInputs",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "nextToken": "CkhjYW5hcnlfdGVzdF9pbnB1dF9saXN0ZGV0ZWN0b3Jtb2RlbHN0ZXN0ZDU3OGZ",
    "maxResults": 3
  },
  "responseElements": null,
  "requestID": "dd6762a1-1f24-4e63-a986-5ea3938a03da",
  "eventID": "c500f6d8-e271-4366-8f20-da4413752469",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: PutLoggingOptions action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `PutLoggingOptions` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
```

```
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:56:43Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "PutLoggingOptions",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "loggingOptions": {
      "roleArn": "arn:aws:iam::123456789012:role/logging__logging_role",
      "level": "INFO",
      "enabled": false
    }
  },
  "responseElements": null,
  "requestID": "df570e50-fb19-4636-9ec0-e150a94bc52c",
  "eventID": "3247f928-26aa-471e-b669-e4a9e6fbc42c",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```


Example: UpdateDetectorModel action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `UpdateDetectorModel` action.


```
{
```

```
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:55:51Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "UpdateDetectorModel",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "detectorModelName": "myDetectorModel",
    "roleArn": "arn:aws:iam::123456789012:role/Events_action_execution_role"
  },
  "responseElements": null,
  "requestID": "add29860-c1c5-4091-9917-d2ef13c356cf",
  "eventID": "7baa9a14-6a52-47dc-aea0-3cace05147c3",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```


Example: UpdateInput action for CloudTrail

The following example shows a CloudTrail log entry that demonstrates the `UpdateInput` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-
ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    }
  },
  "eventTime": "2019-02-07T23:53:00Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "UpdateInput",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "errorCode": "ResourceNotFoundException",
  "errorMessage": "Input of name: NoSuchInput not found",
  "requestParameters": {
    "inputName": "NoSuchInput",
    "inputDescription": "this is a description of an input"
  },
  "responseElements": null,
  "requestID": "58d5d2bb-4110-4c56-896a-ee9156009f41",
  "eventID": "c2df241a-fd53-4fd0-936c-ba309e5dc62d",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

Example: BatchPutMessage action for CloudTrail

AWS IoT Events can use a CloudTrail integration for data plane API logging. This example adds details on data events through the `BatchPutMessage` action.


```
{
            "eventVersion": "1.09",
            "userIdentity": {
                "type": "AssumedRole",
                "principalId": "AKIAI44QH8DHBEXAMPLE:PrincipalId",
                "arn": "arn:aws:sts::123456789012:assumed-role/my-iam-role/my-iam-role-
entity",
                "accountId": "123456789012",
                "accessKeyId": "AKIAI44QH8DHBEXAMPLE",
                "sessionContext": {
                    "sessionIssuer": {
                        "type": "Role",
                        "principalId": "AKIAI44QH8DHBEXAMPLE",
                        "arn": "arn:aws:iam::123456789012:role/my-iam-role",
                        "accountId": "123456789012",
                        "userName": "sample_user_name"
                    },
                    "attributes": {
                        "creationDate": "2024-11-22T18:32:41Z",
                        "mfaAuthenticated": "false"
                    }
                }
            },
            "eventTime": "2024-11-22T18:57:35Z",
            "eventSource": "iotevents.amazonaws.com",
            "eventName": "BatchPutMessage",
            "awsRegion": "us-east-1",
            "sourceIPAddress": "3.239.107.128",
            "userAgent": "aws-internal/3",
            "requestParameters": {
                "messages": [
                    {
                        "messageId": "e306d827-b2e4-4439-9c86-411d4242a397",
                        "payload": "HIDDEN_DUE_TO_SECURITY_REASONS",
                        "inputName": "my_input_name"
                    }
                ]
            },
            "responseElements": {
```

```
                "batchPutMessageErrorEntries": []
            },
            "requestID": "cefc6b63-9ccf-4e31-9177-4aec8e701bfe",
            "eventID": "b994b52c-6011-4e3c-ad5f-e784e732fde0",
            "readOnly": false,
            "resources": [
                {
                    "accountId": "123456789012",
                    "type": "AWS::IoTEvents::Input",
                    "ARN": "arn:aws:iotevents:us-east-1:123456789012:input/
my_input_name"
                }
            ],
            "eventType": "AwsApiCall",
            "managementEvent": false,
            "recipientAccountId": "123456789012",
            "eventCategory": "Data",
            "tlsDetails": {
                "tlsVersion": "TLSv1.3",
                "cipherSuite": "TLS_AES_128_GCM_SHA256",
                "clientProvidedHostHeader": "iotevents.us-east-1.amazonaws.com"
            }
        },
```

Compliance validation for AWS IoT Events

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

Resilience in AWS IoT Events

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS global infrastructure.

Infrastructure security in AWS IoT Events

As a managed service, AWS IoT Events is protected by AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well-Architected Framework*.

You use AWS published API calls to access AWS IoT Events through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

AWS service quotas for AWS IoT Events resources

The *AWS General Reference Guide* provides the default quotas for AWS IoT Events for an AWS account. Unless specified, each quota is per AWS Region. For more information, see AWS IoT Events endpoints and quotas and AWS Service Quotas in the *AWS General Reference Guide*.

To request a service quota increase, submit a support case in the Support center console. For more information, see Requesting a quota increase in the *Service Quotas User Guide*.

Note

- All names for detector models and inputs must be unique within an account.
- You can't change names for detector models and inputs after they're created.

Tagging your AWS IoT Events resources

To help you manage and organize your detector models and inputs you can optionally assign your own metadata to each of these resources in the form of tags. This section describes tags and shows you how to create them.

Tag basics

Tags enable you to categorize your AWS IoT Events resources in different ways, for example, by purpose, owner, or environment. This is useful when you have many resources of the same type. You can quickly identify a specific resource based on the tags you've assigned to it.

Each tag consists of a key and optional value, both of which you define. For example, you could define a set of tags for your inputs that helps you track the devices that send these inputs by their type. We recommend that you create a set of tag keys that meets your needs for each kind of resource. Using a consistent set of tag keys makes it easier for you to manage your resources.

You can search for and filter resources based on the tags you add or apply, use tags to categorize and track your costs, and also use tags to control access to your resources as described in Using tags with IAM policies in the *AWS IoT Developer Guide*.

For ease of use, the Tag Editor in the AWS Management Console provides a central, unified way to create and manage your tags. For more information, see Getting started with Tag Editor in the *Tagging AWS Resources and Tag Editor User Guide*.

You can also work with tags using the AWS CLI and the AWS IoT Events API. You can associate tags with detector models and inputs when you create them by using the `"Tags"` field in the following commands:

- CreateDetectorModel
- CreateInput

You can add, modify, or delete tags for existing resources that support tagging by using the following commands:

- TagResource
- ListTagsForResource

- UntagResource

You can edit tag keys and values, and you can remove tags from a resource at any time. You can set the value of a tag to an empty string, but you can't set the value of a tag to null. If you add a tag that has the same key as an existing tag on that resource, the new value overwrites the old value. If you delete a resource, any tags associated with the resource are also deleted.

For more information, see Best Practices for Tagging AWS Resources

Tag restrictions and limitations

The following basic restrictions apply to tags:

- Maximum number of tags per resource – 50
- Maximum key length – 127 Unicode characters in UTF-8
- Maximum value length – 255 Unicode characters in UTF-8
- Tag keys and values are case sensitive.
- Do not use the `"aws:"` prefix in your tag names or values because it's reserved for AWS use. You can't edit or delete tag names or values with this prefix. Tags with this prefix don't count against your tags per resource limit.
- If your tagging schema is used across multiple services and resources, remember that other services may have restrictions on allowed characters. Generally, allowed characters are: letters, spaces, and numbers representable in UTF-8, and the following special characters: + - = . _ : / @.

Using tags with IAM policies

You can apply tag-based resource-level permissions in the IAM policies you use for AWS IoT Events API actions. This gives you better control over what resources a user can create, modify, or use.

You use the `Condition` element (also called the `Condition` block) with the following condition context keys and values in an IAM policy to control user access (permissions) based on a resource's tags:

- Use `aws:ResourceTag/`*`<tag-key>`*`: `*`<tag-value>`* to allow or deny user actions on resources with specific tags.
- Use `aws:RequestTag/`*`<tag-key>`*`: `*`<tag-value>`* to require that a specific tag be used (or not used) when making an API request to create or modify a resource that allows tags.

- Use `aws:TagKeys: [<tag-key>, ...]` to require that a specific set of tag keys be used (or not used) when making an API request to create or modify a resource that allows tags.

> **Note**
>
> The condition context keys and values in an IAM policy apply only to those AWS IoT Events actions where an identifier for a resource capable of being tagged is a required parameter.

Controlling access using tags in the *AWS Identity and Access Management User Guide* has additional information on using tags. The IAM JSON policy reference section of that guide has detailed syntax, descriptions, and examples of the elements, variables, and evaluation logic of JSON policies in IAM.

The following example policy applies two tag-based restrictions. A user restricted by this policy:

- Cannot give a resource the tag "env=prod" (in the example, see the line `"aws:RequestTag/env" : "prod"`
- Cannot modify or access a resource that has an existing tag "env=prod" (in the example, see the line `"aws:ResourceTag/env" : "prod"`).

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                "iotevents:CreateDetectorModel",
                "iotevents:CreateAlarmModel",
                "iotevents:CreateInput",
                "iotevents:TagResource"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/env": "prod"
                }
            }
```

```
        },
        {
            "Effect": "Deny",
            "Action": [
                "iotevents:DescribeDetectorModel",
                "iotevents:DescribeAlarmModel",
                "iotevents:UpdateDetectorModel",
                "iotevents:UpdateAlarmModel",
                "iotevents:DeleteDetectorModel",
                "iotevents:DeleteAlarmModel",
                "iotevents:ListDetectorModelVersions",
                "iotevents:ListAlarmModelVersions",
                "iotevents:UpdateInput",
                "iotevents:DescribeInput",
                "iotevents:DeleteInput",
                "iotevents:ListTagsForResource",
                "iotevents:TagResource",
                "iotevents:UntagResource",
                "iotevents:UpdateInputRouting"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "aws:ResourceTag/env": "prod"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iotevents:*"
            ],
            "Resource": "*"
        }
    ]
}
```

You can also specify multiple tag values for a given tag key by enclosing them in a list, as follows.

```
        "StringEquals" : {
          "aws:ResourceTag/env" : ["dev", "test"]
```

```
        }
```

Note

If you allow or deny users access to resources based on tags, you must consider explicitly denying users the ability to add those tags to or remove them from the same resources. Otherwise, it's possible for a user to circumvent your restrictions and gain access to a resource by modifying its tags.

Troubleshooting AWS IoT Events

This troubleshooting guide provides solutions for common issues you may encounter when using AWS IoT Events. Browse the topics to identify and resolve problems with detecting events, accessing data, permissions, service integrations, device configurations, and more. With troubleshooting advice for the AWS IoT Events console, API, CLI, errors, latency, and integrations, this guide aims to quickly resolve your issues so you can build reliable and scalable event-driven applications.

Topics

- Common AWS IoT Events issues and solutions
- Troubleshooting a detector model by running analyses in AWS IoT Events

Common AWS IoT Events issues and solutions

See the following section to troubleshoot errors and find possible solutions to resolve issues with AWS IoT Events.

Errors

- Detector model creation errors
- Updates from a deleted detector model
- Action trigger failure (when meeting a condition)
- Action trigger failure (when breeching a threshold)
- Incorrect state usage
- Connection message
- InvalidRequestException message
- Amazon CloudWatch Logs action.setTimer errors
- Amazon CloudWatch payload errors
- Incompatible data types
- Failed to send message to AWS IoT Events

Detector model creation errors

I get errors when I attempt to create a detector model.

When you create a detector model, you must consider the following limitations.

- Only one action is allowed in each `action` field.
- The `condition` is required for `transitionEvents`. It's optional for `OnEnter`, `OnInput`, and `OnExit` events.
- If the `condition` field is empty, the evaluated result of the condition expression is equivalent to `true`.
- The evaluated result of the condition expression should be a Boolean value. If the result isn't a Boolean value, it's equivalent to `false` and doesn't trigger the `actions` or transition to the `nextState` specified in the event.

For more information, see AWS IoT Events detector model restrictions and limitations.

Updates from a deleted detector model

I updated or deleted a detector model a few minutes ago but I'm still getting state updates from the old detector model through MQTT messages or SNS alerts.

If you update, delete, or recreate a detector model (see UpdateDetectorModel), there is a delay before all detector instances are deleted and the new model is used. During this time, inputs might continue to be processed by the instances of the previous version of the detector model. You might continue to receive alerts defined by the previous detector model. Wait for at least seven minutes before you recheck the update or report an error.

Action trigger failure (when meeting a condition)

The detector fails to trigger an action or transition to a new state when the condition is met.

Verify that the evaluated result of the detector's conditional expression is a Boolean value. If the result isn't a Boolean value, it's equivalent to `false` and doesn't trigger the `action` or transition to the `nextState` specified in the event. For more information, see Conditional expression syntax.

Action trigger failure (when breeching a threshold)

The detector doesn't trigger an action or an event transition when the variable in a conditional expression reaches a specified value.

If you update `setVariable` for `onInput`, `onEnter`, or `onExit`, the new value isn't used when evaluating any `condition` during the current processing cycle. Instead, the original value is used until the current cycle is complete. You can change this behavior by setting the `evaluationMethod` parameter in the detector model definition. When `evaluationMethod` is set to `SERIAL`, variables are updated and event conditions evaluated in the order that the events are defined. When `evaluationMethod` is set to `BATCH` (the default), variables are updated and events performed only after all event conditions are evaluated.

Incorrect state usage

The detector enters the wrong states when I attempt to send messages to inputs by using `BatchPutMessage`.

If you use BatchPutMessage to send multiple messages to inputs, the order in which the messages or inputs are processed isn't guaranteed. To guarantee ordering, send messages one at time and wait each time for `BatchPutMessage` to acknowledge success.

Connection message

I get a `('Connection aborted.', error(54, 'Connection reset by peer'))` error when I attempt to call or invoke an API.

Verify that OpenSSL uses TLS 1.1 or a later version to establish the connection. This should be the default under most Linux distributions or Windows version 7 and later. Users of macOS might need to upgrade OpenSSL.

InvalidRequestException message

I get InvalidRequestException when I attempt to call `CreateDetectorModel` and `UpdateDetectorModel` APIs.

Check the following to help resolve the issue. For more information, see CreateDetectorModel and UpdateDetectorModel.

- Make sure that you don't use both `seconds` and `durationExpression` as the parameters of `SetTimerAction` at the same time.
- Make sure that your string expression for `durationExpression` is valid. The string expression can contain numbers, variables (`$variable.<variable-name>`), or input values (`$input.<input-name>.<path-to-datum>`).

Amazon CloudWatch Logs `action.setTimer` errors

You can set up Amazon CloudWatch Logs to monitor AWS IoT Events detector model instances. The following are common errors generated by AWS IoT Events, when you use `action.setTimer`.

- **Error:** Your duration expression for the timer named *`<timer-name>`* could not be evaluated to a number.

  Make sure that your string expression for `durationExpression` can be converted to a number. Other data types, such as Boolean, aren't allowed.
- **Error:** The evaluated result of your duration expression for the timer named *`<timer-name>`* is greater than 31622440. To ensure accuracy, make sure that your duration expression refers to a value between 60-31622400.

  Make sure that the duration of your timer is less than or equal to 31622400 seconds. The evaluated result of the duration is rounded down to the nearest whole number.
- **Error:** The evaluated result of your duration expression for the timer named *`<timer-name>`* is less than 60. To ensure accuracy, make sure that your duration expression refers to a value between 60-31622400.

  Make sure that the duration of your timer is greater than or equal to 60 seconds. The evaluated result of the duration is rounded down to the nearest whole number.
- **Error:** Your duration expression for the timer named *`<timer-name>`* could not be evaluated. Check the variable names, input names, and paths to the data to make sure that you refer to the existing variables and inputs.

  Make sure that your string expression refers to the existing variables and inputs. The string expression can contain numbers, variables (`$variable.`*`variable-name`*), and input values (`$input.`*`input-name`*`.`*`path-to-datum`*).
- **Error:** Failed to set the timer named `<timer-name>`. Check your duration expression, and try again.

See the SetTimerAction action to ensure that you specified the correct parameters, and then set the timer again.

For more information, see Enable Amazon CloudWatch logging when developing AWS IoT Events detector models.

Amazon CloudWatch payload errors

You can set up Amazon CloudWatch Logs to monitor AWS IoT Events detector model instances. The following are common errors and warnings generated by AWS IoT Events, when you configure the action payload.

- **Error:** We couldn't evaluate your expression for the action. Make sure that the variable names, input names, and paths to the data refer to the existing variables and input values. Also, verify that the size of the payload is less than 1 KB, the maximum allowed size of a payload.

  Make sure that you enter the correct variable names, input names, and paths to the data. You might also receive this error message if the action payload is larger than 1 KB.

- **Error:** We couldn't parse your content expression for the payload of *<action-type>*. Enter a content expression with the correct syntax.

  The content expression can contain strings ('*string*'), variables ($variable.*variable-name*), input values ($input.*input-name*.*path-to-datum*), string concatenations, and strings that contain ${}.

- **Error:** Your payload expression {*expression*} isn't valid. The defined payload type is JSON, so you must specify an expression that AWS IoT Events would evaluate to a string.

  If the specified payload type is JSON, AWS IoT Events first checks if the service can evaluate your expression to a string. The evaluated result can't be a Boolean or number. If the validation fails, you might receive this error.

- **Warning:** The action was executed, but we couldn't evaluate your content expression for the action payload to valid JSON. The defined payload type is JSON.

  Make sure that AWS IoT Events can evaluate your content expression for the action payload to valid JSON, if you define the payload type as JSON. AWS IoT Events runs the action even if AWS IoT Events can't evaluate the content expression to valid JSON.

For more information, see Enable Amazon CloudWatch logging when developing AWS IoT Events detector models.

Incompatible data types

Message: Incompatible data types [<inferred-types>] found for <reference> in the following expression: <expression>

You might receive this error for one of the following reasons:

- The evaluated results of your references are not compatible with other operands in your expressions.
- The type of the argument passed to a function is not supported.

When you use references in expressions, check the following:

- When you use a reference as an operand with one or more operators, make sure that all data types that you reference are compatible.

  For example, in the following expression, integer 2 is an operand of both the == and && operators. To ensure that the operands are compatible, `$variable.testVariable + 1` and `$variable.testVariable` must reference an integer or decimal.

  In addition, integer 1 is an operand of the + operator. Therefore, `$variable.testVariable` must reference an integer or decimal.

  ```
  '$variable.testVariable + 1 == 2 && $variable.testVariable'
  ```

- When you use a reference as an argument passed to a function, make sure that the function supports the data types that you reference.

  For example, the following `timeout("time-name")` function requires a string with double quotes as the argument. If you use a reference for the *timer-name* value, you must reference a string with double quotes.

  ```
  timeout("timer-name")
  ```

> **Note**
>
> For the `convert(type, expression)` function, if you use a reference for the *type* value, the evaluated result of your reference must be `String`, `Decimal`, or `Boolean`.

For more information, see AWS IoT Events reference for inputs and variables in expressions.

Failed to send message to AWS IoT Events

Message: Failed to send message to Iot Events

You might experience this error for the following reasons:

- The input message payload does not contain the `Input attribute Key`.
- The `Input attribute Key` is not in the same JSON path as specified in the input definition.
- The input message does not match with the schema, as defined in the AWS IoT Events input.

> **Note**
>
> The data ingestion from other services will also experience failure.

Example

For example in AWS IoT Core, the AWS IoT rule will fail with the following message `Verify the Input Attribute key.`

To resolve this, ensure that the input payload message schema conforms to the AWS IoT Events Input definition and the `Input attribute Key` location matches. For more information, see Create an input for models in AWS IoT Events to learn how to define AWS IoT Events Inputs.

Troubleshooting a detector model by running analyses in AWS IoT Events

AWS IoT Events can analyze your detector model and generate analysis results without sending input data to your detector model. AWS IoT Events performs a series of analyses described in this

section to check your detector model. This advanced troubleshooting solution also summarizes diagnostic information, including the severity level and location, so that you can quickly find and fix potential issues in your detector model. For more information about diagnostic error types and messages for your detector model, see Detector model analysis and diagnostic information for AWS IoT Events.

You can use the AWS IoT Events console, API, AWS Command Line Interface (AWS CLI), or AWS SDK to view diagnostic error messages from the analysis of your detector model.

Note

- You must fix all errors before you can publish your detector model.
- We recommend that you review warnings and take necessary actions before you use your detector model in production environments. Otherwise, the detector model might not work as expected.
- You can have up to 10 analyses in the `RUNNING` status at the same time.

To learn how to analyze your detector model, see Analyze a detector model for AWS IoT Events (Console) or Analyze a detector model in AWS IoT Events (AWS CLI).

Topics

- Detector model analysis and diagnostic information for AWS IoT Events
- Analyze a detector model for AWS IoT Events (Console)
- Analyze a detector model in AWS IoT Events (AWS CLI)

Detector model analysis and diagnostic information for AWS IoT Events

Detector model analyses gather the following diagnostic information:

- **Level** – The severity level of the analysis result. Based on the severity level, analysis results fall into three general categories:
  - **Information** (`INFO`) – An information result tells you about a significant field in your detector model. This type of result usually doesn't require immediate action.
  - **Warning** (`WARNING`) – A warning result draws special attention to fields that might cause issues for your detector model. We recommend that you review warnings and take necessary

actions before you use your detector model in production environments. Otherwise, the detector model might not work as expected.

- **Error** (ERROR) – An error result notifies you about a problem found in your detector model. AWS IoT Events automatically performs this set of analyses when you try to publish the detector model. You must fix all errors before you can publish the detector model.

- **Location** – Contains information that you can use to locate the field in your detector model that the analysis result references. A location typically includes the state name, transition event name, event name, and expression (for example, `in state TemperatureCheck in onEnter in event Init in action setVariable`).
- **Type** – The type of the analysis result. Analysis types fall into the following categories:
  - `supported-actions` – AWS IoT Events can invoke actions when a specified event or transition event is detected. You can define built-in actions to use a timer or set a variable, or send data to other AWS services. You must specify actions that work with other AWS services in an AWS Region where the AWS services are available.
  - `service-limits` – Service quotas, also known as limits, are the maximum or minimum number of service resources or operations for your AWS account. Unless otherwise noted, each quota is Region-specific. Depending on your business needs, you can update your detector model to avoid encountering limits or request a quota increase. You can request increases for some quotas, and other quotas can't be increased. For more information, see Quotas.
- **`structure`** – The detector model must have all required components such as states and follow a structure that AWS IoT Events supports. A detector model must have at least one state and a condition that evaluates the incoming input data to detect significant events. When an event is detected, the detector model transitions to the next state and can invoke actions. These events are known as transition events. A transition event must direct the next state to enter.
- **`expression-syntax`** – AWS IoT Events provides several ways to specify values when you create and update detector models. You can use literals, operators, functions, references, and substitution templates in the expressions. You can use expressions to specify literal values, or AWS IoT Events can evaluate the expressions before you specify particular values. Your expression must follow the required syntax. For more information, see Expressions to filter, transform, and process event data.

  Detector Model expressions in AWS IoT Events can reference specific data or a resource.

  - **`data-type`** – AWS IoT Events supports integer, decimal, string, and Boolean data types. If AWS IoT Events can automatically convert the data of one data type to another during expression evaluation, these data types are compatible.

> **Note**
>
> - Integer and decimal are the only compatible data types supported by AWS IoT Events.
> - AWS IoT Events can't evaluate arithmetic expressions because AWS IoT Events can't convert an integer to a string.

- **`referenced-data`** – You must define the data referenced in your detector model before you can use the data. For example, if you want to send data to a DynamoDB table, you must define a variable that references the table name before you can use the variable in an expression (`$variable.TableName`).
- **`referenced-resource`** – Resources that the detector model uses must be available. You must define resources before you can use them. For example, you want to create a detector model to monitor the temperature of a greenhouse. You must define an input (`$input.TemperatureInput`) to route incoming temperature data to your detector model before you can use the `$input.TemperatureInput.sensorData.temperature` to reference the temperature.

See the following section to troubleshoot errors and find possible solutions from the analysis of your detector model.

Troubleshoot detector model errors in AWS IoT Events

The types of errors described above provide diagnostic information about a detector model and correspond to messages that you might retrieve. Use these messages and suggested solutions to troubleshoot errors with your detector model.

Messages and solutions

- Location
- supported-actions
- service-limits
- structure
- expression-syntax
- data-type
- referenced-data

- referenced-resource

`Location`

An analysis result with information about `Location`, corresponds to the following error message:

- **Message** – Contains additional information about the analysis result. This can be an information, warning, or error message.

  Solution: You might receive this error message if you specified an action that AWS IoT Events currently doesn't support. For a list of supported actions, see Supported actions to receive data and trigger actions in AWS IoT Events.

`supported-actions`

An analysis result with information about `supported-actions`, corresponds to the following error messages:

- **Message:** Invalid action type present in action definition: *`action-definition`*.

  Solution: You might receive this error message if you specified an action that AWS IoT Events currently doesn't support. For a list of supported actions, see Supported actions to receive data and trigger actions in AWS IoT Events.

- **Message:** DetectorModel definition has an *`aws-service`* action, but the *`aws-service`* service is not supported in the region *`region-name`*.

  Solution: You might receive this error message if the action that you specified is supported by AWS IoT Events, but the action isn't available in your current Region. This might occur when you try to send data to an AWS service that isn't available in the Region. You must also choose the same Region for both AWS IoT Events and the AWS services that you're using.

`service-limits`

An analysis result with information about `service-limits`, corresponds to the following error messages:

- **Message:** Content Expression allowed in payload exceeded the limit *`content-expression-size`* bytes in event *`event-name`* in state *`state-name`*.

Solution: You might receive this error message if the content expression for your action payload is greater than 1024 bytes. The size of the content expression for a payload can be up to 1024 bytes.

- **Message:** Number of states allowed in detector model definition exceeded the limit *states-per-detector-model*.

  Solution: You might receive this error message if your detector model has more than 20 states. A detector model can have up to 20 states.

- **Message:** The duration for timer *timer-name* should be at least *minimum-timer-duration* seconds long.

  Solution: You might receive this error message if the duration of your timer is less than 60 seconds. We recommend that the duration of a timer is between 60 and 31622400 seconds. If you specify an expression for the duration of your timer, the evaluated result of the duration expression is rounded down to the nearest whole number.

- **Message:** Number of actions allowed per event exceeded the limit *actions-per-event* in detector model definition

  Solution: You might receive this error message if the event has more than 10 actions. You can have up to 10 actions for each event in your detector model.

- **Message:** Number of transition events allowed per state exceeded the limit *transition-events-per-state* in detector model definition.

  Solution: You might receive this error message if the state has more than 20 transition events. You can have up to 20 transition events for each state in your detector model.

- **Message:** Number of events allowed per state exceeded the limit *events-per-state* in detector model definition

  Solution: You might receive this error message if the state has more than 20 events. You can have up to 20 events for each state in your detector model.

- **Message:** The maximum number of detector models that can be associated with a single input may have reached the limit. Input *input-name* is used in *detector-models-per-input* detector model routes.

  Solution: You might receive this warning message if you tried to route an input to more than 10 detector models. You can have up to 10 different detector models associated with a single detector model.

`structure`

An analysis result with information about `structure`, corresponds to the following error messages:

- **Message:** Actions may only have one type defined, but found an action with *`number-of-types`* types. Please split into separate Actions.

  Solution: You might receive this error message if you specified two or more actions in a single field by using API operations to create or update your detector model. You can define an array of `Action` objects. Make sure that you define each action as a separate object.

- **Message:** The TransitionEvent *`transition-event-name`* transitions to a non-existent state *`state-name`*.

  Solution: You might receive this error message if AWS IoT Events couldn't find the next state that your transition event referenced. Make sure that the next state is defined and that you entered the correct state name.

- **Message:** The DetectorModelDefinition had a shared state name: found state *`state-name`* with *`number-of-states`* repetitions.

  Solution: You might receive this error message if you use the same name for one or more states. Make sure that you give a unique name to each state in your detector model. The state name must have 1-128 characters. Valid characters: a-z, A-Z, 0-9, _ (underscore), and - (hyphen).

- **Message:** The Definition's initialStateName *`initial-state-name`* did not correspond to a defined State.

  Solution: You might receive this error message if the initial state name is incorrect. The detector model remains in the initial (start) state until an input arrives. Once an input arrives, the detector model immediately transitions to the next state. Make sure that the initial state name is the name of a defined state and that you enter the correct name.

- **Message:** Detector Model Definition must use at least one Input in a condition.

  Solution: You might receive this error if you didn't specify an input in a condition. You must use at least one input in at least one condition. Otherwise, AWS IoT Events doesn't evaluate incoming data.

- **Message:** Only one of seconds and durationExpression can be set in SetTimer.

Solution: You might receive this error message if you used both `seconds` and `durationExpression` for your timer. Make sure that you use either `seconds` or `durationExpression` as the parameters of `SetTimerAction`. For more information, see SetTimerAction in the *AWS IoT Events API Reference*.

- **Message:** An action in your detector model is unreachable. Check the condition that initiates the action.

  Solution: If an action in your detector model is unreachable, the event's condition evaluates to false. Check the condition of the event that contains the action, to ensure that it evaluates to true. When the event's condition evaluates to true, the action should become reachable.

- **Message:** An input attribute is being read, but this may be caused by a timer expiration.

  Solution: An input attribute's value can be read when either of the following occurs:

  - A new input value has been received.
  - When a timer in the detector has expired.

  To ensure that an input attribute is being evaluated only when the new value for that input is received, include a call to the `triggerType("Message")` function in your condition as follows:

  The original condition being evaluated in the detector model:

  ```
  if ($input.HeartBeat.status == "OFFLINE")
  ```

  would become similar to the following:

  ```
  if ( triggerType("MESSAGE") &&  $input.HeartBeat.status == "OFFLINE")
  ```

  where a call to the `triggerType("Message")` function comes before the initial input provided in the condition. By using this technique, the `triggerType("Message")` function will evaluate to true and satisfy the condition of receiving a new input value. For more information about the usage of the `triggerType` function, search for `triggerType` in the Expressions section in the *AWS IoT Events Developer Guide*

- **Message:** A state in your detector model is unreachable. Check the condition that will cause a transition to the desired state.

Solution: If a state in your detector model is unreachable, a condition that causes an incoming transition to that state evaluates to false. Check that the conditions of the incoming transitions to that unreachable state in your detector model evaluates to true, so the desired state can become reachable.

- **Message:** An expiring timer can cause an unexpected amount of messages to be sent.

  Solution: To prevent your detector model from entering into an infinite state of sending an unexpected amount of messages because a timer has expired, consider using a call to the `triggerType("Message")` function, in the conditions of your detector model as follows:

  The original condition being evaluated in the detector model:

  ```
  if (timeout("awake"))
  ```

  would be transformed into a condition that looks similar to the following:

  ```
  if (triggerType("MESSAGE") && timeout("awake"))
  ```

  where a call to the `triggerType(“Message”)` function comes before the initial input provided in the condition.

  This change prevents initiating timer actions in your detector, preventing an infinite loop of messages being sent. For more information about how to use timer actions in your detector, see the Using built-in actions page of the *AWS IoT Events Developer Guide*

`expression-syntax`

An analysis result with information about `expression-syntax`, corresponds to the following error messages:

- **Message:** Your payload expression {*`expression`*} isn't valid. The defined payload type is JSON, so you must specify an expression that AWS IoT Events would evaluate to a string.

  Solution: If the specified payload type is JSON, AWS IoT Events first checks if the service can evaluate your expression to a string. The evaluated result can't be a Boolean or number. If the validation doesn't succeed, you might receive this error.

- **Message:** `SetVariableAction.value` must be an expression. Failed to parse value '*`variable-value`*'

  Solution: You can use `SetVariableAction` to define a variable with a `name` and `value`. The `value` can be a string, number, or Boolean value. You can also specify an expression for the `value`. For more information, see SetVariableAction, in the *AWS IoT Events API Reference*.

- **Message:** We couldn't parse your expression of the attributes (*`attribute-name`*) for the DynamoDB action. Enter expression with the correct syntax.

  Solution: You must use expressions for all parameters in `DynamoDBAction`. substitution templates. For more information, see DynamoDBAction in the *AWS IoT Events API Reference*.

- **Message:** We couldn't parse your expression of the tableName for the DynamoDBv2 action. Enter expression with the correct syntax.

  Solution: The `tableName` in `DynamoDBv2Action` must be a string. You must use an expression for the `tableName`. The expressions accept literals, operators, functions, references, and substitution templates. For more information, see DynamoDBv2Action in the *AWS IoT Events API Reference*.

- **Message:** We couldn't evaluate your expression to valid JSON. The DynamoDBv2 action only supports the JSON payload type.

  Solution: The payload type for `DynamoDBv2` must be JSON. Make sure that AWS IoT Events can evaluate your content expression for the payload to valid JSON. For more information, see DynamoDBv2Action, in the *AWS IoT Events API Reference*.

- **Message:** We couldn't parse your content expression for the payload of *`action-type`*. Enter a content expression with the correct syntax.

  Solution: The content expression can contain strings ('*`string`*'), variables ($variable.*`variable-name`*), input values ($input.*`input-name`*.*`path-to-datum`*), string concatenations, and strings that contain `${}`.

- **Message:** Customized Payloads must be non-empty.

  Solution: You might receive this error message, if you chose **Custom payload** for your action and didn't enter a content expression in the AWS IoT Events console. If you choose **Custom payload**, you must enter a content expression under **Custom payload**. For more information, see Payload in the *AWS IoT Events API Reference*.

- **Message:** Failed to parse duration expression '*`duration-expression`*' for timer '*`timer-name`*'.

Solution: The evaluated result of your duration expression for the timer must be a value between 60–31622400. The evaluated result of the duration is rounded down to the nearest whole number.

- **Message:** Failed to parse expression '*`expression`*' for *`action-name`*

  Solution: You might receive this message if the expression for the specified action has incorrect syntax. Make sure that you enter an expression with the correct syntax. For more information, see Syntax to filter device data and define actions in AWS IoT Events.

- **Message:** Your *`fieldName`* for `IotSitewiseAction` couldn't be parsed. You must use correct syntax in your expression.

  Solution: You might receive this error if AWS IoT Events couldn't parse your *`fieldName`* for `IotSitewiseAction`. Make sure the *`fieldName`* uses an expression that AWS IoT Events can parse. For more information, see IotSiteWiseAction in the *AWS IoT Events API Reference*.

`data-type`

An analysis result with information about `data-type`, corresponds to the following error messages:

- **Message:** Duration expression *`duration-expression`* for timer *`timer-name`* is not valid, it must return a number.

  Solution: You might receive this error message if AWS IoT Events couldn't evaluate the duration expression for your timer to a number. Make sure that your `durationExpression` can be converted to a number. Other data types, such as Boolean, aren't supported.

- **Message:** Expression *`condition-expression`* is not a valid condition expression.

  Solution: You might receive this error message if AWS IoT Events couldn't evaluate your `condition-expression` to a Boolean value. The Boolean value must be either `TRUE` or `FALSE`. Make sure that your condition expression can be converted to a Boolean value. If the result isn't a Boolean value, it's equivalent to `FALSE` and doesn't invoke the actions or transition to the `nextState` specified in the event.

- **Message:** Incompatible data types [*`inferred-types`*] found for *`reference`* in the following expression: *`expression`*

Solution: Solution: All expressions for the same input attribute or variable in the detector model must reference the same data type.

Use the following information to resolve the issue:

- When you use a reference as an operand with one or more operators, make sure that all data types that you reference are compatible.

  For example, in the following expression, integer 2 is an operand of both the == and && operators. To ensure that the operands are compatible, `$variable.testVariable + 1` and `$variable.testVariable` must reference an integer or decimal.

  In addition, integer 1 is an operand of the + operator. Therefore, `$variable.testVariable` must reference an integer or decimal.

  ```
  '$variable.testVariable + 1 == 2 && $variable.testVariable'
  ```

- When you use a reference as an argument passed to a function, make sure that the function supports the data types that you reference.

  For example, the following `timeout("time-name")` function requires a string with double quotes as the argument. If you use a reference for the *`timer-name`* value, you must reference a string with double quotes.

  ```
  timeout("timer-name")
  ```

  > **Note**
  >
  > For the `convert(type, expression)` function, if you use a reference for the *`type`* value, the evaluated result of your reference must be `String`, `Decimal`, or `Boolean`.

  For more information, see AWS IoT Events reference for inputs and variables in expressions.

- **Message:** Incompatible data types [*`inferred-types`*] used with *`reference`*. This may lead to a runtime error.

  Solution: You might receive this warning message if two expressions for the same input attribute or variable reference two data types. Make sure that your expressions for the same input attribute or variable reference the same data type in the detector model.

- **Message:** The data types [*inferred-types*] that you entered for the operator [*operator*] aren't compatible for the following expression: '*expression*'

  Solution: You might receive this error message if your expression combines data types that are not compatible with a specified operator. For example, in the following expression, the operator + is compatible with Integer, Decimal, and String data types, but not operands of Boolean data type.

  ```
  true + false
  ```

  You must make sure that the data types you use with an operator are compatible.

- **Message:** The data types [*inferred-types*] found for *input-attribute* aren't compatible and can lead to a runtime error.

  Solution: You might receive this error message if two expressions for the same input attribute reference two data types for either the `OnEnterLifecycle` of a state, or for both the `OnInputLifecycle` and `OnExitLifecycle` of a state. Make sure your expressions in `OnEnterLifecycle` (or, both `OnInputLifecycle` and `OnExitLifecycle`) reference the same data type for each state of your detector model.

- **Message:** The payload expression [*expression*] isn't valid. Specify an expression that would evaluate to a string at runtime because the payload type is JSON format.

  Solution: You might receive this error if your specified payload type is JSON, but AWS IoT Events can't evaluate its expression to a String. Make sure the evaluated result is a String, not a Boolean or a number.

- **Message:** Your interpolated expression {*interpolated-expression*} must evaluate to either an integer or a Boolean value at runtime. Otherwise, your payload expression {*payload-expression*} won't be parseable at runtime as valid JSON.

  Solution: You might receive this error message if AWS IoT Events couldn't evaluate your interpolated expression to an integer or a Boolean value. Make sure your interpolated expression can be converted to an integer or a Boolean value, because other data types, such as tring, aren't supported.

- **Message:** The expression type in the `IotSitewiseAction` field *expression* is defined as type *defined-type* and inferred as type *inferred-type*. The defined type and the inferred type must be the same.

Solution: You might receive this error message if your expression in the `propertyValue` of `IotSitewiseAction` has a data type defined differently than the data type inferred by AWS IoT Events. Make sure you use the same data type for all instances of this expression in your detector model.

- **Message:** The data types [*`inferred-types`*] used for `setTimer` action don't evaluate to `Integer` for the following expression: *`expression`*

  Solution: You might receive this error message if the inferred data type for your duration expression isn't Integer or Decimal. Make sure your `durationExpression` can be converted to a number. Other data types, such as Boolean and String, aren't supported.

- **Message:** The data types [*`inferred-types`*] used with operands of the comparison operator [*`operator`*] are not compatible in the following expression: *`expression`*

  Solution: The inferred data types for the operands of the *`operator`* in the conditional expression (*`expression`*) of your detector model don't match. The operands must be used with the matching data types in all other parts of your detector model.

> **Tip**
>
> You can use `convert` to change the data type of an expression in your detector model. For more information, see Functions to use in AWS IoT Events expressions.

`referenced-data`

An analysis result with information about `referenced-data`, corresponds to the following error messages:

- **Message:** Detected broken Timer: timer *`timer-name`* is used in an expression but is never set.

  Solution: You might receive this error message if you use a timer that isn't set. You must set a timer before you use it in an expression. Also, make sure that you enter the correct timer name.

- **Message:** Detected broken Variable: variable *`variable-name`* is used in an expression but is never set.

Solution: You might receive this error message if you use a variable that isn't set. You must set a variable before you use it in an expression. Also, make sure that you enter the correct variable name.

- **Message:** Detected broken Variable: a variable is used in an expression before being set to a value.

  Solution: Each variable must be assigned to a value before it can be evaluated in an expression. Set the value of the variable before every use so its value can be retrieved. Also, make sure that you enter the correct variable name.

`referenced-resource`

An analysis result with information about `referenced-resource`, corresponds to the following error messages:

- **Message:** Detector Model Definition contains reference to Input that does not exist.

  Solution: You might receive this error message if you use expressions to reference an input that doesn't exist. Make sure that your expression references an existing input and enter the correct input name. If you don't have an input, create one first.

- **Message:** Detector Model Definition contains invalid InputName: *`input-name`*

  Solution: You might receive this error message if your detector model contains an invalid input name. Make sure that you enter the correct input name. The input name must have 1-128 characters. Valid characters: a-z, A-Z, 0-9, _ (underscore), and - (hyphen).

Analyze a detector model for AWS IoT Events (Console)

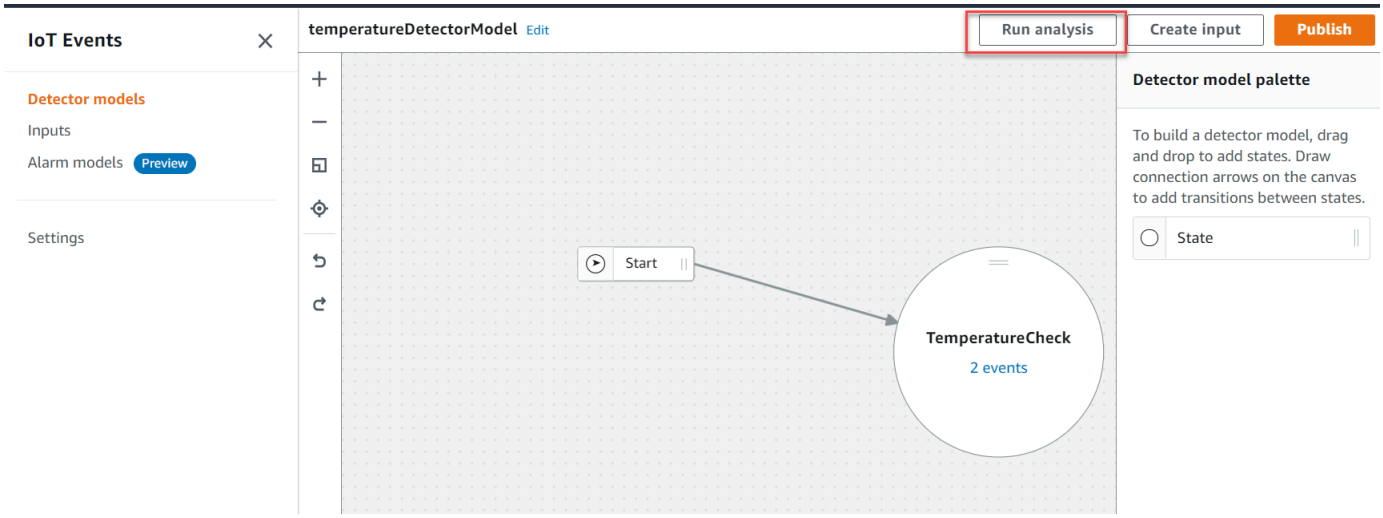
AWS IoT Events allows you to monitor and react to IoT data by detecting events and triggering actions with the AWS IoT Events API. The following steps use the AWS IoT Events console to analyze a detector model.

> **Note**
>
> After AWS IoT Events starts analyzing your detector model, you have up to 24 hours to retrieve the analysis results.

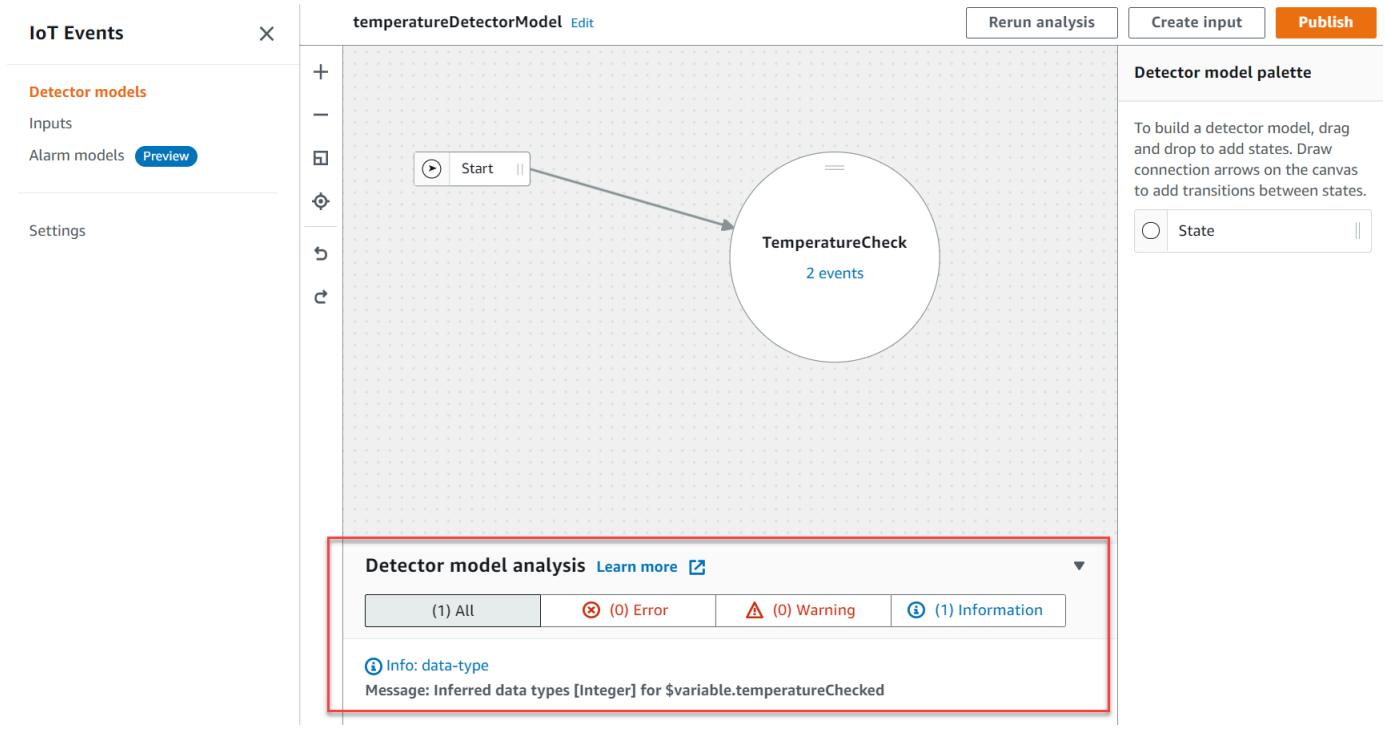
A detector model analysis can help you optimize your models, identify potential issues, and ensure they're functioning as intended. For example, on a windfarm, the detector model analysis could reveal if the model correctly identifies potential gear failures based on abnormal vibration patterns. Or, if the model accurately triggers maintenance alerts when wind speeds exceed safe operating thresholds. By refining a model based on the analysis, you can improve predictive maintenance, reduce downtime, and enhance overall energy production efficiency.

To analyze a detector model

1. Sign in to the AWS IoT Events console.
2. In the navigation pane, choose **Detector models**.
3. Under **Detector models**, choose the target detector model.
4. On your detector model page, choose **Edit**.
5. In the upper-right corner, choose **Run analysis**.



The following is an example analysis result in the AWS IoT Events console.



Analyze a detector model in AWS IoT Events (AWS CLI)

Analyzing your AWS IoT Events detector models programmatically provides valuable insights into their structure, behavior, and performance. This API-based approach allows for automated analysis, integration with your existing workflows, and the ability to perform bulk operations across multiple detector models. By leveraging the StartDetectorModelAnalysis API, you can initiate in-depth examinations of your models, helping you identify potential issues, optimize logic flows, and ensure that your IoT event processing aligns with your business requirements.

The following steps use the AWS CLI to analyze a detector model.

To analyze a detector model using AWS CLI

1. Run the following command to start an analysis.

```
aws iotevents start-detector-model-analysis --cli-input-json file://file-name.json
```

> **Note**
>
> Replace *file-name* with the name of the file that contains the detector model definition.

Example Detector model definition

```
{
    "detectorModelDefinition": {
        "states": [
            {
                "stateName": "TemperatureCheck",
                "onInput": {
                    "events": [
                        {
                            "eventName": "Temperature Received",
                            "condition":
 "isNull($input.TemperatureInput.sensorData.temperature)==false",
                            "actions": [
                                {
                                    "iotTopicPublish": {
                                        "mqttTopic": "IoTEvents/Output"
                                    }
                                }
                            ]
                        }
                    ],
                    "transitionEvents": []
                },
                "onEnter": {
                    "events": [
                        {
                            "eventName": "Init",
                            "condition": "true",
                            "actions": [
                                {
                                    "setVariable": {
                                        "variableName": "temperatureChecked",
                                        "value": "0"
                                    }
```

```
                                }
                            ]
                        }
                    ]
                },
                "onExit": {
                    "events": []
                }
            }
        ],
        "initialStateName": "TemperatureCheck"
    }
}
```

If you use the AWS CLI to analyze an existing detector model, choose one of the following to retrieve the detector model definition:

- If you want to use the AWS IoT Events console, do the following:
  1. In navigation pane, choose **Detector models**.
  2. Under **Detector models**, choose the target detector model.
  3. Choose **Export detector model** from **Action** to download the detector model. The detector model is saved in JSON.
  4. Open the detector model JSON file.
  5. You only need the `detectorModelDefinition` object. Remove the following:
     - The first curly bracket ({) at the top of the page
     - The `detectorModel` line
     - The `detectorModelConfiguration` object
     - The last curly bracket (}) at the bottom of the page
  6. Save the file.
- If you want to use the AWS CLI, do the following:
  1. Run the following command in a terminal.

     ```
     aws iotevents describe-detector-model --detector-model-name detector-model-name
     ```

  2. Replace *detector-model-name* with the name of your detector model.
  3. Copy the `detectorModelDefinition` object to a text editor.

4. Add curly brackets ({}) outside of the `detectorModelDefinition`.
5. Save the file in JSON.

Example response

```
{
    "analysisId": "c1133390-14e3-4204-9a66-31efd92a4fed"
}
```

2. Copy the analysis ID from the output.
3. Run the following command to retrieve the status of the analysis.

```
aws iotevents describe-detector-model-analysis --analysis-id "analysis-id"
```

> **Note**
>
> Replace *analysis-id* with the analysis ID that you copied.

Example response

```
{
    "status": "COMPLETE"
}
```

The status can be one of the following values:

- `RUNNING` – AWS IoT Events is analyzing your detector model. This process can take up to one minute to complete.
- `COMPLETE` – AWS IoT Events finished analyzing your detector model.
- `FAILED` – AWS IoT Events couldn't analyze your detector model. Try again later.

4. Run the following command to retrieve one or more analysis results of the detector model.

> **Note**
>
> Replace *analysis-id* with the analysis ID that you copied.

```
aws iotevents get-detector-model-analysis-results --analysis-id "analysis-id"
```

Example response

```
{
    "analysisResults": [
        {
            "type": "data-type",
            "level": "INFO",
            "message": "Inferred data types [Integer] for
 $variable.temperatureChecked",
            "locations": []
        },
        {
            "type": "referenced-resource",
            "level": "ERROR",
            "message": "Detector Model Definition contains reference to Input
 'TemperatureInput' that does not exist.",
            "locations": [
                {
                    "path": "states[0].onInput.events[0]"
                }
            ]
        }
    ]
}
```

Note

After AWS IoT Events starts analyzing your detector model, you have up to 24 hours to retrieve the analysis results.

AWS IoT Events commands

This chapter provides a comprehensive guide to all the API operations available in AWS IoT Events. It offers detailed explanations, including sample requests, responses, and potential errors for each operation across the supported web services protocols. Understanding these API operations helps you effectively integrate AWS IoT Events into your IoT applications and automate your event detection and response workflows.

AWS IoT Events actions

You can use AWS IoT Events API commands to create, read, update, and delete inputs and detector models, and to list their versions. For more information, see the actions and data types that are supported by AWS IoT Events in the *AWS IoT Events API Reference*.

The AWS IoT Events sections in the *AWS CLI Command Reference* include the AWS CLI commands that you can use to administer and manipulate AWS IoT Events.

AWS IoT Events data

You can use the AWS IoT Events Data API commands to send inputs to detectors, list detectors, and view or update a detector's status. For more information, see the actions and data types that are supported by AWS IoT Events Data in the *AWS IoT Events API Reference*.

The AWS IoT Events data sections in the *AWS CLI Command Reference* includes the AWS CLI commands that you can use to process AWS IoT Events data.

Document history for AWS IoT Events

The following table describes the important changes to the *AWS IoT Events Developer Guide* after September 17, 2020. For more information about updates to this documentation, you can subscribe to an RSS feed.

Change	Description	Date
End of support notice	End of support notice: On May 20, 2026, AWS will discontinue support for AWS IoT Events. After May 20, 2026, you will no longer be able to access the AWS IoT Events console or AWS IoT Events resources.	May 20, 2025
Region launch	AWS IoT Events is now available in the Asia Pacific (Mumbai) region.	September 30, 2021
Region launch	AWS IoT Events is now available in the AWS GovCloud (US-West) Region.	September 22, 2021
Troubleshoot a detector model by running analyses	AWS IoT Events can now analyze your detector model and generate analysis results that you can use to troublesh oot your detector model.	February 23, 2021
Region launch	Launched AWS IoT Events in China (Beijing).	September 30, 2020
Expression usage	Added examples to show you how to write expressions.	September 22, 2020

Monitoring with alarms	Alarms help you monitor your data for changes. You can create alarms that send notifications when a threshold is breached.	June 1, 2020

Earlier updates

The following table describes important changes to the *AWS IoT Events Developer Guide* before September 18, 2020.

Change	Description	Date
Added type validation to the Expressions reference	Added type validation information to the Expressions reference.	August 3, 2020
Added Region warning for other services	Added a warning regarding selecting the same region for AWS IoT Events and other AWS services.	May 7, 2020
Additions, updates	• Payload Customization feature • New event actions: Amazon DynamoDB and AWS IoT SiteWise	April 27, 2020
Added built-in functions for detector model conditional expressions	Added built-in functions for detector model conditional expressions.	September 10, 2019
Added detector model examples	Added examples for the detector model.	August 5, 2019
Added new event actions	Added new event actions for:	July 19, 2019

Change	Description	Date
	• Lambda • Amazon SQS • Kinesis Data Firehose • AWS IoT Events input	
Additions, corrections	• Updated description of `timeout()` function. • Added best practice regarding account inactivity.	June 11, 2019
Updated permissions policy and console debug options	• Updated the console permissions policy. • Updated console debug options page image.	June 5, 2019
Updates	AWS IoT Events service open to general availability.	May 30, 2019
Additions, updates	• Updated security information. • Added annotated detector model example.	May 22, 2019
Added examples and required permissions	Added Amazon SNS payload examples; additions to required permissions for `CreateDetectorModel` .	May 17, 2019
Added additional security information	Added information to the security section.	May 9, 2019
Limited preview release	Limited preview release of the documentation.	March 28, 2019