## AWS IoT Events Developer Guide

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing AWS IoT Events expressions</td>
<td>106</td>
</tr>
<tr>
<td>Detector model examples</td>
<td>108</td>
</tr>
<tr>
<td>HVAC temperature control</td>
<td>108</td>
</tr>
<tr>
<td>Background story</td>
<td>108</td>
</tr>
<tr>
<td>Input definitions</td>
<td>108</td>
</tr>
<tr>
<td>Detector model definition</td>
<td>110</td>
</tr>
<tr>
<td>BatchUpdateDetector example</td>
<td>122</td>
</tr>
<tr>
<td>BatchPutMessage examples</td>
<td>123</td>
</tr>
<tr>
<td>AWS IoT Core rules engine examples</td>
<td>127</td>
</tr>
<tr>
<td>Cranes</td>
<td>129</td>
</tr>
<tr>
<td>Background story</td>
<td>129</td>
</tr>
<tr>
<td>Commands</td>
<td>129</td>
</tr>
<tr>
<td>Detector models</td>
<td>130</td>
</tr>
<tr>
<td>Inputs</td>
<td>134</td>
</tr>
<tr>
<td>Messages</td>
<td>135</td>
</tr>
<tr>
<td>Event detection with sensors and applications</td>
<td>136</td>
</tr>
<tr>
<td>Device HeartBeat</td>
<td>137</td>
</tr>
<tr>
<td>ISA alarm</td>
<td>139</td>
</tr>
<tr>
<td>Simple alarm</td>
<td>145</td>
</tr>
<tr>
<td>Monitoring with alarms</td>
<td>149</td>
</tr>
<tr>
<td>Working with AWS IoT SiteWise</td>
<td>149</td>
</tr>
<tr>
<td>Acknowledge flow</td>
<td>149</td>
</tr>
<tr>
<td>Creating an alarm model</td>
<td>150</td>
</tr>
<tr>
<td>Requirements</td>
<td>150</td>
</tr>
<tr>
<td>Creating an alarm model (console)</td>
<td>151</td>
</tr>
<tr>
<td>Responding to alarms</td>
<td>152</td>
</tr>
<tr>
<td>Responding to alarms (console)</td>
<td>153</td>
</tr>
<tr>
<td>Responding to alarms (API)</td>
<td>153</td>
</tr>
<tr>
<td>Managing alarm notifications</td>
<td>153</td>
</tr>
<tr>
<td>Creating a Lambda function</td>
<td>153</td>
</tr>
<tr>
<td>Using the Lambda function provided by AWS IoT Events</td>
<td>159</td>
</tr>
<tr>
<td>Managing recipients</td>
<td>160</td>
</tr>
<tr>
<td>Security</td>
<td>161</td>
</tr>
<tr>
<td>Identity and access management</td>
<td>161</td>
</tr>
<tr>
<td>Audience</td>
<td>162</td>
</tr>
<tr>
<td>Authenticating with identities</td>
<td>162</td>
</tr>
<tr>
<td>Managing access using policies</td>
<td>164</td>
</tr>
<tr>
<td>Learn more</td>
<td>165</td>
</tr>
<tr>
<td>How AWS IoT Events works with IAM</td>
<td>165</td>
</tr>
<tr>
<td>Identity-based policy examples</td>
<td>168</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>171</td>
</tr>
<tr>
<td>Monitoring</td>
<td>173</td>
</tr>
<tr>
<td>Monitoring tools</td>
<td>174</td>
</tr>
<tr>
<td>Monitoring with Amazon CloudWatch</td>
<td>174</td>
</tr>
<tr>
<td>Logging AWS IoT Events API calls with AWS CloudTrail</td>
<td>175</td>
</tr>
<tr>
<td>Compliance validation</td>
<td>187</td>
</tr>
<tr>
<td>Resilience</td>
<td>188</td>
</tr>
<tr>
<td>Infrastructure security</td>
<td>188</td>
</tr>
<tr>
<td>Quotas</td>
<td>189</td>
</tr>
<tr>
<td>Tagging</td>
<td>190</td>
</tr>
<tr>
<td>Tag basics</td>
<td>190</td>
</tr>
<tr>
<td>Tag restrictions and limitations</td>
<td>191</td>
</tr>
<tr>
<td>Using tags with IAM policies</td>
<td>191</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>193</td>
</tr>
<tr>
<td>Common AWS IoT Events issues and solutions</td>
<td>193</td>
</tr>
<tr>
<td>Detector model creation errors</td>
<td>193</td>
</tr>
<tr>
<td>Updates from a deleted detector model</td>
<td>193</td>
</tr>
</tbody>
</table>

AWS IoT Events Developer Guide
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action trigger failure (when meeting a condition)</td>
<td>194</td>
</tr>
<tr>
<td>Action trigger failure (when breeching a threshold)</td>
<td>194</td>
</tr>
<tr>
<td>Incorrect state usage</td>
<td>194</td>
</tr>
<tr>
<td>Connection message</td>
<td>194</td>
</tr>
<tr>
<td>InvalidRequestException message</td>
<td>195</td>
</tr>
<tr>
<td>Amazon CloudWatch Logs action.setTimer errors</td>
<td>195</td>
</tr>
<tr>
<td>Amazon CloudWatch payload errors</td>
<td>195</td>
</tr>
<tr>
<td>Incompatible data types</td>
<td>196</td>
</tr>
<tr>
<td>Troubleshooting a detector model</td>
<td>197</td>
</tr>
<tr>
<td>Diagnostic information</td>
<td>198</td>
</tr>
<tr>
<td>Analyzing a detector model (Console)</td>
<td>205</td>
</tr>
<tr>
<td>Analyzing a detector model (AWS CLI)</td>
<td>206</td>
</tr>
<tr>
<td>Commands</td>
<td>210</td>
</tr>
<tr>
<td>AWS IoT Events actions</td>
<td>210</td>
</tr>
<tr>
<td>AWS IoT Events data</td>
<td>210</td>
</tr>
<tr>
<td>Document history</td>
<td>211</td>
</tr>
<tr>
<td>Earlier updates</td>
<td>211</td>
</tr>
</tbody>
</table>
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.

You can 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.

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).

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.

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 Amazon Connect, 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.
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.

Use cases

Monitor and Maintain Remote Devices

You need to monitor a fleet of remotely deployed machines. If one stops functioning, and you have no additional context for what's causing the failure, you might have to immediately replace 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 and diagnose the exact problem by using the error codes that are sent over time. Instead of replacing everything, you now have the information you need to send a technician with only the part that needs to be replaced. 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

You deploy robots inside your facilities to automate the movement of packages. To keep the cost of the robots to a minimum, the robots have simple, low-cost sensors that report information to the cloud. But your robots have dozens of sensors and hundreds of operating modes, making it difficult to detect problems as they occur. Using AWS IoT Events, you can build an expert system that processes sensor data in the cloud, and creates alerts to automatically warn technical staff if a failure is imminent.

Track Building Automation Systems

You operate a large number of data centers that must be monitored for high temperature and low humidity to prevent equipment failures that occur when these environmental thresholds are breached. The sensors you use are purchased from many manufacturers, and each type comes with its own management software. However, the management software from different vendors 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.
Setting up AWS IoT Events

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

Setting up permissions for AWS IoT Events

This section describes the roles and 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.

Action permissions

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. For more information, see Using the IAM console to manage roles and permissions (p. 5).

AWS IoT Events supports the following actions that let you use a timer or set a variable:
- setTimer (p. 89) to create a timer.
- resetTimer (p. 90) to reset the timer.
- clearTimer (p. 90) to delete the timer.
- setVariable (p. 90) to create a variable.

AWS IoT Events supports the following actions that let you work with AWS services:
- iotTopicPublish (p. 91) to publish a message on an MQTT topic.
- iotEvents (p. 92) to send data to AWS IoT Events as an input value.
- iotSiteWise (p. 92) to send data to an asset property in AWS IoT SiteWise.
- dynamoDB (p. 94) to send data to an Amazon DynamoDB table.
- dynamoDBv2 (p. 95) to send data to an Amazon DynamoDB table.
• **firehose (p. 96)** to send data to an Amazon Kinesis Data Firehose stream.
• **lambda (p. 97)** to invoke an AWS Lambda function.
• **sns (p. 97)** to send data as a push notification.
• **sqs (p. 98)** to send data to an Amazon SQS queue.

**Example Policy**

```json
{
  "Version": "2012-10-17",
  "Statement": [
  {
    "Effect": "Allow",
    "Action": "iot:Publish",
  },
  {
    "Effect": "Allow",
    "Action": "iotevents:BatchPutMessage",
  },
  {
    "Effect": "Allow",
    "Action": "iotsitewise:BatchPutAssetPropertyValue",
    "Resource": "*"
  },
  {
    "Effect": "Allow",
    "Action": "dynamodb:PutItem",
  },
  {
    "Effect": "Allow",
    "Action": ["firehose:PutRecord", "firehose:PutRecordBatch"],
  },
  {
    "Effect": "Allow",
    "Action": "lambda:InvokeFunction",
    "Resource": "arn:aws:lambda:<region>:<account_id>:function:*"
  },
  {
    "Effect": "Allow",
    "Action": "sns:Publish",
    "Resource": "arn:aws:sns:<region>:<account_id>:*
  },
  {
    "Effect": "Allow",
    "Action": "sqs:SendMessage",
    "Resource": "arn:aws:sqs:<region>:<account_id>:*
  }
  ]
}
```

**Securing input data**

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.

**Using the IAM console to manage roles and permissions**

The following example shows how to use the IAM console to grant permission to AWS IoT Events to generate Amazon SNS alerts on your behalf. You can also attach the role to any entity (user or account owner) that you will use to define and publish an AWS IoT Events detector model that contains event actions that send Amazon SNS alerts.

To complete this example, you need the ARN of an existing Amazon SNS topic to use in AWS IoT Events.

**To use the IAM console to manage roles and permissions**

1. Sign in to the AWS Management Console and open the IAM console at [https://console.aws.amazon.com/iam/](https://console.aws.amazon.com/iam/).
2. In the navigation pane, choose Dashboard, Roles.
3. Choose Create role.

4. On the Create role page, do the following.
   a. For Select type of trusted entity, choose AWS service.
   b. For Choose the service that will use this role, choose IoT.
   c. For Select your use case, choose IoT and choose Next: Permissions.
5. On the Create role page, for Attached permissions, leave these permissions as they are for now and choose Next: Tags.

You add a new policy granting the required permissions in a later step.
6. On the **Create role** page, for **Add tags (optional)**, don't add tags now and choose **Next:Review**.

7. On the **Review** page, enter a **Role name**, an optional **Role description**, and choose **Create role**.
8. On the Roles page, find and select the name of the role that you created.

9. On the Summary page for your role, on the Permissions tab, choose Attach policies.
10. On the **Attach Permissions** page, choose **Create policy**.

11. On the **Create Policy** page, select the **JSON** tab. If a policy validation failure warning appears, choose the **X** icon to dismiss it.
12. On the JSON tab, do the following.

   a. Replace the JSON in the editor with the following example.

   b. Change the Resource value to the ARN of the Amazon SNS topic that to use in AWS IoT Events.

   c. Choose Review policy. You can also use wildcard characters in the Amazon SNS topic ARN to grant broader permissions, but be aware of the security issues this raises.

   ```json
   
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": ["sns:*"],
         "Effect": "Allow",
      }
   ]
   
   Your policy should look like the following.
13. On the **Review policy** page, enter a **Name** for the policy, an optional **Description**, and choose **Create policy**.

14. On the **Policies** page, in the navigation pane, choose **Roles**.
15. On the Roles page, find and select the name of the role that you created.

16. On the Summary page for your role, on the Permissions tab, choose Attach policies.
17. On the **Attach permissions** page for your role, select the check box next to the policy that you created, and choose **Attach policy**.

18. On the **Roles** page, find and select the name of the role that you created.
19. On the **Summary** page for that role, on the **Trust relationships** tab, choose **Edit trust relationship**.

20. On the **Edit Trust Relationship** page, for **Policy Document**, replace the existing JSON with the following and choose **Update Trust Policy**.

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

Your policy should look like the following.
You have now granted AWS IoT Events permission to send alerts to your Amazon SNS topic on your behalf. For enhanced security, remove the unused policies that were attached by default to the role that you created.

**Amazon CloudWatch logging role policy**

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:

```
{
  "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 IAM 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](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_pass.html) in the IAM User Guide.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Action": ["iam:GetRole", "iam:PassRole"],
      "Resource": "arn:aws:iam::<account-id>: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 "\n  {
    "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::<account-id>:role/Role_To_Pass", "level": "INFO", "enabled": true } }"
```

Amazon SNS messaging role policy

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",
        }
    ]
}
```

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/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 that the input will monitor.

**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 an event is detected, it can change the state or trigger custom-built or predefined actions using other AWS services. You can define additional events that trigger 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 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. 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, 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 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 (p. 53). You can use the this JSON example to help you follow the tutorial.

**Topics**
- Prerequisites (p. 20)
- Create an input (p. 21)
- Create a detector model (p. 24)
- Send inputs to test the detector model (p. 40)

---

**Prerequisites**

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

2. Follow the online instructions.
   
   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.
3. 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.
4. Choose an AWS Region that supports AWS IoT Events. For more information, see AWS IoT Events in the AWS General Reference. For help, see Working with the AWS Management Console in the Getting Started with the AWS Management Console.
Create an input

In this topic, you define an input to receive telemetry data (messages).

1. Open the AWS IoT Events console.
2. In the AWS IoT Events console, choose Create detector model.
3. Choose Create new.
4. When you construct your detector models and inputs, you might want to gather files containing example message payloads that your devices or processes send to report their status. This makes it easier for you to define the inputs that you need.
**Example**

For this example, on your local file system, create a file named `input.json` with the following contents.

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

5. Choose **Create input**.

6. For the input, enter an **InputName**, a **Description**, and choose **Upload file**. In the dialog box, choose the JSON file that contains the example message ("input.json").
7. For **Choose input attributes**, select the attributes to use and choose **Create**. This example selects **motorid** and **sensorData.pressure** but doesn't use **sensorData.temperature**.
Create a detector model

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 trigger additional actions. These events are known as transition events.

In your states, you also define events that can execute 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 are executed 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 triggered 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. But its value can only be referenced (for example, in an event's conditional logic) after the detector has reached a state and executed 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**, **Event condition** and, choose **Add action**.

18. For **Choose an action**, choose **Send SNS message**.
   a. For **SNS topic**, enter the target ARN of your 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 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.

```json
{
    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 Detector model restrictions and limitations (p. 62).

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 will be created for you.

b. Choose **Create a detector for each unique key value**. To create and use your own **Role**, follow the steps in *Using the IAM console to manage roles and permissions* (p. 5) 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**.
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.

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

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

```json
{
  "detectorModel": {
    "detectorModelConfiguration": {
      "status": "ACTIVE",
      "lastUpdateTime": 1552072424.212,
      "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole",
      "creationTime": 1552072424.212,
      "key": "motorid",
      "detectorModelName": "motorDetectorModel",
      "detectorModelVersion": "1"
    },
    "detectorModelDefinition": {
      "states": [
```
AWS IoT Events Developer Guide

Create a detector model

```json
{
    "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"
            }
        ]
    }
}
```
Send inputs to test the detector model

There are several ways to receive telemetry data in AWS IoT Events (see Supported actions (p. 89)). 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 inputs to test the detector model.
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](https://aws.amazon.com/iot-core/). In the navigation pane, choose **Act**.

2. On the **Rules** page, choose **Create**.

3. On the **Create a rule** page, enter a **Name** and **Description**.
4. In the **Rule query statement**, enter the following.

```
SELECT *, topic(2) as motorid FROM 'motors/*/status'
```

5. In **Set one or more actions**, choose **Add action**.

6. On the **Select an action** page, select **Send a message to an AWS IoT Events Input** and choose **Configure action**.
7. On the **Configure action** page, do the following:
   a. For **Input name**, enter the name that you created in the previous section.
   b. For **Role**, choose **Create Role** and in the **Create a new role** window, enter a **Name** and choose **Create role**. This creates a role with permission to forward messages to AWS IoT Events.
   c. Back on the **Configure action** page, choose **Add action**.

8. On the **Create a rule** page, choose **Create rule**.

10. On the MQTT client page, choose Publish to a topic.
11. In the Publish section, enter the topic, enter the following payload in the editor, and choose Publish.

```json
{
   "sensorData": {
      "pressure": 23,
      "temperature": 47
   }
}
```
12. 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 (for example, **85**).

13. **Choose Publish**.

   The detector instance that you created generates and sends you an 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 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 SNS message indicating that condition.

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

- See more detector model examples (templates) on the console.
- Follow the steps in Simple step-by-step example (p. 55) to create an input and detector model using the AWS CLI
- Learn details of the Expressions (p. 100) used in events.
- Learn about Supported actions (p. 89).
- If something isn't working, see Troubleshooting AWS IoT Events (p. 193).
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 (p. 3) to create a role with an attached policy that grants permission to create and manage CloudWatch logs for AWS IoT Events.
2. In the AWS IoT Events console, choose the menu icon in the upper-left corner to open the navigation pane.
Enable Amazon CloudWatch logging when developing AWS IoT Events detector models.

If you're on the **Getting started** page, choose the X in the upper right to close that page and go to the **Detector model palette**. Choose the menu icon in the upper-left corner.

3. In the navigation pane, choose **Settings**.

4. On the **Settings** page, choose **Edit**.
5. On the **Edit logging options** page, do the following:

   a. Choose the **Level of verbosity**.
   
   b. For **Select role**, select a role with sufficient permissions to perform the logging actions that you chose.
   
   c. If you chose **Debug** for the **Level of verbosity**, you can also choose **Add Model Option** and add a **Detector Model Name** and (optional) **KeyValue** to specify the detector models and specific detectors (instances) to log.
   
   d. Choose **Update**.
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.
Store your AWS IoT Events data to avoid possible data loss due to a long period of inactivity

If you don’t use AWS IoT Events for a significant period of time, your data, including your detector models, might be deleted automatically. A significant period of time could mean, for example, you don’t incur charges and don’t create detector models. However, we won’t delete data or detector models without providing you with at least a 30 day notice prior. If you need to store data for an extended period of time, consider using AWS storage services.
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

Your devices and processes must have a way 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.
- Write an AWS IoT Core rule action for the AWS IoT Core rules engine that forwards your message data into AWS IoT Events. This identifies 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.
- 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 triggered the action are fed back in to 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 alone 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 trigger custom-built or predefined actions using other AWS services. You can define additional events that trigger 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.

Simple step-by-step example

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 trigger custom-built or predefined actions using other AWS services. You can define additional events that trigger 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 input to capture device data (p. 57)
Create an input to capture device 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 Create an input to capture device data (p. 57), 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".
Create a detector model to represent device states

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 (p. 3) 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.

```json
{
  "detectorModelName": "motorDetectorModel",
  "detectorModelDefinition": {
    "states": [
      {
        "stateName": "Normal",
        "onEnter": {
          "events": [
            {
              "eventName": "init",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "pressureThresholdBreached",
                    "value": "0"
                  }
                }
              ]
            }
          ]
        }
      },
      {
        "stateName": "Dangerous",
        "onEnter": {
          "events": [
            {
              "eventName": "Pressure Threshold Breached",
              "condition": "$input.PressureInput.sensorData.pressure > 70",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "pressureThresholdBreached",
                    "value": "$variable.pressureThresholdBreached + 3"
                  }
                }
              ],
              "nextState": "Dangerous"
            }
          ]
        }
      }
    ],
    "onInput": {
      "transitionEvents": [
        {
          "eventName": "Overpressurized",
          "condition": "$input.PressureInput.sensorData.pressure > 70",
          "actions": [
            {
              "setVariable": {
                "variableName": "pressureThresholdBreached",
                "value": "$variable.pressureThresholdBreached + 3"
              }
            }
          ],
          "nextState": "Dangerous"
        }
      ]
    }
  }
}
```
Create a detector model to represent device states

```
"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"
  }
}
```
Send messages as inputs to a detector

You have now defined an input that identifies the important fields in messages sent from a device (see Create an input to capture device data (p. 57)). 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 (p. 57)).

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
```

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
```

The file normalPressureMessage.json contains the following.

```
{
  "messages": [
```
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-db7a0f55267",
      "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"
      }
   },
   "eventName":"Normal Pressure Restored"
}
```
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**

```json
"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 trigger the actions or transition to the `nextState` specified in the event.

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 executed). You can use the [https://docs.aws.amazon.com/iotevents/latest/apireference/API_iotevents-data_DescribeDetector.html](https://docs.aws.amazon.com/iotevents/latest/apireference/API_iotevents-data_DescribeDetector.html) operation to get similar information about the state of the detector.

Detector model restrictions and limitations

"eventTriggerDetails":{
  "inputName":"PressureInput",
  "messageId":"00004",
  "triggerType":"Message"
},
"State":{
  "stateName":"Dangerous",
  "variables":{
    "pressureThresholdBreached":0
  },
  "timers":{}
},
"eventName":"Normal Pressure Restored"
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 evaluationMethod is set to SERIAL, variables are updated and event conditions evaluated in the order that the events are defined. Otherwise, variables are updated and events performed only after all event conditions are evaluated (when the evaluationMethod is set or defaults to BATCH).

The previous example shows what happens when evaluationMethod is set to or defaults to BATCH. 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).

```json
{
  "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.

```json
"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 control enters the onInput field, any condition only sees the value of a variable as it was when we started 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

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 HVAC temperature control (p. 108).

Background

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.

Input definitions

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",
    "lastUpdateTime": 1557519707.399,
  
```
Background

"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.

Detector model definition

The 'areaDetectorModel' example has comments inline.

CLI command:

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

File: areaDetectorModel.json

```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 > $variable.anomalousLow)",
    // This event is triggered if we have reentered the 'start' state using the
    // 'BatchUpdateDetector' API with 'resetMe' set to true. When we reenter
    // 'BatchUpdateDetector' we do not automatically continue to the 'idle'
    // state, but
    // wait in 'start' until the next input message arrives. This event enables
    // us to
    // transition to 'idle' on the next valid 'temperatureInput' message that
    // arrives.
    "actions": [
        {
            "setVariable": {
                "variableName": "averageTemperature",
                "value": "((($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": {
                    }
                }
            ]
        }
    ]
}
{  "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": "(((averageTemperature * (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": {
        }
    },
    {
        "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": {
      }
    },
    {
      "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": {
      }
    },
    {
      "iotTopicPublish": {
        "mqttTopic": "hvac/cooling/On"
      }
    },
    {
      "setVariable": {
        "variableName": "enteringNewState",
        "value": "true"
      }
    }
  ],
  "nextState": "cooling"
}
"mqttTopic": "hvac/Cooling/On"

// 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": {
    }
  },
  {
    "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"}]
```json
{
    "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",
          "variables": ["something"]
        }
      ]
    }
}
"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": {
}
},
{"sns": {
}
},
{"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": {
      },
    },
    {
      "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,
    "key": "areaId",
    "detectorModelName": "areaDetectorModel",
    "detectorModelVersion": "1"
  }
}

**BatchUpdateDetector example**

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:**

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

**File:** `areaDM.BUD.json`

```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": [
  ]
}


BatchPutMessage examples

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:

```bash
aws iotevents-data batch-put-message --cli-input-json file://seedExample.json
```

File: seedExample.json

```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:

```json
{
  "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:

```bash
aws iotevents-data batch-put-message --cli-input-json file://seedExample.json
```

File: seedExample.json

```json
{
  "messages": [
    {
      "inputName": "temperatureInput",
      "payload": "{"areaId": "Area51", "instanceId": "Sensor2", "temperature": 25.0, "timestamp": "2023-10-01T10:00:00Z"}"
    }
  ]
}
```

Response:

```json
{
  "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:

```bash
aws iotevents-data batch-put-message --cli-input-json file://seedSetDesiredTemp.json
```

File: seedSetDesiredTemp.json

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

Response:

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

Example: Ingesting MQTT messages

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".
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`

```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/anotherRole"
        }
      }
    ],
    "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.

```json
{
  "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
```

File: `spoofExample.json`

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

The following are examples of 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,"resetHeatCool":true,"duration":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,"resetHeatCool":true,"duration":0.7,"desiredTemperature":20.0,"anomalousHigh":60.0,"reportedTemperature":0.1,"anomalousLow":0.0,"sensorId":0,"timers":{}}},"eventName":"resetHeatCool"}

Example: DescribeDetector API

You can use the DescribeDetector operation to see the current state, variable values, and timers for a detector instance.

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"
            }
         ]
   }
}
```
AWS IoT Core rules engine examples

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

```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

```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

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.

**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 Kinesis 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.

Using built-in 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.

Working with other AWS services

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

- `iotTopicPublish` (p. 91) to publish a message on an MQTT topic.
- `iotEvents` (p. 92) to send data to AWS IoT Events as an input value.
- `iotSiteWise` (p. 92) to send data to an asset property in AWS IoT SiteWise.
- `dynamoDB` (p. 94) to send data to an Amazon DynamoDB table.
- `dynamoDBv2` (p. 95) to send data to an Amazon DynamoDB table.
- `firehose` (p. 96) to send data to an Amazon Kinesis Data Firehose stream.
- `lambda` (p. 97) to invoke an AWS Lambda function.
- `sns` (p. 97) to send data as a push notification.
- `sqs` (p. 98) 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 (p. 100) and the Payload data type in the AWS IoT Events API Reference.

AWS IoT Core

IoT topic publish action

The `iotTopicPublish` 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 Message broker for AWS IoT in the AWS IoT Developer Guide.

More information

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 (p. 161)*.

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

---

**AWS IoT Events**

IoT Events action

The *iotEvents* 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 (p. 161)*.

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

---

**AWS IoT SiteWise**

IoT SiteWise action

The *iotSiteWise* 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.

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 (p. 161).

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

Amazon DynamoDB

DynamoDB action

The `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 (p. 161).

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

Amazon DynamoDB(v2)

DynamoDBv2 action

The dynamoDBv2 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 (p. 161).

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

---

Amazon Kinesis Data Firehose

Firehose action

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

Amazon Kinesis 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 Kinesis Data Firehose? in the Amazon Kinesis Data Firehose Developer Guide.

More information (3)

When you send data to a Kinesis Data Firehose delivery stream, you must specify the following parameters.

**deliveryStreamName**

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

**separator**

(Optional) You can use a character separator to separate continuous data sent to the Kinesis Data Firehose delivery stream. The separator value must be `\n` (newline), `\t` (tab), `\r\n` (Windows new line), or `comma` (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 (p. 161).

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

AWS Lambda

Lambda action

The 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 (p. 161).

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

Amazon Simple Notification Service

SNS action

The 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.
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 (p. 161).

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

---

**Amazon Simple Queue Service**

**SQS action**

The `sns` action lets you send data to an Amazon SQS queue. For the list of supported Regions, see [Amazon Simple Queue Service endpoints and quotas](https://aws.amazon.com/documentation/sqs/endpoints/) 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*.

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 (p. 161).

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 Amazon Connect, 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

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.

Syntax

You can use literals, operators, functions, references, and substitution templates in the AWS IoT Events expressions.

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

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".
### Functions

- 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($input.MyInput.status, "offline")
```
startsWith($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.

### References

#### 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.TechinicianID
```

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 `time-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.

<table>
<thead>
<tr>
<th>Incompatible pairs of types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer, string</td>
</tr>
<tr>
<td>Integer, Boolean</td>
</tr>
<tr>
<td>Decimal, string</td>
</tr>
<tr>
<td>Decimal, Boolean</td>
</tr>
<tr>
<td>String, Boolean</td>
</tr>
</tbody>
</table>
Substitution templates

'${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.

<table>
<thead>
<tr>
<th>Substitution template</th>
<th>Result string</th>
</tr>
</thead>
<tbody>
<tr>
<td>'${Sensor ' + $variable.SensorID}'</td>
<td>&quot;Sensor 10&quot;</td>
</tr>
<tr>
<td>'Sensor ' + '${$variable.SensorID + 1}'</td>
<td>&quot;Sensor 11&quot;</td>
</tr>
<tr>
<td>'Sensor 10: ${$variable.SensorID == 10}'</td>
<td>&quot;Sensor 10: true&quot;</td>
</tr>
<tr>
<td>'{&quot;sensor&quot;:&quot;${$variable.SensorID + 1}&quot;}'</td>
<td>'{&quot;sensor&quot;:11}&quot;</td>
</tr>
<tr>
<td>'{&quot;sensor&quot;:${$variable.SensorID + 1}}'</td>
<td>'{&quot;sensor&quot;:11}&quot;</td>
</tr>
</tbody>
</table>

Expression usage

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.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>'123'</td>
<td># Integer</td>
</tr>
<tr>
<td>'123.12'</td>
<td># Decimal</td>
</tr>
<tr>
<td>'hello'</td>
<td># String</td>
</tr>
<tr>
<td>'true'</td>
<td># Boolean</td>
</tr>
</tbody>
</table>
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, Greengrass Temperature Table 2000-01-01.

```
'Greenhouse Temperature Table ' + $input.GreenhouseInput.date
```
Detector model examples

This section contains examples of detector models and inputs.

Topics
- HVAC temperature control (p. 108)
- Cranes (p. 129)
- Event detection with sensors and applications (p. 136)
- Device HeartBeat (p. 137)
- ISA alarm (p. 139)
- Simple alarm (p. 145)

HVAC temperature control

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

A "seedTemperatureInput" is used to create a detector instance for an area and define its operational parameters.
CLI command used:

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

File: `seedInput.json`

```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:

```json
{
  "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:

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

File: `temperatureInput.json`

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

Response:
Detector model definition

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:

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

File: areaDetectorModel.json

```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}"
        }
    }
],
"nextState": "idle",

"eventName": "reset",
"condition": "($variable.resetMe == true) &&
(\${input.temperatureInput.sensorData.temperature} < \$variable.anomalousHigh &&
\${input.temperatureInput.sensorData.temperature} > \$variable.anomalousLow)",
"actions": [
Detector model definition

{
    "setVariable": {
        "variableName": "averageTemperature",
        "value": "(((variable.averageTemperature * (sensorCount - 1)) + input.temperatureInput.sensorData.temperature) / sensorCount)"
    },
    "nextState": "idle"
},

"onExit": {
    "events": [
    {
        "eventName": "resetHeatCool",
        "condition": "true",
        "actions": [
            {
                "sns": {
                }
            },
            {
                "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 > $variable.rangeHigh",
    "actions": [
      {
        "iotTopicPublish": {
          "mqttTopic": "temperatureSensor/spike"
        }
      },
      {"sns": {
      }
    },
    {"iotTopicPublish": {
      "mqttTopic": "hvac/Cooling/On"
    }
  },
],
{
    "setVariable": {
        "variableName": "enteringNewState",
        "value": "true"
    }
},
"nextState": "cooling"
},
{
    "eventName": "lowTemperatureSpike",
    "condition": "$input.temperatureInput.sensorData.temperature < $variable.rangeLow",
    "actions": [
        {
            "iotTopicPublish": {
                "mqttTopic": "temperatureSensor/spike"
            }
        },
        {
            "sns": {
            }
        },
        {
            "iotTopicPublish": {
                "mqttTopic": "hvac/Heating/On"
            }
        },
        {
            "setVariable": {
                "variableName": "enteringNewState",
                "value": "true"
            }
        }
    ],
    "nextState": "heating"
},
{
    "eventName": "highTemperatureThreshold",
    "condition": "(((($variable.averageTemperature * ($variable.sensorCount - 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) > ($variable.desiredTemperature + $variable.allowedError))",
    "actions": [
        {
            "sns": {
            }
        },
        {
            "iotTopicPublish": {
                "mqttTopic": "hvac/Cooling/On"
            }
        },
        {
            "setVariable": {
                "variableName": "enteringNewState",
                "value": "true"
            }
        }
    ],
    "nextState": "cooling"
}
Detector model definition

```json
{
  "eventName": "lowTemperatureThreshold",
  "condition": "(((($variable.averageTemperature * ($variable.sensorCount - 1)) + $input.temperatureInput.sensorData.temperature) / $variable.sensorCount) < ($variable.desiredTemperature - $variable.allowedError))",
  "actions": [
    {
      "sns": {
      }
    },
    {
      "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",
     "stateName": "cooling"
    }
  ]
}
```
"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": "coolingTimer",
            "value": "false"
          }
        }
      ]
    }
  ]
}
"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": {
        }
      },
      {
        "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"
            }
          }
        ]
      }
    ],
    "onExit": {
      "events": [
        {
          "eventName": "exit",
          "condition": "variable.into_IDLE",
          "actions": [
            {
              "sns": {
                "targetArn": "arn:aws:sns:us-west-2:123456789012:exit/temperature"
              }
            },
            {
              "iotTopicPublish": {
                "mqttTopic": "hvac/Temp/Exit"
              }
            }
          ]
        }
      ],
      "onExit": {
        "events": [
          {
            "eventName": "exit",
            "condition": "variable.into_IDLE",
            "actions": [
              {
                "sns": {
                  "targetArn": "arn:aws:sns:us-west-2:123456789012:exit/temperature"
                }
              },
              {
                "iotTopicPublish": {
                  "mqttTopic": "hvac/Temp/Exit"
                }
              }
            ]
          }
        ],
        "nextState": "idle"
      }
    }
  }
}
}
{
    "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)"
                    }
                }
            ]
        }
    ]
}
Detector model definition

```json
{
  "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": {
          }
        },
        {
          "sns": {
          }
        },
        {
          "iotTopicPublish": {
            "mqttTopic": "hvac/Heating/Off"
          }
        },
        {
          "iotTopicPublish": {
            "mqttTopic": "hvac/Cooling/On"
          }
        },
        {
          "setVariable": {
            "variableName": "enteringNewState",
            "value": "true"
          }
        }
      ]
    }
  ]
}```
Detector model definition

```json
{
    "detectorModelConfiguration": {
        "status": "ACTIVATING",
        "lastUpdateTime": 1557523491.168,
        "roleArn": "arn:aws:iam::123456789012:role/IoTEventsRole",
        "creationTime": 1557523491.168,
        "key": "areaId",
        "detectorModelName": "areaDetectorModel",
        "detectorModelVersion": "1"
    }
}
Response:

```
BatchUpdateDetector example

In this example, "BatchUpdateDetector" is used to change operational parameters for a working detector instance.

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",
            122
```
BatchPutMessage examples

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

**CLI command used:**

```bash
aws iotevents-data batch-put-message --cli-input-json file://seedExample.json
```

**File:** seedExample.json

```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:**

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

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

**CLI command used:**

```bash
aws iotevents-data batch-put-message --cli-input-json file://temperatureExample.json
```

**Response:**

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

File: temperatureExample.json

```json
{
   "messages": [
      {
         "messageId": "00001",
         "inputName": "seedTemperatureInput",
         "payload": "{\"areaid\": \"Area51\", \"reportedTemperature\": "0.1", \"resetMe\": true}\n      },
      {
         "timers": [
            {}
         ]
      }
   ]
}
```
File: temperatureExample.json

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

Response:

```json
{
  "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
```

File: seedSetDesiredTemp.json

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

Response:

```json
{
  "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"
    }
  }
}
```
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:

```json
{
    "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"
                }
            ]
        }
    }
}```
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"
      }
    }]
  }
}
```
"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"
        }
      }
    ]
  }
}
```
Background story

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.

Commands

```bash
# 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

# Send HighTemperature message
aws iotevents-data batch-put-message --cli-input-json file://messages/highTemperatureMessage.json

# Send LowPressure message
aws iotevents-data batch-put-message --cli-input-json file://messages/lowPressureMessage.json

# Send LowTemperature message
aws iotevents-data batch-put-message --cli-input-json file://messages/lowTemperatureMessage.json

## Detector models

**File: craneDetectorModel.json**

```json
{
  "detectorModelName": "craneDetectorModel",
  "detectorModelDefinition": {
    "states": [
      {
        "stateName": "Running",
        "onEnter": {
          "events": [
            {
              "eventName": "init",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "craneThresholdBreached",
                    "value": "0"
                  }
                }
              ]
            }
          ]
        }
      }
    ]
  }
}
```
To update an existing detector model. File: updateCraneDetectorModel.json

```json
{
  "detectorModelName": "craneDetectorModel",
  "detectorModelDefinition": {
    "states": [
      {
        "stateName": "Running",
        "onEnter": {
          "events": [
            {
              "eventName": "init",
              "condition": "true",
              "actions": [
                {
                  "setVariable": {
                    "variableName": "craneThresholdBreached",
                    "value": "0"
                  }
                }
              ]
            }
          ]
        }
      }
    ],
    "initialStateName": "Running"
  },
  "key": "craneid",
  "roleArn": "arn:aws:iam::123456789012:role/columboSNSRole"
}
```
Detector models

```json
{
  "definition": {
    "type": "detector",
    "name": "TemperatureThresholdDetector",
    "initialStateName": "Running",
    "roleArn": "arn:aws:iam::123456789012:role/columboSNSRole",
    "onInput": {
      "events": ["Overheated", "Crane Threshold Breached", "Underheated"],
      "actions": [
        { "setVariable": { "variableName": "craneThresholdBreached", "value": "0" } },
        { "setVariable": { "variableName": "alarmRaised", "value": "false" } }
      ]
    }
  }
}
```


Detector models

File: motorDetectorModel.json

```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
```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"
}

Inputs

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"}
      ]
   }
}

Messages

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\"}"
   }
   ]
}
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.

```json
{
    "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": []
                }
            }]
    }
}
```
Device HeartBeat

This detector model is one of the templates available from the AWS IoT Events console. It's included here for your convenience.
"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": {  "onTransitionEvents": [
  {
    "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"
          }
        }
    ]
  ]
}
ISA alarm

This detector model is one of the templates available from the AWS IoT Events console. It’s included here for your convenience.
"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 != $variable.lower_threshold"
    }
  ],
  "stateName": "Normal",
  "onEnter": {
    "events": [
      {
        "eventName": "Create Config variables",
        "actions": [
          {
            "setVariable": {
              "variableName": "lower_threshold",
              "value": "$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"
      }
    ]
  }
```

---

141
"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": []
}
"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 == \"return_to_service\"
    }
  ],
  "events": []
}
"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": []
    },
Simple alarm

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"
      }
    ]
  }
}
Simple alarm

```json
{
   "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"
                  }
               }
            }
         }
      ]
   }
}
```
Simple alarm

```json
{},
  "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

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

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, you must create an input 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.

For example, you can use the `sensorData.temperature` attribute in the following AWS IoT Events input:

```json
{
   "input": {
      "inputConfiguration": {
         "inputName": "TemperatureInput",
         "inputDescription": "Temperature readings from a sensor",
         "inputArn": "arn:aws:iotevents:us-east-1:123456789012:input/TemperatureInput",
         "creationTime": "2020-11-29T20:07:07.161000-08:00",
         "lastUpdateTime": "2020-11-29T20:07:07.161000-08:00",
         "status": "ACTIVE"
      },
      "inputDefinition": {
         "attributes": [
            {
               "jsonPath": "sensorData.temperature"
            }
         ]
      }
   }
}
```

- 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.
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:
   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.
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 AWS SSO store before you can add them to your alarm model. For more information, see Managing recipients (p. 160).

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 (p. 149)**.

   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 Working with other AWS services (p. 91).

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**

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.
Responding to alarms (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 (p. 149).

Responding to alarms (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

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.

Creating a Lambda function

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 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"
         ],
         "Resource": "*"
      },
      {
         "Effect": "Deny",
         "Action": [
            "sns:Publish"
         ],
         "Resource": "arn:aws:sns:*:*:*"
      }
   ]
}
```

- 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.

## Deploying a Lambda function

This tutorial uses an AWS 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 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 Configuring the AWS CLI 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 (p. 157).
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

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:**

```python
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'):
        logger.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
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 \n'.format(alarm_state['alarmModelName'], alarm_state['stateName'], timestamp)
    if (alarm_state['ruleEvaluation']):
        property = alarm_state['ruleEvaluation']['simpleRule']['inputProperty']
        default_msg += 'Sev: ' + str(nep['severity']) + ' \n'
        default_msg += alarm_msg + 'Current Value: ' + str(property) + ' \n'
        operator = alarm_state['ruleEvaluation']['simpleRule']['operator']
        threshold = alarm_state['ruleEvaluation']['simpleRule']['threshold']
        alarm_msg += '{\{} \{} \} \n'.format(str(property), operator, str(threshold))
        default_msg += alarm_msg + '({} {} {})'.format(str(property), operator, str(threshold))
    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 + 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 + 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.

```yaml
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"
              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
```

157
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
    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'] + '
' + alarm_state['stateUpdateTime'] + ' UTC '.format(alarm_state['stateName'], timestamp)
    alarm_msg = '{} {} at {} UTC'.format(nep['alarmModelName'], nep.get('keyValue', 'Singleton'), alarm_state['stateName'], timestamp)
    default_msg += 'Sev: ' + str(nep['severity']) + '
' + alarm_msg + '
'
    if (alarm_state['ruleEvaluation']):
        property = alarm_state['ruleEvaluation']['simpleRule']['inputProperty']
        default_msg += 'Current Value: ' + str(property) + '
' + alarm_msg + '
'
    operator = alarm_state['ruleEvaluation']['simpleRule']['operator']
    threshold = alarm_state['ruleEvaluation']['simpleRule']['threshold']
    default_msg += '}()}.format(str(property), operator, str(threshold))
    default_msg += alarm_msg + '
'
    emails = event.get('emailConfigurations', [])
    logger.info('Start Sending Emails')
AWS IoT Events Developer Guide
Using the Lambda function provided by AWS IoT Events

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 + '
' + 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 + '
' + 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.SMSSenderId': ('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

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 email addresses in Amazon SES, 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:

<table>
<thead>
<tr>
<th>Country</th>
<th>Local phone number</th>
<th>E.164 formatted number</th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>206-555-0100</td>
<td>+12065550100</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>020-1234-1234</td>
<td>+442012341234</td>
</tr>
<tr>
<td>Lithuania</td>
<td>8+601+12345</td>
<td>+37060112345</td>
</tr>
</tbody>
</table>
To find a country calling code, go to [countrycode.org](http://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.

Managing recipients

AWS IoT Events uses AWS Single Sign-On (AWS SSO) to manage the SSO access of alarms recipients. To enable the alarm to send notifications to the recipients, you must enable AWS SSO and add recipients to your AWS SSO store. For more information, see Add Users in [AWS Single Sign-On User Guide](https://docs.aws.amazon.com/singlesignon/latest/userguide/

**Important**

- You must choose the same AWS Region for AWS IoT Events, AWS Lambda, and AWS SSO.
- AWS Organizations only supports one AWS SSO Region at a time. If you want to make AWS SSO available in a different Region, you must first delete your current AWS SSO configuration. For more information, see [AWS SSO Region Data](https://docs.aws.amazon.com/singlesignon/latest/userguide/SSO_REGION_DATA.html) in [AWS Single Sign-On User Guide](https://docs.aws.amazon.com/singlesignon/latest/userguide/).
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 (p. 161)
- Monitoring AWS IoT Events (p. 173)
- Compliance validation for AWS IoT Events (p. 187)
- Resilience in AWS IoT Events (p. 188)
- Infrastructure security in AWS IoT Events (p. 188)

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 (p. 162)
- Authenticating with identities (p. 162)
- Managing access using policies (p. 164)
- Learn more (p. 165)
- How AWS IoT Events works with IAM (p. 165)
- AWS IoT Events identity-based policy examples (p. 168)
• Troubleshooting AWS IoT Events identity and access (p. 171)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work you do in AWS IoT Events.

Service user – If you use the AWS IoT Events service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS IoT Events features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in AWS IoT Events, see Troubleshooting AWS IoT Events identity and access (p. 171).

Service administrator – If you're in charge of AWS IoT Events resources at your company, you probably have full access to AWS IoT Events. It's your job to determine which AWS IoT Events features and resources your employees should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with AWS IoT Events, see How AWS IoT Events works with IAM (p. 165).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS IoT Events. To view example AWS IoT Events identity-based policies that you can use in IAM, see AWS IoT Events identity-based policy examples (p. 168).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see The IAM console and sign-in page in the IAM User Guide.

You must be authenticated (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication, or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email or your IAM user name. You can access AWS programmatically using your root user or IAM user access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using Signature Version 4, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the AWS General Reference.

Regardless of the authentication method that you use, you might also be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Using multi-factor authentication (MFA) in AWS in the IAM User Guide.

AWS account root user

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.
IAM users and groups

An IAM user is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the IAM User Guide. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

An IAM group is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named IAMAdmins and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the IAM User Guide.

IAM roles

An IAM role is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the IAM User Guide.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as federated users. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated users and roles in the IAM User Guide.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the IAM User Guide.
- **AWS service access** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the IAM User Guide.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the IAM User Guide.

To learn whether to use IAM roles, see When to create an IAM role (instead of a user) in the IAM User Guide.
Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when an entity (root user, IAM user, or IAM role) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the IAM User Guide.

An IAM administrator can use policies to specify who has access to AWS resources, and what actions they can perform on those resources. Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, role, or group. These policies control what actions that identity can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the IAM User Guide.

Identity-based policies can be further categorized as inline policies or managed policies. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the IAM User Guide.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity’s identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the IAM User Guide.

- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the AWS Organizations User Guide.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. 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.

Learn more

For more information about identity and access management for AWS IoT Events, continue to the following pages:
- How AWS IoT Events works with IAM (p. 165)
- Troubleshooting AWS IoT Events identity and access (p. 171)

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 (p. 165)
- AWS IoT Events resource-based policies (p. 167)
- Authorization based on AWS IoT Events tags (p. 167)
- AWS IoT Events IAM roles (p. 167)

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 Amazon Resource Names (ARNs) and AWS service namespaces.

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 (*):

```
```

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 an IAM 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 an IAM user permission to access a resource only if it is tagged with their IAM 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 (p. 168).

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 (p. 190).

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Viewing AWS IoT Events inputs based on tags (p. 171).

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, IAM 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 IAM 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 (p. 168)
- Using the AWS IoT Events console (p. 168)
- Allow users to view their own permissions (p. 169)
- Accessing one AWS IoT Events input (p. 170)
- Viewing AWS IoT Events inputs based on tags (p. 171)

**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 IAM 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 (IAM 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-data:BatchPutMessage",
            "iotevents-data:BatchUpdateDetector",
            "iotevents:CreateDetectorModel",
            "iotevents:CreateInput",
            "iotevents:DeleteDetectorModel",
            "iotevents:DeleteInput",
            "iotevents-data:DescribeDetector",
            "iotevents:DescribeDetectorModel",
            "iotevents:DescribeInput",
            "iotevents:DescribeLoggingOptions",
            "iotevents-listDetectorModelVersions",
            "iotevents-listDetectorModels",
            "iotevents-data-listDetectors",
            "iotevents-listInputs",
            "iotevents-listTagsForResource",
            "iotevents-putLoggingOptions",
            "iotevents:TagResource",
            "iotevents-UntagResource",
            "iotevents-updateDetectorModel",
            "iotevents-updateInput",
            "iotevents-updateInputRouting"
         ],
         "Resource": "arn:${Partition}:iotevents:${Region}:${Account}:detectorModel/${detectorModelName}"
      },
      {
         "Effect": "Allow",
         "Action": [
            "iam:GetUserPolicy",
            "iam:ListGroupsForUser",
            "iam:ListAttachedUserPolicies",
            "iam:ListUserPolicies",
            "iam:GetUser"
         ],
         "Resource": "arn:${Partition}:iotevents:${Region}:${Account}:input/${inputName}"
      }
   ]
}
```

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**

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. 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:${Partition}:iotevents:${Region}:${Account}:input/${inputName}"
      }
   ]
}
```
Accessing one AWS IoT Events input

In this example, you want to grant an IAM user in your AWS account access to one of your AWS IoT Events inputs, exampleInput. You also want to 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 S3 service that grants permissions to users and tests them using the console, see An example walkthrough: Using user policies to control access to your bucket.

```json
{
    "Version":"2012-10-17",
    "Statement":[
        {
            "Sid":"ListInputsInConsole",
            "Effect":"Allow",
            "Action": [ 
                "iotevents:ListInputs"
            ],
            "Resource":"arn:aws:iotevents::*:**"
        },
        {
            "Sid":"ViewSpecificInputInfo",
            "Effect":"Allow",
            "Action": [ 
                "iotevents:DescribeInput"
            ],
            "Resource":"arn:aws:iotevents::exampleInput"
        },
        {
            "Sid":"ManageInputs",
            "Effect":"Allow",
            "Action": [ 
                "iotevents:CreateInput",
                "iotevents:DeleteInput",
                "iotevents:DescribeInput",
                "iotevents:ListInputs",
                "iotevents:UpdateInput"
            ],
            "Resource":"arn:aws:iotevents::*:**"
        }
    ]
}
```
Viewing AWS IoT Events *inputs* based on tags

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.

```
{
"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 IAM 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*.

Troubleshooting 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 (p. 171)
- I Am not authorized to perform iam:PassRole (p. 172)
- I want to view my access keys (p. 172)
- I'm an administrator and want to allow others to access AWS IoT Events (p. 172)
- I want to allow people outside of my AWS account to access my AWS IoT Events resources (p. 173)

**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, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies 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 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 asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

**I want to view my access keys**

After you create your IAM user access keys, you can view your access key ID at any time. However, you can’t view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfYiZwYX3vY mapper`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

**Important**

Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing access keys in the IAM User Guide.

**I'm an administrator and want to allow others to access AWS IoT Events**

To allow others to access AWS IoT Events, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in AWS IoT Events.

To get started right away, see Creating your first IAM delegated user and group in the IAM User Guide.
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.

To learn more, consult the following:

• To learn whether AWS IoT Events supports these features, see How AWS IoT Events works with IAM (p. 165).
• 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 How IAM roles differ from resource-based policies in the IAM User Guide.

Monitoring AWS IoT Events

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

• Monitoring tools (p. 174)
• Monitoring with Amazon CloudWatch (p. 174)
• Logging AWS IoT Events API calls with AWS CloudTrail (p. 175)
Monitoring tools

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 Monitoring log files in the Amazon CloudWatch User Guide.
- **Amazon CloudWatch Events** – Match events and route them to one or more target functions or streams to make changes, capture state information, and take corrective action. For more information, see What is Amazon CloudWatch Events 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 customized dashboards 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 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 Amazon Web Services (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 (p. 48) 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.

```json
{
  "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 Viewing events 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:

- Overview for creating a trail
- 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.

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"
}
```

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-Lambda/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/IotEvents-Lambda",
        "accountId": "123456789012",
        "userName": "Admin"
      }
    }
  },
  "eventTime": "2019-02-08T19:02:44Z",
  "eventSource": "iotevents.amazonaws.com",
  "eventName": "CreateDetectorModel",
  "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"
}
```
Logging AWS IoT Events API calls with AWS CloudTrail

The following example shows a CloudTrail log entry that demonstrates the CreateInput action.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DBEXAMPLE:IoTEvents-Lambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEvents-RoleForIotEvents-ABC123DEFG456/IotEvents-Lambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-ABC123DEFG456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABC123DEFG456"
      }
    },
    "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",
      "roleArn": "arn:aws:iam::123456789012:role/events_action_execution_role"
    },
    "responseElements": null,
    "requestID": "cecfbfa1-e452-4fa6-b86b-89a89f392b66",
    "eventID": "8138d46-b-50a3-4af0-9c5e-5af5e5a55",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
  }
}
```

The following example shows a CloudTrail log entry that demonstrates the CreateInput action.
The following example shows a CloudTrail log entry that demonstrates the `DeleteDetectorModel` action.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DBEXAMPLE",
        "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": "7669db89-0cc0-4c42-904b-f24b764dd808",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

The following example shows a CloudTrail log entry that demonstrates the `DeleteInput` action.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DBEXAMPLE",
        "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": "DeleteInput",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "192.168.0.1",
  "userAgent": "aws-internal/3",
  "requestParameters": {
    "inputName": "batchputmessagedetectorupdated",
    "inputDescription": "batchputmessagedetectorupdated"
  },
  "responseElements": null,
  "requestID": "149064c1-4e24-4160-a5b2-1065e63ee2e4",
  "eventID": "7669db89-0cc0-4c42-904b-f24b764dd808",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```
The following example shows a CloudTrail log entry that demonstrates the DescribeDetectorModel action.
The following example shows a CloudTrail log entry that demonstrates the DescribeInput action.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AAKIAI44QH8DHBEEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEEXAMPLE",
        "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": "bad80ff8-3e2b-4073-ac38-9cb3385beb04",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

The following example shows a CloudTrail log entry that demonstrates the DescribeLoggingOptions action.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DHBEEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AAKIAI44QH8DHBEEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DHBEEXAMPLE",
        "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": "bad80ff8-3e2b-4073-ac38-9cb3385beb04",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```
The following example shows a CloudTrail log entry that demonstrates the `ListDetectorModels` action.

```json
{
"eventVersion": "1.05",
"userIdentity": {
 "type": "AssumedRole",
 "principalId": "AKIAI44QH8DHXBEXAMPLE:IoTEvents-EventsLambda",
 "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
 "accountId": "123456789012",
 "accessKeyId": "AKIAI44QH8DHXBEXAMPLE",
 "sessionContext": {
  "attributes": {
   "mfaAuthenticated": "false",
   "creationDate": "2019-02-07T22:22:30Z"
  },
  "sessionIssuer": {
   "type": "Role",
   "principalId": "AKIAI44QH8DHXBEXAMPLE",
   "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456",
   "accountId": "123456789012",
   "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
  }
},
"eventTime": "2019-02-07T22:53:23Z",
"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",
"eventType": "AwsApiCall",
"recipientAccountId": "123456789012"
}
```
The following example shows a CloudTrail log entry that demonstrates the **ListDetectorModelVersions** action.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAI44QH8DHBEEXAMPLE:IoTEvents-EventsLambda",
        "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
        "accountId": "123456789012",
        "accessKeyId": "AKIAI44QH8DHBEEXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2019-02-07T22:22:30Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AKIAI44QH8DHBEEXAMPLE",
                "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": {
        "nextToken": "CkZEZXRLy3Rvck1vZGVsMl9saXN0ZGV0ZWN0b3JtZb3RhHN0ZXNoX2VlOWJkZTk1YT",
        "maxResults": 2
    },
    "responseElements": null,
    "requestID": "ebecb277-6bd8-44ea-8abd-fbf40ac044ee",
    "eventID": "fc6281a2-3fac-4e1e-98e0-ca6560b8b8be",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

The following example shows a CloudTrail log entry that demonstrates the **ListDetectors** action.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AKIAI44QH8DHBEEXAMPLE:IoTEvents-EventsLambda",
        "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
        "accountId": "123456789012",
        "accessKeyId": "AKIAI44QH8DHBEEXAMPLE",
        "sessionContext": {
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2019-02-07T22:22:30Z"
            },
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AKIAI44QH8DHBEEXAMPLE",
                "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": "ListDetectorModels",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "192.168.0.1",
    "userAgent": "aws-internal/3",
    "requestParameters": {
        "nextToken": "CkZEZXRLy3Rvck1vZGVsMl9saXN0ZGV0ZWN0b3JtZb3RhHN0ZXNoX2VlOWJkZTk1YT",
        "maxResults": 3
    },
    "responseElements": null,
    "requestID": "ebecb277-6bd8-44ea-8abd-fbf40ac044ee",
    "eventID": "fc6281a2-3fac-4e1e-98e0-ca6560b8b8be",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

The following example shows a CloudTrail log entry that demonstrates the **ListDetectors** action.
The following example shows a CloudTrail log entry that demonstrates the `ListInputs` action.

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DBEXAMPLE:IotEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DBEXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456",
        "accountId": "123456789012",
        "userName": "IotEventsLambda-RoleForIotEvents-ABCD123DEF456"
      }
    },
    "eventTime": "2019-02-07T22:53:54Z",
    "eventName": "ListInputs",
    "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-afe6-4ae6-afd2-a0bb1e9614a9",
    "eventSource": "iotevents.amazonaws.com",
    "eventSourceArn": "arn:aws:iot::123456789012:detector-instance/hello"
  }
}
```
The following example shows a CloudTrail log entry that demonstrates the `PutLoggingOptions` action:

```json
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "AKIAI44QH8DH8EXAMPLE:IoTEvents-EventsLambda",
    "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
    "accountId": "123456789012",
    "accessKeyId": "AKIAI44QH8DH8EXAMPLE",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2019-02-07T22:22:30Z"
      },
      "sessionIssuer": {
        "type": "Role",
        "principalId": "AKIAI44QH8DH8EXAMPLE",
        "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": "d5f70e50-fb19-4636-9ec0-e150a94bc52c",
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```
The following example shows a CloudTrail log entry that demonstrates the `UpdateDetectorModel` action.

```json
{
"eventVersion": "1.05",
"userIdentity": {
  "type": "AssumedRole",
  "principalId": "AKIAI44QH8DH8BEXAMPLE:IotEvents-EventsLambda",
  "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
  "accessKeyId": "AKIAI44QH8DH8BEXAMPLE",
  "sessionContext": {
    "attributes": {
      "mfaAuthenticated": "false",
      "creationDate": "2019-02-07T22:22:30Z"
    },
    "sessionIssuer": {
      "type": "Role",
      "principalId": "AKIAI44QH8DH8BEXAMPLE",
      "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": "7baaa9a14-6a52-47dc-aee0-3cace05147c3",
"eventType": "AwsApiCall",
"recipientAccountId": "123456789012"
}
```

The following example shows a CloudTrail log entry that demonstrates the `UpdateInput` action.

```json
{
"eventVersion": "1.05",
"userIdentity": {
  "type": "AssumedRole",
  "principalId": "AKIAI44QH8DH8BEXAMPLE:IotEvents-EventsLambda",
  "arn": "arn:aws:sts::123456789012:assumed-role/IotEventsLambda-RoleForIotEvents-ABCD123DEF456/IotEvents-EventsLambda",
  "accessKeyId": "AKIAI44QH8DH8BEXAMPLE",
  "sessionContext": {
    "attributes": {
      "mfaAuthenticated": "false",
      "creationDate": "2019-02-07T22:22:30Z"
    },
    "sessionIssuer": {
      "type": "Role",
      "principalId": "AKIAI44QH8DH8BEXAMPLE",
      "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": "UpdateInput",
"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": "7baaa9a14-6a52-47dc-aee0-3cace05147c3",
"eventType": "AwsApiCall",
"recipientAccountId": "123456789012"
}
```
Compliance validation for AWS IoT Events

Third-party auditors assess the security and compliance of AWS services as part of multiple AWS compliance programs, such as SOC, PCI, FedRAMP, and HIPAA.

To learn whether AWS IoT Events or other AWS services are in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. 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. AWS provides the following resources to help with compliance:

- **Security and Compliance Quick Start Guides** – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.

- **Architecting for HIPAA Security and Compliance Whitepaper** – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.

  **Note**
  Not all services are compliant with HIPAA.

- **AWS Compliance Resources** – This collection of workbooks and guides might apply to your industry and location.

- **Evaluating Resources with Rules** in the AWS Config Developer Guide – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.

- **AWS Security Hub** – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.
AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

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 the AWS global network security procedures that are described in the Amazon Web Services: Overview of security processes whitepaper.

You use AWS published API calls to access AWS IoT Events through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.
AWS IoT Events quotas

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 Working with Tag Editor in Working with the AWS Management Console.

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.

Additional information is available in AWS tagging strategies.
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. An IAM 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:*",
            "Resource" : "*",
```
You can also specify multiple tag values for a given tag key by enclosing them in a list, as follows.

```json
"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

Use the information in these sections to troubleshoot and resolve issues with AWS IoT Events.

Topics
- Common AWS IoT Events issues and solutions (p. 193)
- Troubleshooting a detector model by running analyses (p. 197)

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 (p. 193)
- Updates from a deleted detector model (p. 193)
- Action trigger failure (when meeting a condition) (p. 194)
- Action trigger failure (when breaching a threshold) (p. 194)
- Incorrect state usage (p. 194)
- Connection message (p. 194)
- InvalidRequestException message (p. 195)
- Amazon CloudWatch Logs action.setTimer errors (p. 195)
- Amazon CloudWatch payload errors (p. 196)
- Incompatible data types (p. 196)

Detector model creation errors

I get errors when I attempt to create a detector model.

Solution

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 Detector model restrictions and limitations (p. 62).

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.
Solution

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.

Solution

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.

Solution

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.

Solution

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.

Solution

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.

Solution

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.

Solution

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.

Solution

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.

Solution

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.

Solution

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.
Solution

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.

  **Solution**

  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.

  **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 `{}`.

  - **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.

    **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 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.

      **Solution**

      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>`
Solution

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 References (p. 104).

Troubleshooting a detector model by running analyses

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 (p. 198).

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 Analyzing a detector model (Console) (p. 205) or Analyzing a detector model (AWS CLI) (p. 206).

Topics
- Detector model analysis and diagnostic information (p. 198)
- Analyzing a detector model (Console) (p. 205)
- Analyzing a detector model (AWS CLI) (p. 206)

Detector model analysis and diagnostic information

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 (p. 100).
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**

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** (p. 199)
- **supported-actions** (p. 200)
- **service-limits** (p. 200)
- **structure** (p. 201)
- **expression-syntax** (p. 201)
- **data-type** (p. 203)
- **referenced-data** (p. 205)
- **referenced-resource** (p. 205)

**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 (p. 89).
**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 (p. 89).

- **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](#).

### 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 (p. 100).

- **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 \textit{duration-expression} for timer \textit{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 \textit{durationExpression} can be converted to a number. Other data types, such as Boolean, aren't supported.

- **Message:** Expression \textit{condition-expression} is not a valid condition expression.

  **Solution:** You might receive this error message if AWS IoT Events couldn't evaluate your \textit{condition-expression} to a Boolean value. The Boolean value must be either \textsc{true} or \textsc{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 \textsc{false} and doesn't invoke the actions or transition to the \textit{nextState} specified in the event.

- **Message:** Incompatible data types [\textit{inferred-types}] found for \textit{reference} in the following expression: \textit{expression}

  **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 \texttt== and \texttt&& 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 \texttt{timeout("time-name")} function requires a string with double quotes as the argument. If you use a reference for the \textit{timer-name} value, you must reference a string with double quotes.

  \texttt{timeout("time-name")}

  **Note**

  For the \texttt{convert(type, expression)} function, if you use a reference for the \textit{type} value, the evaluated result of your reference must be \textsc{string}, \textsc{decimal}, or \textsc{boolean}.

  For more information, see References (p. 104).

- **Message:** Incompatible data types [\textit{inferred-types}] used with \textit{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 string, 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 (p. 101).
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.

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).

### Analyzing a detector model (Console)

The following steps use the AWS IoT Events console to analyze a detector model.

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.

**Note**
After AWS IoT Events starts analyzing your detector model, you have up to 24 hours to retrieve the analysis results.

### Analyzing a detector model (AWS CLI)

The following steps use the AWS CLI to analyze a detector model.

1. Run the following command to start an analysis.

```bash
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

```json
{
   "detectorModelDefinition": {
      "states": [
      {
         "stateName": "TemperatureCheck",
         "onInput": {
            "events": [
            {
               "eventName": "Temperature Received",
               "condition": "isNull($input.TemperatureInput.sensorData.temperature)==false",
               "actions": [
               {
                  "iotTopicPublish": {
                     "mqttTopic": "IoTEvents/Output"
                  }
               ]
            }
         }
      }
   ]
}
```
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 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 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.

```
aws iotevents get-detector-model-analysis-results --analysis-id "analysis-id"
```

**Note**
Replace `analysis-id` with the analysis ID that you copied.

Example 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 directs you to all the API operations for AWS IoT Events in detail, including sample requests, responses, and errors for the supported web services protocols.

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

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.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region launch</td>
<td>AWS IoT Events is now available in the AWS GovCloud (US-West) Region.</td>
<td>September 22, 2021</td>
</tr>
<tr>
<td>Troubleshoot a detector model</td>
<td>AWS IoT Events can now analyze your detector model and generate analysis results that you can use to troubleshoot your detector model.</td>
<td>February 23, 2021</td>
</tr>
<tr>
<td>Region launch</td>
<td>Launched AWS IoT Events in China (Beijing).</td>
<td>September 30, 2020</td>
</tr>
<tr>
<td>Expression usage</td>
<td>Added examples to show you how to write expressions.</td>
<td>September 22, 2020</td>
</tr>
<tr>
<td>Monitoring with alarms</td>
<td>Alarms help you monitor your data for changes. You can create alarms that send notifications when a threshold is breached.</td>
<td>June 1, 2020</td>
</tr>
</tbody>
</table>

Earlier updates

The following table describes important changes to the *AWS IoT Events Developer Guide* before September 18, 2020.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additions</td>
<td>Added type validation to References (p. 104).</td>
<td>August 3, 2020</td>
</tr>
<tr>
<td>Additions</td>
<td>Added Region information to Working with other AWS services (p. 91).</td>
<td>May 7, 2020</td>
</tr>
<tr>
<td>Additions, updates</td>
<td>Added the Payload Customization feature and new event actions: Amazon DynamoDB and AWS IoT SiteWise.</td>
<td>April 27, 2020</td>
</tr>
<tr>
<td>Edits</td>
<td>Added new descriptions of state machine concepts. General editing of content.</td>
<td>October 31, 2019</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Additions</td>
<td>Added new built-in functions for detector model conditional expressions.</td>
<td>September 10, 2019</td>
</tr>
<tr>
<td>Additions</td>
<td>Added detector model examples.</td>
<td>August 5, 2019</td>
</tr>
<tr>
<td>Additions</td>
<td>Added new event actions: Lambda, Amazon SQS, Kinesis Data Firehose and AWS IoT Events input.</td>
<td>July 19, 2019</td>
</tr>
<tr>
<td>Additions, corrections</td>
<td>Corrected description of <code>timeout()</code> function. Added best practice regarding account inactivity.</td>
<td>June 11, 2019</td>
</tr>
<tr>
<td>Corrections</td>
<td>Updated: console debug options page image; console permissions policy.</td>
<td>June 5, 2019</td>
</tr>
<tr>
<td>Updates</td>
<td>AWS IoT Events service open to general availability.</td>
<td>May 30, 2019</td>
</tr>
<tr>
<td>Additions, updates</td>
<td>Updated security information; Added annotated detector model example.</td>
<td>May 22, 2019</td>
</tr>
<tr>
<td>Corrections</td>
<td>Updated: link to limited preview download; Amazon SNS payload examples; additions to required permissions for CreateDetectorModel.</td>
<td>May 17, 2019</td>
</tr>
<tr>
<td>Additions</td>
<td>Added information on security.</td>
<td>May 9, 2019</td>
</tr>
<tr>
<td>Corrections</td>
<td>Link to limited preview download corrected.</td>
<td>April 19, 2019</td>
</tr>
<tr>
<td>Edits</td>
<td>Editorial improvements.</td>
<td>April 16, 2019</td>
</tr>
<tr>
<td>Limited preview release</td>
<td>Limited preview release of the documentation.</td>
<td>March 28, 2019</td>
</tr>
<tr>
<td>Edits</td>
<td>Editorial improvements.</td>
<td>May 18, 2018</td>
</tr>
</tbody>
</table>