



Streaming Analytics

Last updated: 03/03/2026

This content applies to the latest CD version of Cumulocity.

Specifications contained herein are subject to change and these changes will be reported in subsequent versions.

Copyright © 2026 Cumulocity GmbH.

The name Cumulocity GmbH and all Cumulocity GmbH product names are either trademarks or registered trademarks of Cumulocity GmbH and/or its subsidiaries and/or its affiliates and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

This software may include portions of third-party products. Third-party terms are set out in a 3rd-party-licenses file linked to or included with each installation package.

Table of Contents

Table of Contents	3
INTRODUCTION	8
UNDERSTANDING PROCESSING MODES IN STREAMING ANALYTICS	8
PREREQUISITES	8
BROWSERS	8
PERMISSIONS	8
HOME SCREEN	9
ANALYTICS BUILDER	10
EPL APPS	10
MICROSERVICE RUNTIME AND APPLICATIONS	10
ANALYTICS BUILDER	12
GETTING STARTED WITH ANALYTICS BUILDER	12
WHAT IS ANALYTICS BUILDER	12
FIRST STEPS: CREATING YOUR FIRST MODEL	12
FIRST STEPS: CREATING A MODEL FROM A SAMPLE	15
UNDERSTANDING MODELS	17
MODELS	18
TEMPLATE MODEL INSTANCES	18
BLOCKS	18
WIRES	22
SAMPLE USE CASE	22
USING THE MODEL MANAGER	23
THE MODEL MANAGER USER INTERFACE	23
FILTERING THE MODELS AND SAMPLES	25
ADDING A NEW MODEL	25
EDITING AN EXISTING MODEL	26
EDITING THE INSTANCES OF A MODEL	26
DEPLOYING A MODEL	27
UNDEPLOYING A MODEL	28
DUPLICATING A MODEL	28
DOWNLOADING A MODEL	28
COPYING A MODEL	29
UPLOADING A MODEL	29
PASTING A MODEL	29
DELETING A MODEL	30
RELOADING ALL MODELS	30
VIEWING A SAMPLE	30
CREATING A MODEL FROM A SAMPLE	30
USING THE MODEL EDITOR	30
THE MODEL EDITOR USER INTERFACE	31
WORKING WITH MODELS	31
WORKING WITH BLOCKS AND WIRES	33
WORKING WITH GROUPS	42
MANAGING THE CANVAS	45
USING THE INSTANCE EDITOR	46
THE INSTANCE EDITOR USER INTERFACE	46
ADDING AN INSTANCE	47
EDITING AN INSTANCE	47
DEPLOYING AN INSTANCE	48
UNDEPLOYING AN INSTANCE	48
FILTERING AND SORTING THE INSTANCES	48
DUPLICATING AN INSTANCE	49
DELETING AN INSTANCE	49
SAVING THE INSTANCES	49
RELOADING THE INSTANCES	50
LEAVING THE INSTANCE EDITOR	50

WIRES AND BLOCKS	50
VALUES SENT ON A WIRE	50
TYPE CONVERSIONS	55
PROCESSING ORDER OF WIRES	56
WIRE RESTRICTIONS	56
BLOCK INPUTS AND OUTPUTS	57
COMMON BLOCK INPUTS AND PARAMETERS	57
INPUT BLOCKS AND EVENT TIMING	58
OUTPUT BLOCKS AND EVENT TIMING	59
FRAGMENT PROPERTIES ON WIRES	59
KEYS FOR IDENTIFYING A SERIES OF EVENTS	59
DETAILS OF VALUES AND BLOCKS	60
INTRODUCTION	60
VALUES AS REPRESENTATIONS OF CONTINUOUS-TIME PHYSICAL QUANTITIES	60
WINDOW BLOCK OUTPUT TIMINGS	66
PULSE SIGNALS	68
DISCRETE-TIME MEASUREMENTS	69
MODELS AND DEVICES	70
MODEL EXECUTION FOR DIFFERENT DEVICES	70
BROADCAST DEVICES	72
VIRTUAL DEVICES	73
CONNECTIONS BETWEEN MODELS	74
CONFIGURING THE NUMBER OF SHOWN INPUT SOURCES AND OUTPUT DESTINATIONS	75
SEARCHING FOR DEVICES, GROUPS AND/OR ASSETS	75
SUPPORT FOR DYNAMIC CHANGES TO GROUP AND ASSET HIERARCHY	76
MODEL SIMULATION	77
ABOUT SIMULATION MODE	77
SIMULATION PARAMETERS	77
CONFIGURING THE MAXIMUM NUMBER OF SIMULATION MODELS	77
MONITORING DROPPED INPUTS	78
MONITORING AND CONFIGURATION	78
MONITORING	78
CONFIGURATION	82
ANALYTICS BUILDER BLOCK REFERENCE	86
OVERVIEW OF ALL BLOCKS	86
INPUT	88
ALARM INPUT	88
EVENT INPUT	89
MANAGED OBJECT INPUT	90
MEASUREMENT INPUT	91
OPERATION INPUT	92
POSITION INPUT	93
OUTPUT	94
ALARM OUTPUT	95
EVENT OUTPUT	96
MANAGED OBJECT OUTPUT	97
MEASUREMENT OUTPUT	98
OPERATION OUTPUT	99
SEND EMAIL	100
SEND SMS	100
LOGIC	101
AND	101
NOT	102
OR	102
CALCULATION	103
CROSSING COUNTER	103
DELTA	104
DIFFERENCE	105
DIRECTION DETECTION	105

EXPRESSION	106
FROM BASE N	107
KPI	108
LIMIT	109
RANGE	110
RANGE LOOKUP	111
ROUNDING	111
THRESHOLD	112
TO BASE N	113
AGGREGATE	114
AVERAGE (MEAN)	114
COUNTER	115
DISCRETE STATISTICS	116
GRADIENT	117
GROUP STATISTICS	118
INTEGRAL	119
MINIMUM / MAXIMUM	120
STANDARD DEVIATION	121
FLOW MANIPULATION	122
COMBINER	122
GATE	123
LATCH VALUES	125
PULSE	125
SELECTOR	126
SWITCH	127
TIME DELAY	128
UTILITY	129
CONSTANT VALUE	129
CRON TIMER	130
DURATION	131
EXTRACT PROPERTY	132
GEOFENCE	133
MISSING DATA	134
SET PROPERTIES	134
TEXT SUBSTITUTION	136
TOGGLE	137
SMART RULES (NEW) PLUGIN	138
GETTING STARTED WITH THE SMART RULES (NEW) PLUGIN	138
WHAT IS THE SMART RULES (NEW) PLUGIN	138
PREREQUISITES	138
CREATING YOUR FIRST SMART RULE	138
CREATING SMART RULES FROM EXISTING ANALYTICS BUILDER SAMPLES	141
TROUBLESHOOTING	142
UNDERSTANDING THE SMART RULES (NEW) PLUGIN	142
ANALYTICS BUILDER WORKFLOW VERSUS SMART RULES PLUGIN	142
UNDERSTANDING THE SMART RULES INTERFACE	143
INTEGRATION WITH ANALYTICS BUILDER INSTANCE EDITOR	143
EPL APPS	145
USING THE APAMA EVENT PROCESSING LANGUAGE (EPL)	145
HOW CAN I CREATE DERIVED DATA FROM EPL?	146
HOW CAN I CONTROL DEVICES FROM EPL?	146
HOW CAN I QUERY DATA FROM EPL?	147
BASIC FUNCTIONALITY	148
DEVELOPING APPS	148
DEPLOYING APPS	151
TESTING APPS	153
CONNECTIVITY BUNDLES FOR COMMUNICATING WITH CUMULOCITY	154
SUPPORTED REST SERVICES	154

EVENTS AND CHANNELS	160
EXAMPLE	162
BUILT-IN ACTIONS	162
OVERVIEW	162
QUERYING CUMULOCITY DATA	162
INVOKING OTHER PARTS OF THE CUMULOCITY REST API	163
INVOKING HTTP SERVICES	164
UTILITY FUNCTIONS	164
ADVANCED FEATURES	166
CUSTOM FRAGMENTS	166
MEASUREMENT FRAGMENTS	167
LISTENERS	167
CREATING OWN EVENT TYPES	169
CREATING OWN ACTIONS	169
VARIABLES	170
SPAWNING MONITOR INSTANCES AND CONTEXTS	171
BEST PRACTICES AND GUIDELINES	171
EPL MONITORS	171
NUMBER FORMATS	171
SUBSCRIBING TO CHANNELS AND CONTEXTS	172
APAMA LIMITATIONS IN CUMULOCITY	172
EXAMPLES	173
CALCULATING AN HOURLY AVERAGE OF MEASUREMENTS	173
CREATING ALARMS FROM BIT MEASUREMENTS	174
CONSUMPTION MEASUREMENTS	175
MISCELLANEOUS SAMPLE APPS	176
STUDY - CIRCULAR GEOFENCE ALARMS	176
OVERVIEW	176
PREREQUISITES	176
STEP 1: FILTERING THE INPUT	177
STEP 2: COLLECTING NECESSARY DATA	178
STEP 3: CHECKING IF THE DEVICE SUPPORTS C8Y_GEOFENCE	178
STEP 4: CREATING THE TRIGGER	178
STEP 5: CREATING THE ALARM	179
STEP 6: CLEARING THE ALARM	179
PUTTING EVERYTHING TOGETHER	179
CONNECTING APAMA TO OTHER MICROSERVICES	181
OVERVIEW	181
CREATING AN EPL APP	181
CONNECTING TO THE CUMULOCITY PLATFORM	181
MAKING MICROSERVICE REQUESTS	182
EXAMPLE REQUEST TO A MICROSERVICE ENDPOINT	182
OTHER MICROSERVICES	183
USING CUMULOCITY MQTT SERVICE	183
CUSTOMIZATION AND PERMISSIONS	185
OPTIMIZING REQUESTS WITH CONCURRENT CONNECTIONS	185
CONTROLLING ACCESS TO THE STREAMING ANALYTICS APPLICATION	185
CUSTOMIZING THE HOME SCREEN OF THE STREAMING ANALYTICS APPLICATION	186
CONFIGURATION REQUIREMENTS FOR NOTIFICATIONS 2.0	187
MODIFYING MICROSERVICE PERMISSIONS AND RESOURCE USAGE	188
TROUBLESHOOTING AND DIAGNOSTICS	189
DOWNLOADING DIAGNOSTICS AND LOGS	189
LOG FILES OF THE APAMA-CTRL MICROSERVICE	190
AUDIT LOGS FOR STREAMING ANALYTICS	190
DIAGNOSTICS REST ENDPOINTS	190
MONITORING REST ENDPOINTS	191
ALARMS GENERATED BY THE APAMA-CTRL MICROSERVICE	191
ALARM SEVERITIES	192

INTRODUCTION

Using Cumulocity Streaming Analytics, you can add your own logic to your IoT solution for immediate processing of incoming data from devices or other data sources. This logic can, for example, alert applications of new incoming data, create new data based on the received data (such as sending an alarm when a threshold for a sensor is exceeded), or trigger operations on devices.

Typical real-time analytics use cases include:

- Remote control: Turn a device off if its temperature rises over 40 degrees.
- Validation: Discard negative meter readings or meter readings that are lower than the previous.
- Derived data: Calculate the volume of sales transactions per vending machine per day.
- Aggregation: Sum up the sales of vending machines for a customer per day.
- Notifications: Send me an email if there is a power outage in one of my machines.
- Compression: Store location updates of all cars only once every five minutes (but still send real-time data for the car that I am looking at to the user interface).

Cumulocity Streaming Analytics includes a built-in event processing engine that powers both [Analytics Builder](#) and custom [EPL \(Event Processing Language\) apps](#). This engine, called the Apama correlator, runs automatically within the Cumulocity platform and does not require manual setup or management.

Streaming Analytics is available in both Cumulocity Core (cloud) and Cumulocity Edge (local installation), providing flexibility for different deployment needs.

UNDERSTANDING PROCESSING MODES IN STREAMING ANALYTICS

When working with Streaming Analytics in Cumulocity, the availability and persistence of data depend on the processing mode chosen at ingestion. Refer to [HTTP usage > Processing mode](#) in the Cumulocity OpenAPI Specification and [OPC UA > Sample payloads](#) for more information.

This directly impacts what data you can access, analyze, and act upon in realtime.

PERSISTENT (default) and QUIESCENT modes:

Data is stored in the Cumulocity database and sent to the Streaming Analytics engine. This is the normal expectation where all data is persisted in the platform before Streaming Analytics processes it.

TRANSIENT and CEP modes:

Data is sent to the Streaming Analytics engine, but it is not stored in the database. This allows processing of data before it's stored in the platform to reduce storage with aggregates or to do data conversion or cleaning tasks before it is stored in the database.

PERSISTENT and TRANSIENT modes also generate notifications to non-Streaming-Analytics subscribers to notifications, whereas CEP and QUIESCENT modes will only be sent to Streaming Analytics. The latter two modes are only applicable to measurements and events.

PREREQUISITES

BROWSERS

The Streaming Analytics application supports the same browsers as Cumulocity, with the following exception: browsers on smartphones and tablets are not supported.

PERMISSIONS

To use Analytics Builder, you must at least have the following permissions:


Permission type	Permission level
CEP management	ADMIN
Option management	READ
Inventory	READ

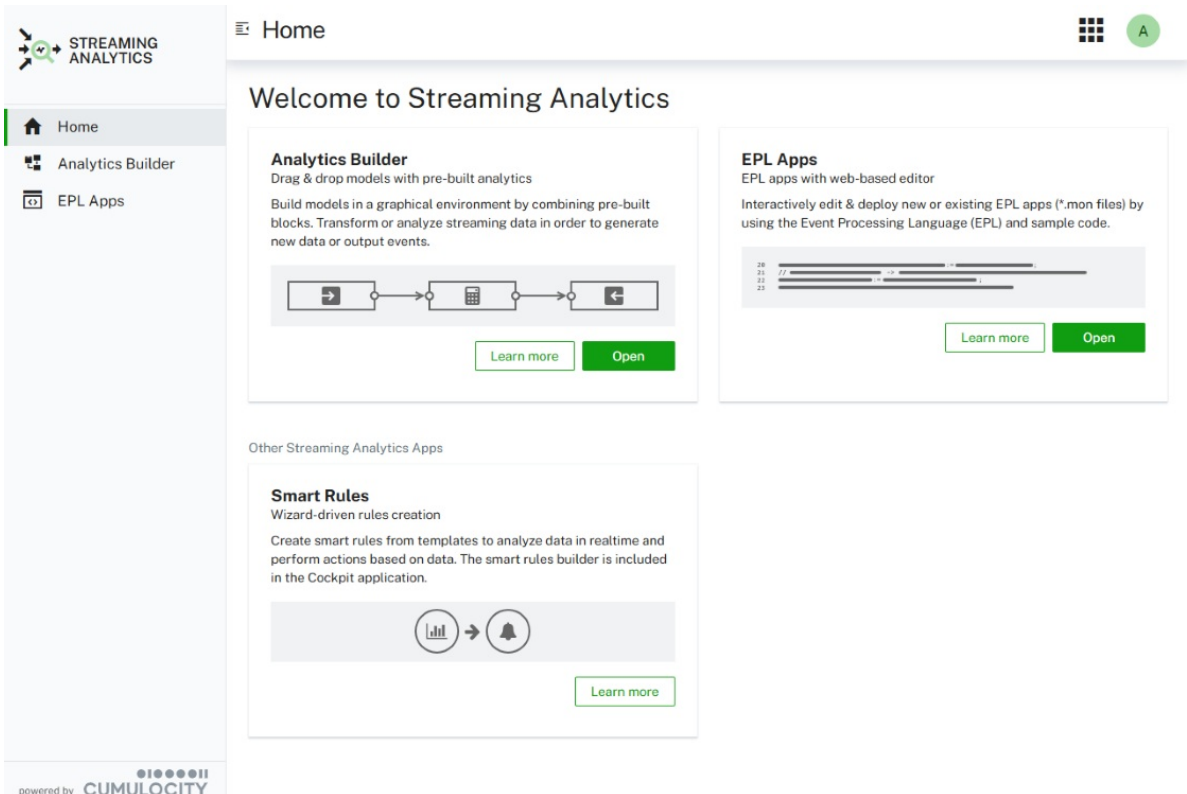
This is typically achieved by using a global role which has those permissions, and where the role has access to the Analytics Builder application. See [Managing permissions and roles](#) for details.

To use EPL Apps, you only need the following permission:

Permission type	Permission level
CEP management	ADMIN

HOME SCREEN

To access the home screen of the Streaming Analytics application, click the Streaming Analytics icon  in the [application switcher](#).



The screenshot displays the Streaming Analytics application's home screen. On the left, a vertical sidebar contains the application logo and three navigation options: 'Home' (selected), 'Analytics Builder', and 'EPL Apps'. The main area is titled 'Home' and 'Welcome to Streaming Analytics'. It features three prominent cards: 'Analytics Builder' (with a diagram of a data flow), 'EPL Apps' (with a code snippet), and 'Smart Rules' (with a diagram of a rule flow). Each card includes a brief description, a 'Learn more' button, and an 'Open' button. At the bottom, the Cumulocity logo and 'powered by CUMULOCITY' are visible.

The home screen of the Streaming Analytics application lets you navigate to the different pages of the application. These are [Analytics Builder](#) and [EPL Apps](#). You can access these pages using the corresponding **Open** button on the home screen or by using the navigator on the left.

If you need more space for a page, you can hide the navigator. Click the small arrow at the very left of the top bar to toggle the display of the navigator.

The home screen also provides information on smart rules. You create and maintain them with the Cockpit application. See [Smart rules](#) for detailed information.

The home screen and navigator only show entries for the items that you are allowed to use, depending on the version of the Apama-ctrl microservice to which your tenant is subscribed. See also [Microservice runtime and applications](#) and [Customizing the home screen of the Streaming Analytics application](#).

ANALYTICS BUILDER

The **Analytics Builder** page of the Streaming Analytics application allows you to build analytic models that transform or analyze streaming data in order to generate new data or output events. The models are capable of processing data in real time.

The models interact with the devices and sensor measurements. Models can receive **Measurement** and **Event** objects from devices, which provide the inputs to calculations or pattern detection performed within a model. Models can create new **Measurement** objects which can represent derived values from sensors (for example, an average temperature) or the measurements can be used as an input to other analytic models (see [Connections between models](#)). Models can create new **Operation** objects which are sent to devices to control the devices (for example, to sound an alarm bell, display a message on a screen, or switch a device off). The models are also stored in the Cumulocity inventory, but can be uploaded or downloaded via the model manager.

You build the models in a graphical environment by combining pre-built *blocks* into *models*. The blocks in a model package up small bits of logic, and have a number of inputs, outputs and parameters. Each block implements a specific piece of functionality, such as receiving data from a sensor, performing a calculation, detecting a condition, or generating an output signal. You define the configuration of the blocks and connect the blocks using *wires*. You can edit the models, simulate deployment with historic data, or run them against live systems. See [Analytics Builder](#) for detailed information.

It is also possible to build custom blocks if none of the blocks delivered with Analytics Builder implement the logic required; see [Creating your own blocks](#).

You can customize several aspects of Analytics Builder by setting various tenant options. See [Configuration](#) for detailed information.

EPL APPS

The **EPL Apps** page of the Streaming Analytics application allows you to write business logic in Apama's Event Processing Language (Apama EPL) which gives more power and flexibility in a text-based programming language. This is an alternative if more complex logic is required or the logic does not fit into the pattern of an analytic model.

You can develop EPL apps (that is, single *.mon files) directly within Cumulocity, written in Apama EPL. You can also import existing *.mon files as EPL apps into Cumulocity. When you activate an EPL app from the Streaming Analytics application, you deploy it to Cumulocity. See [EPL Apps](#) for detailed information, including examples.

A quick way to get started is to explore the code of the EPL samples that can be accessed from the EPL editor. See [Developing apps with the Streaming Analytics application](#) for information on how to create an EPL app and access the samples. For a start, use one of the simpler samples with temperature measurements, such as "Create an alarm if a measurement exceeds a threshold value". You can immediately see results using this sample. Add your own EPL code to the sample and try out your changes.

You use the Apama API for interacting with Cumulocity. For detailed information, see the `com.apama.cumulocity` package in the [API Reference for EPL \(ApamaDoc\)](#), which is part of the Apama documentation.

MICROSERVICE RUNTIME AND APPLICATIONS

Analytic models, EPL apps and smart rules are executed in an Apama-ctrl microservice. All microservices come with pre-configured resource limits (for example, CPU and memory). The resource usage of analytic models, EPL apps and smart rules depends on various factors such as application complexity, number of devices, etc. It is recommended that customers profile their applications to avoid out-of-

memory issues.

To do this	you need the following
Build analytic models	Apama-ctrl microservice variant (for example, Apama-ctrl-starter) and Streaming Analytics application
Develop EPL apps	Apama-ctrl microservice variant (for example, Apama-ctrl-250mc-1g) and Streaming Analytics application
Use smart rules	Any Apama-ctrl microservice variant and Smartrule microservice (included in Cumulocity's Standard tenant)

See also the following sections:

- [Subscribed applications](#)
- [Subscribed microservices](#)
- [Smart rules](#)

If your tenant is subscribed to the Apama-ctrl-starter microservice, then the following applies:

- Limited number of at most 3 active analytic models. Custom blocks written with the Analytics Builder Block SDK cannot be used.
- The **EPL Apps** page is not available in the Streaming Analytics application.
- Smart rules are supported.

If your tenant is subscribed to the Apama-ctrl-250mc-1g microservice (or one of its larger variants), then the following applies:

- Analytic models, EPL apps and smart rules are supported.
- Custom blocks written with the Analytics Builder Block SDK can be used.

If your tenant is subscribed to the Apama-ctrl-mt microservice, then the following applies:

- Multi-tenant support.
- EPL apps are only enabled on the tenant that owns the microservice, but disabled on the subtenants.
- Analytic models and smart rules are supported.

If your tenant is subscribed to the Apama-ctrl-smartrules or Apama-ctrl-smartrulesmt microservice, then the following applies:

- Smart rules are supported.
- The **Analytics Builder** and **EPL Apps** pages are not available in the Streaming Analytics application.

Contact [product support](#) to discuss adding more capabilities.

ANALYTICS BUILDER

GETTING STARTED WITH ANALYTICS BUILDER

WHAT IS ANALYTICS BUILDER

Analytics Builder allows you to build analytic models that transform or analyze streaming data in order to generate new data or output events. The models are capable of processing data in real time.

You build the models in a graphical environment by combining pre-built blocks into models. The blocks in a model package up small bits of logic, and have a number of inputs, outputs and parameters. Each block implements a specific piece of functionality, such as receiving data from a sensor, performing a calculation, detecting a condition, or generating an output signal. You define the configuration of the blocks and connect the blocks using wires. You can edit the models, simulate deployment with historic data, or run them against live systems. See [Understanding models](#) for more detailed information.

Analytics Builder consists of the following tools:

- **Model manager.** When you invoke Analytics Builder, the model manager is shown first. It lists all available models and lets you manage them. For example, you can test and deploy the models from the model manager, or you can duplicate or delete them. You can create new models or edit existing models; in this case, the model editor is invoked. Samples are also available which help you get started with creating your own models. See [Using the model manager](#) for detailed information.
- **Model editor.** The model editor lets you define the blocks that are used within a model and how they are wired together. User-visible documentation (the so-called *block reference*) is available in the model editor, describing the functionality of each block. See [Using the model editor](#) for detailed information.
- **Instance editor.** If template parameters have been defined in a model, the instance editor lets you set up different instances of the same model which can then be activated and managed separately. The instance editor uses the template parameters that have been defined in the model editor. See [Using the instance editor](#) for detailed information.

The blocks are implemented in the Event Processing Language (EPL) of Apama. At runtime, the EPL code runs in an Apama correlator to execute the models. Some runtime behavior and restrictions are important to understand. These are documented in later topics.

FIRST STEPS: CREATING YOUR FIRST MODEL

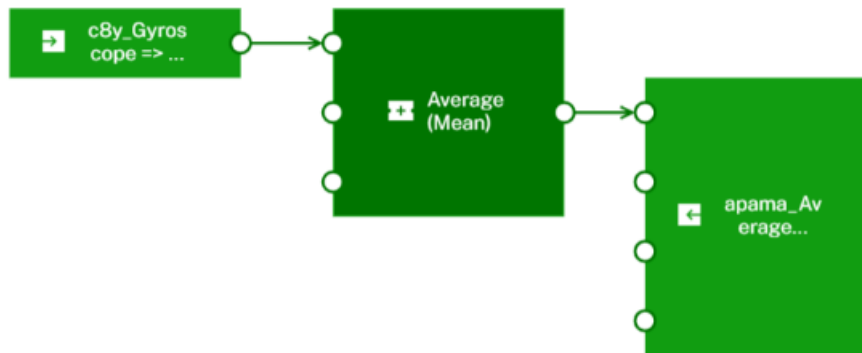
This topic gives a brief overview of how to add and design a new model, and how to view its output. It is not intended to be a comprehensive description of the full range of possibilities provided by Analytics Builder. Therefore, explanations are kept to a minimum. For more detailed information, see the remainder of this documentation.

The steps below require that at least one device has already been registered in Cumulocity. Preferably, this is a device which is already sending measurement values to Cumulocity. These first steps assume that you are using a smartphone on which the Cumulocity Sensor App has been installed, see [Cumulocity Sensor App](#) for details.

The model that you add will contain three blocks:

- An input block which receives measurement values from a device.
- A block that calculates the mean of the measurement values over time.
- An output block that sends the calculated mean values to Cumulocity's Device Management application so that they can be viewed there.

When you have completed all steps below, your model will look similar to the following:



Step 1: Switch to Analytics Builder

On the home screen of the Streaming Analytics application, click the **Open** button that is shown below the Analytics Builder heading.

Alternatively, click **Analytics Builder** in the navigator on the left.

INFO

If the navigator is currently hidden, click the small arrow at the very left of the top bar to toggle the display of the navigator.

Step 2: Add a new model

The first page that is shown when you invoke Analytics Builder is the model manager.

1. On the **Models** tab, click **New model** in the toolbar.
2. In the resulting dialog box, enter a model name and click **OK**.

Step 3: Add the input block

You design your model in the model editor. The model editor is shown after you have entered the model name. The palette which is shown on the left contains all blocks that can be added to a model. You add a block by dragging it from the palette onto the canvas.

1. In the palette, expand **Input**.
2. Drag the **Measurement Input** block onto the canvas. The block parameter editor is automatically shown.

INFO

If the block parameter editor is not shown (for example, because you clicked an empty space on the canvas after dragging the input block onto the canvas), click the block using the left mouse button to show the block parameter editor.


3. Click the menu **☰** that is shown for **Input Source**. In the resulting dialog box, click the **Select device** button for the device that you want to use. This button is shown when you hover over a device.

INFO

By default, an input block is listening to all input sources, that is, the **All Inputs** option is set. However, these first steps assume that you are using your smartphone. Therefore, you must select a single

device as described above.

4. From the **Fragment and Series** drop-down list box, select the fragment and series for which the input block is to listen. If the device has previously sent data, the drop-down list box offers one or more values for selection. An example for the Cumulocity Sensor App would be **c8y_Gyroscope => gyroscopeY**.
5. Select the **Ignore Timestamp** checkbox. This makes sure that the measurements are processed in the same order as they are received.

If you need detailed information on the currently selected block, view the block reference in the documentation pane on the right. If the documentation pane is currently not shown, click the document icon .

Step 4: Add the block that calculates the mean of the measurement values

1. In the palette, expand **Aggregate**.
2. Drag the **Average (Mean)** block onto the canvas.
3. In the block parameter editor, specify a value for **Window Duration (secs)**, for example "10". The specified number of seconds will be used to control what duration the measurement is averaged over. Smaller values will react quicker to changes in values, larger values will give more smoothing of the value.

Step 5: Add the output block

1. In the palette, expand **Output**.
2. Drag the **Measurement Output** block onto the canvas.
3. As the output destination, select the same device as for your input block.

INFO

If you have kept the default option of **All Inputs** for the input block, you must set the output destination to **Trigger Device**. However, these first steps assume that you are using a single device, so you must select the same device as for your input block.



4. Specify "apama_Average" as the fragment name.
5. Specify "value" as the series name.

Step 6: Connect the blocks

To pass the values from one block to another, you must connect the blocks with wires. You attach the wires to the ports, that is, to the small circles that are shown to the left and/or right of a block.

1. Click the **Value** output port of the input block and drag the mouse to the **Value** input port of the **Average (Mean)** block.
2. Click the **Average** output port of the **Average (Mean)** block and drag the mouse to the **Value** input port of the output block.

Step 7: Save the model and go back to the model manager

1. In the toolbar of the model editor, click the save icon  to save your newly created model.
2. In the toolbar of the model editor, click the close icon  to leave the model editor and thus to return to the model manager.

INFO

Only saved models are listed on the **Models** tab of the model manager. When you add a new model and then leave the model editor without saving the model, it will not be listed in the model manager, and all the edits you made will be lost.

Step 8: Activate the model in production mode

A card for the newly added model is shown on the **Models** tab of the model manager. A new model is automatically set to draft mode and inactive state. You will now activate your new model in production mode. This deploys the model so that the measurements from your device are processed.

1. Click the drop-down menu on the card which currently shows **Draft** and select **Production**.
2. Click the toggle button on the card which currently shows **Inactive**. This changes the state to **Active**.

Step 9: Go to the Device Management application and view the measurements

To view the measurements that are sent from your active model, you must switch to the Device Management application. See [Device Management application](#) for detailed information.

1. Go to the Device Management application.
2. In the navigator on the left, click **Devices** and then **All devices**.
3. Locate your device and click its name to display the device details.
4. Click **Measurements** on the left. This is a dynamic tab which is only shown when measurements are available for the device. The resulting page shows several charts, visualizing the data sent from your device. It should now also show a chart titled "Apama_average" in which you can view the values that are sent from your newly created model. You may have to reload the page to see this new chart. See [Measurements](#) for more information on the **Measurements** tab.

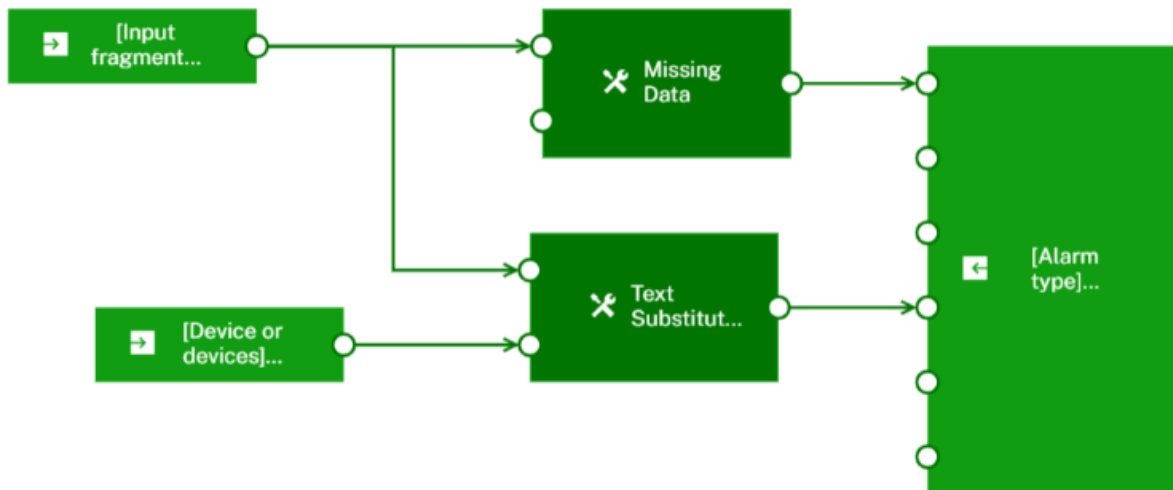
FIRST STEPS: CREATING A MODEL FROM A SAMPLE

This topic gives a brief overview of how to create a model from a sample. It is based on the **On missing measurements create alarm** sample. Your new model will create an alarm if no new measurement data has been received for a specified time period.

This topic is not intended to be a comprehensive description of the full range of possibilities provided by Analytics Builder. Therefore, explanations are kept to a minimum. For more detailed information, see the remainder of this documentation.

The steps below require that at least one device has already been registered in Cumulocity. Preferably, this is a device which is already sending measurement values to Cumulocity. These first steps assume that you are using a smartphone on which the Cumulocity Sensor App has been installed, see [Cumulocity Sensor App](#) for details.

The following image shows the blocks that are defined in the **On missing measurements create alarm** sample.



The sample uses predefined template parameters. After you have created a model from the sample, you can create multiple instances of the model and you can specify different values for the template parameters. See also [Models](#) which explains the difference between models without template parameters and models with template parameters.




The following is a brief description of the blocks that are defined within the sample:

- The input starts with the **Measurement Input** block waiting for new incoming measurements that match a given value that is defined with the **Input fragment and series** template parameter. The name of that parameter is the label that you can see on the input block.

- The output from the **Measurement Input** block is then passed to the **Missing Data** block which triggers an output if no input is received within the time defined with the **Duration (seconds)** template parameter.
- The output from the **Missing Data** block is used as the trigger for the **Create Alarm** input port of the **Alarm Output** block. The name of the **Alarm type** template parameter is the label that you can see on the output block.
- The output from the **Measurement Input** block is also passed as input to the **Object** input port of the **Text Substitution** block, along with the input from the **Managed Object Input** block which is passed to the **Source** input port of the **Text Substitution** block. The name of the **Device or group of devices** template parameter is the label that you can see on the input block.
- The **Text Substitution** block supports replacement of placeholders. For example, if the input text is “Missing measurements of type: #{type}”, then the **#{type}** value is replaced by the actual **type** of the measurement. See [Text Substitution](#) for more details.
- The **Text Substitution** block is configured to use the **Alarm text** template parameter as user input. It applies the required substitutions and then sends the string containing the substitutions from its **Output** output port to the **Message** input port of the **Alarm Output** block.
- The **Alarm Output** block requires the **Alarm type** and **Alarm severity** template parameters to be configured and creates an alarm whenever it is triggered by the **Missing Data** block.

Step 1: Create a new model from a sample

The **Samples** tab of the model manager lists all sample models that are provided with Analytics Builder. You can view a sample by simply clicking on its card, but you cannot edit or deploy it. However, you can use the samples as a basis for developing your own models, by creating a model from a sample.

1. Go to the **Samples** tab of the model manager.
2. Click the actions menu  of the **On missing measurements create alarm** sample and then click **Create model from sample**. The new model is immediately shown in the model editor. It has the same name, description and tags as the sample.
3. If you want to rename the model, click **Model settings** which is shown at the left of the toolbar. You can then specify a new name in the resulting dialog box.
4. In the toolbar of the model editor, click the save icon  to save the new model.
5. In the toolbar of the model editor, click the close icon  to leave the model editor and thus to return to the model manager.

INFO

Keep in mind that only saved models are listed on the **Models** tab of the model manager.







Step 2: Create a new instance of the model

The sample model uses template parameters. So when you turn the sample into a model, you create a so-called template model. You cannot activate a template model directly in the model manager. Instead, you must create at least one instance of the model, and you can then activate that instance using the instance editor.

1. On the **Models** tab of the model manager, locate the card for your newly created model.
2. To invoke the instance editor, click **0 Instances** which is currently shown on the card.
3. Click **New Instance** to create the first instance of your new model.

Step 3: Fill in the template parameter values

In the instance editor, a row is shown for each instance that you create. A column is provided for each template parameter that is defined in the template model, with the name of the template parameter being the column header. As long as an instance is not active, you can adjust the values for that instance.

	DEVICE OR GROUP OF DEVICES 	INPUT FRAGMENT AND SERIES 	DURATION (S)	RUN MODE 	STATUS 		
1	<input type="text" value="..."/>	<input type="text" value=""/>	3600	<input type="text" value="Draft"/>	<input type="button" value="Inactive"/>		

Use the horizontal scroll bar below the instance table if not all template parameters (columns) are shown on the screen.

1. Click the field below the **Device or group of devices** column header. In the resulting dialog, click **Select device** for the device that you want to use for this instance.
2. In the text box below the **Input fragment and series** column header, specify the details of the measurement input that you want to monitor in the following format:

```
<valueFragmentName>.<valueSeriesName>
```

For example, if the measurement fragment is `c8y_Gyroscope` and the series is `gyroscopeX`, then you must enter the following:

```
c8y_Gyroscope.gyroscopeX
```

TIP

If you want to find out which fragments and series are available to your device, without changing the predefined template parameters of the **Measurement Input** block, go back to the model editor, drag the input block for your device from the palette onto the canvas and open the **Fragment and Series** drop-down list box. This lists all the values that you can use. However, instead of the `=>` that you can see in the drop-down list box, you must use a dot (.) in this case. Don't forget to remove this block again after you have decided which value to use.

3. The fields below the **Duration (seconds)**, **Alarm type**, **Alarm text** and **Alarm severity** column headers already contain default values (see also the above description of the blocks). Adapt them to your requirements. For example, change the duration to 30 seconds, rename the alarm type to "MyAlarmType", keep the predefined alarm text, and set the alarm severity to **Minor**.
4. In the toolbar of the instance editor, click **Save**.

Step 4: Activate the instance

You will now activate the instance in production mode. This deploys the instance so that the measurements from your device are processed.

1. In the **Run Mode** column of the instance editor, click the drop-down menu for the instance and select **Production**.
2. In the **Status** column of the instance editor, click the button which currently shows **Inactive** to change the status to **Active**.

Step 5: Send in the data from your device

Once the instance has been activated, send in the data from your device. The instance starts monitoring the device once measurement data starts arriving and creates an alarm if no data is received within the configured duration.

For our example case with the gyroscope measurements from a smartphone, it should be sufficient that you simply turn off the smartphone display while the Cumulocity Sensor App is still running.

Step 6: Go to the Device Management application and view the alarms

To view the alarms that are sent from your active instance, you must switch to the Device Management application. See [Device Management application](#) for detailed information.

1. Go to the Device Management application.
2. In the navigator on the left, click **Devices** and then **All devices**.
3. Locate your device and click its name to display the device details.
4. Click **Alarms** on the left.
5. On the resulting page, check the alarms that are sent from your device. If you have edited your instance as described above, you should see a MINOR alarm after 30 seconds, saying "Missing measurements of type: c8y_Gyroscope". See [Working with alarms](#) for more information on the **Alarms** tab.

UNDERSTANDING MODELS

MODELS

A model is a container which can have a network of blocks connected to each other with wires.

The behavior of a block inside a model does not depend on other blocks. There can be multiple instances of the same block in a model where each instance may behave differently, depending on the configurable parameters or the inputs connected to the block.

You can create two different types of models: models without template parameters and models with template parameters.

Models without template parameters

All blocks in the model use defined input devices or ranges of devices and contain defined parameter values. Such a model can be activated immediately in the model manager.

Models with template parameters

A model in which one or more template parameters are defined is called a "template model". Template parameters can be bound to any number of block parameters, provided that the type of the block parameter is the same as that of the template parameter.

For example, you can define a template parameter for the device name and another for the threshold value. These template parameters can later be set individually in the different instances of the model. For example, one template parameter can specify a device which can then be used for several input and output blocks. Or one instance can use device ABC with a threshold value of 100, and another instance can use device XYZ with a threshold of 200. Models with template parameters are not activated directly in the model manager. You must create at least one instance of the model, and you can then activate each instance separately using the instance editor.

The scope of the template parameters is local to the model in which they are defined. In other words, template parameters defined in one model cannot be used in any other model that is deployed in same tenant or subtenant. The names of the template parameters must be unique within the scope of the model in which they are defined.

There are two relevant roles for this type of model, this can be the same person or different persons:

- **Model author.** The model author creates the model and defines all of its blocks, parameters and wires. Most importantly, the model author creates the template parameters and binds them to the appropriate parameters in selected blocks.
- **Instance maintainer.** The instance maintainer creates the instances of the model and assigns values to the template parameters that are to be used by each instance.

The model author has the following options to define a template parameter:

- It can have default value which is provided as the default value in the instance editor. The instance maintainer may then leave it at the default value or change it to another value.
- It can be optional. The instance maintainer then has the possibility to either provide a value or leave it blank.
- It can be required. The instance maintainer must then provide a value. A required value is one that is not optional and has no default value.

TEMPLATE MODEL INSTANCES

Template model instances hold the values to be used in models with template parameters.

For example, two devices may have similar checks on data from the devices, but use different threshold values for those checks. In this case, you would configure an instance for each of the devices, specifying which device and what threshold to use.

Each instance can be activated, deactivated, or use different run modes, independently.

BLOCKS

Blocks are the basic processing units of the model. Each block has some predefined functionality and processes data accordingly. A block can have a set of parameters and a set of input ports and output ports.

The palette of the model editor offers for selection the following types of blocks:

- **Input blocks,** which receive data from external sources. An input block normally represents a device that has been registered in the Cumulocity inventory, a group, an asset, or all input sources. See also [Input blocks](#).

- **Output blocks**, which send data to external sources. An output block normally represents a device that has been registered in the Cumulocity inventory. But there are also blocks for sending an email or SMS to specified receivers. See also [Output blocks](#).
- **Processing blocks**, which receive data from the input blocks and send the resulting data to the output blocks. See also [Processing blocks](#).

INFO

For detailed information on each block, see [Overview of all blocks](#) which provides links to the descriptions of all the blocks in the block reference.

A block can receive data from another block through its input ports. A block can send data to another block through its output ports. Different blocks will have different numbers of input or output ports, and some blocks have only input ports or only output ports. For most blocks, it is not required to connect all of the input or output ports.

A block can have configurable parameters that define the behavior of the block. These parameters are either optional or mandatory, depending on the requirement of the block. A parameter can be configured with a value or a template parameter.

When using the same block multiple times, you can specify different values for the same parameter. For example, the **Threshold** block has a configurable parameter named **Threshold Value**. If you are using two instances of the **Threshold** block and configure this parameter differently for each block, the blocks will report different breaches of the threshold.

INFO

Two output ports cannot be connected to the same input port, whereas one output port can be connected to multiple input ports.

Input blocks

An input block is a special type of block that receives data from an external source. It converts the data into a format understandable to wires and transfers the data to the connected blocks. For example, when an input block receives a **Measurement** event from Cumulocity, it extracts the required information from the event and then transfers the information to the connected blocks for further processing.

Models can process data from multiple devices, and scale up (using multiple cores) when doing so. For detailed information, see [Model execution for different devices](#).

INFO

By default, the **All Inputs** option is selected, which means that the input block is listening to all input sources.

In addition, Analytics Builder supports input devices that are referred to as "broadcast devices". Signals from these devices are available to all models across all devices. For detailed information, see [Broadcast devices](#).

Output blocks

An output block is a special type of block that receives data from a connected processing block. It converts the data into a format understandable to an external source and transfers the data to the external source. For example, when an output block receives data from a connected processing block, it packages the data into an **Operation** object and then sends the operation to Cumulocity.

You can specify a **Trigger Device** for an output block. This is a special device which can be used to send the output back to the device which triggered the output. Models can process data from multiple devices, and scale up (using multiple cores) when doing so. For detailed information, see [Model execution for different devices](#).

INFO

If you use the default option of **All Inputs** as the input source for an input block, you must set the output destination of the output block to **Trigger Device**.

Other output blocks are **Send Email** and **Send SMS** to send emails and text messages. These blocks depend on the tenant environment being correctly configured to be able to deliver the emails and text messages, see also [SMS provider](#). Unlike the other blocks, these are not associated with devices within the Cumulocity platform.

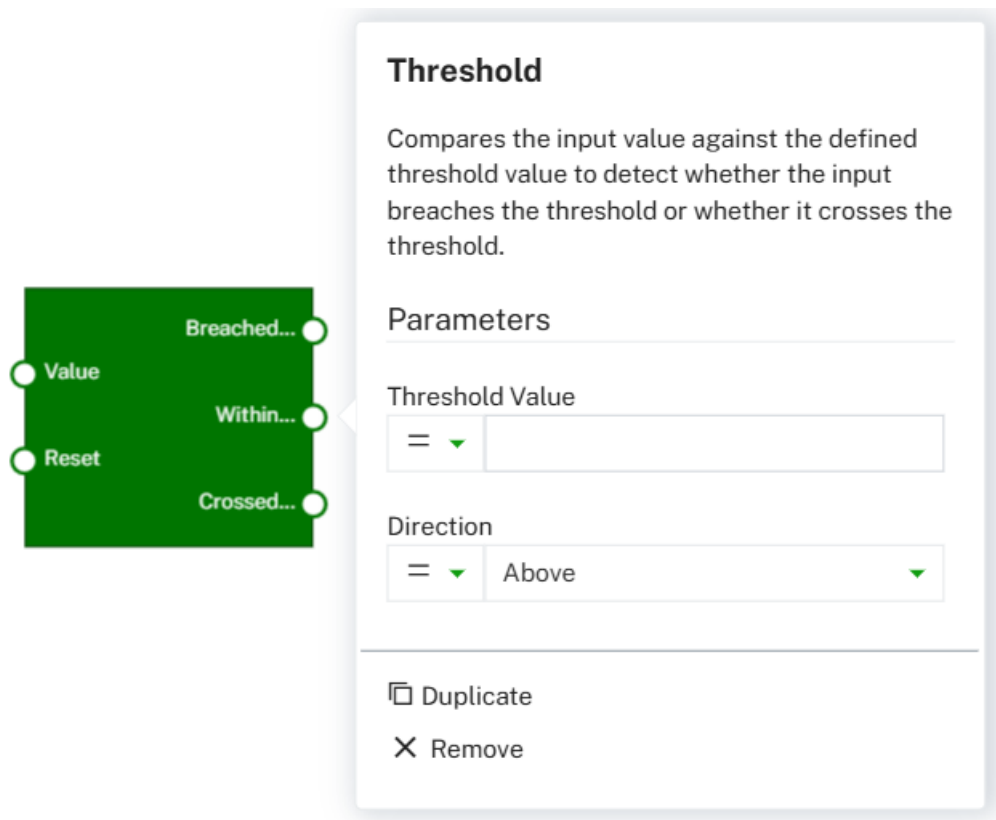
Processing blocks

There are different types of processing blocks. They are grouped into different categories in the palette in the model editor, depending on their functionality.

This category	includes blocks that
Logic	perform logical operations on the data. Blocks such as AND and OR are in this category.
Calculation	perform mathematical operations on the data. Blocks such as Difference , Threshold , Direction Detection , Delta and Expression are in this category.
Aggregate	perform aggregation of the data over a window of values. Blocks such as Average (Mean) and Integral are in this category.
Flow Manipulation	manipulate the flow of the data. Blocks such as Time Delay , Gate , Pulse and Latch Values are in this category.
Utility	provide miscellaneous utility functions. Blocks such as Toggle and Missing Data are in this category.

Example of a processing block - the Threshold block

The following example shows what a block looks like in the model editor, together with the block parameter editor. It shows the **Threshold** block, which detects whether the input value breaches the threshold or whether it crosses the threshold.



The parameters are:

- **Threshold Value.** `float` type. This value is compared against the input value.
- **Direction.** The direction in which to look: whether the input value is above or below the defined threshold, or whether it crosses the threshold.

The input ports are:

- **Value.** `float` type. The input value to the block, to be compared against the defined threshold value.
- **Reset.** `pulse` type. When a signal is received, the state of the block is reset so that any previously received input values are no longer used.

The output ports are:

- **Breached Threshold.** `boolean` type. Is set to `true` when the threshold has been breached. That is, the input value is beyond the range of the defined threshold value.
- **Within Threshold.** `boolean` type. Is set to `true` when the threshold has not been breached. That is, the input value is within the range of the defined threshold value.
- **Crossed Threshold.** `pulse` type. Sends a signal when the input value crosses the threshold, going from one side of the threshold to the other.

Creating your own blocks

You can use the Analytics Builder Block SDK to write, test, and package custom blocks and to upload these blocks into Analytics Builder.

The Block SDK is available from GitHub at <https://github.com/Cumulocity-IoT/apama-analytics-builder-block-sdk>. See the documentation in GitHub for detailed information.

You write the custom blocks in Apama's Event Processing Language (EPL). Once you have written a block, you can package it into an extension and upload it. An example command line to build and upload an extension is:

```
analytics_builder build extension --input path --cumulocity_url $C8Y_URL --username $C8Y_USERNAME --password $C8Y_PASSWORD --name customBlocks --restart
```

To upload an extension, the user specified in the `--username` argument must have CREATE permission for "Inventory" in Cumulocity, in addition to the permissions listed in [Prerequisites](#).

The Apama-ctrl microservice is restarted after running the above command. The user must have the ADMIN permission for "CEP management" to request a restart.

INFO

When using the multi-tenant Apama-ctrl-mt microservice, only extensions uploaded to the tenant that owns the microservice will be used.

WIRES

One block is connected to another block with the help of wires. All data transfer between the output port of one block and the input port of another block is done using wires. All connections must be made between compatible types. See [Wires and blocks](#) for detailed information.

INFO

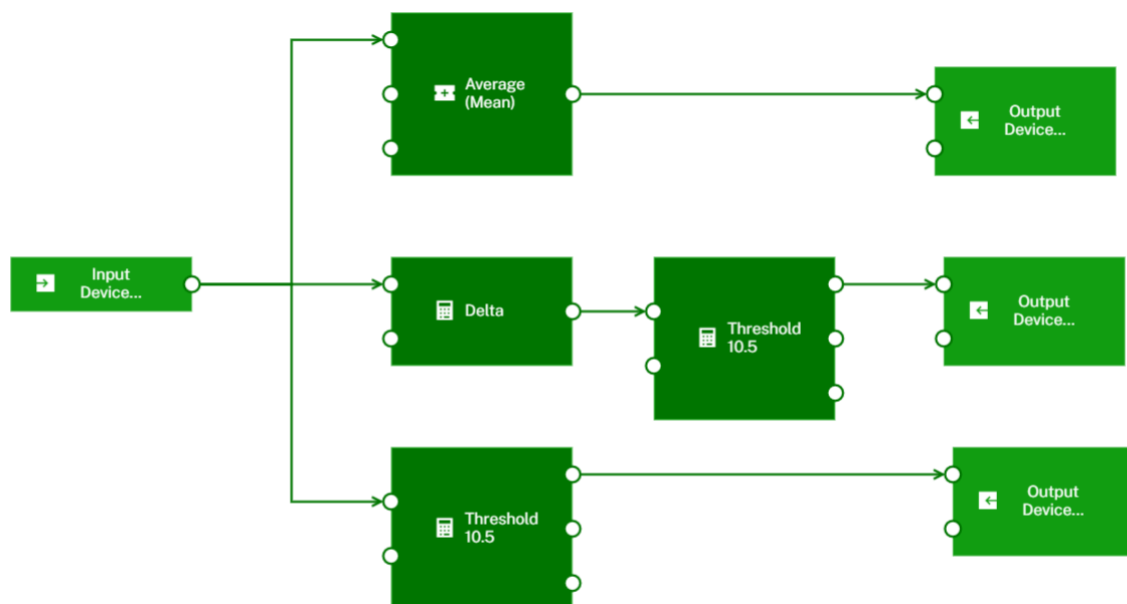
The network of blocks in a model cannot contain any kind of cycles. See [Wire restrictions](#) for more information.

SAMPLE USE CASE

Consider a situation where you are getting real-time sensor data and you want to analyze this data. For the sake of simplicity, let us assume that there is only one sensor and that you are interested in the following:

- You want to know the average value of the sensor readings over a period of time.
- You want to detect sudden changes in the sensor readings using a defined threshold value.
- You want to ensure that the sensor readings are within a certain range and that an alert is created if the readings go beyond that range. For example, you are getting pressure readings and you want to ensure that the maximum pressure does not go beyond the range that the device can handle.

The model for this example has the following blocks:



- The input block shows **Input Device** as the device name. The incoming data is in real time and continuous. The input block receives the data from the sensor. It passes the data to the **Average (Mean)**, **Delta** and **Threshold** blocks. The input ports of these blocks are connected to the output port of the input block.
- The **Average (Mean)** block finds the average (or mean) of the readings that it receives over a period of time and passes this to the connected output block.
- The **Delta** block calculates the difference between successive input values and passes the calculated value to the connected **Threshold** block.
- The model has two different instances of a **Threshold** block. A **Threshold** block compares the input value against the defined threshold value to detect whether the input breaches the threshold or not. The first instance is connected to the **Delta** block and reports a breach if the delta value goes beyond the threshold. The second instance is connected to the input block and reports a breach if the input value is not within the threshold.
- The model has three instances of an output block which show **Output Device** as the device name. The first instance sends the average of the sensor reading. The second instance generates an output if the values of successive sensor readings change by more than the configured threshold. The third instance generates an output if the sensor value goes beyond the configured threshold.

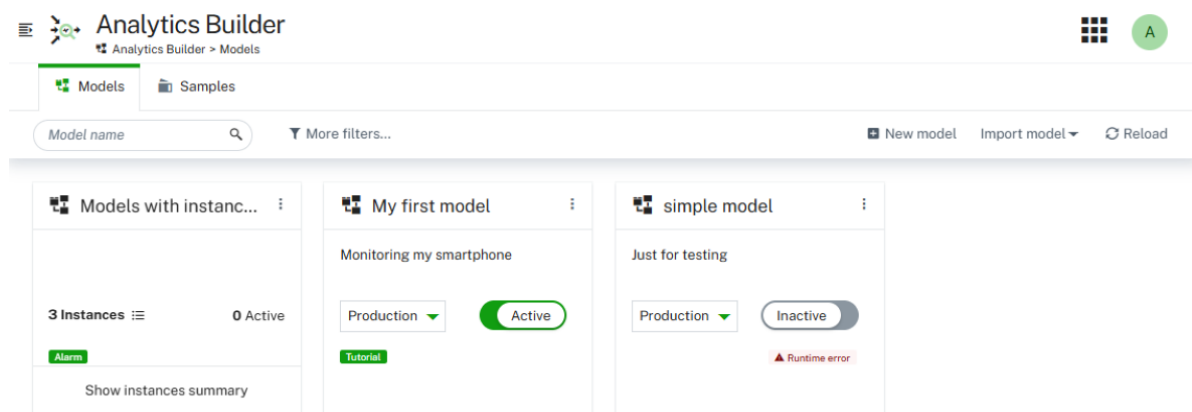
USING THE MODEL MANAGER

THE MODEL MANAGER USER INTERFACE

The model manager contains two tabs: the **Models** tab which shows all currently defined models and the **Samples** tab which shows sample models that are intended to help you get started with creating your own models.

The Models tab

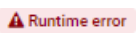
The **Models** tab lists all available analytic models within the current Cumulocity environment as cards. You add new models and manage the existing models from here.



To edit a model, you can simply click on the card that is shown for the model (see also [Editing an existing model](#)). When you add a new model or edit an existing model, the model editor is invoked in which you define the blocks and wires that make up a model. See [Using the model editor](#) for detailed information.

There are two types of models, and the cards for these models look different:

- When a card shows a mode (such as **Draft** or **Production**) and state (**Active** or **Inactive**), it pertains to a model that has no template parameters. Such a model can be activated immediately in the model manager. See [Deploying a model](#) for more information.


If a runtime time error icon  is shown on the card of a deployed model, this model is no longer processing events. Click the runtime error icon to display information on what went wrong.

- When a model has template parameters, it acts as a template. In this case, the number of defined and active instances is shown on the card. A template model is not activated directly in the model manager. Instead, you use the instance editor to create a


number of instances, where each instance provides values for the template parameters. Each instance has a mode and can be activated and deactivated in the instance editor, as with models without any template parameters.

To edit the instances, you can simply click the total number of instances (see also [Editing the instances of a model](#)). This invokes the instance editor. See [Using the instance editor](#) for detailed information.

You can flip the card for a template model to show more details. Click **Show instances summary** to do this. You can then see the number of instances in the different modes.

If an error icon such as  **1 error** is shown on the card of a template model, at least one of the instances is no longer processing events. Click the error icon to display information on what went wrong.

As long as a model has no template parameters, there will be zero instances and the card shows the controls for selecting a mode and activating it.

Each card that is shown for a model has an actions menu  at the top right which contains commands for managing the model (for example, to download or delete the model).

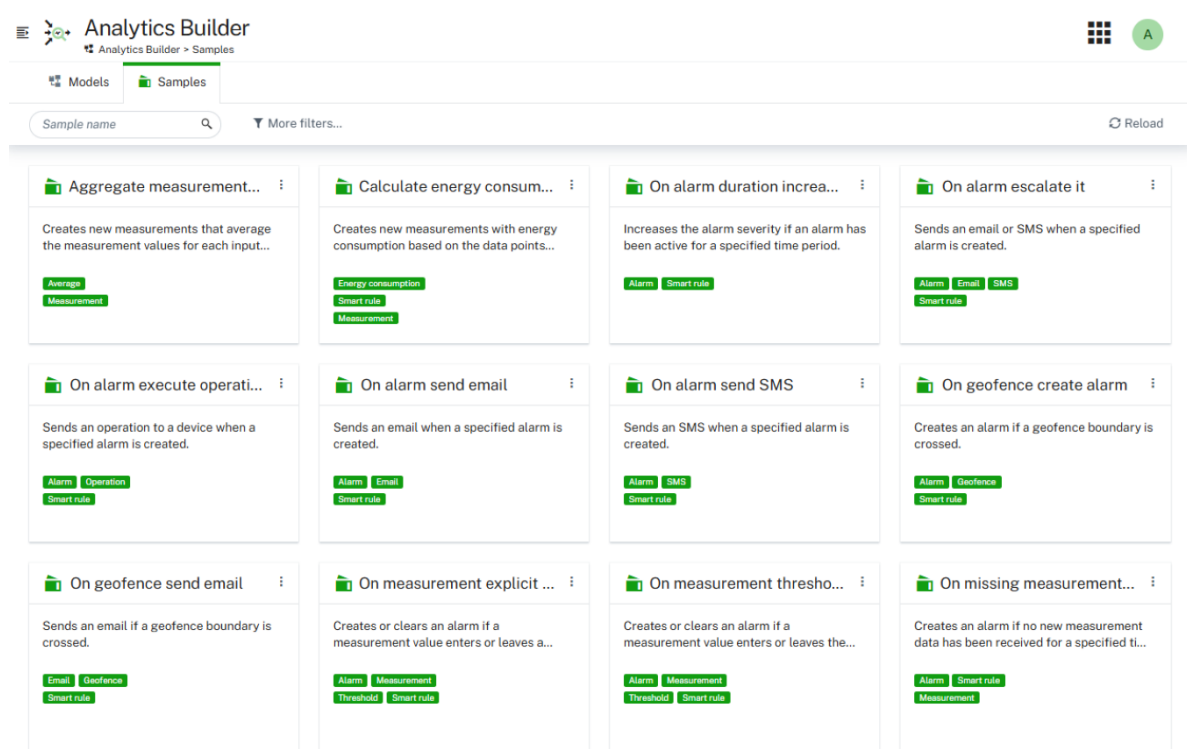
If a description or tags have been defined for the model, this is shown on the card for that model. If you want to change the name, the description or the tags of a model, you must do this in the model editor. See [Changing the name, description, and tags of a model](#).

If you have a long list of cards, you can easily locate the model that you are looking for by entering its name in the **Model name** search box. Or you can enter part of the model name. For example, enter the word “test” to find all models that have this word in their names. The characters that you type in may be contained at any position within the model name. These search criteria are not case-sensitive. When search criteria are currently applied, **Clear search** is shown next to the search box; click this to clear the search and thus to show all available cards.

You can also reduce the number of shown cards by using a filter. See [Filtering the models and samples](#) for detailed information.

The Samples tab

The **Samples** tab lists all sample models that are provided with Analytics Builder as cards.



If the name of a sample or its description is not fully shown on the card, you can hover over the name or description to see the full name or description in a tooltip.

You can view the samples, but you cannot edit or deploy them. To view a sample, you can simply click on the card that is shown for the sample. The model editor is then invoked in read-only mode. See [Viewing a sample](#) for more information.

If you want to use a sample as a basis for further development, you can create a model from the sample. You can then edit the new model according to your requirements and deploy it. See [Creating a model from a sample](#) for more information.

You can easily locate a sample by entering its name or part of the name in the **Sample name** search box (for example, “geofence” or “email”). You enter and clear the search criteria in the same way as described above for the **Model name** search box on the **Models** tab. You can also filter the samples by their tags; see [Filtering the models and samples](#) for more information.

FILTERING THE MODELS AND SAMPLES

The model manager offers several ways to reduce the number of cards that are shown on the **Models** and **Samples** tabs, thus letting you quickly locate the models or samples that you are looking for.

Filtering also works in combination with a model or sample name that you specify in the **Model name** or **Sample name** search box which is explained in [The model manager user interface](#).

To filter the models or samples

1. On the **Models** or **Samples** tab of the model manager, click **More filters** in the toolbar.
2. In the resulting dialog, select one or more filters for the models. For samples, it is only possible to filter by tag. On the **Models** tab, you can filter the models according to the following criteria:
 - **Mode**. You can show only the models that are in a specific mode. For example, if you only want to see the models that are in simulation and test mode, select the corresponding checkboxes.
 - **Status**. You can show only the models that are either active or inactive. For example, if you only want to see active models, select the corresponding checkbox.
 - **Source or destination**. You can show only the models that use specific input sources or output destinations. Open the **Filter by source or destination** drop-down list box, select one or more items and click **Apply**.
 - **Data point**. You can show only the models that use specific data points, such as `c8y_TemperatureMeasurement`. This requires that at least one item has been selected in the **Filter by source or destination** drop-down list box. Open the **Filter by data points** drop-down list box, select one or more data points, and click **Apply**.
 - **Tags**. You can show only the models for which specific tags have been defined in the **Create model** dialog box which is shown when you add a new model or when you invoke the **Edit model** dialog box from the model editor (see also [Adding a new model](#) and [Changing the name, description, and tags of a model](#)). Open the **Filter by tag** drop-down list box, select one or more tags, and click **Apply**.

You can combine several types of filters, for example, to show only active models in production mode that use a specific device.

On the **Samples** tab, you can filter the samples by tag only. Open the **Filter by tag** drop-down list box, select one or more tags, and click **Apply**.

All of the above-mentioned drop-down list boxes include a **Filter** search box that you can use to reduce the number of items that are offered for selection. You can enter a name or part of a name. For example, enter the word “test” to show only the items that have this word in their names. The characters that you type in may be contained at any position within the name. These filter criteria are not case-sensitive. Clicking the **All** checkbox selects all items that are currently shown in the drop-down list box, depending on the contents of the **Filter** search box.

3. Click **Apply filters**. The toolbar of the **Models** or **Samples** tab now shows the filters that are currently applied. This is an example

of the **Models** tab:



Click **Clear**

filters in the toolbar if you want to clear all filters. Or to clear a specific filter, click the X that is shown in the label for this filter, or click the filter name in the label and deselect that filter (and other filters if required) in the resulting dialog box. Clicking **Reset filters** in that dialog box clears all filters.

ADDING A NEW MODEL

When you add a new model, the model editor is invoked. See [Using the model editor](#) for detailed information.

INFO

The new model will only be listed in the model manager, when you save the model in the model editor. See also [Saving a model](#).

You can also create a new model from a sample. See [Creating a model from a sample](#) for more information.

To add a new model

1. On the **Models** tab of the model manager, click **New model** in the toolbar.
2. In the resulting **Create model** dialog box, enter a unique model name.

You can optionally enter a description for the model and one or more tags. Tags are helpful for filtering the models in the model manager to show only the models for which a specific tag has been defined (see also [Filtering the models and samples](#)). To add a tag, you simply type its name and press Enter or the Tab key. The tag is then shown in a colored rectangle. To remove a tag, click the X that is shown in the rectangle. The dialog prevents you from entering duplicate tags for a model; if you enter such a tag name, the duplicate tag is not added and the original tag blinks one time.

3. Click **OK**. The model editor appears. See [Overview of steps for adding a model](#) for a brief overview of how to add blocks and wires to the new model.

INFO

When you click **Cancel** without specifying a model name, the model editor also appears and the default name “New” is then shown in the breadcrumb. You can add blocks to the model, but as long as you do not specify a model name, you will not be able to save the model. Click **Model settings** and specify a model name. See also [Changing the name, description, and tags of a model](#).

EDITING AN EXISTING MODEL

You can edit (or view) each model that is currently listed in the model manager.

When a model is active, editing will set the model to read-only mode. In this case, the model editor only allows you to view the contents of the model (for example, you can view the block parameters). You can navigate and zoom the model as usual, but you cannot change anything.

To edit a model

On the **Models** tab of the model manager, simply click the card that is shown for the model (but not on the toggle button for changing the state or the drop-down menu for changing the mode).

Alternatively, click the actions menu  of the card and then click **Edit**.

When the model is active, a dialog appears informing you that you can only view the model. When you click **Continue**, the model editor appears and you can view the model, but you cannot change it. See [Using the model editor](#) for further information.

INFO

If you do not have sufficient permissions (that is, you only have READ permission for “CEP management” instead of ADMIN permission), the actions menu provides a **View** command instead of the **Edit** command.

EDITING THE INSTANCES OF A MODEL

When one or more blocks in a model use template parameters (see also [Managing template parameters](#)), you can set up different instances of that model.

Each instance can then use different values for the template parameters and can be activated independently from the other instances. The instances are defined and activated in the instance editor.


INFO

The actions below are only available when template parameters have been defined for the model, that is, when the card for the model shows the number of defined instances.

To edit the instances of a model

On the **Models** tab of the model manager, click the total number of instances on the front of the card.

Alternatively, you can also do one the following:

- Click the actions menu  of the card and then click **Instances**.
- Or click **Show instances summary** to flip the card and then click the **Edit Instances** button on the back of the card.

INFO

Show instances summary is only visible (and thus you can only get to the back of the card) if there are any instances (regardless of state).

This invokes the instance editor. See [Using the instance editor](#) for further information.

DEPLOYING A MODEL

A model (or instance) can have one of two states. The current state is always indicated on the card that is shown for a model:

- **Active.** This state indicates that the model has been deployed.
- **Inactive.** This state indicates that the model is currently not deployed.

The inputs that a model receives and what happens to its outputs depends on the mode to which the model is set. Each model can be set to one of the following modes:

- **Draft.** The model is still under development. (New models are created in draft mode.)
- **Test.** This mode is only permitted for models using a single device. When active, the model is deployed to the Apama correlator so that the measurements and events from the device are processed. The output of the model is only stored (and recorded as an **Operation** or **Measurement** object of a “virtual device”) and not sent back to the device.

INFO

Test mode is not supported for a model which contains a custom block which consumes input data and also produces output data. Custom blocks are created with the Block SDK; see also [Creating your own blocks](#).

- **Simulation.** This mode is only permitted for models using a single device. When active, the model uses historical input data (replayed in real time from previously received data) and is deployed to the Apama correlator. The output of the model is only stored (and recorded as an **Operation** or **Measurement** object of a “virtual device”) and not sent back to the device. To start a simulation, you must define the time range from which the input data is to be used. When all data from the time range has been replayed, the model is automatically undeployed from Apama and the model state is changed to **Inactive**. The timestamps of the historical data entries remain unchanged for easier comparison of simulation runs. See also [Model simulation](#).
- **Production.** When active, the model is deployed to the Apama correlator so that the measurements and events from the devices are processed. The output of the model is stored and sent back to the devices.


A model in draft mode can only be in the inactive state. A model in test, simulation or production mode can be in either the active or inactive state.

INFO

The above information on the different states and modes similarly applies for the instances of a template model. The following instruction, however, only applies for non-template models. If you want to deploy the instances of a template model, see [Deploying an instance](#).

When a model is imported by loading a JSON file, it is always imported as an inactive model.

To deploy a model

1. On the **Models** tab of the model manager, click the drop-down menu on the card for the model that you want to deploy and select one of **Production**, **Test** or **Simulation**.
2. If you have selected simulation mode, click the calendar icon  which is now shown, specify the time span that is to be used, and click **Apply**. See also [Simulation parameters](#).
3. When the toggle button currently shows **Inactive**, click this button to change the state to **Active**. For simulation mode, you can only set the state to **Active** when a valid time range has been defined.

UNDEPLOYING A MODEL

You can undeploy (that is, deactivate) each model that is currently in production, test or simulation mode and for which the toggle button shows **Active**.

When you undeploy a model, the model is stopped and no longer processes incoming data. Any state built up in the model is lost. For simulation mode, this means that the model is stopped before all historical data from the specified time range has been replayed.

INFO

If you want to undeploy the instances of a template model, see [Undeploying an instance](#).

To undeploy a model

In the **Models** tab of the model manager, click the toggle button on the card for the model that you want to undeploy so that **Inactive** is then shown on the button.

DUPLICATING A MODEL

You can duplicate each model that is currently listed in the model manager.

The duplicated model gets the same name as the original model followed by the number sign (#) and a number. For example, when the name of the original model is "My Model", the name of the first duplicate is "My Model #1". The number in the model name is increased by one with each subsequent duplicate that you create. The duplicated model gets the same description as the original model. It is recommended that you edit the duplicate and give the model a meaningful name and description.

To duplicate a model

On the **Models** tab of the model manager, click the actions menu  of the model that you want to duplicate and then click **Duplicate**.

A card for the duplicated model is immediately shown in the model manager.

DOWNLOADING A MODEL

You can download each model that is currently listed in the model manager. This is helpful, for example, if you want to transfer a model from the current Cumulocity tenant to a different tenant. The model is saved in JSON format.

To download a model

On the **Models** tab of the model manager, click the actions menu  of the model that you want to download and then click **Download**.

The resulting behavior depends on your browser. The model is usually downloaded to the download location of your browser.


INFO

The **Download** command is also available when the model is active (read-only mode). This allows you to download the model at any time.

COPYING A MODEL

Instead of downloading a model, you can also copy the JSON code of the model to the clipboard and then paste it into an editor of your choice.

To copy a model

On the **Models** tab of the model manager, click the actions menu  of the model that you want to copy and then click **Copy**.

INFO

The **Copy** command is also available when the model is active (read-only mode). This allows you to copy the JSON code at any time.

UPLOADING A MODEL

You can upload a model that has previously been downloaded in JSON format. This is helpful, for example, if you want to upload a model from a different Cumulocity tenant.

To upload a model

1. On the **Models** tab of the model manager, click **Import model** in the toolbar, and then click **Upload**.
2. In the resulting dialog box, navigate to the location where the model that you want to upload is stored.
3. Select the model and click **Open**.

A card for the uploaded model is shown in the model manager.

PASTING A MODEL

Instead of uploading a model, you can also paste the JSON code for a model from the clipboard.

To paste a model

On the **Models** tab of the model manager, click **Import model** in the toolbar, and then click **Paste**.

INFO


The **Paste** command is enabled only if the clipboard contains content.

If the clipboard contains valid JSON code for a model, a card for the pasted model is shown in the model manager.

DELETING A MODEL

You can delete each model that is currently listed in the model manager. When you delete a model that is currently deployed, it is first undeployed and then deleted.

To delete a model

1. On the **Models** tab of the model manager, click the actions menu  of the model that you want to delete and then click **Delete**.
2. In the resulting dialog box, click **Delete** to confirm the deletion.

RELOADING ALL MODELS

You can refresh the display to show any changes other users have made since the page loaded, or to see whether deployed models have entered a failed state.

To reload all models

On the **Models** tab of the model manager, click **Reload** in the toolbar.


VIEWING A SAMPLE

The samples are always in read-only mode. You can view the contents of each sample that is currently listed in the model manager.

For example, you can look at the block parameters and view the documentation for each block that is used in the sample. You can navigate and zoom the sample in the same way as a regular model, but you cannot add or edit anything. However, you can create a new model from a sample (see also [Creating a model from a sample](#)).

To view a sample

On the **Samples** tab of the model manager, simply click the card that is shown for the sample.

Alternatively, click the actions menu  of the card and then click **View**.


CREATING A MODEL FROM A SAMPLE

You can create a new model from each sample that is currently listed in the model manager. The new model gets the same name, description and tags as the sample.

INFO

You must save the new model so that it is listed in the model manager. If a model with that name already exists, you are prompted to save the new model with a different name.

To create a model from a sample

On the **Samples** tab of the model manager, click the actions menu  of the sample from which you want to create a new model and then click **Create model from sample**.

Alternatively, when the sample is currently shown in the model editor, click **Create model from sample** in the toolbar.

The new model is immediately shown in the model editor and you can now change each aspect of the model.

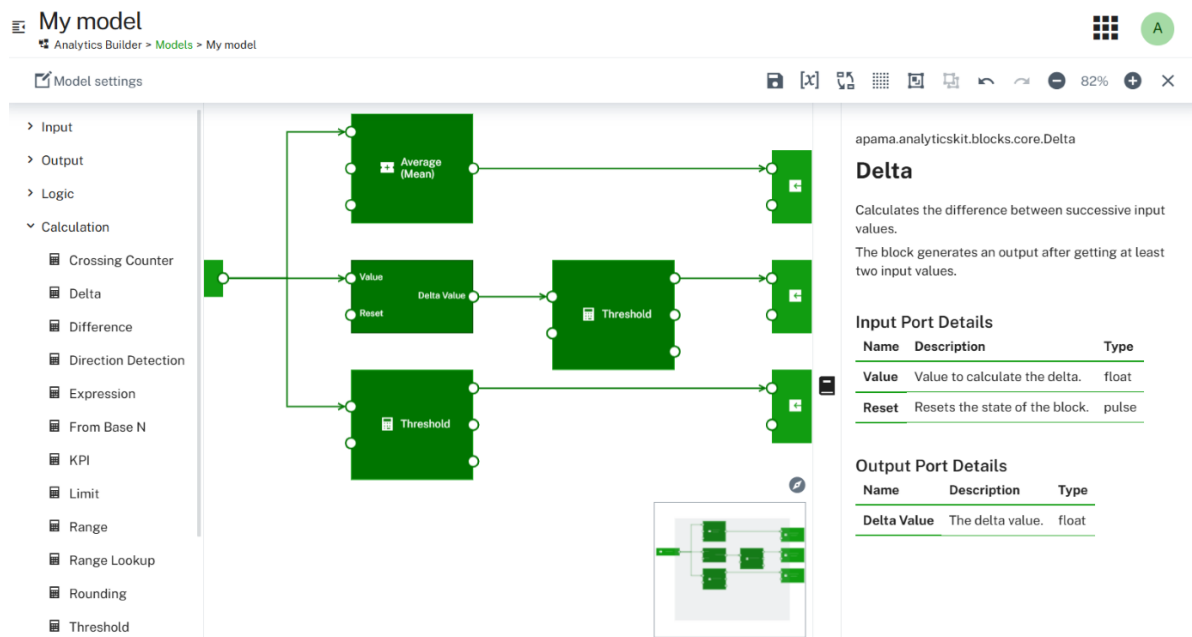
USING THE MODEL EDITOR

THE MODEL EDITOR USER INTERFACE

The model editor allows you to create analytic models graphically. It is invoked when you add or edit a model in the model manager. See also [Adding a new model](#) and [Editing an existing model](#).

INFO

The model editor is also invoked when you view a sample. In this case, the model editor is invoked in read-only mode and the palette on the left is not shown. See also [Viewing a sample](#).




The palette on the left contains the blocks that you can add to your model. It has several expandable/collapsible categories for the different types of blocks.

The canvas in the middle is the area in which you “draw” your model. You drag the blocks from the palette onto the canvas, specify the parameters for the blocks, and wire the blocks together. The content of the canvas is aligned to a grid, see also [Showing and hiding the grid](#).

The overview area at bottom right of the canvas shows the entire model. This is helpful if your model is too large to fit on the currently visible area of the canvas. See also [Navigating large models](#).

The documentation pane on the right allows you to view reference information for the currently selected block. See also [Viewing the documentation for a block](#).

CAUTION

Changes are only saved when you click the save icon . See also [Saving a model](#). The editor warns you if you attempt to navigate away from the editor and there are unsaved changes. However, you should always ensure that your changes are saved before disconnecting the browser from the network or suspending a laptop.

WORKING WITH MODELS

Overview of steps for adding a model

This topic gives a brief overview of how to add and design a new model. For more detailed information, see the topics that are referenced in the steps below.

You add and design a model as follows:

1. On the **Models** tab of the model manager, click **New model**. Enter a model name in the resulting dialog box. See also [Adding a new model](#).
2. In the model editor, drag the required blocks from the palette onto the canvas. See also [Adding a block](#).
3. Refer to the block documentation as necessary. See also [Viewing the documentation for a block](#).
4. Use the block parameter editor to specify the parameters of the block. See also [Editing the parameters of a block](#).
5. Connect the appropriate blocks with wires. See also [Adding a wire between two blocks](#).
6. Save your changes. See also [Saving a model](#).

INFO

Only saved models are listed in the model manager. When you add a new model and then leave the model editor without saving the model, it will not be listed in the model manager, and all the edits you made will be lost.

7. Leave the model editor. This takes you back to the model manager. See also [Leaving the model editor](#).
8. A newly added model is automatically set to draft mode in the model manager. If you want to test it, simulate it, or make it available in production, see [Deploying a model](#).

For detailed background information, including restrictions, see [Wires and blocks](#).

Changing the name, description, and tags of a model

You can rename each model that you are currently editing in the model editor, and you can also change the description of each model.

You can also add or remove tags. Tags are helpful in the model manager, to show only the models for which a specific tag has been defined, see also [Filtering the models and samples](#).

To change the name, description, and tags of a model

1. In the model editor, click **Model settings** which is shown at the left of the toolbar.
2. In the resulting **Edit model** dialog box, specify a new unique name for the model, change the description, and/or change the tags.

To add a tag, you can either:

- Type its name and press Enter. The tag is then shown in a colored rectangle.
- Type its name and submit the dialog without pressing Enter. Any pending text in the **Tags** field will automatically be added as a tag before submission.

To remove a tag, click on the delete icon (X) that is shown in the rectangle.

The dialog prevents you from entering duplicate tags for a model. If you enter such a tag name, the duplicate tag is not added, and the original tag blinks once.

3. Click **OK**.

Saving a model


When you save a model in the model editor, it is stored in the Cumulocity inventory for your tenant, in JSON format.

IMPORTANT

It may happen that you and another user are editing the same model at the same time. In this case, the changes that are saved last will be stored. So your changes might be overwritten by a later save by another user.

To save a model

In the toolbar of the model editor, click the save icon .

The save icon  is only enabled when changes have been applied to the model and the model has been given a name.

Leaving the model editor

When you leave the model editor as described below, you are returned to the model manager. You can then, for example, edit a different model, or change the mode or state of the current model.

⚠ CAUTION

All unsaved changes are lost when you navigate to a different URL or close the browser window.

To leave the model editor

In the toolbar of the model editor, click the close icon .

In case there are still unsaved changes, you are asked whether to save or discard them.

WORKING WITH BLOCKS AND WIRES

Adding a block

The blocks in the palette are grouped into different categories. When you move the mouse pointer over a block in the palette, a tooltip appears which briefly explains the purpose of the block. The tooltip also shows the entire name of the block.

Detailed information for each block is available in the block reference, which is shown in the documentation pane. See also [Viewing the documentation for a block](#).

To add a block

1. In the palette of the model editor, expand the category which contains the block that you want to add.
2. Drag the block from the palette and drop it on the canvas.

When you drop the block on an existing block on the canvas, the new block is created on top of that block. When you drop the block on a collapsed group, the new block is created below that group. In both cases, you should move the new block to a free space of the canvas. See also [Moving a block](#).

When you drop the block on an expanded group (where the contents of the group are visible), the new block is added to that group. For more information on groups, see [Working with groups](#).

3. Specify all required parameters for the block. See [Editing the parameters of a block](#).

i INFO

The block parameter editor is automatically shown when you add a block for which parameters must be specified. It is not shown, however, if the block does not require any parameters (such as the **OR** block).

Editing the parameters of a block

Most blocks (but not all) have parameters that you must set according to your requirements.

When "Missing" is shown on an input or output block on the canvas, this means that the defined input source or output destination cannot be found in the Cumulocity inventory. You should then either go to the Cumulocity inventory and make sure that the device is registered or that the group or asset exists, or you should select a different, existing input source or output destination in the block parameter editor

(see below).

The labels of some blocks on the canvas show the value of the most important parameter. For example, the **Expression** block shows the defined expression, and the **Time Delay** block shows the defined delay in seconds.

The block parameter editor also contains commands for duplicating and removing the currently selected block. See [Duplicating a block](#) and [Removing a block or wire](#) for detailed information.

For the input and output blocks, you can globally replace the input sources and output destinations that are used. See [Replacing sources or destinations](#) for detailed information.

To edit the parameters of a block

1. On the canvas of the model editor, click the block that you want to edit using the left mouse button. The block parameter editor appears, providing input fields for all parameters that can be specified for that block.
2. For the input and output blocks, you can select a different input source or output destination from a dialog box.

The tree in the dialog box reflects the parent/child hierarchy in the Cumulocity inventory. For example, the list of devices includes any defined child devices, and the list of groups includes any defined sub-groups. These are available from expandable/collapsible nodes. By default, 10 items are shown within each node, sorted alphabetically. With a large inventory, you will have to click **Load more** to display any items that are not shown initially.

Using the checkboxes at the top of the dialog, you can filter the entries that are to be shown. By default, all defined devices, groups and assets are displayed in the dialog. If you also want to display any other managed objects in the dialog, select the "Other" checkbox.

The search box can be used to show any managed objects in the Cumulocity inventory which match your search and filter criteria. The search is case-sensitive. The characters that you type in may be contained at any position within the name. The tree is updated with each character that you type. With a large search result, you will have to click **Load more** to display any managed objects that are initially not shown.

The managed objects that are shown when searching depend on a tenant option. You can restrict the search to show only managed objects of a specific type. For more information, see [Searching for devices, groups and/or assets](#).

Click the button which is shown when you hover over an entry to select the input source or output destination that you want to use. The name of that button depends on the entry that is currently selected:

Button name	Shown for	Description
Select	All Inputs option for input blocks	Data is received from all input sources.
Select	Trigger Device option for output blocks	Output is sent to the device which triggered the output.
Select device	Devices in input and output blocks	Data is received from the device or output is sent to the device.
Select group's devices	Groups in input blocks	Data is received from all devices within the group hierarchy. You cannot directly receive data from a group.
Select asset	Assets in input and output blocks	Data is received from the asset itself or output is sent to the asset itself. The devices of the assets are ignored.
Select asset's devices	Assets in input blocks	Data is received from all devices within the asset hierarchy. The block does not receive data from the asset itself.



INFO

For output blocks, you cannot select a group. A button is not provided in this case. Select the **Trigger Device** option instead to send the output to the device which triggered the output.



The maximum number of shown input sources and output destinations depends on a tenant option. For more information, see [Configuring the number of shown input sources and output destinations](#).

3. It is possible to use a template parameter instead of specifying a value for a block parameter. This allows different values to be used for this block parameter in different instances of the model (see [Using the instance editor](#) for more information). Create a template parameter of a matching type in the **Template Parameters** dialog box (see [Managing template parameters](#)), switch the block parameter to use a template parameter (see below) and select the desired template parameter from the drop-down list box. Or create the template parameter directly in the block parameter editor (see below).

The block parameter editor provides the following options in a drop-down list box:

-  When selected, you can specify a value for this parameter using the adjacent control. This value is validated in the block parameter editor.
-  When selected, you can select a template parameter from the adjacent drop-down list box. You can only select a template parameter that is of the same type as the block parameter to which you want to assign it; template parameters of unsuitable types are not available for selection. Template parameters are not validated in the block parameter editor.

If you want to add a new template parameter directly in the block parameter editor, type a name in the text box of the above drop-down list box. As soon as you start typing and if a template parameter with that name does not yet exist, the option **Add template parameter name** is shown below the text box. Click this option to add the new template parameter and thus make it available in the **Template Parameters** dialog box. The new template parameter will have the same type, optional and default values as the block parameter. If a template parameter with the name that you are specifying exists already, but with an incompatible type, the name and type is shown below the text box but cannot be selected.

4. Some blocks support multi-line input for certain block parameters. For example, the **Send Email** block supports this in the **Text** parameter and the **Alarm Output** block supports this in the **Message** parameter. Your input is automatically wrapped in the text box and you can press Enter to start text on a new line. When you add a new template parameter for such block parameter directly in the block parameter editor (as described above), the type of the new template parameter is automatically set to **Multi-line String**.
5. For some blocks (such as the **Range Lookup** block), the block parameter editor shows text boxes for specifying key-value pairs. If you want to specify more key-value pairs, click **Add row**. The key-value pair in the first row is processed first. You can drag a row to a different position using the move icon  that is shown next to that row. You can delete a row that you do not need any more by clicking the delete icon  next to that row. Empty rows are automatically deleted when you leave the block parameter editor.
6. Specify all required parameters. Detailed reference information for each block is available from the documentation pane. See also [Viewing the documentation for a block](#).

Your input is kept in memory when you leave the block parameter editor (for example, when you click on another block or the canvas).

INFO

Keep in mind that your changes are only written to the inventory when you save the model. See also [Saving a model](#).

Viewing the documentation for a block


The documentation pane allows you to view detailed information for the currently selected block. It shows the so-called *block reference* which provides documentation of a block's parameters, input ports and output ports. You can resize the documentation pane, and you can also toggle its display.

INFO

You can also view the block reference directly in this documentation. See [Analytics Builder block reference](#).

To view the documentation for a block

1. In the model editor, click the block for which you want to view the documentation. You can do this in the palette or on the canvas.

2. If the documentation pane is currently not shown, click the area that contains the document icon  (shown at the right of the canvas) to display the documentation pane. Clicking that area again hides the documentation pane.
3. If you want to resize the documentation pane (for example, to make it larger), move the mouse pointer over the area that contains the document icon. Click and hold down the mouse button and drag the mouse to the left or right (to make the documentation pane wider or smaller).

Selecting blocks and wires

If you want to move, duplicate or remove one or more blocks that are currently shown on the canvas of the model editor, you must first select the required blocks.

To select a single block on the canvas, just click the block. With a block, the resulting behavior depends on the mouse button that you use:

- When you click the block using the left mouse button, the block is selected and the block parameter editor is shown (see also [Editing the parameters of a block](#)).
- When you click the block using the right mouse button, the block is selected only (the block parameter editor is not shown). This is helpful if the editor would be in the way, for example, when adding a wire to another block.

To select a single wire, just click the wire (you can use either mouse button in this case).

To select several blocks and/or wires at the same time, do one of the following:

- Press Ctrl and click each block and/or wire that you want to select.
- Or to select an area containing several blocks and wires, click and hold down the mouse button over an empty space of the canvas and wait until the mouse pointer changes to a cross. Then drag the mouse to select the desired area. Release the mouse button when all required blocks and wires have been selected.
- Or to select all blocks and wires, press Ctrl+A.

To deselect your selection:

- Press Ctrl and click the currently selected block or wire.
- Or to deselect all selections, click an empty space of the canvas.

Moving a block

You can move each block that is currently shown on the canvas to different location. When one or more wires are attached to a block that is moved, the wires are also moved.

To move a block

On the canvas of the model editor, click the block that you want to move, hold down the mouse button and drag the block to the new location.

Alternatively, to move several blocks at the same time, select them as described in [Selecting blocks and wires](#). Then click and hold down the mouse button and immediately drag the blocks to the new location (do not wait until the mouse pointer changes).

Duplicating a block

You can duplicate each block that is currently shown on the canvas. The original block and its duplicate will then both have the same parameters.

When you duplicate a single block, the attached wires are not automatically duplicated. When you duplicate several blocks at the same time, however, the attached wires between the selected blocks are automatically duplicated.

To duplicate a block

- On the canvas of the model editor, click the block that you want to duplicate and then do one of the following:
 - Click the **Duplicate** command which is shown at the bottom of the block parameter editor.
 - Or press Ctrl+C to copy the block, and then press Ctrl+V to paste the block.
 - Or press Ctrl and drag the block to be duplicated to the position at which you want to place the duplicate.
- Or to duplicate several blocks at the same time, select them as described in [Selecting blocks and wires](#) and then proceed as described above. Exception: the **Duplicate** command is only available when you select a single block.

Adding a wire between two blocks

The blocks on the canvas can be wired together to indicate that the output from one block is used as the input for the other block.

The wires are attached to ports, that is, to the circles that are shown to the left and/or right of a block. Each block can have zero, one or more of the following:

- output ports (shown at the right side of a block)
- input ports (shown at the left side of a block)

To see the names of the ports, click the block to select it. Or move the mouse pointer over a port to see the port name in a tooltip (the name is displayed first, followed by the description of the port).

See [Wires and blocks](#) for detailed information on the types of values that can be sent between two blocks, the processing order of wires, restrictions, and more.

To add a wire between two blocks

On the canvas of the model editor, click the output port of the block that you want to connect and drag the mouse to the input port of another block.

Changing a wire

You can change the path that a wire takes to the block to which it is currently connected. And you can also rewire a block so that it is connected to a different block or to a different port of the same block.

Wires cannot create cycles. See [Wire restrictions](#) for detailed information.

To change a wire

1. On the canvas of the model editor, click the wire that you want to change. The port names of the attached blocks are then shown, and the ports attached to each end of the wire are highlighted.
2. To change the path that a wire takes between two blocks, drag one of the square resize icons (■) that are now shown on the selected wire to a different position. Or to move the wire to a different port, drag the diamond-shaped move icon (◆) that is now shown at the input or output port (a hand pointer is shown in this case) to a different port.

Removing a block or wire

You can remove each block or wire that is currently shown on the canvas. When you remove a block, all wires that are attached to this block are automatically removed.

To remove a block or wire

- On the canvas of the model editor, click the block or wire that you want to remove and press Del. In the case of a block, you can alternatively click the **Remove** command which is shown at the bottom of the block parameter editor.
- Or to remove several blocks and/or wires at the same time, select them as described in [Selecting blocks and wires](#) and then press Del.

Undoing and redoing an operation



You can undo and redo each change that has been applied to the canvas. For example, you can undo the removal of blocks, undo changed parameter values, or undo the rerouting of a wire.

It is not possible to undo/redo the change to a model name or its description.

INFO

To use the key combinations mentioned below, the canvas must have the focus. When the documentation pane or the palette currently has the focus, the change on the canvas is not undone/redone.

To undo or redo an operation

- To undo the last operation, click the undo icon  in the toolbar of the model editor or press Ctrl+Z.
- To redo the last operation, click the redo icon  in the toolbar of the model editor or press Ctrl+Y.

The above icons are only enabled when there is an operation that can be undone or redone.

Replacing sources or destinations

You can find the input sources or output destinations that are used in the current model and replace them with other input sources or output destinations that are currently registered in the Cumulocity inventory (visualized in the Device Management application).

INFO

In the rules below, the term device refers to a device or other asset (but not to a group).

The following rules apply:

- You can replace a device with another device.
- You can replace a group with another group.
- You can replace a group or a device with the **All Inputs** option.
- You can replace the **All Inputs** option with a group or a device.
- When you replace a device with a group or the **All Inputs** option:
 - all matching input devices are changed to groups or **All Inputs**, whichever is selected, and
 - all matching output devices are changed to trigger devices.
- When you replace a group or the **All Inputs** option with a device:
 - the group or **All Inputs**, whichever is selected, is changed to a device, and
 - all matching trigger devices are changed to the specified device.


INFO

If you change more than one group to a device at a time, then only the first specified device will be used to replace all trigger devices.

- The **Trigger Device** option is not available for selection in the dialog.


After you have replaced the devices, you must verify that the measurements that are used by the input and output blocks of the current model still refer to the appropriate measurements. The Cumulocity fragment and series are not changed by the replacement, which may or may not apply to the newly defined device.

To replace sources or destinations

1. In the toolbar of the model editor, click the replace icon . This icon is only enabled when at least one source or destination has been defined in the current model. Any defined trigger devices are not considered in this case.
2. In the **Current** drop-down list box of the resulting dialog box, select the source or destination that you want to replace. All input sources and output destinations that are used in the model are available for selection.
3. Click the **Replace with** box to display a dialog box. The dialog box is the same as when selecting a input source or output destination in the block parameter editor. See [Editing the parameters of a block](#) for more information on this dialog box. Click the button which is shown when you hover over an entry to select the source or destination that you want to use instead.
4. If you want to replace further sources or destinations, click **Add row**. This is only shown if more than one source or destination has been defined in the current model.

A new row is shown, containing additional **Current** and **Replace with** drop-down list boxes, and you can now select one more source or destination to be replaced. Any sources or destinations that you have previously selected for replacement are no longer offered for selection in the **Current** drop-down list box.

Repeat this step until all required sources and destinations have been selected for replacement. You can add as many rows as there are sources or destinations in the current model.

5. If you want to delete a row (for example, when you no longer want to replace a selected device), click the delete icon  next to that row. This is only available if the dialog box currently shows more than one row.
6. Click **Replace**.

Managing template parameters

A model may specify zero or more template parameters, which can be used in place of defined values for block parameters. For example, instead of defining 100 as the threshold value for a **Threshold** block in the block parameter editor, you can assign a template parameter.

If a model has any template parameters, then multiple instances of the model can be created, and different values can be specified for the template parameters. See also [Understanding models](#) which explains the difference between models without template parameters and models with template parameters, and the different roles: model author and instance maintainer.



If you want to use template parameters, you must first define them in the **Template Parameters** dialog box as described below. After you defined a template parameter, you can assign it to specific parameters of the same type in the block parameter editor (see also [Editing the parameters of a block](#)).

INFO

It is also possible to define template parameters directly in the block parameter editor.

Models with no template parameters can be directly activated in the model manager, with a single instance of the model running. This is different when you save the model after you have assigned at least one template parameter to one or more block parameters, you can no longer activate the model directly in the model manager. Instead, you must create at least one instance of the model, and you then activate that instance using the instance editor. See [Using the instance editor](#) for detailed information.

To define the template parameters for the instances of the current model

1. In the toolbar of the model editor, click the template parameters icon  to invoke the **Template Parameters** dialog box. When at least one template parameter has been defined, a checkmark is shown on the above icon: .

This dialog box is initially empty and you must create the template parameters that you want to use in your model. When template parameters have already been defined, they are all shown in this dialog box.

If you have a long list of template parameters, you can easily locate the template parameter that you are looking for by entering its name or part of the name in the search box. When search criteria are currently applied, an X is shown in the search box; click this to clear the search and thus to show all available template parameters.

After a template parameter has been assigned to one or more block parameters, the **Usage Count** column indicates the number of places in which the template parameter is used. For example, when a template parameter is used in two places, this means that it has been assigned to two block parameters. These can be within the same block or in different blocks.

2. To create a template parameter, click **Create new template parameter**. This adds an empty row at the bottom of the dialog box. You can click **Create new template parameter** several times to add several empty rows that you can fill one after the other.
3. Specify the following information for each template parameter:

- **Name.** Type a unique name to identify the template parameter within the current model. This name can later be selected in the block parameter editor.
- **Type.** Select the value type of the template parameter from the drop-down list box. The type can be, for example, a string or float, a source or destination, or the parameter of a specific block.

Multi-line String is a special type which is used, for example, with the **Text** parameter of the **Send Email** block. Long input is automatically wrapped in the **Default Value** text box and you can press Enter to start text on a new line.

An input block specifies a device or a range of devices, while an output block specifies a device, a trigger device or an asset. For template parameters, the same template parameter and thus value can be used for both input and output blocks. If a template parameter is set to refer to a range of devices, then using it in an output block will be treated as the trigger device. Typically, a single template parameter would be used for all input and output blocks, and may be a single device or a range of devices, in which case the block output goes to the device within the range that triggered a model evaluation (so a model calculating an average of a measurement and outputting to a measurement would generate a new measurement for each device independently). Even if a different template parameter whose value refers to a different range was used, the model output would only be sent to the device that triggered a model's evaluation.

- **Source or Destination Type.** You can select one or more of the following "Source or Destination" types from the **Restrict to** dropdown: "Device", "Groups", "Assets", or "Other". This selection restricts the template parameter to the selected types. This restriction is mandatory when "From Context" is selected from the **Value Selection** dropdown.

Template Parameters

+ New template parameter

Search

Name	Type	Value Selection	Default Value	Usage Count
Input Source	Source or Destination	From Context ?		2 ↕ ✖
Input Fragment and Series		Required ▼		1 ↕ ✖
Threshold Value		Required ▼		1 ↕ ✖

Device
Groups
Assets
Other

- **Value Selection** - Select one of the following options from the dropdown:

- **Required** - Default. Instance deployment requires a value for this template parameter.
- **Optional** - The template parameter can remain blank or can be set later by the instance maintainer. When you select this option, it is not possible to specify a default value.
- **From Context** - Applicable only for “Source or Destination” type. The input will be automatically selected based on the context where the instance is created. Example: Device context or group context.

i INFO

Only one template parameter per model can have a “From Context” value selection.

- **Default Value.** You can only specify a default value when the **Value Selection** option is set to “Required”.

Exception: Boolean types always have a value and cannot be optional. They are “false” by default (that is, the checkbox for the default value is not selected).

If you specify a default value, this default value will be provided in the instance editor when the instance maintainer creates a new instance. The instance maintainer can then either leave this default value unmodified or change it as required for that instance.

When setting the default value for a source or destination, an additional dialog box appears when you click the **Default Value** field. The dialog box is the same as when selecting a different input source or output destination in the block parameter editor (see [Editing the parameters of a block](#) for more information on this dialog box). Click the button which is shown when you hover over an entry to select the source or destination that you want to use.

You can also define your own selection lists for specific types of template parameters. See [Adding a selection list for a template parameter](#) for detailed information.

i INFO

If there is a block parameter for which a required value has not been specified, then the instance cannot be activated. Attempting to do so will report an error.


- You can update a template parameter at any time. This includes the name, whether it is optional or not, and the default value. All blocks in which the updated template parameter is defined are automatically adapted to use the new values. The only exception is the type. You can only change the type if the template parameter is not used in any block of the model.
- When the model is inactive, you can reorder the template parameters. This affects the sequence in which they are shown in the instance editor. Drag a row to a different position using the move icon ⬇ which is shown next to the row. See also [Filtering and sorting the instances](#).
- You can only delete a template parameter if it is not used in any block of the model. To delete a template parameter, click the delete icon ✖ which is shown next to the row.
- Click **OK** to store the changes in memory and to close the dialog box.

INFO







Keep in mind that your changes are only written to the inventory when you save the model. See also [Saving a model](#).

Adding a selection list for a template parameter

Several types that you can select in the **Template Parameters** dialog box (see [Managing template parameters](#)) allow you to select a predefined value. An example of such a type is the **Direction** parameter of the **Crossing Counter** block which offers predefined values such as **Upwards** and **Downwards** for selection. In addition to the predefined values, you can also add your own selection lists for types such as string, float, source or destination, or geofence, and you can also select a specific value to be the default value. The values that you define for a selection list are then available for selection when you create instances of the model.

When it is possible to add a selection list, a selection list icon  is shown in the **Template Parameters** dialog box, next to the text box in which you can add a default value. When a selection list is already available, a drop-down arrow is shown instead of the above icon.

To add a selection list for a template parameter

1. In the **Template Parameters** dialog box, click the selection list icon  in the **Default Value** column for a defined template parameter.
2. In the resulting **Selection List** dialog box, click **Add selection**.
3. Specify a unique name and a value for the new selection. The value that you can specify depends on the type of the currently selected template parameter. For example, if **float** is selected as the type, you can only specify a float value. Or if **Source or Destination** is selected as the type, you can select an input source or output destination from a dialog box.
4. Add at least one more selection, that is, click **Add selection** once more and specify a unique name and a value for the new selection.
5. Repeat the above step until all required selections have been added.
6. To reorder the selection list, drag a row to a different position using the move icon  which is shown next to the row.
7. To delete a selection, click the delete icon  which is shown next to the row.
8. Click **OK** to save the selection list and to close the **Selection List** dialog box. In the **Template Parameters** dialog box, you can now select one of your selections as the default value from a drop-down list box.
9. To clear the selected default value in the **Template Parameters** dialog box, click the clear icon  that is shown next to the selection.
10. To edit your selection list, click the actions menu  in the **Default Value** column of the **Template Parameters** dialog box and then click **Edit**. The **Selection List** dialog box is then shown.
11. To permanently delete your selection list, click the actions menu  in the **Default Value** column of the **Template Parameters** dialog box and then click **Delete**. In the resulting dialog box, click **OK**.

Copying items to a different model

You can copy any items on the canvas (blocks, groups, and attached wires) and paste them in a different model. The prerequisite for this is that all is done in the same session. It will not work if you try to paste the items in a different tab or in a different browser.

CAUTION

There may be performance issues if you copy many input blocks and output blocks. This is because this operation requires access to the inventory service of Cumulocity to get the information about the devices that are represented by these blocks.

To copy items to a different model

1. On the canvas of the model editor, select all items that you want to copy and press Ctrl+C. This also works if the model is currently in read-only mode.
2. Leave the model editor. See also [Leaving the model editor](#).
3. In the model manager, switch to the model into which you want to paste the copied items. This can be an existing model (see also [Editing an existing model](#)) or a new model that you must first create (see also [Adding a new model](#)).

4. When the model editor is shown, press Ctrl+V to paste the copied items into the model.

WORKING WITH GROUPS

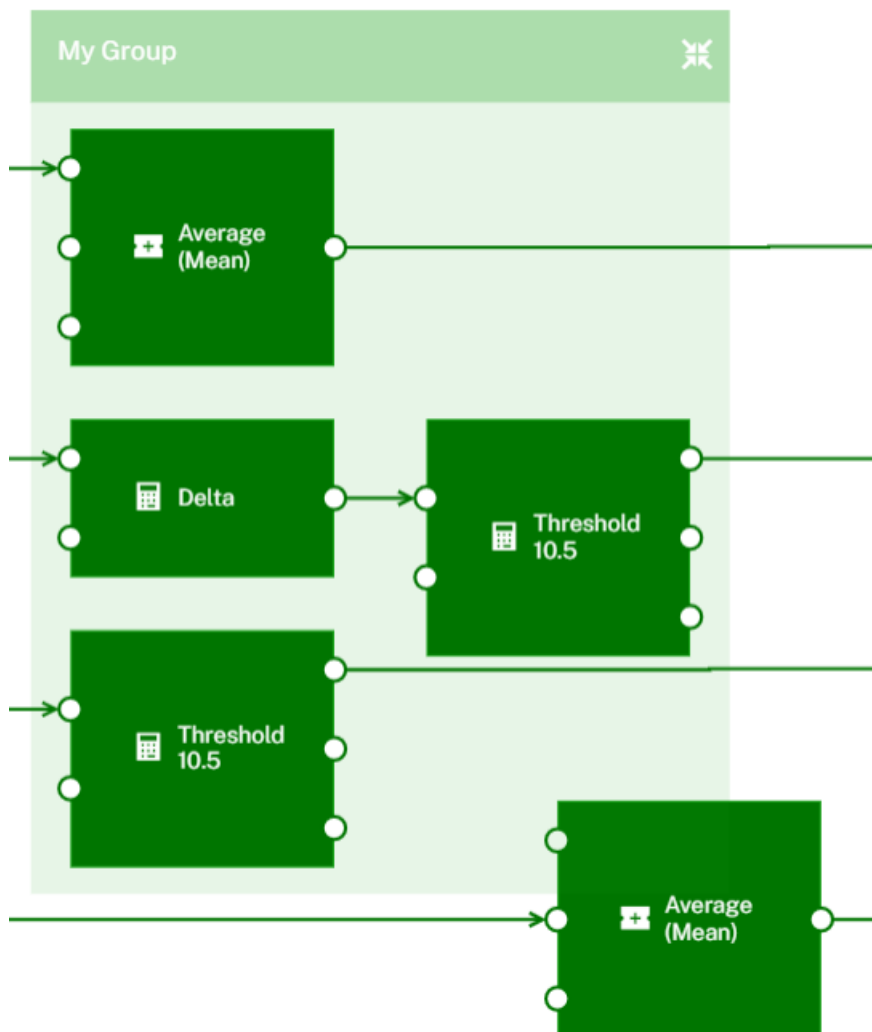
What is a group

You can arrange blocks and their attached wires in a group. A group is a special type of block which can be collapsed and expanded. When a group is expanded, you can change its contents in the same way as you would on the canvas, for example, you can add wires or edit the block parameters. You can also add more blocks to the group or remove blocks from the group. When a group is collapsed, it occupies less space on the canvas, however, the blocks and wiring within the group are not visible in this case.

Groups are helpful if commonly required functionality needs to be made available in multiple places. You can give each group a name by which it can be identified. You can copy a group and paste it in either the same model or in a different model.

INFO

Do not confuse this type of group with a group (or range) of devices. Input sources may sometimes be a group of devices, which is different. See also [Editing the parameters of a block](#).



The size of the box that is shown for a group is determined by its contents. If you move a block within the group to a different position, the box size is automatically adapted (that is, the box is made larger or smaller). The same applies if you change the path that a wire takes to

another block within the same group.

You move a group on the canvas in the same way as you move a block (see also [Moving a block](#)). When you move a group, the group is always shown on top of all other items on the canvas. As the group box is transparent, you can easily see which blocks belong to the group and which are just overlayed by the box.

It is not possible to nest groups.

INFO


There is one exception when managing the contents of a group: When you duplicate one or more blocks that are contained in a group using Ctrl+C and Ctrl+V or if you use the **Duplicate** command in the block parameter editor, the duplicate is not added to the group. It is added to the canvas instead. However, when you press Ctrl and then drag the blocks to be duplicated, you can place the duplicate either within the group (this can be the same or a different group) or on the canvas. See also [Duplicating a block](#).

Adding a group

You can add any blocks that are currently shown on the canvas (including the wires between the blocks) to a group.

It is not possible to create an empty group. You must first add a group as described below. Once the group exists, you can add more blocks to the group, either from the palette or from the canvas, as described in [Adding a block](#) and [Moving blocks into a group](#).

To add a group

1. On the canvas of the model editor, select one or more blocks that you want to add to a group. You need not select wires; all existing wires are retained. See also [Selecting blocks and wires](#).
2. In the toolbar of the model editor, click the group icon . Or press Ctrl+G.

Collapsing and expanding a group

If you need more space on the canvas and do not need the group contents to be visible, you can collapse the group.



When a group is collapsed, a number is shown on the collapsed group indicating the number of blocks in that group. For example:



If you want to make the group contents visible again (for example, to edit block parameters or to add wires), you must expand the group.

When you save the model, the state of each group (that is, whether it is currently collapsed or expanded) is stored. The next time you edit the model, its contents will be shown as after the last save.

To collapse or expand a group

- To collapse a group, click the collapse icon  which is shown next to the group name.
- To expand a group, click the expand icon  which is shown above the top right of the collapsed group.

Renaming a group

When you add a group, its default name is "Group". You can rename each group and give it a unique name.

If a group name is longer than can be shown in the group label, move the mouse pointer over the group name to view the entire name in a tooltip.

It is not possible to have groups without names. If you delete a group name, the previous name is automatically used again.

To rename a group

1. In the model editor, select the group and then click on the group name. You can either do this when the group is collapsed or expanded (see also [Collapsing and expanding a group](#)). This selects the entire name for editing.
2. Specify a new group name and press Enter.


Moving blocks into a group

You can move one or more blocks from the canvas into an existing group. All existing wires are retained.

You can also drag a block from the palette into an existing group. See [Adding a block](#).

You can only move/drag blocks into a group when its contents are visible, that is, when the group is currently expanded. See also [Collapsing and expanding a group](#).

To move blocks into a group


1. Make sure that the group into which you want to move the blocks is not collapsed.
2. On the canvas of the model editor, select the blocks that you want to move into the group (see also [Selecting blocks and wires](#)). You need not select the wires between the blocks; they are automatically moved together with the blocks.
3. Do one of the following:
 - Drag the selection into the group and drop it there.
 - Or select the group into which you want to move the blocks. Then click the group icon  in the toolbar of the model editor, or press Ctrl+G.

Moving blocks from a group to the canvas

You can move a block from a group to the canvas. All existing wires are retained.

When the last item of a group has been moved to the canvas, the group is automatically removed. If you want to move all items to the canvas at the same time, you can simply ungroup the entire group. See [Ungrouping a group](#).

To move blocks from a group to the canvas

- To move one or more blocks at the same time:
 1. In the expanded group, select the blocks that you want to move.
 2. In the toolbar of the model editor, click the ungroup icon . Or press Ctrl+Shift+G.
- Or to move a single block:
 1. In the expanded group, select the block that you want to move.
 2. Click the **Ungroup** command which is then shown at the bottom of the block parameter editor.

Removing blocks and wires from a group

You remove blocks wires from a group in the same way as removing them directly on the canvas. The only prerequisite is that the group is currently expanded. See [Removing a block or wire](#).

If the last item in a group is removed, the group is automatically removed.

Duplicating a group

You can duplicate each group that is currently shown on the canvas. The original group and its duplicate will then both have the same contents.

Wires coming in from blocks outside of the group or going from the group to blocks outside of the group are not duplicated.

You can also copy a group into a different model, see [Copying items to a different model](#).

To duplicate a group

On the canvas of the model editor, click the group that you want to duplicate (it does not matter whether the group is currently collapsed or expanded) and then do one of the following:


- Press Ctrl+C to copy the group, and then press Ctrl+V to paste the group.
- Or press Ctrl and drag the group to be duplicated to the position at which you want to place the duplicate.

Ungrouping a group

When you ungroup a group, the group is removed and all the blocks from that group are shown directly on the canvas. All attached wires are retained.

You can ungroup several groups at the same time. In this case, it is important that no block or wire is selected either within or without the selected groups, otherwise ungrouping is not possible.

To ungroup a group

1. On the canvas of the model editor, select one or more groups that you want to ungroup. It does not matter whether a group is currently collapsed or expanded.
2. In the toolbar of the model editor, click the ungroup icon . Or press Ctrl+Shift+G.

Removing a group

You can remove each group that is currently shown on the canvas. Wires to blocks outside of the group are removed.

CAUTION

When you remove a group, all blocks and wires within this group are removed from the model.

To remove a group

On the canvas of the model editor, click the group that you want to remove (it does not matter whether it is currently collapsed or expanded) and press Del.

MANAGING THE CANVAS

Navigating large models

If your model is too large to fit onto the visible part of the canvas, you can use the mouse to drag the parts of the model into view that are currently outside of the window. You can do this either directly on the canvas or in the overview area. The overview area always shows the entire model. If the overview area is currently not shown, see [Showing and hiding the overview](#).

To navigate in a large model

1. In the model editor, position the mouse over a free spot of the canvas (which does not contain a block or wire) or anywhere over the overview area.
2. Click and hold down the mouse button, and immediately drag the mouse into the desired direction. Release the mouse button when the required area is visible on the canvas.


INFO


When you hold down the mouse button for a longer time over a free spot of the canvas, the mouse pointer changes and you can select an area instead (for example, several blocks and attached wires). See also [Selecting blocks and wires](#).

Showing and hiding the overview

The overview area, which shows the entire model, is shown at bottom right of the canvas. If you do not need the overview, you can hide it.

To show or hide the overview

To hide the overview, click the hide icon  which is shown directly above the overview area.

To show the overview, click the show icon  at the bottom right of the canvas.

Zooming the canvas


The toolbar of the model editor indicates the current zoom percentage for the canvas. The zoom icons in the toolbar allow you to


- zoom out, which makes everything on the canvas smaller so that more items can be shown, and to
- zoom in, which makes everything on the canvas larger, but less items can then be shown.

INFO

When you use the key combinations mentioned below, the currently selected area defines what is to be zoomed. When the canvas has the focus (for example, when you have just selected a block or wire), only the content of the canvas is zoomed. When the documentation pane or the palette currently has the focus, the browser's zoom functionality is used and all of the browser content is zoomed (and the zoom percentage in the toolbar remains unchanged).

To zoom the canvas

To zoom out, click the zoom out icon  in the toolbar of the model editor. Alternatively, press Ctrl and the minus key.


To zoom in, click the zoom in icon  in the toolbar of the model editor. Alternatively, press Ctrl and the plus key.


Showing and hiding the grid

The blocks, wires and groups on the canvas always snap to a grid. You can decide whether the grid is to be shown or not. The grid is not shown by default. When you zoom the canvas, the grid is zoomed accordingly.

To show or hide the grid

In the model editor, click the toolbar icon for toggling the display of the grid.

When the grid is hidden, the icon looks as follows: .

When the grid is shown, the icon looks as follows: .

When the model is active (read-only mode), it is not possible to toggle the display of the grid and this icon is therefore not shown.

USING THE INSTANCE EDITOR

THE INSTANCE EDITOR USER INTERFACE

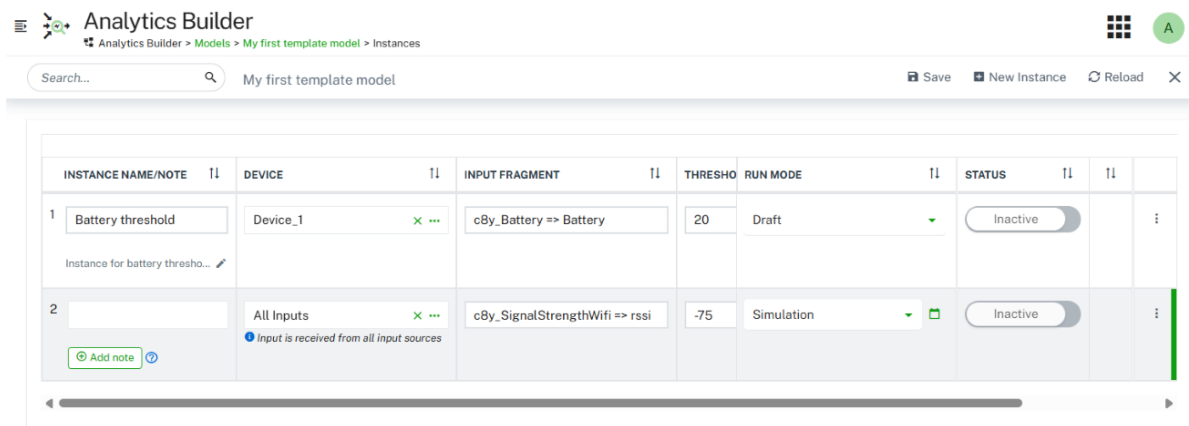
A prerequisite for invoking the instance editor is that one or more template parameters have been defined in the model editor (see also [Managing template parameters](#)).

The instance editor allows you to set up different instances of the same model. The blocks in each instance can then use different values for the template parameters, and the instances can be activated independently from the other instances. You invoke the instance editor from the model manager. See also [Editing the instances of a model](#).

INFO

To edit the instances, you must have ADMIN permission for “CEP management”. If you have READ permission, you will only be able to view the instances.

The instance editor shows the instances for a selected model. If there are any instances, a table shows the values of the instances, the mode and whether the instance is active.




The screenshot shows the Analytics Builder interface. At the top, there's a header with the Analytics Builder logo and a breadcrumb trail: Analytics Builder > Models > My first template model > Instances. Below the header is a search bar and a toolbar with buttons for Save, New Instance, Reload, and a close icon. The main area contains a table with the following columns: INSTANCE NAME/NOTE, DEVICE, INPUT FRAGMENT, THRESHOLD, RUN MODE, STATUS, and an actions menu (represented by a vertical ellipsis). The table has two rows. The first row is for an instance named 'Battery threshold' with device 'Device_1', input fragment 'cBy_Battery ==> Battery', threshold '20', run mode 'Draft', and status 'Inactive'. The second row is for an instance named 'All Inputs' with device 'All Inputs', input fragment 'cBy_SignalStrengthWifi ==> rssi', threshold '-75', run mode 'Simulation', and status 'Inactive'. A horizontal scrollbar is visible at the bottom of the table.

INSTANCE NAME/NOTE	DEVICE	INPUT FRAGMENT	THRESHOLD	RUN MODE	STATUS	
1 Battery threshold	Device_1	cBy_Battery ==> Battery	20	Draft	Inactive	
2	All Inputs	cBy_SignalStrengthWifi ==> rssi	-75	Simulation	Inactive	


A row is shown for each instance. A column is provided for each template parameter that is defined in the template model, with the name of the template parameter being the column header. When an instance is not active, you can adjust the values for that instance.

A horizontal scrollbar is available if not all template parameters (columns) can be shown on the screen.

The right side of the table shows the mode and status of each instance. You activate (deploy) and deactivate (undeploy) the instances from here. See also [Deploying an instance](#) and [Undeploying an instance](#).

Each row that is shown for an instance has an actions menu  at the very right which contains commands for managing the instance (for example, to delete the instance).

You can control the list of instances by filtering and sorting. See [Filtering and sorting the instances](#) for more information.

If the error icon  is shown near the end of a row, the corresponding instance is no longer processing events. Click that icon to get more information.

When you open the instance editor, it may happen that template parameters have been changed since you last edited the instances and that they no longer use the same values types as before. If the values specified in the instance editor are still compatible, they are converted to the new value types. Incompatible values (including checkboxes for boolean types and values that are shown in drop-down list boxes) are automatically removed. Each field from which the value has been removed shows an error underline and a corresponding error message.

ADDING AN INSTANCE

When you add a new instance, a new row is added to the instance editor table. You can then either immediately fill in the required values, or you can first add all required rows and then fill the rows one after the other.

To add an instance

1. In the toolbar of the instance editor, click **New Instance**. This adds a new row at the bottom of the table. New instances (rows) are shown with a background color until they have been saved.
2. Fill in the instance name and template parameter values, as defined by the model. See also [Editing an instance](#).

EDITING AN INSTANCE

You provide the parameter values for instances in the same way as you provide values for blocks in the model editor (see also [Editing the parameters of a block](#)).

The instance editor table provides different types of input controls, depending on the type of template parameter:

- Text boxes are provided in which you can enter values, depending on the setting of the template parameter (for example, a string or a float or a multi-line string). Your input is validated as you type. For example, it is not possible to enter a string value in a text box that expects a float value.
- Checkboxes are provided for boolean values. Selecting a checkbox corresponds to setting the value to `true`.
- Drop-down list boxes are provided when you can select a different value (for example, to select a different rule for rounding).
- When you edit an input source or output destination, an additional dialog box appears. The dialog box is the same as when selecting a different input source or output destination in the block parameter editor (see [Editing the parameters of a block](#) for more information on this dialog box). Click the button which is shown when you hover over an entry to select the input source or output destination that you want to use.

Instances (rows) that have been edited but have not yet been saved are shown with a background color until they have been saved.

When a text box requires a value that has not yet been specified, a message is shown, indicating that this is a required field. It is possible to save the instances and leave the instance editor, and set all of the required values at a later point in time. As long as the missing fields of an instance have not been specified, it is not possible to activate that instance.

DEPLOYING AN INSTANCE



You can activate (that is, deploy) each instance separately. For example, one instance can be in production mode and another in test mode. See [Deploying a model](#) for more information on the different modes; that information applies to both regular models and template models.

When you activate an instance, all changes for that instance are first saved and the instance is then activated.

When an instance is activated, the template parameter values, where supplied, are taken and applied to those block parameters in the model which use a template parameter binding. If no template parameter value is provided, then a default value for that template parameter is used, if there is one. If no template parameter value is supplied in the case of a required template parameter, then the instance will fail to activate.

Once an instance is active, you cannot modify the template parameter values or mode without deactivating the instance first. If any instances are active, then the model is read-only and cannot be modified until all instances are deactivated.

To deploy an instance

1. In the **Run Mode** column of the instance editor, click the drop-down menu for the instance that you want to deploy and select one of **Production**, **Test** or **Simulation**. You cannot activate instances that are in draft mode.
2. If you have selected simulation mode, click the calendar icon  which is now shown, specify the time span that is to be used, and click **Apply**. See also [Simulation parameters](#).
3. When the toggle button in the **Status** column currently shows **Inactive**, click this button to change the state to **Active**. For simulation mode, you can only set the state to **Active** when a valid time range has been defined. In the case of an error, the error icon  is shown at the right of the table and the instance cannot be activated. Click the error icon to get more information.

UNDEPLOYING AN INSTANCE

You can deactivate (that is, undeploy) each instance that is currently in production, test or simulation mode and for which the toggle button in the **Status** column of the instance editor shows **Active**.


When you undeploy an instance, the instance is stopped and no longer processes incoming data. Any state built up in the instance is lost. For simulation mode, this means that the instance is stopped before all historical data from the specified time range has been replayed.

To undeploy an instance

In the **Status** column of the instance editor, click the toggle button for the instance that you want to undeploy so that **Inactive** is then shown on the button.

FILTERING AND SORTING THE INSTANCES

If you have a long list of instances, you can easily locate the instances that you are looking for by entering a value in the search box. Or you can enter part of the value. This searches all input fields in the instance editor and only lists the instances (rows) that contain this value. All values that match the filter are highlighted. The search criteria are not case-sensitive. When search criteria are currently applied, an X is shown in the search box; click this to clear the search and thus to show all available instances.

You can also sort the columns of the instance editor table. To do so, click one of the sort icons  that are shown in the column header. This sorts the instances according to the values in that column (for example, alphabetically or by number). Clicking again sorts the column in the opposite direction. Fields with required values in that column that have not yet been specified can thus be shown either at the very top or bottom of the column. You can also sort the instances alphabetically according to run mode and status, for example, to show the active instances at the top.

Editing a value will not affect the display of rows in the instance editor table. If you want to reapply the search and sorting, you must save and reload the instances.

Adding a new instance will not affect the display of rows in the instance editor table. If you add a row after sorting, the row is always added at the bottom of the table, unless you reload the instances.

INFO

You can also reorder the template parameters in the **Template Parameters** dialog box (see [Managing template parameters](#)). This affects the sequence in which they are shown in the instance editor.

DUPLICATING AN INSTANCE

You can duplicate each instance (row) that is currently listed in the instance editor. The original instance and its duplicate will then both have the same template parameter values and the same mode. However, the duplicated instance is always inactive even if the original instance is active.

To duplicate an instance


In the instance editor, click the actions menu  of the instance that you want to duplicate and then click **Duplicate**.

A new row for the duplicated instance is immediately shown at the bottom of the instance editor table.

DELETING AN INSTANCE

You can delete each instance that is currently listed in the instance editor. When you delete an instance that is currently deployed, it is first undeployed and then deleted.

To delete an instance

1. In the instance editor, click the actions menu  of the instance that you want to delete and then click **Delete**.
2. In the resulting dialog box, click **Delete** to confirm the deletion.

SAVING THE INSTANCES

You can save the instances even if there are still rows in which required information must be specified. This is helpful if you want to add that information at a later point in time.

INFO

When you activate an instance, all of your recent changes are automatically saved. See also [Deploying an instance](#).

To save the instances

In the toolbar of the instance editor, click **Save**.

This command is only enabled when changes have been applied to the instances. It saves only those instances where the rows that are highlighted with a background color.

RELOADING THE INSTANCES

You can refresh the display to show the latest state of all instances, or to see whether deployed instances have entered a failed state.

To reload the instances


In the toolbar of the instance editor, click **Reload**.

If there are unsaved changes when reloading, you are prompted to save these changes first.

LEAVING THE INSTANCE EDITOR

When you leave the instance editor as described below, you are returned to the model manager.

To leave the instance editor

In the toolbar of the instance editor, click the close icon .

If there are unsaved changes when leaving, you are prompted to save these changes first.

WIRES AND BLOCKS

VALUES SENT ON A WIRE

Blocks within a model are connected from block outputs to block inputs with wires.

INFO

These block outputs and inputs are also called output ports and input ports. See also [Adding a wire between two blocks](#).

Wires allow blocks to pass signals and values between blocks. The value sent on a wire is one of the following types, according to the block output from which it is connected:

Type	Description
<code>boolean</code>	A true or false value. A boolean value stays true or false until changed.
<code>float</code>	A numeric value, which can be fractional (and processed using fixed precision). A float value maintains its current value until changed.
<code>string</code>	A textual value. A string value maintains its current value until changed.
<code>pulse</code>	A signal of a point in time. Pulses are only active momentarily. Unlike the above types, they only represent a single instance in time. See also The pulse type .

Type	Description
<code>any</code>	A value that may be any of the above types. See also The any type .

The type of a wire depends on the output to which it is connected. This can be viewed in the block reference. Similarly, the type (or supported types) of a block's input can be viewed in the block reference.

Value types

The following types are referred to as value types:

- `boolean`
- `string`
- `float`
- `any` when used to hold a `boolean`, `string` or `float` value

Value types are useful for modeling measurements such as sensor values, which may be read intermittently, or sampled. In between readings, the physical property being measured (such as temperature) will still have some value, as it is a continuous property. For practical reasons, a sensor may not give a continuous stream of output but instead a periodic sampling, or provide new readings only if the value being measured has changed (within whatever measurement resolution the sensor provides). Between sample points, blocks will use the most recent value, as that is the most up to date value being provided. In general, blocks assume that a value stays at whatever the most recent reading of that value is until a new value is received.

For example, consider a pair of temperature sensors. One provides a reading every 10 seconds regardless, while another only provides a new reading if the value has changed by 0.5 degrees. If we connect these to a **Difference** block, then we may have inputs as shown in the following table, with the corresponding result from the **Difference** block's **Absolute Difference** output:

Time	Sensor 1 (reads every 10s)	Sensor 2 (output if changed by 0.5)	Difference block: Absolute Difference output
10:00:00	20.0		
10:00:03		22.0	2
10:00:10	20.0		2
10:00:20	20.0		2
10:00:23		22.5	2.5
10:00:28		23.0	3
10:00:30	21.1		1.9
10:00:35		23.5	2.4
10:00:40	22.8	24.0	1.2

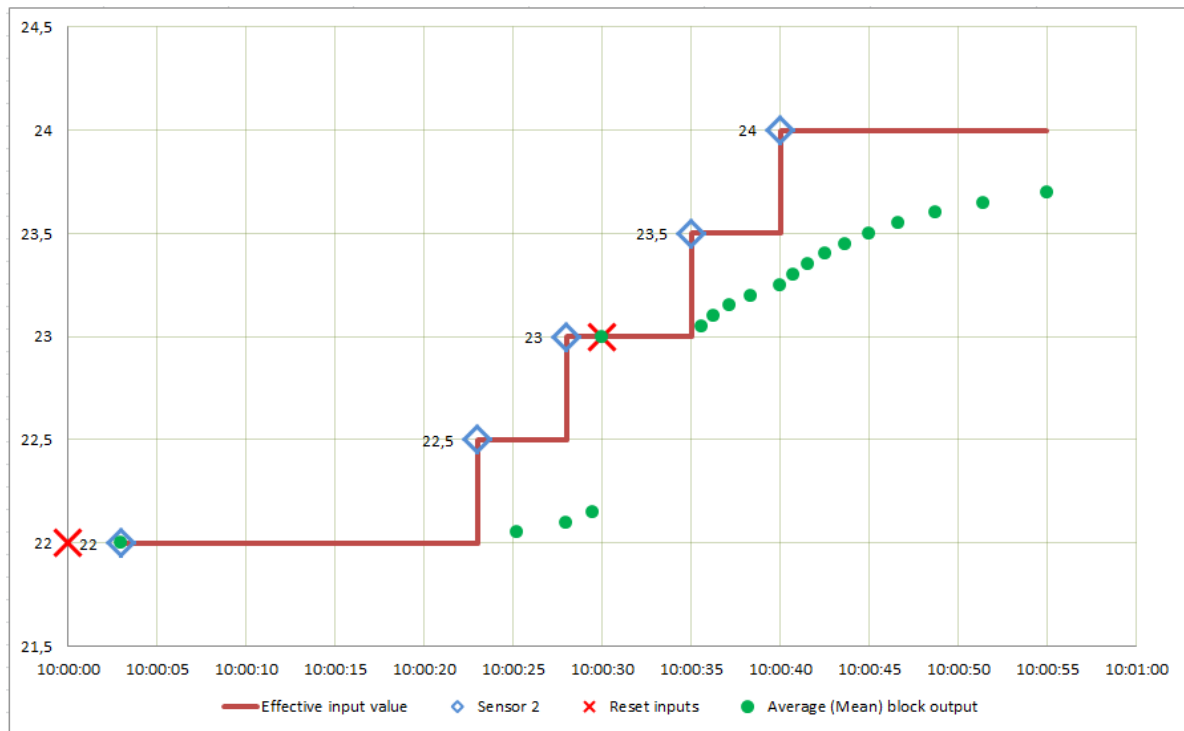
Note that two inputs (to different input ports of the block) to the same block with the same timestamp only generate a single output. For each wire within a model (and each input block), there can only be a single value for a given point in time. An input block cannot generate more than one output for the same timestamp. If it receives multiple events at the same time, then it is undefined which of the events is picked.

In general, blocks will not consider there to be any significance to a wire receiving the same `boolean`, `float` or `string` value as before. Most blocks will not change behavior. This is true for any arithmetic blocks, such as the **Difference** block in the example above: the output is still 2 on the repeated readings from sensor 1. There are some exceptions, such as the **Missing Data** block when the **Ignore Repeated Inputs** checkbox is not selected (`false`).

If a single block has a numeric value input and pulse signals such as reset, the absence of a new value when a pulse signal occurs means that the value is treated as having the same value still. Thus, when an **Average (Mean)** block is reset, its output will be equal to the most recently received input (assuming it has received an input since the model has started). In the example below, the **Average (Mean)** block's duration has not been set, while the output threshold is set to 0.05; this means the block will generate new output even if there is no new input (see [Common block inputs and parameters](#)).

Time	Reset signal	Sensor 2	Average (Mean) block output	Notes
10:00:00	Reset			No output. There has been no input value yet.
10:00:03		22.0	22.00	With no history, the output value is the input value.
10:00:23		22.5		All of the values up to this point have been 22, so the average value is still 22 (thus, no new output is generated).
10:00:25.22			22.05	Average of 20 seconds at value 22 and 2.22 seconds at value 22.5.
10:00:28		23.0	22.10	Average of 20 seconds at value 22 and 5 seconds at value 22.5.
10:00:30	Reset		23.00	The input is still 23 (we just have not received a new event), and reset only discards the history. With no history, the output value is the input value.
10:00:35		23.5		
10:00:35.56			23.05	Average at various points in time, when the output changes by 0.05.
10:00:36.25			23.10	
10:00:37.14			23.15	
10:00:38.33			23.20	
10:00:40		24.0	23.25	Average of 5 seconds at value 23 (from reset at :30 to :35) and 5 seconds at value 23.5 (from :35 to :40).

The following graph illustrates the inputs to the **Average (Mean)** block and the output of this block:



Note how the effective input value is unchanged until a new measurement input occurs, and the **Average (Mean)** block operates on this effective value (the red line in the above graph). When reset, the block outputs the current effective input, which at the second reset at 10:00:30 is 23. Note that when the **Output Threshold** parameter is set, new outputs can be generated even if no new input occurs, and will asymptotically approach the last input value. Note that this behavior differs from Apama queries or stream queries.

If the **Average (Mean)** block was configured with a window of 10 seconds, then the window would apply as illustrated below:

Time	Reset signal	Sensor 2	Effective input value	Average (Mean) block output	Values in window history	Notes
10:00:00	Reset					
10:00:03		22	22	22.00		First value after start: the window is empty, so the Average (Mean) block uses the input value for the output.
10:00:23		22.5	22.5		22	
10:00:23 - 10:00:28			22.5	increasing from 22.00 to 22.20	22, 22.5	Proportion of window that is 22 or 22.5 changes over time, thus the output changes.
10:00:28		23	23	22.25	22, 22.5	
10:00:28 - 10:00:30			23	increasing from 22.25 to 22.40	22, 22.5, 23	

Time	Reset signal	Sensor 2	Effective input value	Average (Mean) block output	Values in window history	Notes
10:00:30	Reset		23	23.00		Window is reset and thus now empty; the current (effective) input is 23, so the Average (Mean) block uses that for the output.
10:00:35		23.5	23.5		23	
10:00:35 - 10:00:40			23.5	increasing from 23.00 to 23.20	23, 23.5	
10:00:40		24	24	23.25	23, 23.5	Window is now full (10 seconds since reset).
10:00:40 - 10:00:45			24	increasing from 23.25 to 23.75	23, 23.5, 24	
10:00:45			24	23.75	23.5, 24	Value 23 is now finally expired from the window (this was the effective input until 10:00:35, which is 10 seconds ago).
10:00:45 - 10:00:50			24	increasing from 23.75 to 24	23.5, 24	
10:00:50			24	24	24	Value 23.5 is now finally expired from the window (this was the effective input until 10:00:40, which is 10 seconds ago). The window now contains 10 seconds worth of measurements, all with value 24.

In the above, note how the current value only has any weighting in the window (that is, contributing to the output value) after the measurement is received. At the point the measurement is received, it has zero weighting compared to the previous history. As before, the sensor's value remains the effective input until it is replaced with a newer value (note that this is different to aggregates with time-based windows in Apache queries or stream queries). For example, the block has an effective input value of 23.5 from 10:00:35 to 10:00:40, and the value 23.5 is thus only finally expired from the window at 10:00:50, 10 seconds after it ceased to be the current effective input value, rather than 10 seconds after it first entered the window. Finally, note that when the window is empty, the effective input is used as the output instead, as the window is zero-length.

The pulse type

In contrast to value types, the `pulse` type represents a single point in time. For example, this may be a result of:

- a user pressing a momentary-action button,
- a state transition of a device,
- a sensor detecting a person walking through a door,
- a heartbeat event to denote a remote device is still alive, or
- a state transition of a block within a model.

Typically, blocks act upon every pulse sent to one of their inputs. Pulses are commonly used to trigger an output from a model using an output block, or used to reset the state of blocks within a model.

Pulses are active momentarily. In some regards, they are similar to a boolean value which is automatically reset to `false` after a model has processed a value.


Repeated pulses are typically significant, though they may not necessarily result in any change, depending on how they are being used. For example, repeatedly resetting an **Average (Mean)** block while its input value is unchanged will result in the output value remaining the same.













The any type

The `any` type is used on blocks which pass through a value of any type (for example, a **Time Delay** block or a **Gate** block).

Values of the `any` type can represent a value type or a `pulse` type.

TYPE CONVERSIONS

It is legal to connect a block output to a block input if they are the same type. Most other connections are also permissible, which result in the conversions as described in the table below. An  indicates that a connection is not legal; trying to deploy a model with such a wiring connection will fail.

		From block with output type				
		pulse	boolean	float	string	any
Connect to input of type	pulse		pulse occurs when output changes to true	pulse occurs when output changes value	pulse occurs when output changes value	pulse occurs when output changes value (excluding changes to false)
	boolean	true when the pulse has occurred, otherwise false		true if non-zero	true if not an empty string	true if value non-zero/empty
	float		0 for false, 1 for true			permitted if the value is of type float or boolean, other values fail at runtime
	string		"true" or "false"	number converted to a string (may be in scientific notation)		string value (may be in scientific notation)
	any					

Only conversions that will always succeed are allowed. String values are not converted to float values; while the input conversion may work sometimes, it cannot be guaranteed to always work.

In many cases, you need not worry about type conversions and where a wire makes sense. Any type conversion that is needed happens automatically.

Some blocks accept different types of inputs, and may change their output type or behavior depending on the input types. For example, the logical **OR** block can operate on either boolean or pulse inputs, and its output is the same as its input types.

In some cases, it is desirable to force a value to be interpreted as a specific type, in which case a converter block can be used to force a conversion to a specific type. For example, the **Pulse** block can convert boolean or float values to pulses, according to the conversions above. This means: for boolean, generate a pulse when the boolean value changes to true; for float, generate a pulse when the value changes. Thus, connecting two float outputs to an **OR** block directly will generate a boolean output which is true when either of the float outputs is non-zero. Alternatively, connecting two float outputs each to a **Pulse** block and from them to the inputs of an **OR** block, will send a pulse whenever either float output changes value. This is the default behavior of the **Pulse** block.

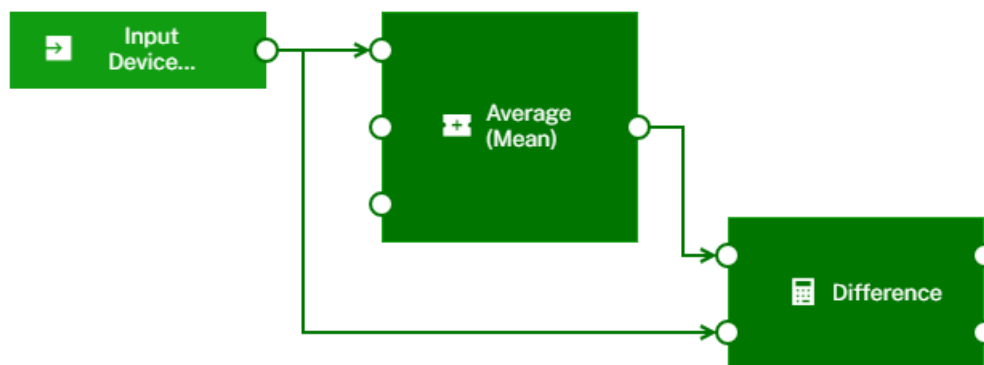
Different types of pulse conversions are possible with the **Pulse** block, depending on the setting of its **Mode** parameter. The conversions in the different modes are described in the table below:

		From block with output type				
		pulse	boolean	float	string	any
Connect to Pulse block in mode	On value change (default)	✓	pulse occurs when output changes to true	pulse occurs when output changes value	pulse occurs when output changes value	pulse occurs when output changes value (excluding changes to false)
	On every input	✓	pulse occurs on every input	pulse occurs on every input	pulse occurs on every input	pulse occurs on every input
	On non-zero values	✓	pulse occurs on every true input	pulse occurs on every non-zero input	pulse occurs on every non-empty input	pulse occurs depending on value's type as described in cells to the left

PROCESSING ORDER OF WIRES

Where a block has multiple inputs connected, all of these inputs are calculated before the block performs any calculations based on the inputs. It may be that the inputs for a block occur out of step with each other (such as in the example for two temperature sensors in [Value types](#)), in which case a block uses the latest value for value type inputs.

Where a single value is sent on two or more paths which both lead to the same block, the block performs calculations based on the latest value for both paths. This ensures consistent behavior when multiple paths to a single block exist. For example:



When the device measurement is received, the **Average (Mean)** block calculation is completed to generate an average before the **Difference** block computes the difference between the value and its average.

WIRE RESTRICTIONS

While a block's output can be connected to multiple other blocks, a block's input can only have a single connection.

It is also legal to leave a block's input or output unconnected if that is not required (the **Average (Mean)** block in the example that is given in [Processing order of wires](#) does not have anything connected to its **Sample** or **Reset** inputs).

Wires cannot create cycles. This means, the output of a block cannot be connected to

- the input of the same block, or to
- the input of any block that is connected directly or indirectly to one of the source block's inputs.

For example, there are three blocks: Block1, Block2 and Block3. A model would contain a cycle in the following cases:

- The output of Block1 is connected to the input of Block2, and the output of Block2 is connected to the input of Block1.

- The output of Block1 is connected to the input of Block2, the output of Block2 is connected to the input of Block3, and the output of Block3 is connected back to the input of Block1.

There are many possible connections which may lead to cycles in the model. The model editor, however, prevents you from creating cycles.

BLOCK INPUTS AND OUTPUTS

Many blocks have inputs or outputs that do not have to be used.

Some blocks generate several different outputs, and a model may only require some of the outputs available.

Some blocks have inputs, especially inputs of the **pulse** type, which do not have to be used. Leaving these not connected to anything is fine, and the operation associated with those inputs (such as **Reset**, see [Common block inputs and parameters](#)) will never be triggered.

Blocks can, when needed, detect which inputs are connected. For example, the **AND** block has five inputs, but it only requires the inputs that are connected to be **true** to generate a **true** output.

COMMON BLOCK INPUTS AND PARAMETERS

The inputs listed below are the names of common input ports that are shown on the left side of a block.

- **Value** input

Most calculation blocks have one main input which is called **Value**. This is the value on which the block performs its main calculation.

- **Value 1** and **Value 2** inputs

Blocks may have a number of similar inputs, which may be labeled **Value 1**, **Value 2**, and so on. You can find such inputs with the **Difference** block (see also the example in [Value types](#)) or with the **AND** and **OR** logic blocks. Typically, there is nothing significant as to which input is used.

- **Reset** input

Blocks that maintain some internal state may also have a **Reset** input, which is typically a **pulse** type. This does not have to be connected, but can be used to explicitly control on which range of readings a block should perform a calculation. For example, a model that monitors vehicle journeys may reset on the engine starting, which signifies the beginning of a journey. See also [Value types](#) for an example that illustrates the **Reset** input.

- **Sample** input and **Output Threshold** parameter

Blocks typically re-calculate their output when a new input is received. Some blocks may also generate output at some point after receiving an input, either because of time delay parameters set (for example, with the **Missing Data** or **Time Delay** blocks), or because their output may change over time even if the input value is constant. For example, the **Integral** block with a positive input generates an ever-increasing output until its window is full (or indefinitely if no duration has been set, when the block is calculating the integral over an unbounded window).

As with real-world sensors, it is not practical to create a continuously changing output. As well as generating an output if their input value changes, such blocks may also have a **Sample** input which triggers the block to re-evaluate and generate a new output, even if the input has not received any new value and the output has not changed by a significant amount. This is useful if there is a specific point in time when the output of the block should be calculated, as its output is going to be used at a later point in the model.

Alternatively, such blocks may have an **Output Threshold** parameter, which is used to control how frequently the output is re-calculated. When set, the block determines when its output will change by the output threshold, and when that occurs, even if it is not as a result of any new input value, the block generates an output value.

The **Output Threshold** should be set taking into account what error margins will exist on the input value (real-world physical sensors have some limited precision and accuracy in the property they are measuring), and what precision is required in the output.

Take care to avoid **Output Threshold** values that are too large or too small. If the values are too large, the block does not generate a new output when needed (unless the **Sample** input is used). If the values are too small, the block limits how frequently it generates output. If you want to change the values, send a **POST** request to Cumulocity that changes the value for the `minimum_wait_time_secs` key. See [Configuration](#) for detailed information.

The scale of appropriate values varies depending on what the magnitude of the input value is. If **Output Threshold** is not set, then the block only generates new outputs if it receives an input (this may be appropriate if it is receiving frequent inputs on the value, or if the **Sample** input is being used).

- **Ignore Timestamp** parameter

For Cumulocity measurements, events and alarms, the block by default uses the source timestamp available on the input. The block reorders the input based on the timestamp (see also [Input blocks and event timing](#)), but drops events that are delayed by too much. If this behavior is not desirable (for example, if a device's clock is not well synchronized, or if data from a device may be delayed), then you can disable this behavior by selecting the **Ignore Timestamp** parameter. If this is selected, the timestamp of the data is ignored, and the model processes the input data as soon as it is received, regardless of what timestamp it has. This may give different results compared to the default behavior of using timestamps. The behavior which is most desirable will depend on the nature of the device and its connectivity to Cumulocity.

Note that when a model is running in simulation mode, the setting of the **Ignore Timestamp** parameter is ignored. The block will always use the source timestamp, so that when replaying simulation events, the data is guaranteed to be processed in order and this will yield more realistic results (and there is no record of when the data was received, only the source timestamp). See also [About simulation mode](#).

INPUT BLOCKS AND EVENT TIMING

Input blocks make data from external sources (such as Cumulocity measurements) available to the model. Many data sources have timestamps on each piece of data, which reports the time that a measurement or event actually occurred. There may be delays in transmitting the data to the Apama system for processing, leading to events being received by Apama out of order.

Data sources with timestamps, such as measurements, can be reordered. Operations, for example, do not have timestamps and are therefore processed as they are received, without reordering.

Analytics Builder delivers several input blocks which consume data sources with timestamps. These blocks provide an **Ignore Timestamp** parameter which allows you to disable data reordering and thus to process the inputs as they are received. See also [Common block inputs and parameters](#).

The following table lists the available input blocks and indicates whether they are able to reorder the input:

Input block	Reordering is possible
Alarm Input	Yes
Event Input	Yes
Managed Object Input	No
Measurement Input	Yes
Operation Input	No
Position Input	Yes

INFO

The **Position Input** block is a specialized **Event Input** block. You can also use the **Cron Timer** block to activate a model periodically. Unlike the above blocks, the **Cron Timer** block is not associated with a device and can be found in the **Utility** category of the palette.

For data sources that have timestamps associated with a piece of data, the input block can handle events received out of order. In order to do this, the input blocks hold all received events in a reorder buffer and delay processing them until a predefined delay time after their source timestamp. By delaying the processing of the event relative to the source timestamp, the input block allows events to be reordered. The key parameter to this process is the amount of time by which the events are delayed. To configure the time in seconds by which the input blocks delay inputs, send a **POST** request to Cumulocity that changes the value for the `timedelay_secs` key. See [Configuration](#) for detailed information.

The input blocks assume that while events may be delivered out of order, they are received by Apama within the defined time delay value. If an event is received after a delay of more than the defined number of seconds (that is, the difference between the timestamp in

the event and the time on the system running Apama), then it is dropped if an event for the same timestamp or a more recent timestamp has already been processed by the model. Thus, it is possible that an old event might be processed by one model but dropped by another model.

If the time delay value is set too low, then a small delay may result in Apama dropping an event, which can lead to erroneous results. The higher the time delay value is, the larger is the delay before an event is processed. Thus, it is important to pick a suitable value for the time delay to match the environment for events being delivered into Apama.

The correlator logs the number of dropped events periodically to the log file. See [Configuration](#) for configuring logging throttling and [Log files of the Apama-ctrl microservice](#).

OUTPUT BLOCKS AND EVENT TIMING

Output blocks make data (such as Cumulocity measurements or operations) from the model available to external systems (such as Cumulocity). Outputs blocks can either produce synchronous or asynchronous values.

The values from an output block which generates synchronous output (such as measurements) can also be consumed by another model in a time-synchronous manner and can be processed by the model with any other data from the same timestamp. See also [Connections between models](#).

The values from an output block which generates asynchronous output can also be consumed by another model, but only in a time-asynchronous manner when data is received back from the external system.

The following table lists the available output blocks and indicates whether the output is synchronous or asynchronous:

Output block	Type of output
Alarm Output	Synchronous
Event Output	Synchronous
Managed Object Output	Asynchronous
Measurement Output	Synchronous
Operation Output	Asynchronous

FRAGMENT PROPERTIES ON WIRES

Each wire has a primary value that is of the type of the wire: one of `float`, `boolean`, `string` or `pulse`.

In addition to this, some blocks may provide other fragments of information alongside the value. These are named properties on the value. They may be other pieces of information provided from an input block, such as the unit in which a measurement is measured, or some extra contextual information for a data source.

Most blocks only operate on the primary value from their input wires, but some blocks can make use of these fragment properties values and extract them into separate output ports (for an example, see the **Extract Property** block). This gives more flexibility in processing more complex data from external sources.

KEYS FOR IDENTIFYING A SERIES OF EVENTS

Input and output blocks identify a series of events by specifying a key for the series (or stream) of events. This series of events is used to identify correct events to deliver to an input block. The key is made up of multiple block parameters, and identifies that series of events distinct from other series of events through the same block type. For example:

- For `Measurement` object input and outputs, the key is the device, the fragment, and the series. The **Unit** parameter specified in an output block is not considered part of the key (it is for information only) and is not required to match the parameters of the **Measurement Input** block.
- For `Event` objects, the key is the device and the event type.
- For `Alarm` objects, the key is the device and the alarm type.

❗ IMPORTANT

In Analytics Builder, for synchronous output types such as measurements, events and alarms (see also [Output blocks and event timing](#)), it is allowed to have more than one output block that generates output with any given key. As there can be connections between the models, ambiguities may occur while processing events in the input blocks if there are multiple output blocks (in different models) generating the same output stream at the same point in time. When using output blocks, ensure that no two blocks generate output of the same stream type at the same time.

DETAILS OF VALUES AND BLOCKS

INTRODUCTION

Analytics Builder provides an environment for connecting blocks together to form models that can process and react to inputs. Analytics Builder uses a few types of values internally, and it is important to understand the differences between these. The following topics cover the distinctions between value types representing continuous-time and discrete-time values, and the `pulse` type. They also cover some of the details of block implementations around windowing and when blocks generate output.

Summary

In Analytics Builder, the value types `float`, `boolean`, and `string` are used to represent continuous-time values. They have the following properties, which you should consider when writing models or creating custom blocks:

- A value wire holds its value until there is new input.
- Repeated inputs of the same value should not affect the output (for most cases).
- There is no guarantee that a new input will occur within a defined time period.

Contrast these to the `pulse` type, which represents discrete events and has the following properties:

- A pulse represents a single point in time.
- Multiple inputs of a pulse have significance, even if there is no difference to any value associated with the pulse.
- If a block has multiple input ports for pulses, inputs occurring at different times to different ports will only be “seen” one at a time. An input port for a pulse on a block acts as if it is automatically “reset” after evaluation.

These properties and the rationale behind them are explored and explained in the following topics. These topics also explain how to handle cases that do not fit into these distinctions, such as discrete numeric measurements.

VALUES AS REPRESENTATIONS OF CONTINUOUS-TIME PHYSICAL QUANTITIES

A continuous-time value type, especially of the `float` (that is, numeric) type, is typically used to represent the measurement of some continuous physical quantity or property by a sensor. For example, a value may represent one of the following:

- The pressure in a pipe.
- The temperature measured by a thermometer.
- The rotational speed of an axle.
- The position of an object.

These are all continuous measurable properties which are analog in nature. There will be some degree of precision as to how accurately they can be measured in both time and value (and within physical limits). By time accuracy, we mean how frequently a measurement can be made and how precisely the time of the measurement is recorded. There may also be latency - a delay between a change in the actual property and when that can be measured. By value accuracy, we mean with what level of precision the value can be measured - typically at least 2 significant figures, and rarely more than 4 or 5 significant figures of precision can be distinguished. By continuous, we mean that it is valid to measure the property at any point in time.

Taking discrete measurements at different times rather than continuously may be referred to as “sampling” (see also [https://en.wikipedia.org/wiki/Sampling_\(signal_processing\)](https://en.wikipedia.org/wiki/Sampling_(signal_processing))), and limits as to the value precision may be referred to as “quantization error”

(see also [https://en.wikipedia.org/wiki/Quantization_\(signal_processing\)](https://en.wikipedia.org/wiki/Quantization_(signal_processing))). When measuring a continuous value, the rate at which measurements are obtained should not make significant differences to the output of a block or a model. More measurements may give a more accurate output, but should not make a gross change to calculations.

For example, a sensor measuring the rotational speed of an axle may be able to provide a new measurement every one tenth (0.1) of a second, and only measure in the range 0 to 10000 rpm to the nearest 50 rpm. A change of 10 or 20 rpm may not result in any change of measured value, as the change is less than the level of precision. Applying brakes to a rotating axle to stop it may not be detected immediately, but result in one reading of 1000 rpm, followed by a reading 0.1 seconds later at 0 rpm after the axle has stopped (while the axle would take a few tens of milliseconds to stop, slowing down over that time period).

A sensor may be connected in such a way that it provides a new measurement at a regular frequency (for example, audio sampling at 8,000 Hz or a camera taking video at 50 frames a second). This is a regular sampling input.

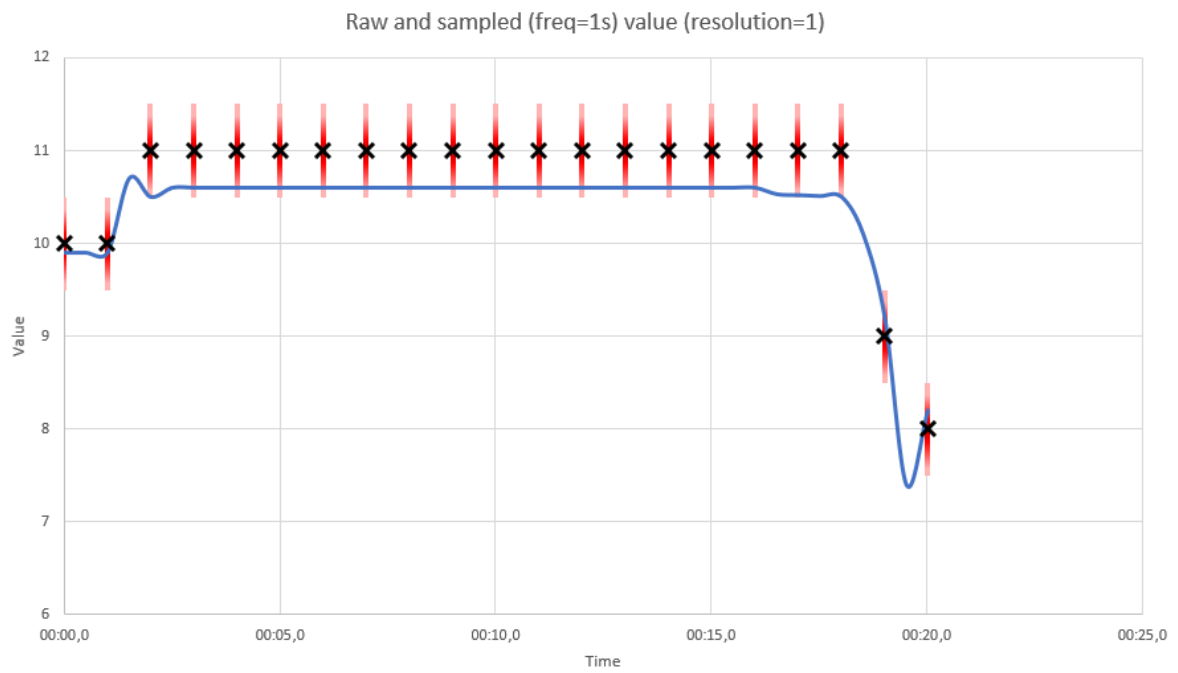
A simple and common optimization is that a sensor or device may generate a new measurement value only if the value is different to the previous value. For many sensors, it would be normal to measure something that often maintains a steady or constant value (at least to within the quantization limit), and there is little value in repeatedly sending the same value. This is an on-change input. There will still be an underlying sampling frequency, but new values are only transmitted from the sensor if they are different.

It is also possible to combine the regular sampling and on-change forms together: a sensor that generates a new input if the measurement is different, or periodically. This is a hybrid input. For example, the rotational sensor described above may only send a value if the rotation speed changes, or every 10 seconds regardless.

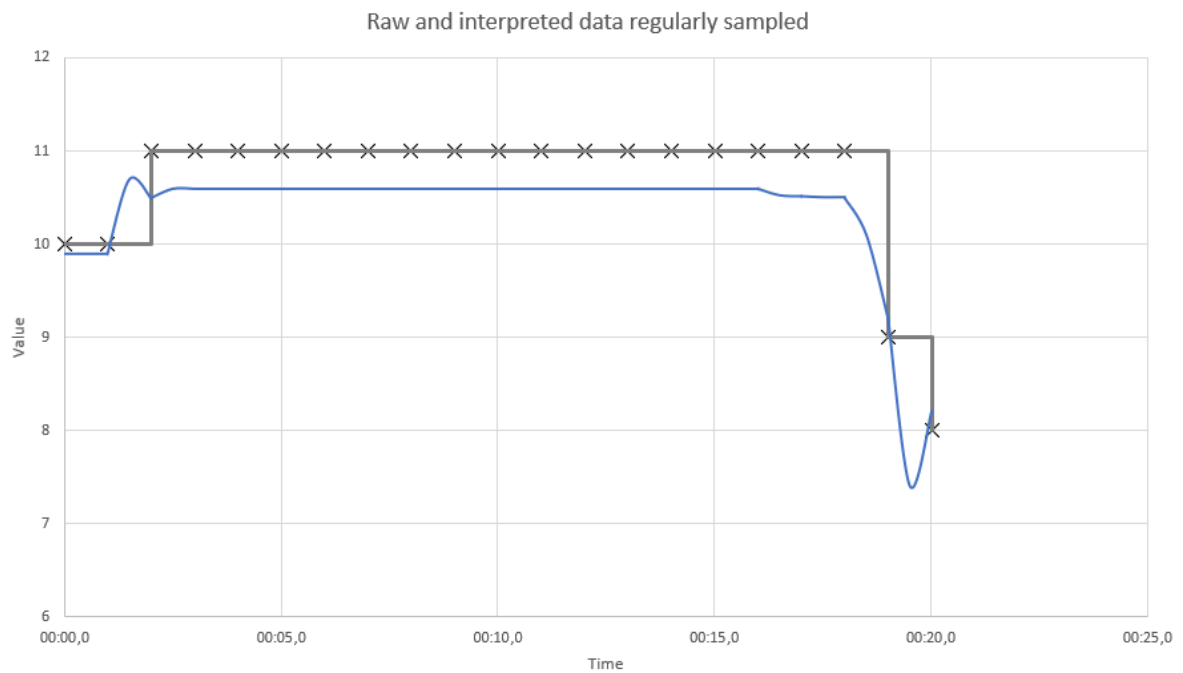
As an example, consider a raw value that changes over time as so:



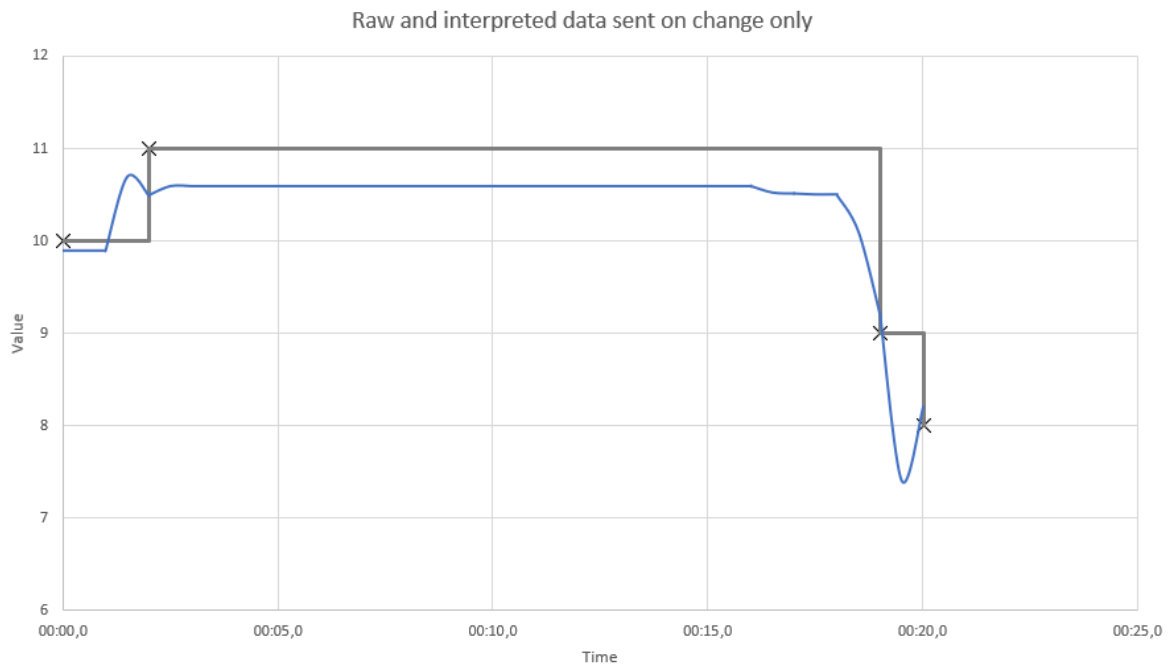
But suppose the sensor can only measure to the nearest whole number, and only once a second. The value thus has some error, shown by the red error bars:



A regular sampling sensor would generate uniform inputs:



While an on-change sensor would generate inputs only when a value changes:



The grey line shows how a real-time processing system such as Apama interprets such values. A value is assumed to maintain the latest value until it is replaced by a newer value. It is also common to draw lines between measurements. So there is a straight line decreasing from value 11 at 00:01 to value 9 at 00:19. However, a real-time system cannot do this. It does not know what the next value is, whereby viewing historic data can interpolate between values. At time 00:19.5, the only information it has is that the value was 11 and then 9. It does not yet know that the value will be 8 at time 00:20. Note that there is no difference between the interpreted grey line in the regularly sampled and on-change-only case. Note that in the middle of the graph, there is a significant quantization error (the `true` value of 10.6 is read as 11), and the sampling frequency of only once a second means that the minimum point of 7.4 at 00:19.5 seconds is lost.

Input values at different times

Consider two position sensors which give the position of two robot arms, and both are on-change sensors. If the two arms move together in unison in the same direction and speed, then the position sensors should update to new values at the same time such that they are a constant distance apart (or at least, close to a constant distance). If the **Difference** block has inputs connected to both sensors, then even if the robot arms move, the output of the **Difference** block should be approximately constant. Analytics Builder evaluates all values with the same timestamp, so even though there may be a small delay in receiving the values from the two sensors, provided they supply timestamps from the same clock (and the **Ignore Timestamp** parameter of the **Measurement Input** block is not set), then the **Difference** block will always generate a synchronized output, as shown in the table below:

Time	Position sensor 1	Position sensor 2	Output of the Difference block
00:00	4	14	10
00:01	6	16	10
00:02	9	19	10

By contrast, consider if the two robot arms do not move in unison - one moves, then another. The distance between the arms may vary as either arm moves. As the sensors are on-change inputs, there will only be a new value when the position changes. However, the absence of an input does not mean that the corresponding robot arm does not have a position. It has remained where it is (within error margins). For example:

Time	Position sensor 1	Position sensor 2	Output of the Difference block
00:00	9	19	10

Time	Position sensor 1	Position sensor 2	Output of the Difference block
00:01	11	19	8
00:02	13	19	6
00:05	13	18	5
00:06	13	17	4
00:07	13	15	2

The bold numbers indicate the effective value. The last value latches if it has not been replaced by a more up-to-date value.

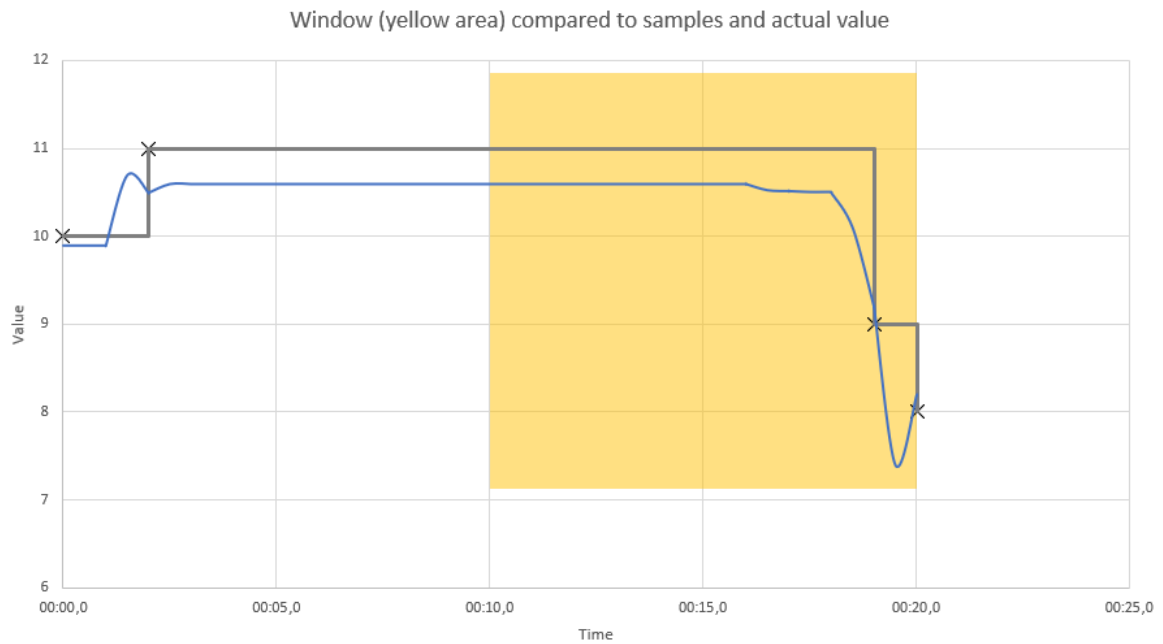
On-change inputs and time windows

If an on-change input is connected to an aggregate block such as the **Average (Mean)** block, then the block should treat the input as continuously having the most recent value it received. This is significant for blocks that maintain a time window. Even if the block last received an input (and thus had its `$process` action called) more than the time window ago, the contents of the window will contain the most recent value. For example, consider the **Average (Mean)** and **Integral** blocks with window duration set to 10 seconds, and input as so:

Time	Input value	Window contents	Output of the Average (Mean) block	Output of the Integral block
00:00	10	0: 10	10	0
00:02	11	0-2: 10	10	20
00:10	11	0-2: 10, 2-10: 11	10.8	108
00:12	11	2-12: 11	11	110
00:19	9	9-19: 11	11	110
00:20	8	10-19: 11; 19-20: 9	10.8	108

In this case, note how a measurement received at time 00:02 still has influence on the output at 00:19 and later - because it is not replaced until 00:19. Also note that when a new value occurs, it has zero influence on the average or integral - it has not been that value for any time yet. The only exception is for the **Average (Mean)** block when it starts - with an empty window, the output is the input value.

Also refer to the diagram below for what values the window covers at time 00:20:



While only the measurement updates with values 9 and 8 were received within the window, the average value within the window is close to the 11 value. The measurement update for that was received at time 00:02, but as it is a continuous value, it continues to hold the 11 value until time 00:19.

Note that for a block such as **Missing Data**, the absence of input for some time may affect the behavior of the block. If the **Missing Data** block is configured with a 10 second duration, then it would trigger at time 00:12.

If the **Average (Mean)** and **Integral** blocks receive a regular input from a regular sampling sensor, then the block will receive more measurement values, and the comparable table is:

Time	Input value	Window contents	Output of the Average (Mean) block	Output of the Integral block
00:00	10	0: 10	10	0
00:01	10	0-1: 10	10	10
00:02	11	0-2: 10	10	20
00:03	11	0-2: 10, 2-3: 11	10.333	31
00:04	11	0-2: 10, 2-4: 11	10.5	42
00:05	11	0-2: 10, 2-5: 11	10.6	53
00:06	11	0-2: 10, 2-6: 11	10.667	64
00:07	11	0-2: 10, 2-7: 11	10.714	75
00:08	11	0-2: 10, 2-8: 11	10.75	86
00:09	11	0-2: 10, 2-9: 11	10.778	97
00:10	11	0-2: 10, 2-10: 11	10.8	108
00:11	11	1-2: 10, 2-11: 11	10.9	109

Time	Input value	Window contents	Output of the Average (Mean) block	Output of the Integral block
00:12	11	2-12: 11	11	110
00:13	11	3-13: 11	11	110
00:14	11	4-14: 1	11	110
00:15	11	5-15: 11	11	110
00:16	11	6-16: 11	11	110
00:17	11	7-17: 11	11	110
00:18	11	8-18: 11	11	110
00:19	9	9-19: 11	11	110
00:20	8	10-19: 11; 19-20: 9	10.8	108

Note that the highlighted lines are the same as without the repeated measurements. Repeated measurements of the same value received by these blocks make no difference to what the block would calculate if re-evaluated.

WINDOW BLOCK OUTPUT TIMINGS

For aggregate blocks such as the **Average (Mean)** block, the effect of a change of input value means that, if regularly re-evaluated, the output of the block will change, approaching the new value. If there have been different input values received by the block in the past, then a re-evaluation of the block at any point in time is possible, and each may generate a different output.

With the previous example from [On-change inputs and time windows](#), repeatedly re-evaluating an **Average (Mean)** block with a 10 second window will yield the following:

Time	Input value	Output of the Average (Mean) block
00:00	10	10
00:01	10	10
00:02	11	10
00:03	11	10.333
00:04	11	10.5
00:05	11	10.6
00:06	11	10.667
00:07	11	10.714
00:08	11	10.75
00:09	11	10.778
00:10	11	10.8
00:11	11	10.9

Time	Input value	Output of the Average (Mean) block
00:12	11	11

It would also be possible to re-evaluate the block every half a second, or any fraction of a second - to milliseconds or even smaller times between re-calculations. It is impractical to “continuously” re-evaluate the block (to re-calculate the average value at a given point in time and generate a new output). So when is an appropriate time for the block to evaluate and generate an output?

The **Average (Mean)** block (and others) provide an **Output Threshold** parameter. If this is set, then the block emulates a sensor which generates a new measurement reading if the output changes by the output threshold amount. Thus, if set at 0.1, we get several outputs between 00:02 and 00:03 (when the output is changing from 10 to 10.333), another output between 00:03 and 00:04 (when it reaches 10.4), outputs at 00:04 and 00:05 exactly, another between 00:06 and 00:07 (10.7), then at 00:10 and 00:11 and 00:12. The block calculates at what time the output would vary by more than the output threshold compared to the most recent output, and re-evaluates at that point in time. Thus, the output may occur quite irregularly in time, but output at times such that the values output always differs by an amount equal to the output threshold. The block also re-evaluates on any new inputs, even if there is not a different value to the last input.

As models may wish to perform a calculation with the output of the **Average (Mean)** block at any point in time (for example, to compare to another measurement), a **Sample** input port is also provided, to force a re-evaluation and generate an output value.

Windows and buckets

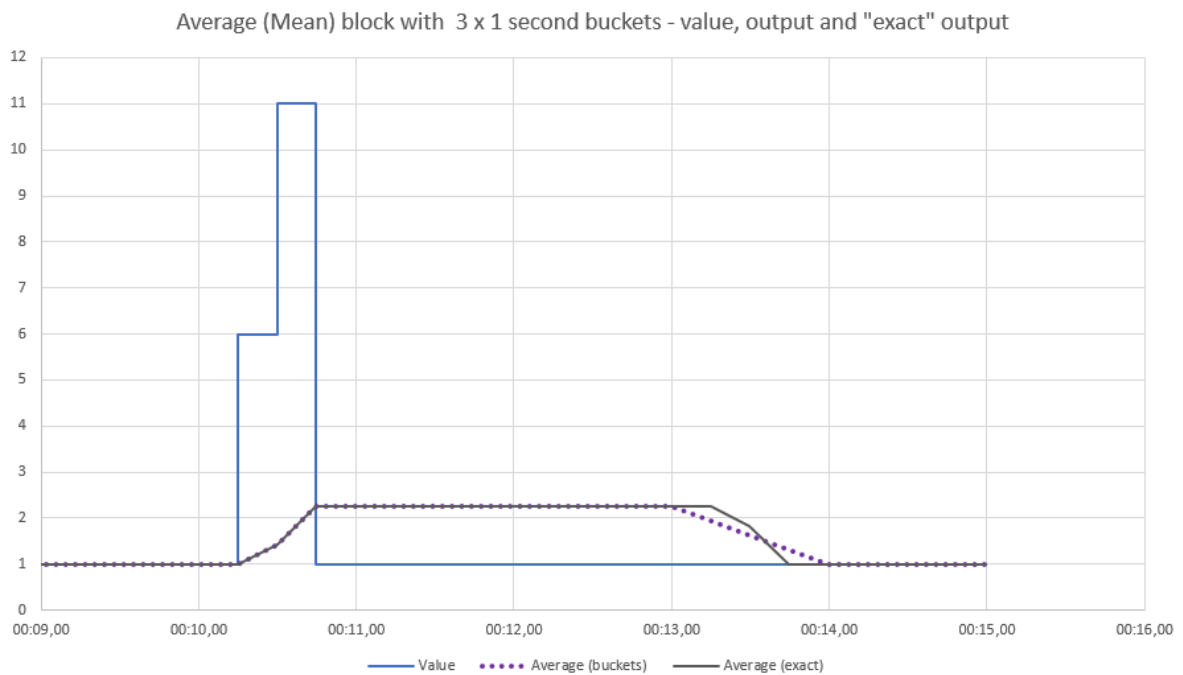
A number of blocks, primarily those in the **Aggregate** category, maintain a time-based window of input values received in the past. Their output is a calculation based on values within this window. Typically, such blocks offer two distinct ways of managing this window:

- A parameter value specifying the duration of the window. If set, the window automatically expires interpreted values older than the time specified (where interpreted values are the latest received value at any point in time, as described in [On-change inputs and time windows](#)). If the parameter is not specified, then the block does not automatically expire any data.
- A reset input. When a signal is received, the contents of the window is cleared and the block resets to having no contents.

It is possible to use these in combination, or neither, but more typically one or the other.

For the case of the window duration being specified, the block must be able to expire old data. A strictly exact implementation of this would be to store each different measurement input along with the time it occurred. For long windows and/or high frequency inputs, this can result in a large amount of data being stored. To avoid an excessive amount of data being stored, the product blocks do not store all measurement values and times. Instead, the window duration is divided into equal-size buckets. The blocks store state per bucket and use that information to re-calculate the output of the block. A historic bucket can either be completely within the window or be partly expired. If a bucket is partly expired, then the block applies a fractional proportion of the values within that bucket. The practical effect of this is that if the value is changing without significant fluctuations, there is only small difference between an exact (but more resource-intensive) implementation of the block and one that uses buckets. If there is a significant fluctuation in the input value that causes a shift in the output, then the exact time of individual measurement inputs is lost, and the effect that the significant value has as it expires will be spread in time by up to one bucket duration. The product blocks use 20 buckets as a reasonable compromise between accuracy and efficiency.

To illustrate this, we exaggerate the effect by simulating an **Average (Mean)** block with 3 buckets and a 3 second window, so each bucket is 1 second in duration. A few anomalous readings (after a continuous input of value 1) affect the average for both exact and bucket-based **Average (Mean)** blocks in the same way, but we can compare the result of “exactly” expiring each value exactly 3 seconds after it occurred with using buckets, where the change in output is smoothed over the bucket duration:



Note that not only is the timing of the expiry of the anomalous values less precise, the exact shape of the output is lost. The bucketed average changes uniformly between time 00:13 and 00:14. Remember, the product blocks use 20 buckets, so the effect would be less pronounced in this case.

PULSE SIGNALS

A pulse is used to signal a point in time or a change of state. Examples of use cases for pulses are:

- A person goes through a gate (for example, at a train station).
- A button is pressed (for example, an emergency stop button).
- A machine goes into a new state (for example, a gateway is reset or powered on).
- A device has made a connection to the network.

In Cumulocity, [events](#), [alarms](#) and [operations](#) are used as the sources of pulses.

A pulse may be merely a point in time, but it can also convey extra information, for example, the version number of the software or which network node it has connected to. These can be obtained using the **Extract Property** block. If you are writing your own custom blocks, these are accessible if the input is declared as a `Value` type, which has a `properties` field. This can be used with numeric value types as well. See the documentation for the Analytics Builder Block SDK for more information on the `Value` type.

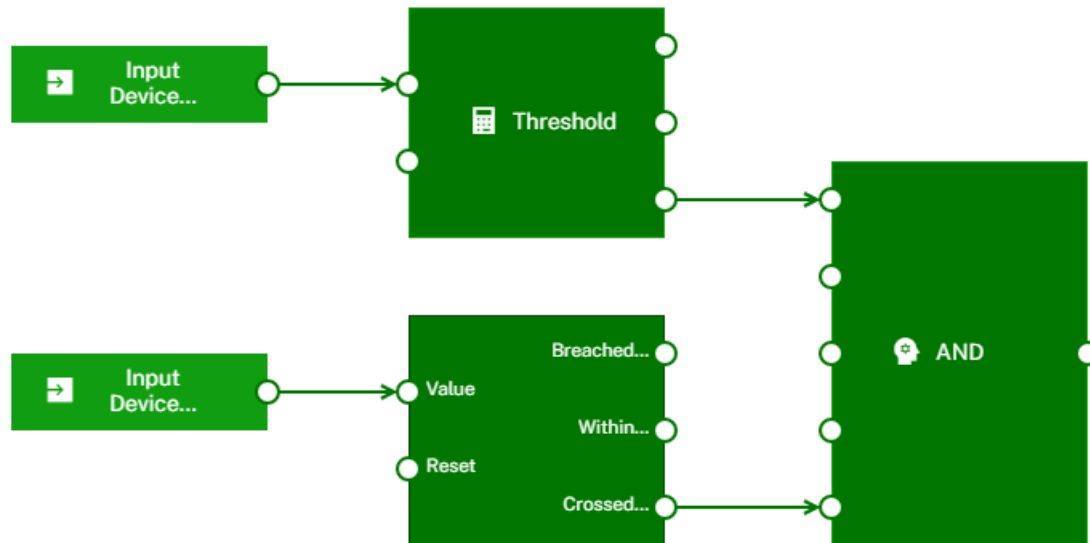
In contrast to measurement values, the timing and number of pulses is very significant, and even though the only difference between subsequent pulses may be the time they were received, each is still significant (whereas multiple measurements with the same value are of little interest).

In contrast to measurement values, a pulse is only active for a single evaluation of a model, where a model evaluation processes all blocks that have a timer that fires (including input blocks) and any blocks connected to outputs that have changed. While both pulses and boolean measurement values are represented by the `boolean` type in the EPL of the blocks, their behavior is different:

- If a boolean measurement value is received by a block, it will "stick" to its value until replaced. For example, if looking at whether temperature sensors 1 and 2 are both over a threshold using an **AND** block, that value is still true after receiving high measurements.
- If a pulse is received by a block, it is reset after it is evaluated. For example, if an **Average (Mean)** block is reset by an event, then the reset happens when the event is received. After that, if no further event is received, the block is not reset on future value inputs.

It is still valid and sensible to combine multiple pulses, for example, with an **AND** block. If two pulses occur at the same point in time, that will be a single evaluation. For example, the **Threshold** block has a **Crossed Threshold** output port which is a pulse that is sent only when a continuous value input goes from one side of the threshold value to another. Two sensors on the same device (thus with the same timestamp) may cross thresholds at the same time, so the **AND** of the output of two such thresholds will only trigger if both inputs cross the threshold with new values of the same timestamp. Note that if one sensor crosses the threshold and then later the other sensor,

the blocks below would never give an output from the **AND** block. They would only do this if the two occurred at the same time.



DISCRETE-TIME MEASUREMENTS

There are some cases where a measurement would be used where a numeric measurement value does not represent a continuous-time property. For example:

- The weight of parcels passing a weighing machine on a conveyor belt.
- The size of objects passing a measuring point.
- The value of a ticket scanned or printed by a machine.

In contrast to the continuous-time values, each of these are significant, even if two measurements are of the same value. The time of each measurement may have some significance, but the time between subsequent measurements is of no great significance. If the measurements were received with slightly different timings, or even potentially out of order, this would not signify a difference (for example, the sum of the value of tickets does not change if they are processed in a different order or with different timings, and the time between values is unlikely to be uniform). Note that by discrete-time we are only referring to the time of the measurements. The value may still be continuous. For example, weight is a continuous value, but we may weigh individual parcels - while the weight of a parcel may be representable to fractions of a gram of weight. If we are between two parcels on a conveyor belt, there is no "current" value for the weight of the parcel at that point. The value could also be discrete. For example, the ticket value would typically be a discrete value (for example, to the nearest cent, or one of a few predefined ticket values).

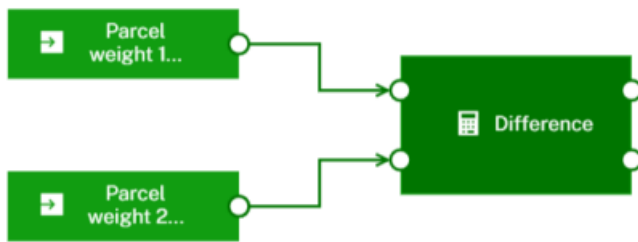
Compare also: https://en.wikipedia.org/wiki/Discrete_time_and_continuous_time and https://en.wikipedia.org/wiki/Continuous_or_discrete_variable. In practice, all measurements are samples of a continuous-time property.

When dealing with discrete-time inputs, you should use the **Discrete Statistics** block rather than the **Average (Mean)** block. While it is possible to connect an input from a parcel weight sensor to the **Average (Mean)** block, the **Average** block weights by time. For example:

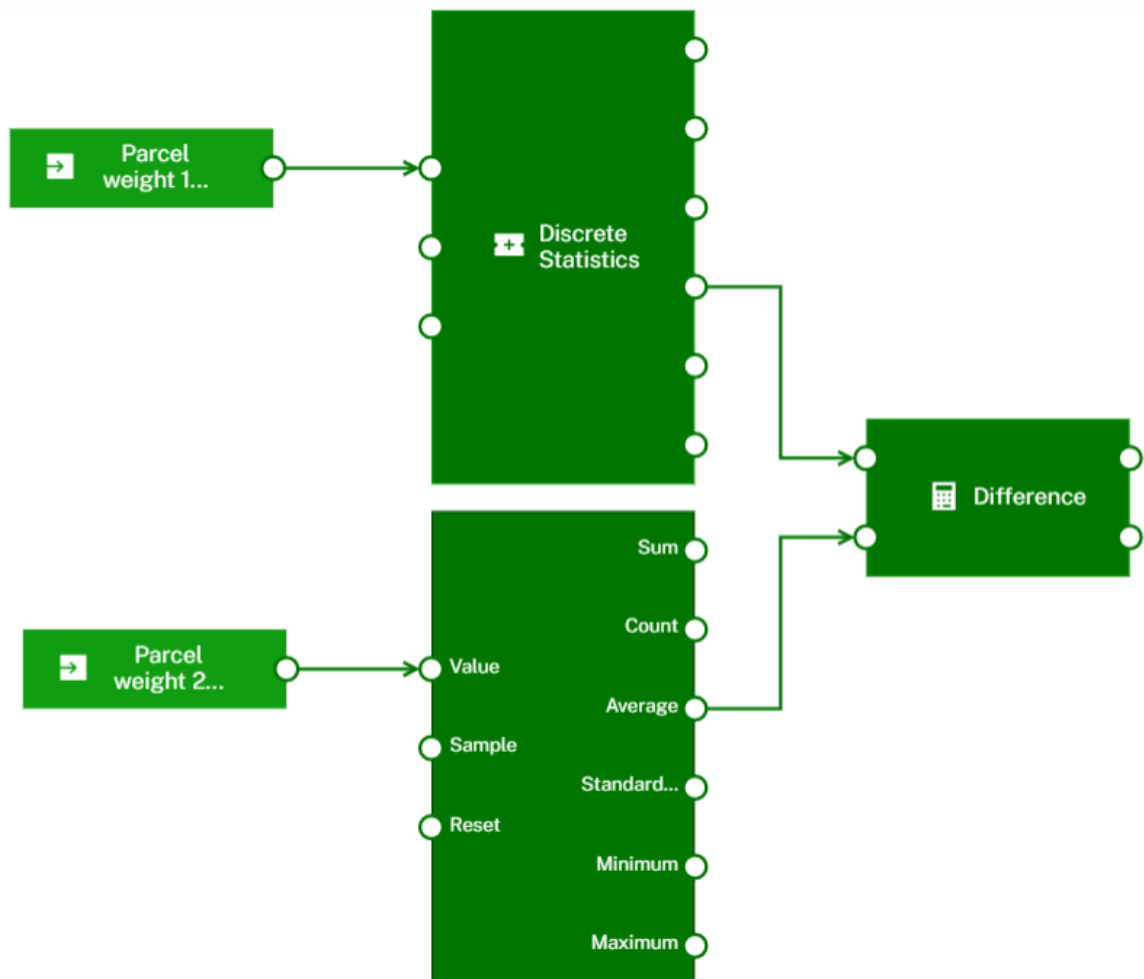
Time	Input value	Average of continuous-time input	Average of discrete-time input
00:10	11	11	11
00:19	9	11	10
00:20	8	10.8	9.33

Compare this to the table in [On-change inputs and time windows](#), looking at times 00:10 onwards (that is, what would be in a window from 00:10 to 00:20). Note that the continuous-time block would generate a different output if the inputs occurred at different times, while a block averaging values based on discrete-time would not.

Note that by default measurements are treated as continuous-time values. So it is possible, for example, to calculate the difference between two values:



The above example gives the difference between the most recent weight received by two sensors. This may not be a particularly useful distinction if these are genuinely discrete-time inputs. However, it can make sense to compare the difference of averages (or means) between two discrete-time inputs. The **Average** output port of the **Discrete Statistics** block gives a continuous-time value:



MODELS AND DEVICES

MODEL EXECUTION FOR DIFFERENT DEVICES

Models are executed independently of each other. That is, models for specific devices can execute in parallel, making use of hardware parallelism where possible, if models are processing data (such as **Measurement**, **Event**, or **Operation** objects) for a different set of devices. When defining a model, you can configure it to use data from a set of specific devices or from a range of devices, with each device being handled independently.

Each model must either:

- receive input from a set of specific devices and send output to a set of specific devices, or
- receive input from each device within a range of devices and send output to the trigger device or an asset. Note that asset output can only be used for sending cross-device aggregates.

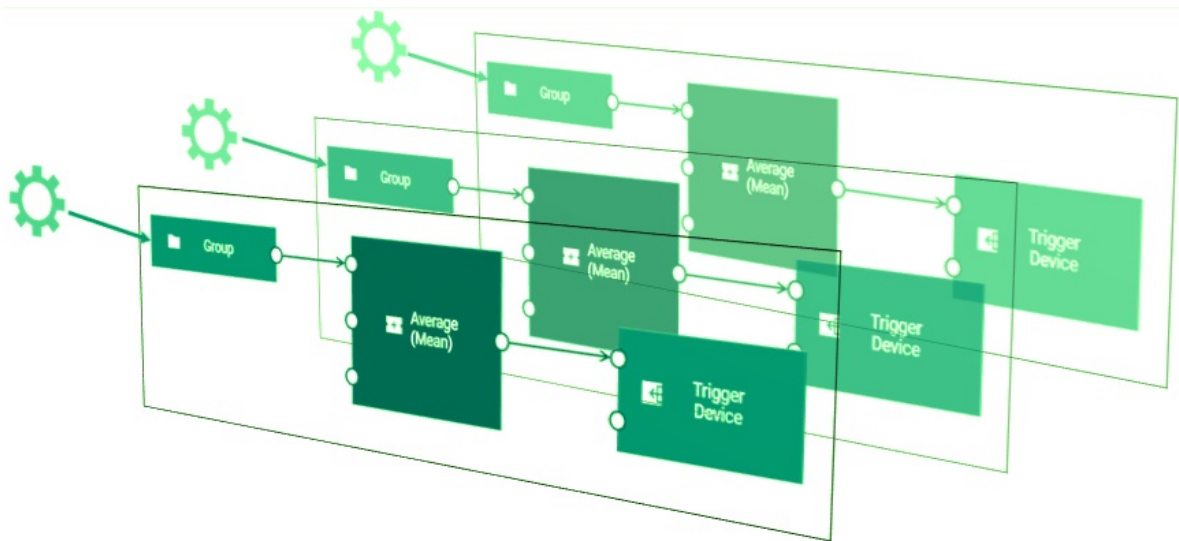
A range of devices can include a Cumulocity group, an asset, or all input sources on the tenant. When a model uses a range of devices, the model acts on all devices referred to by the range, either directly or indirectly through members of the group that are themselves groups and have device members (or even “grand-children” group members). A device can be a member of zero, one or many groups. For more information, see [Grouping devices](#) and [Managing assets](#).

INFO

A model that acts on a range of devices only determines the group membership when the model is activated. If the membership of a group changes while a model is running, the model will not behave any differently for any new or removed members of the group. If a group membership is changed, then models that refer to that group should be de-activated and re-activated.

It is not possible to mix the two types of input blocks above (but see [Broadcast devices](#)). However, data from a model processing specific devices can be sent to and received from other models, including models for ranges of devices, and vice versa (see [Connections between models](#)).

When a model consumes data from a range, the model behaves as if multiple instances of the model are running, as illustrated below, each one processing data from each device independently. Each instance processes data for a different device, but all share the same blocks and block parameters. The values of the wires are independent for different instances. Any blocks that are stateful, such as the **Average (Mean)** block, operate independently of the data from other devices. As with models using specific devices, if any block causes a runtime error or exception, then the entire model goes into a failed state - it stops processing data for all devices.



Typically, when using ranges of devices for inputs, all input blocks would use the same group. It is possible to use different groups. If there are devices in one group but not in another, those blocks will never generate a signal for devices that are not in that group. For some blocks, such as the **Expression** block, this is not useful - an **Expression** block will only generate an output if all of the required inputs have received a value, but it may be useful for **pulse** inputs of a **Gate** block.

When a model has inputs that are consuming data from specific devices, then the output blocks generating outputs can specify the same or different specific devices.

When a model has inputs that are consuming data from a range of devices, all synchronous output blocks must specify the trigger device or asset. The trigger device generates data (**Measurement** , **Event** or **Operation**) for whichever device that instance applies to - or whichever device sent the data to trigger that instance. Asynchronous output blocks in such models can specify the trigger device, asset or any other specific device.

When a model has inputs that are consuming data from each device belonging to an asset, the output blocks can send the output to a specific asset or trigger device. Keep in mind that asset output can only be used for sending cross-device aggregates.

When a template parameter is used for an output block, then if the parameter's value is a range of devices, then this is treated the same as if it were set to the trigger device. The output goes to whichever device triggered the model's evaluation, with each device within a range being treated independently. Typically, the same template parameter will be used for both input and output, so these will refer to the same range, and each device is processed independently.

You can use the model editor to change input and output blocks from one input source or output destination to another. When changing between a range of devices and a device or asset, output blocks will switch between the trigger device and the device or asset specified, so that the model is kept in a usable state. See also [Replacing sources or destinations](#).

The test and simulation modes are only permitted for models using specific devices. If you wish to test or simulate a model using a range of devices, then use the model editor to modify it to apply to a single device within the range, and then activate the model in test or simulation mode. See [Deploying a model](#) for more information on these modes.

Configuring the concurrency level

By default, the Analytics Builder runtime uses 1 CPU core to execute models. If you want to change the number of CPU cores, send a `POST` request to Cumulocity that changes the value for the `numWorkerThreads` key. See [Configuration](#) for detailed information.

Typically, this configuration value would be set to the number of CPU cores available for the system, but it may be useful to configure this either higher or lower according to what resources are available. It does not need to scale to the number of devices (that is, it is quite reasonable to have 4 worker threads with hundreds of devices, assuming a moderate event rate per device).

With the concurrency level set to 1, it is still possible to create models which use ranges of devices as inputs, but these continue to operate independently for each device within the range, and it is still not possible to mix range and single device input or output.

INFO

Using multiple specific devices in a model with the concurrency level set to more than 1 can lead to connections between models which are deployed across multiple workers. Chains of models using multiple specific devices with high throughput usually scale less well than chains of models all using a single specific device.

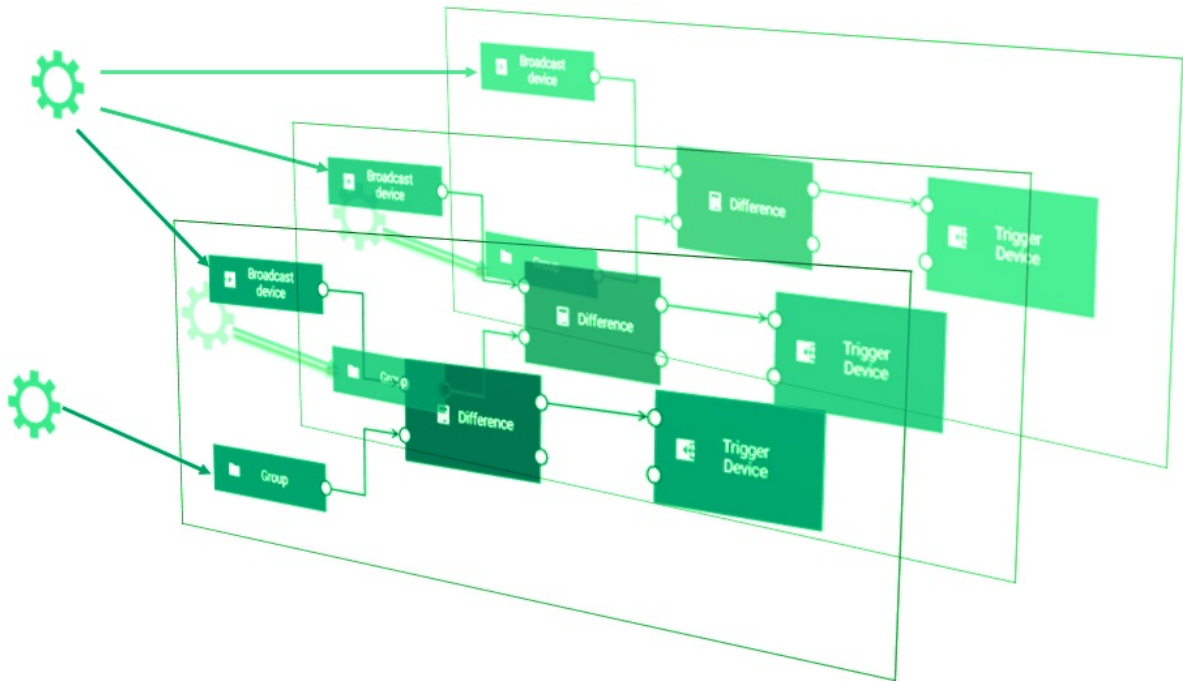
INFO

The concurrency level has a fix value of 1 for microservices with multi-tenant support.

BROADCAST DEVICES

It is sometimes useful to have signals that can apply to all models. These may be signals from devices, or from other systems that are presented as if they were signals from a device. Analytics Builder thus supports devices that are referred to as broadcast devices and signals from these devices are available to all models across all devices.

Broadcast devices can be used as inputs in any model, together with either inputs from a specific device or a range of devices. The diagram below illustrates how a broadcast device applies to all devices within a range of devices. It is possible to combine signals from devices in a range of devices with signals from a broadcast device by providing them as different inputs into a processing block such as the **Expression** block.



Unlike other devices, a broadcast device can only be used for synchronous output of a model that only consumes data from broadcast devices. Broadcast output of the asynchronous type can be generated by a model consuming non-broadcast inputs.

It is also not possible to connect models together using synchronous data from a broadcast device output (that is, no model may use a measurement from a broadcast device that is the output of a different model). Models can be connected together using asynchronous outputs from a broadcast device (that is, models may use an operation from a broadcast device that is the output of a different model).

Identifying broadcast devices

Broadcast devices are identified by the presence of a property on the device object in the inventory for that device; the presence of either the `pas_broadcastDevice` or `c8y_Kpi` property. Thus, whether a device is considered a broadcast device or not is global for that device across all models. It is not permitted to use a range of devices that contains a broadcast device. `c8y_Kpi` objects are typically used with the `KPI` block. Thus, it is possible to use a KPI object to compare measurements from a range of devices - one KPI object is used for all devices in the range.

VIRTUAL DEVICES

A virtual device is used when a model is deployed in test or simulation mode. See also [Deploying a model](#).

Virtual devices are objects in the Cumulocity inventory with a `c8y_VirtualDevice` property. This property refers to the identifier of the real device of which the virtual device is a copy.

Use the `creationDate` to find out what `virtualDevice` was created for a model activation and which measurements have that device as their source.

By default, the virtual devices are kept for 30 days. If you want to change this default, you must change the tenant options. That is, you must send a `POST /tenant/options` request. For detailed information, see the information on the [tenant options](#) in the Cumulocity OpenAPI Specification. For example, specify the following to set the retention period for the virtual devices to 1 day:

```
{
  "category": "analytics.builder",
  "key": "retention.virtualDevicesMaxDays",
  "value": "1"
}
```

See also [Configuration](#).

Virtual devices are not shown in the Device Management application. Use [REST](#) operations to find these entries.

CONNECTIONS BETWEEN MODELS

You can connect multiple models together using output blocks and input blocks. A model that contains an output block such as **Measurement Output** (for **Measurement** objects of Cumulocity) will generate a series of events, and this can be consumed by a suitable input block (such as **Measurement Input**) in another model. For more details, see [Keys for identifying a series of events](#).

When models are connected together using inputs and outputs for the same stream of events, the term “chain” is used to refer to all of the models that are connected to each other in this way. There may be multiple chains if there are separate groups of models that are connected to each other.

INFO

The events from one model can only be consumed by another model when all involved models are deployed in production mode. When the models are deployed in test or simulation mode, virtual devices are used and the events from one model can therefore not be consumed by another model.

When one model has a synchronous output block generating a series of events for a given key and a second model has an input block consuming from that same series of events (that is, with the same key parameters), then this forms a connection from the first model to the second. When the first model triggers the output block, this causes the second model to be evaluated with a new input on its input block.

It is also possible to form connections between models using the output from an asynchronous output block. In this case, when the first model triggers the asynchronous output block, the output is generated and sent to the external system (such as Cumulocity). The data is received back from the external system at some later point in time and causes the evaluation of any other models consuming the data.

Similar to the processing order of wires within a model (see also [Processing order of wires](#)), the following applies when an output block in one model generates a series of events that an input block in another model consumes:

- A single model can send the same events to more than one other model. This means, it is possible to have a single model perform some common pre-processing, such as unit conversion or calculating an average (with the **Average (Mean)** block), and that value to be used by multiple other models.
- Models are executed in order with respect to the connections between models formed using synchronous output so that the source of a connection is always evaluated before the target of a connection. If a model has connections from multiple blocks all triggered from the same initial event, then they will all evaluate first, and the receiving model will evaluate with all of the inputs once.

Connections formed using asynchronous output do not have a specific execution order. A model consuming the output is executed only when the output is received back from the external system.

Similar to the wire restrictions within a model (see also [Wire restrictions](#)), there are restrictions on how output blocks and input blocks can be used to connect models together:

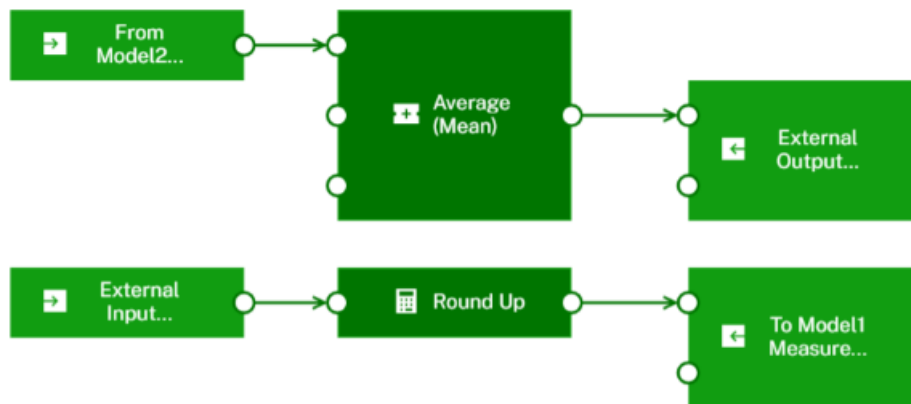
- One block across all models is permitted to generate a series of synchronous events for a given key. See also [Keys for identifying a series of events](#). Multiple output blocks generating asynchronous events can be used within a single model or across multiple models.
- No cycles can be created between models using synchronous output. A model that receives events via an input block synchronously generated from another model cannot include an output block that generates synchronous events that the other model would consume. This applies even if one of the models contains two separate parts, such that there is no actual cycle in terms of wires and connections between models. Cycles among models can be created because of asynchronous outputs. Therefore, care must be taken not to introduce indefinite cyclic executions of models.

Any model that does not meet these restrictions when used in combination with the already activated models will cause an error on trying to activate it. This will count for the last element in a cycle of models. For such errors, the problem may be in interactions between models rather than a problem specific to a single model, but existing models that have already been activated will not automatically be deactivated. For example, if multiple models all generate the same series of synchronous events (with the same key), then the first model to be activated can be deployed, but all subsequent models will report an error upon trying to activate them.

For example, there are three models: Model1, Model2, and Model3. A cycle may exist if:

- An output block of Model1 produces a series of synchronous events that is consumed by an input block in Model2, and Model2 contains an output block that generates a second series of synchronous events, and
- Model3 contains an input block that consumes a series of events from Model2, and Model3 also contains an output block that generates a series of synchronous events used by an input block in Model1.

Note that only activating any two of these models can be done without error. If activated in order, only Model3 would have an error. But if Model1 or Model2 were deactivated, then Model3 could be activated. The error will occur even if one of the models does not contain a link from the input block that is part of the chain to the output block that forms part of the chain, such as the example for Model3 below: the events from Model2 do not form a cycle to the **To Model1 Measurement** output block, but they count as a cycle as they are both in the same model. (In this case, the issue could be resolved by splitting that model into two models, thus removing the cycle).



INFO

Using multiple specific devices in a model with the concurrency level set to more than 1 can lead to connections between models which are deployed across multiple workers. Chains of models using multiple specific devices with high throughput usually scale less well than chains of models all using a single specific device.

CONFIGURING THE NUMBER OF SHOWN INPUT SOURCES AND OUTPUT DESTINATIONS

By default, a maximum of 10 items are shown in the following cases:

- When you select a different input source or output destination (see [Editing the parameters of a block](#)).
- When you replace devices, groups or assets (see [Replacing sources or destinations](#)).

When you use the search box in the above cases, this default also applies to the maximum number of items that are shown in the search result. When you click **Load more**, up to 10 more items are shown.

If you want to change this default value (to show either more or less items), you must change the tenant options. That is, you must send a `POST /tenant/options` request. For detailed information, see the information on the [tenant options](#) in the Cumulocity OpenAPI Specification.

For example, specify the following to set the value to 20:

```
{
  "category": "analytics.builder",
  "key": "c8yAnalyticsBlocks.queryInventoryPageSize",
  "value": "20"
}
```

See also [Configuration](#).

SEARCHING FOR DEVICES, GROUPS AND/OR ASSETS

By default, only devices, groups and assets are shown in the following cases:

- When you select a different input source or output destination (see [Editing the parameters of a block](#)).
- When you replace devices or groups (see [Replacing sources or destinations](#)).

When you use the search box in the above cases, all managed objects in the Cumulocity inventory which match the search and filter criteria are shown, grouped by type. You can thus build analytic models by defining any managed objects in the inventory as input blocks or output blocks.

If you want to restrict the search to show only managed objects of a specific type (for example, to show only devices), you must change the tenant options. That is, you must send a `POST /tenant/options` request. For detailed information, see the information on the [tenant options](#) in the Cumulocity OpenAPI Specification.

For example, specify the following if you only want to show devices:

```
{
  "category": "analytics.builder",
  "key": "c8yAnalyticsBlocks.queryInventoryNameSearchAdditionalFilter",
  "value": "has(c8y_IsDevice)"
}
```

The `c8y_IsDevice` in the value is a so-called fragment. You can specify any fragment that is known to Cumulocity, including any fragments that you have created yourself.

You can combine several values. For example, specify the following if you only want to show devices and groups:

```
{
  "category": "analytics.builder",
  "key": "c8yAnalyticsBlocks.queryInventoryNameSearchAdditionalFilter",
  "value": "has(c8y_IsDevice) or has(c8y_IsDeviceGroup)"
}
```

The default value of this tenant option is `not has(c8y_IsVirtualDevice)`. As long as you do not change this tenant option, virtual devices are not shown as they would not make sense in an analytic model. If you change the value for this tenant option, make sure to specify all managed objects that you want to see in the search result.

See also [Configuration](#).

SUPPORT FOR DYNAMIC CHANGES TO GROUP AND ASSET HIERARCHY

By default, input blocks in Analytics Builder now support dynamic changes to the group hierarchy. This includes the addition, deletion, and update of devices or assets, as well as structural changes within groups. Analytics Builder models automatically adapt to these changes, processing data based on the most current group or asset hierarchy.

INFO

This is also applicable to the [Smart rules plugin](#).

Key behavior and limitations

- *Deletion of source/target:* If a specific device, group, or asset configured as an input or output in the model is deleted, the deployed model will automatically transition to a FAILED state.
- *Empty groups:* If a monitored group becomes empty, the model remains in the ACTIVE state. It will resume processing data as soon as a new device, group, or asset is added to the group.
- *Nested & cascaded deletions:* Dynamic detection does not support the deletion of nested groups or the cascaded deletion of devices. In these scenarios, the model must be undeployed and redeployed to recognize the updated hierarchy.

To enable this behavior for custom blocks, refer to the [Analytics Builder Block SDK documentation](#).

MODEL SIMULATION

ABOUT SIMULATION MODE

You can deploy a model in simulation mode to run it against historical input data (such as Cumulocity measurements). This allows testing the behavior of a newly developed model against historical data or fine-tuning an existing model. Or it allows testing a model against a set of historical data with known properties.

You use the model manager to deploy a model in simulation mode. See [Deploying a model](#) for more details.

INFO

Simulation mode is only permitted for models using specific devices. If you wish to simulate a model using a range of devices, then use the model editor to modify it to apply to a single device within the range, and then activate the model in simulation mode.

When a model is deployed in simulation mode, it uses data from a virtual device (see also [Virtual devices](#)). Thus, a simulated model can run alongside other non-simulated models without interfering with them.

A simulated model runs as if it is running at the time of the historical data. The input data are processed in the order of their historical time. The simulated model also uses the historical time for the timestamps of the generated output.

Events, alarms and operations are created with a timestamp. However, with time there can be updates to these objects. For example, an alarm can be cleared or the status of an operation can be changed. As a history of changes to event, alarm and operation objects is not maintained, the object is only replayed at its initial timestamp, with the latest version of its properties. Thus, changes to these objects are not replayed and simulation mode is of limited use if your models depend on changes to objects.

INFO

Simulation mode is not permitted for models with **Managed Object Input** blocks.

When running a simulation, historical data is replayed into the Apama correlator from the Cumulocity database. If there is a significant delay in the data being queried from the database or high load in the system, this can lead to dropping the input in exceptional circumstances. A simulated model processes input data at normal speed. For example, if the historical data entries are separated by one second, they are processed one second apart. This means that simulating a model with one hour of historical data will take approximately one hour of simulation time.

SIMULATION PARAMETERS

To deploy a model in simulation mode, you must provide values for two parameters in the model manager: start time and end time. These values determine the time range for which historical data is to be sent into the simulated model.

- The start time from which historical data is to be sent into the model.
- The end time until which historical data is to be sent into the model.

Sending of data into the simulated model is stopped when all historical data from the specified time range has been sent.

CONFIGURING THE MAXIMUM NUMBER OF SIMULATION MODELS

By default, a maximum of 3 simulation models can be deployed at a time.

If you want to change this default value (to deploy either more or fewer simulation models at a time), you must change the tenant options. That is, you must send a `POST /tenant/options` request. For detailed information, see the information on the [tenant options](#) in the Cumulocity OpenAPI Specification.

For example, specify the following to set the value to 5:

```
{
  "category": "analytics.builder",
  "key": "simulation.maxInstances",
  "value": "5"
}
```

See also [Configuration](#).

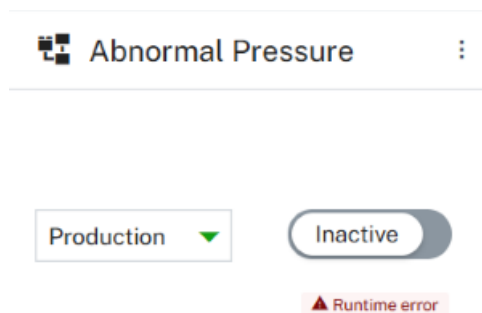
MONITORING DROPPED INPUTS


The simulated model may drop delayed input events in exceptional cases. Alarms are periodically raised with the number of input events dropped across all the models. See also [Monitoring dropped events](#).

MONITORING AND CONFIGURATION

MONITORING

You can monitor the current status of each model in the model manager. The card for a model shows the current mode for this model (such as production mode) and whether it is active (deployed) or inactive.



If a model failed to deploy or failed while running, a runtime error icon  **Runtime error** is shown on the card for the model.

To find out whether a model has failed while processing data, reload all models in the model manager to show their latest states. See also [Reloading all models](#).

Monitoring periodic status

In addition to the status that is shown on the card for a model, it is possible to enable generation of periodic status published as Cumulocity operations or events. See [Configuration](#) on setting the `status_device_name` and `status_period_secs` tenant options.

Each operation has the following parameters:

Parameter	Description
<code>models_running</code>	Information about deployed models that are running.
<code>models_failed</code>	Information about deployed models that have failed.
<code>chain_diagnostics</code>	Information about model chains. See Connections between models for more information.

Parameter	Description
<code>apama_status</code>	The Apama correlator status metrics. Many status names correspond to the key names in the Apama REST API. The values are returned by the <code>getValues()</code> action of the <code>com.apama.correlator.EngineStatus</code> event and exposed via the REST API.

Model status

The following information is published for each deployed model that is currently running or has failed:

Name	Description
<code>mode</code>	The mode of the deployed model. It is <code>SIMULATION</code> for models deployed in simulation mode. Otherwise, it is <code>PRODUCTION</code> .
<code>modeProperties</code>	Any mode-specific properties of the model. This includes the start and end time of the simulation for models running in the <code>SIMULATION</code> mode.
<code>numModelEvaluations</code>	The total number of times the model has been evaluated since it was deployed.
<code>numBlockEvaluations</code>	The total number of times that the blocks have been evaluated in the model since it was deployed. This is the sum of the count of evaluation for each block in the model.
<code>avgBlockEvaluations</code>	The average number of blocks that have been evaluated per model evaluation.
<code>numOutputGenerated</code>	The total number of outputs generated by the model since it was deployed.

This information about each model provides insight into the performance or working of models. For example, a model with a much larger number of `numBlockEvaluations` than another model might indicate that it is consuming most resources even though it might have low `numModelEvaluations`. Similarly, it can be used to find out whether a model is producing output at the expected rate relative to the number of times it is evaluated.

You can monitor the status using the Apama REST API or the Management interface which is an EPL plug-in. See the following topics in the Apama product documentation for further information:

- [Managing and Monitoring over REST](#)
- [Using the Management interface](#)

Chain diagnostics

The following information is published for all chains that are present:

Name	Description
<code>creationTime</code>	The time when this chain was created.
<code>executionCount</code>	The number of times the chain was evaluated. (1)
<code>modelsInEvalOrder</code>	A list of model identifiers in the order in which the models were evaluated.
<code>pendingTimersCount</code>	The number of pending timers which are behind the current time.
<code>maxTime</code>	The maximum time taken to evaluate the chain. (1)
<code>minTime</code>	The minimum time taken to evaluate the chain. (1)
<code>meanTime</code>	The mean time taken to evaluate the chain. (1)
<code>execBucket</code>	The execution time statistics of the chain. (1), (2)

(1) The fields are updated if the chain is evaluated fully or partially. Partial evaluation of a chain means that only some models of the chain are evaluated.

(2) There are 21 buckets which store the number of times when the execution time falls within the bucket range. Each bucket has size of `timedelay_secs` divided by 10 seconds, except for the last bucket which stretches to infinity. For example, if `timedelay_secs` is 2 seconds, then the first bucket holds the number of times when the chain execution took up to 0.2 seconds, the second bucket holds the number of times when the chain execution took more than 0.2 seconds but up to 0.4 seconds, and so on. See also the following example:

Bucket	Execution time range
1	0 - 0.2
2	0.2 - 0.4
3	0.4 - 0.6
...	...
20	3.8 - 4.0
21	4.0 - infinity

For more information on `timedelay_secs`, see [Keys for model timeouts](#).

Slowest chain status

When chains of models with a high throughput are deployed across multiple workers, it may happen that the chain falls behind in processing input events, creating a backlog of input events that are still to be processed. These chains are referred to as "slow chains". A message is written to the correlator log if the slowest chain is delayed by more than 1 second. For example: "Analytics Builder chain of models "Model 1", "Model 2", "Model 3" is slow by 3 seconds." See [Log files of the Apama-ctrl microservice](#) for information on where to find the log.

The following information on the slowest chain is also available in the periodic status that is published as Cumulocity operations or events, within the `apama_status` parameter:

Name	Description
<code>user-analyticsbuilder.slowestChain.models</code>	All models contained in the slowest chain.
<code>user-analyticsbuilder.slowestChain.delaySec</code>	The number of seconds the chain lags behind in processing the input events.

Example

The following is an example of the status operation data that is published by Cumulocity:

```
{
  "creationTime": "2021-01-05T21:48:54.620+02:00",
  "deviceId": "6518",
  "deviceName": "apama_status",
  "id": "8579",
  "self": "https://myown.iot.com/devicecontrol/operations/8579",
  "status": "PENDING",
  "models_running": {
    "Package Tracking": {
      "mode": "SIMULATION",
      "modeProperties": {"startTime":1533160604, "endTime":1533160614},
      "numModelEvaluations": 68,
      "numBlockEvaluations": 967,
      "avgBlockEvaluations": 14.2,
      "numOutputGenerated": 50
    }
  }
},
```



```

"models_failed": {
  "Build Pipeline ": {
    "mode": "PRODUCTION",
    "numModelEvaluations": 214,
    "numBlockEvaluations": 671,
    "avgBlockEvaluations": 3.13,
    "numOutputGenerated": 4
  }
},
"chain_diagnostics": {
  "780858_780858": {
    "creationTime": 1600252455.164188,
    "executionCount": 4,
    "modelsInEvalOrder": ["780858_780858", "780860_780860"],
    "pendingTimersCount": 1,
    "timeData": {
      "execBucket": [2,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
      "maxTime": 0.00014781951904296875,
      "meanTime": 0.0001152356465657552,
      "minTime": 6.29425048828125e-05
    }
  }
},
"apama_status": {
  "user-analyticsbuilder.slowestChain.models": "\"Model 1\", \"Model 2\", \"Model 3\"",
  "user-analyticsbuilder.slowestChain.delaySec": "3",
  "numJavaApplications": "1",
  "numMonitors": "27",
  "user-httpServer.eventsTowardsHost": "1646",
  "numFastTracked": "183",
  "user-httpServer.authenticationFailures": "4",
  "numContexts": "5",
  "slowestReceiverQueueSize": "0",
  "numQueuedFastTrack": "0",
  "mostBackedUpInputContext": "<none>",
  "user-httpServer.failedRequests": "4",
  "slowestReceiver": "<none>",
  "numInputQueuedInput": "0",
  "user-httpServer.staticFileRequests": "0",
  "numReceived": "1690",
  "user-httpServer.failedRequests.marginal": "1",
  "numEmits": "1687",
  "numOutEventsUnAked": "1",
  "user-httpServer.authenticationFailures.marginal": "1",
  "user-httpServer.status": "ONLINE",
  "numProcesses": "48",
  "numEventTypes": "228",
  "virtualMemorySize": "3177968",
  "numQueuedInput": "0",
  "numConsumers": "3",
  "numOutEventsQueued": "1",
  "uptime": "1383561",
  "numListeners": "207",
  "numOutEventsSent": "1686",
  "mostBackedUpICQueueSize": "0",
  "numSnapshots": "0",
  "mostBackedUpICLatency": "0",
  "numProcessed": "1940",
  "numSubListeners": "207"
}
}

```

Monitoring dropped events

When a model receives an event, it may be dropped if the correlator delivers or processes it too late. See [Input blocks and event timing](#).

Alarms are periodically raised with the total number of dropped events and a sample of the last dropped event. See [Analytics Builder dropped events](#) for information on alarms.

All dropped input events are also sent to channel `AnalyticsDroppedEvents`, allowing you to implement your own monitoring of the dropped events. A dropped input event sent to the channel `AnalyticsDroppedEvents` is packaged inside an event of type `apama.analyticsbuilder.DroppedEvent`. This allows you to extract the original dropped event and perform any analysis on it, for example, categorizing the number of dropped events per device. This can be achieved by writing EPL that listens for the `DroppedEvent` events, aggregates by device identifier and/or time, and sends measurements to Cumulocity that can be monitored. See also [Deploying apps](#).

Monitoring the model lifecycle

Life-cycle messages are written to the correlator log whenever a model is created or deleted, or when it fails. The log messages may look as follows:

```
Model "Build Pipeline" with PRODUCTION mode has started.

Model "Build Pipeline" with PRODUCTION mode has ended.

Model "Build Pipeline" with PRODUCTION mode has failed with an error:
IllegalArgumentException - Error while validating parameters for the
block "toggle" of type "apama.analyticskit.blocks.core.Toggle":
The "Set Delay" must be finite and positive: -1.
```

Deploying a model can combine existing models or chains to form a new chain. The formation of a chain may take a while to complete as it may combine multiple existing models and chains. Activation messages are written to the correlator log whenever the activation of a chain is started and completed. For example:

```
Analytics Builder chain of models "Model 1", "Model 2", "Model 3" is being activated.
Analytics Builder chain of models "Model 1", "Model 2", "Model 3" has been activated.
```

See [Log files of the Apama-ctrl microservice](#) for information on where to find the log.

CONFIGURATION

You can customize the settings of Analytics Builder, the so-called “tenant options”, by sending REST requests to Cumulocity. The key names that you can use with the REST requests are listed in the topics below. A category name is needed along with the key name; this is always `analytics.builder`.

You can find some concrete examples in [Using curl commands for setting various tenant options](#). However, you can use any tool you like.

To change the tenant options, you need ADMIN permission for “Option management”. See [Managing permissions and roles](#) for more information.

⚠ CAUTION

After you have changed a tenant option using a REST request, the correlator will automatically restart. An alarm with a MAJOR severity will be created in this case; you can view it on the **Alarms** page of the Cockpit application (see [Working with alarms](#) for more information).

Keys for status reporting

Key name	Description
<code>status_device_name</code>	The name of the Cumulocity device to which the status operations are to be published. The default name is <code>apama_status</code> .

Key name	Description
<code>status_period_secs</code>	The frequency in seconds at which the status is to be published. The default value is 0 seconds, meaning that status reporting is disabled. You can enable status reporting by setting the frequency to a positive value.
<code>status_send_type</code>	How the status is to be published. The default value is <code>OPERATION</code> , meaning that the status is published as a Cumulocity operation. You can change this to one of the following values: <ul style="list-style-type: none"> <code>EVENT</code> - Publish the status as a Cumulocity event. <code>MEASUREMENT</code> - Publish the status as a Cumulocity measurement.
<code>status_send_keys</code>	A comma-separated list of field names to be used when publishing the status. If not set or empty, the status for all fields is published. For example, if you specify the following, then the status includes only the values for these fields in one measurement. <code>numQueuedInput,numListeners,numMonitors</code>
<code>status_event_type</code>	The event type if the status is published as a Cumulocity event, or the measurement type if the status is published as a Cumulocity measurement. The default type is <code>apama_status</code> .
<code>status_event_text</code>	The event text if the status is published as a Cumulocity event. The default text is <code>Apama Status</code> .

Keys for model timeouts

Key name	Description
<code>timedelay_secs</code>	The maximum delay in seconds before the input block considers an event to be old. The default value is 1 second. See also Input blocks and event timing .
<code>logging_throttle_secs</code>	Logging throttling in seconds. Periodic log messages (for example, those reporting changes in the number of events being dropped by the input block) will not appear more frequently than defined by this constant. The default value is 1 second. See also Input blocks and event timing .
<code>minimum_wait_time_secs</code>	The minimum wait time in seconds. Some blocks can generate output automatically, based on the rate of change of the output. This sets a lower limit on the time between such outputs. See also Common block inputs and parameters .

Keys for simulation mode

Key name	Description
<code>simulation.maxInstances</code>	The maximum number of simulation models to be deployed at a time. The default value is 3 models. See also Configuring the maximum number of simulation models .

Other keys

Key name	Description
<code>numWorkerThreads</code>	The number of worker threads. The default value is 1. See also Configuring the concurrency level . Note that the number of worker threads cannot be changed for the microservice with multi-tenant support.

Key name	Description
<code>retention.virtualDevicesMaxDays</code>	The retention period in days for keeping virtual devices. The default value is 30 days. See also Virtual devices .
<code>c8yAnalyticsBlocks.queryInventoryPageSize</code>	The number of items that are shown in the Select Input Source and Select Output Destination dialog boxes. The default value is 10. See also Configuring the number of shown input sources and output destinations .
<code>c8yAnalyticsBlocks.queryInventoryNameSearchAdditionalFilter</code>	The managed objects that are shown when you use the search box and filter checkboxes of the Select Input Source or Select Output Destination dialog box. See also Searching for devices, groups and/or assets .

Logged tenant options

The values for some of the tenant options are logged. These are the following:

- `status_device_name`
- `status_period_secs`
- `timedelay_secs`
- `numWorkerThreads`

If you want to find out which values are currently used for these tenant options, you can look them up in the log. See also [Log files of the Apama-ctrl microservice](#).

Using curl commands for setting various tenant options

You can set or change various tenant options by sending `POST` requests to Cumulocity. This topic explains how you can do this using the curl command-line tool. See <https://curl.se/> for detailed information on curl. See also the information on the [tenant options](#) in the Cumulocity OpenAPI Specification.

The syntax of the curl command depends on the environment in which you are working. The syntax for a Bash UNIX shell, for example, is as follows:

```
curl --user username -X POST -H 'Content-Type: application/json' -d '{"category": "analytics.builder", "key": "keyname", "value": "value"}' -k https://hostname/tenant/options
```

where:

- `username` is the name of a user who has ADMIN permission for “Option management” in Cumulocity. curl will prompt for a password. Or you can provide a password in the `username` argument by appending it with a colon (:) and the password. For example:

```
--user User123:secretpw
```

If your tenant does not have its own unique host name, you must provide the tenant identifier in the `username` argument. For example:

```
--user management/User123
```

or

```
--user t12345/User123
```

- `keyname` is one of the keys listed in the previous topics.
- `value` is the value that is to be set for the key, which can be a number or a string, depending on the key.
- `hostname` is the host name of your tenant where your user application is deployed.
- The `category` is always `analytics.builder`.

Example (Bash shell):

```
curl --user User123 -X POST -H 'Content-Type: application/json' -d '{"category": "analytics.builder", "key": "numWorkerThreads", "value": "4"}' -k https://mytenant/tenant/options
```


ANALYTICS BUILDER BLOCK REFERENCE

OVERVIEW OF ALL BLOCKS

The following table gives a brief description of all blocks that can be selected from the palette of the model editor, sorted alphabetically (excluding custom blocks that you have created yourself).

Block Name	Description
Alarm Input	Receives Alarm objects from a device, asset, devices in a group, or all input sources and reorders them based on the timestamp.
Alarm Output	Creates a new Alarm object for a specified device, asset or for the trigger device with a pre-configured alarm name and parameters.
AND	Performs a logical 'and' on the inputs.
Average (Mean)	Calculates the mean of the values over time.
Combiner	Calculates the output based on the selected mode and the connected inputs.
Constant Value	Outputs a value, either when the Trigger input port receives a signal or at startup.
Counter	Gives a count of the total inputs and repeated inputs.
Cron Timer	Sends a signal output based on cron-like periodic timer syntax.
Crossing Counter	Detects and counts the number of threshold crossings in the specified direction.
Delta	Calculates the difference between successive input values.
Difference	Calculates the absolute and signed differences between the connected inputs.
Direction Detection	Detects whether the input value changes direction.
Discrete Statistics	Generates statistics of sum, count, average (mean), standard deviation, minimum and maximum for discrete input values.
Duration	Measures the time elapsed from a set start time.
Event Input	Receives Event objects from a device, asset, or devices in a group and reorders them based on the timestamp.
Event Output	Creates a new Event object for a specified device, asset or for the trigger device.
Expression	Evaluates an expression to perform arithmetic or logical calculations or string operations.
Extract Property	Extracts the specified property from the input value and converts it to the specified type.
From Base N	Converts a base N string to a float.
Gate	Blocks the input from going to output unless the gate is open and enabled.

Block Name	Description
Geofence	Compares the input value against the defined geofence value to detect whether the device is within the geofence, and whether the device entered or exited the geofence.
Gradient	Calculates the weighted linear regression gradient for the values.
Group Statistics	Generates periodic aggregate values across all the devices in a group for which the block has received input values.
Integral	Calculates the integral of the input value over time.
KPI	Compares a value against either a KPI (Key Performance Indicator) or the data point of a device, asset or group of devices.
Latch Values	Latches the latest input value received while the block is enabled.
Limit	Outputs a value that is kept within the defined upper and lower limits.
Managed Object Input	Receives ManagedObject objects from a device, asset, devices in a group, or all input sources.
Managed Object Output	Updates a ManagedObject object for a specified device, asset or for the trigger device.
Measurement Input	Receives Measurement objects from a device, asset, devices in a group, or all input sources and reorders them based on the timestamp.
Measurement Output	Creates a new Measurement object for a specified device, asset or for the trigger device.
Minimum / Maximum	Calculates the minimum and maximum of a value over time.
Missing Data	Generates an output if the input has not occurred for a set amount of time.
NOT	Performs a logical 'not' on the input.
Operation Input	Receives Operation objects from a device, asset, devices in a group, or all input sources.
Operation Output	Creates a new Operation object for a specified device, asset or for the trigger device.
OR	Performs a logical 'or' on the inputs.
Position Input	Receives Event objects from a device, asset, devices in a group, or all input sources and extracts the c8y_Position fragment into a Value object.
Pulse	Converts a non-pulse input into a pulse output.
Range	Compares the input value against the defined lower and upper range values to detect whether the input is within or out of the range, or whether it crosses the range.
Range Lookup	Finds the range in which the input value lies.
Rounding	Rounds the input to a specified number of decimal points or to an integer, using a selectable rule.
Selector	Outputs a parameter value depending on which input port has a true value, lowest number taking precedence.
Send Email	Sends an email to the specified email addresses.

Block Name	Description
Send SMS	Sends an SMS (Short Message Service) to the specified phone number.
Set Properties	Outputs a pulse with properties set from values on the input ports.
Standard Deviation	Calculates the standard deviation and variance of the values over time.
Switch	Outputs the values from a given input, or acts as a circuit breaker.
Text Substitution	Substitutes identifiers marked with a hash and braces (for example, <code>#{name}</code>) in the text template with corresponding entries from the input values.
Threshold	Compares the input value against the defined threshold value to detect whether the input breaches the threshold or whether it crosses the threshold.
Time Delay	Delays the input by the specified amount of time.
To Base N	Converts a float to a base N string.
Toggle	Converts two pulse inputs to a boolean output based on the set and reset signals, with optional delays.

INPUT

This category contains the following blocks:

Block Name	Description
Alarm Input	Receives Alarm objects from a device, asset, devices in a group, or all input sources and reorders them based on the timestamp.
Event Input	Receives Event objects from a device, asset, or devices in a group and reorders them based on the timestamp.
Managed Object Input	Receives ManagedObject objects from a device, asset, devices in a group, or all input sources.
Measurement Input	Receives Measurement objects from a device, asset, devices in a group, or all input sources and reorders them based on the timestamp.
Operation Input	Receives Operation objects from a device, asset, devices in a group, or all input sources.
Position Input	Receives Event objects from a device, asset, devices in a group, or all input sources and extracts the <code>c8y_Position</code> fragment into a Value object.

ALARM INPUT

`apama.analyticskit.blocks.cumulocity.AlarmInput`

Receives Alarm objects from a device, asset, devices in a group, or all input sources and reorders them based on the timestamp.

If the Alarm Status parameter is Active, then the alarms are reordered based on the timestamp (and dropped if they are too old), unless the Ignore Timestamp parameter is set. Otherwise, data is processed as it is received.

The parameters that define the input stream of the block are "Input Source" and "Alarm Type". If this block is configured with the same "Input Source" and "Alarm Type" parameters as an Alarm Output block in another model, then a connection between the models is formed, as each block refers to the same stream of Alarm objects.

Note: When running in simulation mode, because only the creation time of the alarm is stored, the alarm status must be Active.

Parameters

Name	Description	Type	Notes
Alarm Type	The alarm type the block is listening for.	string	
Input Source	Defines the source from which the alarm has been received. This can be a single device, an asset, an object that references or contains a group of devices, or all input sources.	any	
Severity	The severity of the alarm. If not specified, the block listens for all alarm severities.	Option - one of: <ul style="list-style-type: none"> • Critical • Major • Minor • Warning 	Optional
Alarm Status	The status of the alarm. If not specified, the block listens for alarms with any status.	Option - one of: <ul style="list-style-type: none"> • Active • Acknowledged • Cleared 	Optional
Notification Mode	Filters Alarm events such that only new alarms, updated alarms, or all alarms are processed. The default is that all alarms are processed.	Option - one of: <ul style="list-style-type: none"> • All • Updates only • New alarms only 	Default: All
Ignore Timestamp	If selected, the timestamp of the incoming alarm is ignored. Note: When running in simulation mode, because historical input data is used, timestamps are not ignored.	boolean	Default: False

Output Port Details

Name	Description	Type
Alarms	Generates a pulse output for each Alarm object received, with extra properties.	pulse

EVENT INPUT

`apama.analyticskit.blocks.cumulocity.DeviceEventInput`

Receives Event objects from a device, asset, or devices in a group and reorders them based on the timestamp.

If the Ignore Timestamp parameter is set, the block ignores the timestamp of the event and processes the events as they are received. Otherwise, it drops old events.

The parameters that define the input stream of the block are "Input Source" and "Event Type". If this block is configured with the same "Input Source" and "Event Type" parameters as an Event Output block in another model, then a connection between the models is formed, as each block refers to the same stream of Event objects.

Note: When running in simulation mode, because historical input data is used, timestamps are not ignored.

Note: A history of changes is not maintained for Event objects, and it is thus not possible to retrieve the original objects from the inventory. For this reason, a model which contains this input block type may behave differently in simulation mode than it would in production mode.

Parameters

Name	Description	Type	Notes
Input Source	Defines the source from which the event has been received. This can be a single device, an asset, an object that references or contains a group of devices, or all input sources.	any	
Event Type	The event type the block is listening for.	string	
Notification Mode	Filters Event events such that only new events, updated events, or all events are processed. The default is that all events are processed.	Option - one of: <ul style="list-style-type: none"> • All • Updates only • New events only 	Default: All
Ignore Timestamp	If selected, the timestamp of the incoming event is ignored. Note: When running in simulation mode, because historical input data is used, timestamps are not ignored.	boolean	Default: False

Output Port Details

Name	Description	Type
Events	Generates a pulse output for each Event object received, with extra properties.	pulse

MANAGED OBJECT INPUT

`apama.analyticskit.blocks.cumulocity.ManagedObjectInput`

Receives ManagedObject objects from a device, asset, devices in a group, or all input sources.

The block does not reorder the received ManagedObject objects and processes them as they are received. If the Property Name parameter is supplied, then the block does not produce new output if the value of the specified property has not changed since the last output, even if other properties on the same ManagedObject object have changed.

The Value output from the block contains all properties on the ManagedObject object, including the property specified by the Property Name parameter. Property values can be accessed using the Extract Property block.

Properties with values of type string, boolean or float can be accessed by specifying the name of the property in the Extract Property block. For example, if the name of the property is `ap_State`, then specify `ap_State` as the value for the Property Path parameter of the Extract Property block.

If a property value is of type JSON object or sequence, then nested values can be accessed by specifying the full path to the nested values as the name of the property.

For example, if the name of the property is `c8y_SpeedMeasurement` and the value is `{ "Speed": { "value": 1234, "unit": "km/h" } }` (in JSON form), then specify `c8y_SpeedMeasurement.Speed.unit` as the value for the Property Path parameter of the Extract Property block to extract the value of the unit.

Any position data associated with the ManagedObject object is available as a `c8y_Position` property and can be extracted using the Extract Property block.

If the value of the property specified by the Property Name parameter of this block is of type string, boolean or float, then the value is also

directly available in the Value output port and can be directly consumed by blocks consuming values of that type without using the Extract Property block, for example, the Expression or Difference blocks.

The parameters that define the input stream of the block are "Input Source" and "Property Name".

Parameters

Name	Description	Type	Notes
Input Source	Defines the source from which the managed object has been received. This can be a single device, an asset, an object that references or contains a group of devices, or all input sources.	any	
Property Name	The name of the property for which to listen. The ManagedObject object must have a property of this name. Otherwise, it is ignored. If not set, the objects are not filtered. Every update generates a pulse output with all of the properties from the ManagedObject.	string	Optional
Capture Start Value	Outputs the initial value when the model is activated. This parameter must not be selected if the Input Source parameter is set to "All Inputs".	boolean	Default: False

Output Port Details

Name	Description	Type
Value	Generates an output for each ManagedObject object received. All properties of the managed object are available as extra properties. You can use the Extract Property block to access their values.	any

MEASUREMENT INPUT

`apama.analyticskit.blocks.cumulocity.DeviceMeasurementInput`

Receives Measurement objects from a device, asset, devices in a group, or all input sources and reorders them based on the timestamp.

If the Ignore Timestamp parameter is set, the block ignores the timestamp of the measurement and processes the measurements as they are received. Otherwise, it drops old measurements.

If using a group for input, select a device within the group to select the fragment and series, and then change to the desired group.

The parameters that define the output stream of the block are "Input Source" and "Fragment and Series". If this block is configured with the same "Input Source" and "Fragment and Series" parameters as a Measurement Output block in another model, then a connection between the models is formed, as each block refers to the same stream of Measurement objects.

Note: When running in simulation mode, because historical input data is used, timestamps are not ignored.

Parameters

Name	Description	Type	Notes
Input Source	Defines the source from which the measurement has been received. This can be a single device, an asset, an object that references or contains a group of devices, or all input sources.	any	

Name	Description	Type	Notes
Fragment and Series	The fragment the block is listening for. This only shows fragments and series for measurements associated with the object (device or group) selected. Any measurements on a device within a group are only shown when a device is selected (unless there are measurements with the group as the source). For example, if a temperature measurement is sent in Celsius, the fragment is T and the series is C(Celsius). This means, that this parameter can be set as T.C or T=>C.	string	
Ignore Timestamp	If selected, the timestamp of the incoming measurement is ignored. Note: When running in simulation mode, because historical input data is used, timestamps are not ignored.	boolean	Default: False

Output Port Details

Name	Description	Type
Value	The numeric value from the measurement object.	float

OPERATION INPUT

`apama.analyticskit.blocks.cumulocity.OperationInput`

Receives Operation objects from a device, asset, devices in a group, or all input sources.

The block does not reorder the received Operation objects and processes the operations as they are received. The block can be optionally configured to only process operations having a specified status or property.

The output from the block contains all properties on the Operation object. Property values can be accessed using the Extract Property block.

Properties with values of type string, boolean or float can be accessed by specifying the name of the property in the Extract Property block. For example, if the name of the property is `ap_State`, then specify `ap_State` as the value for the Property Path parameter of the Extract Property block.

If a property value is of type JSON object or sequence, then nested values can be accessed by specifying the full path to the nested values as the name of the property.

For example, if the name of the property is `c8y_SpeedMeasurement` and the value is `{ "Speed": { "value": 1234, "unit": "km/h" } }` (in JSON form), then specify `c8y_SpeedMeasurement.Speed.unit` as the value for the Property Path parameter of the Extract Property block to extract the value of the unit.

The parameter that defines the input stream of the block is Input Source.

Note: A history of changes is not maintained for Operation objects, and it is thus not possible to retrieve the original objects from the inventory. For this reason, a model which contains this input block type may behave differently in simulation mode than it would in production mode.

Parameters

Name	Description	Type	Notes
Input Source	Defines the source from which the operation has been received. This can be a single device, an asset, an object that references or contains a group of devices, or all input sources.	any	

Name	Description	Type	Notes
Operation Name	The name of the operation the block is listening for. If specified, the Operation object must have a property of this name. Otherwise, it is ignored.	string	Optional
Operation Status	The status for which to listen. If not specified, the block listens for operations with any status.	Option - one of: <ul style="list-style-type: none"> • SUCCESSFUL • FAILED • EXECUTING • PENDING 	Optional
Notification Mode	Filters Operation events such that only new operations, updated operations, or all operations are processed. The default is that all operations are processed.	Option - one of: <ul style="list-style-type: none"> • All • Updates only • New operations only 	Default: All

Output Port Details

Name	Description	Type
Operations	Generates a pulse output for each Operation object received. All properties of the Operation object are available as extra properties. You can use the Extract Property block to access their values.	pulse

POSITION INPUT

`apama.analyticsbuilder.blocks.PositionInput`

Receives Event objects from a device, asset, devices in a group, or all input sources and extracts the `c8y_Position` fragment into a Value object.

If no `c8y_Position` fragment is present, the event is ignored. If the fragment does not contain at least a valid latitude and valid longitude, the event is ignored. If the Primary Value parameter is set to Altitude and the fragment does not contain an altitude, the event is ignored. Latitudes must be between -90 and 90 degrees inclusive. Longitudes must be between -180 and 180 degrees inclusive.

The primary value of the output Value object can be set to be the latitude, longitude or altitude. All members of the `c8y_Position` fragment are added to the properties dictionary of the Value object.

If the Ignore Timestamp parameter is set, the block ignores the timestamp of the event and processes the measurements as they are received. Otherwise, it reorders the events and drops old measurements.

The parameter that defines the input stream of the block is Input Source, and Event Type if set.

Note: When running in simulation mode, because historical input data is used, timestamps are not ignored.

Note: A history of changes is not maintained for Event objects, and it is thus not possible to retrieve the original objects from the inventory. For this reason, a model which contains this input block type may behave differently in simulation mode than it would in production mode.

Parameters

Name	Description	Type	Notes
------	-------------	------	-------

Name	Description	Type	Notes
Input Source	Defines the source from which the position is received. This can be a single device, an asset, an object that references or contains a group of devices, or all input sources.	any	
Event Type	The event type for which the block listens. If left unset, then there is no filtering by type. To consume events from another model, this property must be set.	string	Optional
Notification Mode	Filters Event events such that only new events, updated events, or all events are processed. The default is that all events are processed.	Option - one of: <ul style="list-style-type: none"> • All • Updates only • New events only 	Default: New events only
Ignore Timestamp	If selected, the timestamp of the incoming measurement is ignored. Note: When running in simulation mode, because historical input data is used, timestamps are not ignored.	boolean	Default: False
Primary Value	The primary value to be output by the block: latitude, longitude or altitude, or empty if not set.	Option - one of: <ul style="list-style-type: none"> • Latitude • Longitude • Altitude 	Optional

Output Port Details

Name	Description	Type
Position	An object containing at least latitude and longitude.	any

OUTPUT

This category contains the following blocks:

Block Name	Description
Alarm Output	Creates a new Alarm object for a specified device, asset or for the trigger device with a pre-configured alarm name and parameters.
Event Output	Creates a new Event object for a specified device, asset or for the trigger device.
Managed Object Output	Updates a ManagedObject object for a specified device, asset or for the trigger device.
Measurement Output	Creates a new Measurement object for a specified device, asset or for the trigger device.
Operation Output	Creates a new Operation object for a specified device, asset or for the trigger device.
Send Email	Sends an email to the specified email addresses.

Block Name	Description
Send SMS	Sends an SMS (Short Message Service) to the specified phone number.

ALARM OUTPUT

`apama.analyticskit.blocks.cumulocity.CreateAlarm`

Creates a new Alarm object for a specified device, asset or for the trigger device with a pre-configured alarm name and parameters.

This block produces synchronous output. The parameters that define the output stream are "Output Destination" and "Alarm Type".

If "Create Alarm" is provided repeatedly, then the count of the alarm is increased (unless it has been cleared). If the severity or message has changed, then the count is not increased - the existing alarm is modified instead. If the alarm has been acknowledged, the status of the alarm is only made active on an increase in severity. Use the Always Activate Alarm parameter to make every modification of an existing alarm reactivate the alarm.

The default behavior of the Alarm Output block is to produce a synchronous output and the Analytics Builder framework explicitly disallows models that can cause a loop when synchronous outputs are produced. However, such validations are disabled when asynchronous outputs are produced. Therefore, special care must be taken when using the Create Asynchronous Output parameter. This may result in infinite loops during model execution (for the same model or across multiple models), leading to performance degradation or the application running out of memory. For example, a loop can occur if the model consumes an alarm input of a given type and produces an alarm output of the same type for the same device; the output generated by the model is then consumed by the same model as an input, resulting in a loop.

Parameters

Name	Description	Type	Notes
Output Destination	The device (or for models handling group of devices, trigger device or asset) that the alarm is associated with. Assets can be used only for sending cross-device aggregates. The model editor uses the device name. This is mapped internally to the device identifier.	any	
Alarm Type	Identifies the type of this alarm, for example "com_cumulocity_events_TamperEvent".	string	
Message	The text that is displayed if the alarm is triggered. The message is set either by parameter or input. If both are not set, the model name is used. You cannot set both.	string	Optional
Severity	The severity of the alarm. The severity is set either by this parameter or the input port. You must set only one of them.	Option - one of: <ul style="list-style-type: none"> • Critical • Major • Minor • Warning 	Optional
Params Fragment	If this parameter is set, then the incoming properties on the Properties input port are added to this fragment in the generated Alarm. If this is not set, the properties are copied to the top level of the Alarm params.	string	Optional

Name	Description	Type	Notes
Create Asynchronous Output	<p>Allow Create Alarm or Clear Alarm to be done asynchronously. Asynchronous output events are events that do not have source timestamps and can only be consumed by another model in a time-asynchronous manner when they are received back from the platform.</p> <p>Caution: The default behavior of the Alarm Output block is to produce a synchronous output and the Analytics Builder framework explicitly disallows models that can cause a loop when synchronous outputs are produced. However, such validations are disabled when asynchronous outputs are being produced. Therefore, special care must be taken when using the Create Asynchronous Output parameter. This may result in infinite loops during model execution (for the same model or across multiple models), leading to performance degradation or the application running out of memory. For example, a loop can occur if the model consumes an alarm input of type foo and produces an alarm output of type foo for the same device; the output generated by the model is then again consumed by the same model as an input, resulting in a loop.</p>	boolean	Optional
Always Activate Alarm	<p>If selected, all alarms produced by the block are set to ACTIVE. By default (unselected), if an alarm has been acknowledged, the alarm is only reactivated if the severity has increased.</p>	boolean	Optional

Input Port Details

Name	Description	Type
Create Alarm	Creates an alarm when a signal is received and a severity value is set.	pulse
Clear Alarm	Updates the status of an existing alarm to CLEARED when a signal is received.	pulse
Severity	The severity of the alarm - should be one of the severity values (WARNING, MINOR, MAJOR, CRITICAL) or CLEARED.	string
Message	The message for the alarm.	string
Time	Sets the timestamp of the alarm. If not connected, the current model time is used.	float
Properties	The properties to set on the alarm.	any

EVENT OUTPUT

```
apama.analyticskit.blocks.cumulocity.CreateEvent
```

Creates a new Event object for a specified device, asset or for the trigger device.

The text of the event is determined by the Text Input input port or by the Message parameter. You must not set both. If neither are set, the model name is used as the text.

This block produces synchronous output. The parameters that define the output stream of the block are "Output Destination" and "Event Type".

Parameters

Name	Description	Type	Notes
Output Destination	The device (or for models handling group of devices, trigger device or asset) to which the event is to be sent. Assets can be used only for sending cross-device aggregates. The model editor uses the current device name. This is mapped internally to the device identifier.	any	
Event Type	Identifies the type of this event.	string	
Message	The text that is displayed when the event is created. This requires that the Text Input input is not connected. If neither are set, the model name is used as the text.	string	Optional
Params Fragment	If this parameter is set, then the incoming properties on the Properties input port are added to this fragment in the generated Event. If this is not set, the properties are copied to the top level of the Event params.	string	Optional

Input Port Details

Name	Description	Type
Create Event	Creates an event when a signal is received.	pulse
Text Input	Sets the text of the event. The Message parameter must not be set if this is used.	string
Time	Sets the timestamp of the event. If not connected, the current model time is used.	float
Properties	The properties to set on the event.	any

MANAGED OBJECT OUTPUT

`apama.analyticskit.blocks.cumulocity.ManagedObjectOutput`

Updates a ManagedObject object for a specified device, asset or for the trigger device.

If the Property Name parameter is set, then the Value input port (or its properties if it has an empty value) is used to set that property on the managed object.

If the Property Name parameter is not set, then all properties from the Value input port are used to update the managed object.

Note: The following reference properties on a managed object cannot be updated using this block: childDeviceIds, childAssetIds, deviceParentIds and assetParentIds.

This block does not participate in time-synchronous model-to-model communication. Multiple blocks can be used in a single model or multiple models to update the same property of the same device. Cycles among models can be formed because of this block, so care must be taken not to introduce indefinite cyclic execution of models.

This block produces asynchronous output. The parameter that defines the output stream of the block is "Output Destination".

Parameters

Name	Description	Type	Notes
------	-------------	------	-------

Name	Description	Type	Notes
Output Destination	The device (or for models handling group of devices, trigger device or asset) that the managed object is associated with. Assets can be used only for sending cross-device aggregates. The model editor uses the current device name. This is mapped internally to the device identifier.	any	
Property Name	The name of the property to update. If not set, then the properties from the Value input port are mapped to the top level properties of the managed object.	string	Optional

Input Port Details

Name	Description	Type
Value	The value of the property to set.	any
Update Property	Signals that the property is to be updated. If not connected, every new value from the Value input port updates the property.	pulse

MEASUREMENT OUTPUT

`apama.analyticskit.blocks.cumulocity.CreateMeasurement`

Creates a new Measurement object for a specified device, asset or for the trigger device.

This block sends a new measurement for the Value input port. If the Send input port is connected, this block only sends an output on receiving a send signal. The measurement is sent to the current device or the device specified.

This block produces synchronous output. The parameters that define the output stream of the block are "Output Destination", "Fragment Name" and "Series Name".

Non-finite values are ignored.

Parameters

Name	Description	Type	Notes
Output Destination	The device (or for models handling group of devices, trigger device or asset) to which the measurement is to be sent. Assets can be used only for sending cross-device aggregates. The model editor uses the current device name. This is mapped internally to the device identifier.	any	
Fragment Name	The name of the fragment in the measurement.	string	
Series Name	The name of the series in the measurement.	string	
Unit	The name of the unit of measurement (for example, "mm" or "lux").	string	Optional
Params Fragment	If this parameter is set, then the incoming properties on the Properties input port are added to this fragment in the generated Measurement. If this is not set, the properties are copied to the top level of the Measurement params.	string	Optional

Input Port Details

Name	Description	Type
Value	The measurement to be sent.	float
Send	Signals that a new measurement is to be created. If not connected, every new Value input creates a measurement.	pulse
Time	Sets the timestamp of the measurement. If not connected, the current model time is used.	float
Properties	The properties to set on the measurement.	any

OPERATION OUTPUT

```
apama.analyticskit.blocks.cumulocity.CreateOperationStaticValue
```

Creates a new Operation object for a specified device, asset or for the trigger device.

If none of the Operation Name, Parameter Name or Parameter Value parameters are set, then the operation uses the Properties input port to populate the Operation object.

The block does not participate in time-synchronous model-to-model communication. Multiple blocks can be used in a single model or multiple models to create new operations for the same device. Cycles among models can be formed because of this block, so care must be taken not to introduce indefinite cyclic execution of models.

The block produces asynchronous output. The parameter that defines the output stream of the block is "Output Destination".

Parameters

Name	Description	Type	Notes
Output Destination	The device (or for models handling group of devices, trigger device or asset) to which the operation is to be sent. Assets can be used only for sending cross-device aggregates. The model editor uses the device name. This is mapped internally to the device identifier.	any	
Operation Name	The name of the property to create on the Operation object.	string	Optional
Parameter Name	The name of the parameter for the operation.	string	Optional
Parameter Value	The value of the parameter for the operation.	string	Optional
Description	The description of the operation to create.	string	Optional

Input Port Details

Name	Description	Type
Create Operation	Creates an operation when a signal is received.	pulse
Properties	The properties to set on the operation.	any

SEND EMAIL

```
apama.analyticskit.blocks.cumulocity.Send_Email
```

Sends an email to the specified email addresses.

The subject and text must each be provided using either a parameter or via an input port, not both.

Note: When running in simulation or test mode, the block logs the output instead of sending an email.

Parameters

Name	Description	Type	Notes
Subject	The subject of the email.	string	Optional
Text	The text of the email, which is processed as HTML. For example, if you want to insert a new line, use the tag.	string	Optional
Reply to	The reply-to address for the email.	string	Optional
To	The recipients of the email. One or more email addresses separated by commas.	string	Optional
CC	(Carbon copy) The recipients that are to receive a copy of the email. One or more email addresses separated by commas.	string	Optional
BCC	(Blind carbon copy) The recipients that are to receive a blind copy of the email. One or more email addresses separated by commas. Recipients listed in To and CC are not able to see the BCC list.	string	Optional

Input Port Details

Name	Description	Type
Send	Sends an email when a signal is received.	pulse
Subject	Subject of the email.	string
Text	Text of the email.	string

SEND SMS

```
apama.analyticskit.blocks.cumulocity.Send_SMS
```

Sends an SMS (Short Message Service) to the specified phone number.

If the Send SMS input port is connected, this block sends an output to the specified phone number when receiving a signal.

Note: When running in simulation or test mode, the block logs the output instead of sending an SMS.

Parameters

Name	Description	Type	Notes
Phone Number	The phone number to which the SMS is to be sent.	string	

Name	Description	Type	Notes
Text	The content of the SMS. The maximum length is 160 characters. Do not specify this parameter if using the SMS Text input port.	string	Optional

Input Port Details

Name	Description	Type
Send SMS	Triggers sending an SMS.	pulse
SMS Text	Content of the SMS. If not connected, specify the content in the Text parameter.	string

LOGIC

This category contains the following blocks:

Block Name	Description
AND	Performs a logical 'and' on the inputs.
NOT	Performs a logical 'not' on the input.
OR	Performs a logical 'or' on the inputs.

AND

`apama.analyticskit.blocks.core.And`

Performs a logical 'and' on the inputs.

If all the connected inputs are true, then the output is true, else false. If any of the inputs is of type pulse, then the output is a pulse. Any combination of pulse and boolean inputs is valid. The block functions even if some inputs are not connected.

Input Port Details

Name	Description	Type
Value 1	First input value to the block.	boolean
Value 2	Second input value to the block.	boolean
Value 3	Third input value to the block.	boolean
Value 4	Fourth input value to the block.	boolean
Value 5	Fifth input value to the block.	boolean

Output Port Details

Name	Description	Type
AND	The logical 'and' of the inputs.	pulseOrBoolean(value1, value2, value3, value4, value5)

NOT

`apama.analyticskit.blocks.core.Not`

Performs a logical 'not' on the input.

If the connected input is true, the output is false. If the connected input is false, the output is true.

Input Port Details

Name	Description	Type
Value	Input value to the block.	boolean

Output Port Details

Name	Description	Type
NOT	The logical 'not' of the input.	boolean

OR

`apama.analyticskit.blocks.core.Or`

Performs a logical 'or' on the inputs.

If any of the connected inputs is true, the output is true, else false. If all the inputs are of type pulse, the output is a pulse. Inputs must be of the same type. The block functions even if some inputs are not connected.

Input Port Details

Name	Description	Type
Value 1	First input value to the block.	boolean
Value 2	Second input value to the block.	boolean
Value 3	Third input value to the block.	boolean
Value 4	Fourth input value to the block.	boolean
Value 5	Fifth input value to the block.	boolean

Output Port Details

Name	Description	Type
------	-------------	------

Name	Description	Type
OR	The logical 'or' of the inputs.	sameAsAll(value1, value2, value3, value4, value5)

CALCULATION

This category contains the following blocks:

Block Name	Description
Crossing Counter	Detects and counts the number of threshold crossings in the specified direction.
Delta	Calculates the difference between successive input values.
Difference	Calculates the absolute and signed differences between the connected inputs.
Direction Detection	Detects whether the input value changes direction.
Expression	Evaluates an expression to perform arithmetic or logical calculations or string operations.
From Base N	Converts a base N string to a float.
KPI	Compares a value against either a KPI (Key Performance Indicator) or the data point of a device, asset or group of devices.
Limit	Outputs a value that is kept within the defined upper and lower limits.
Range	Compares the input value against the defined lower and upper range values to detect whether the input is within or out of the range, or whether it crosses the range.
Range Lookup	Finds the range in which the input value lies.
Rounding	Rounds the input to a specified number of decimal points or to an integer, using a selectable rule.
Threshold	Compares the input value against the defined threshold value to detect whether the input breaches the threshold or whether it crosses the threshold.
To Base N	Converts a float to a base N string.

CROSSING COUNTER

`apama.analyticskit.blocks.core.CrossingCounter`

Detects and counts the number of threshold crossings in the specified direction.

Crossing is defined as a change in the input value from one side of the threshold to the other side of the threshold (that is, from less than to greater than or vice versa).

The block can operate over a time-bounded window that is specified with the Window Duration parameter. If this parameter is not specified, it uses an unbounded window. The Reset input clears the window contents.

If a window is configured, the block uses a set of 20 buckets, so the time of expired values is an approximation to the nearest bucket interval.

Parameters

Name	Description	Type	Notes
Threshold Value	This value is compared against the input value.	float	Default: 0.5
Direction	The direction in which to check for a threshold crossing: whether to detect a crossing in the upwards direction, in the downwards direction, or in both directions.	Option - one of: <ul style="list-style-type: none"> • Upwards • Downwards • Both 	
Window Duration (secs)	If present, the amount of time (in seconds) for which values are to be kept in the window. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Value	The input value for which to detect a crossing.	float
Reset	Resets the count of crossings.	pulse
Sample	Forces re-evaluation of the current value and sends the output.	boolean

Output Port Details

Name	Description	Type
Crossing Count	The number of crossings.	float
Crossing	Sends a pulse when a crossing is detected.	pulse

DELTA

`apama.analyticskit.blocks.core.Delta`

Calculates the difference between successive input values.

The block generates an output after getting at least two input values.

Input Port Details

Name	Description	Type
Value	Value to calculate the delta.	float
Reset	Resets the state of the block.	pulse

Output Port Details

Name	Description	Type
Delta Value	The delta value.	float

DIFFERENCE

`apama.analyticskit.blocks.core.Difference`

Calculates the absolute and signed differences between the connected inputs.

Only generates an output if both inputs receive a value.

Input Port Details

Name	Description	Type
Value 1	First input to the block.	float
Value 2	Second input to the block.	float

Output Port Details

Name	Description	Type
Absolute Difference	The absolute difference of the inputs.	float
Signed Difference	The signed difference of the inputs. Positive if the Value 1 input is larger than the Value 2 input.	float

DIRECTION DETECTION

`apama.analyticskit.blocks.core.DirectionDetector`

Detects whether the input value changes direction.

Outputs the change in direction and the last significant inflection point, ignoring minor variations if the changes are less than the defined hysteresis value. Repeated inputs of the same value are ignored.

Parameters

Name	Description	Type	Notes
Hysteresis	Only counts a change in direction if the input value changes by the defined hysteresis value since the point of changing direction. Must either be zero (meaning all changes of direction are counted) or a positive number.	float	Default: 0.0

Input Port Details

Name	Description	Type
Value	Numeric value for which to detect the change in direction.	float

Name	Description	Type
Reset	Resets the state of the block.	pulse

Output Port Details

Name	Description	Type
Upward Direction	Is true if the input value changes towards the upward direction, else false.	boolean
Inflection Point	The last inflection point detected by the block.	float

EXPRESSION

`apama.analyticskit.blocks.core.Expression`

Evaluates an expression to perform arithmetic or logical calculations or string operations.

On change of input values (once all in-use inputs have been received), the expression specified in the parameter is re-calculated.

The expression language is much like EPL (see [EPL reference](#)), but is restricted to float, integer, string and boolean types.

Note: All numeric literals are treated as float type values, even if they have no fractional part. Integer values can only be obtained as the result of functions such as `floor()`. Similar to EPL, integer and float are not implicitly convertible within an expression. If the result of an expression is an integer value, it is converted to a float automatically (there might be a loss of precision).

Boolean values can be specified using the boolean literals `true` and `false`. Boolean literals are case insensitive, so for example, `TRUE` and `True` are allowed. String values can be specified by enclosing string literals in double quotes, for example `"my value"`. Special characters are encoded with a backslash (`\`). The following special characters (along with their encoding) are supported in string literals:

- Double quotes - `\"`
- Backslash - `\\`
- Newline - `\n`
- Tab - `\t`

Similar to EPL, each value type can be concatenated with a string type. The non-string value is first converted to a string and then appended to the string value.

The values of the inputs are available as `input1`, `input2`, `input3`, `input4` and `input5`. The input values can be of type float, string, boolean and any. Logical, relational, numerical and equality operators can be used on the values of the supported types. Logical operators are case insensitive, so for example, `AND` and `And` are allowed. Built-in methods on the float, integer, string and boolean types can be called, including `x.abs()` (absolute value of `x`), `x.pow(y)` (raise `x` to the power `y`), `x.sin()` (sine of `x` in radians), `x.round()` (rounds `x` to the nearest integer), and `s.ltrim()` (remove whitespace from the start of the string `s`). Built-in static methods of the supported types can be called by specifying the type name, followed by a dot (`.`) and the method name, for example, `float.max(input1, input2)` (find the larger of two input values). Built-in constants on the supported types can be accessed by specifying the type name, followed by a dot (`.`) and the constant name, for example, `float.E` (Euler's constant). Values of type `any` are unpacked at runtime to evaluate the expression. After unpacking, the value must be of type float, string or boolean. The type checker tries to validate the expressions during the validation phase, but this is not always possible with the `any` type. So if an expression contains the `any` type, even if it passes the validation phase, it can still fail at runtime due to a wrong type of variable being passed or an unsupported operation being performed. For a full list of built-in methods and constants, consult the [API Reference for EPL \(ApamaDoc\)](#).

Some examples:

- Convert Fahrenheit to Celsius: `(input1 - 32) * 5/9`
- Convert days to seconds: `input1 * 86400`
- Average of 4 inputs: `(input1 + input2 + input3 + input4) / 4`
- Threshold comparison: `input1 > 3.1412` (but also see the Threshold block)
- Pythagoras to compute the hypotenuse of a right-angled triangle: `(input1.pow(2) + input2.pow(2)).sqrt()`
- Comparison to 3 decimal places: `(input1 * 1000 - (input1 * 1000).fractionalPart()) = (input2 * 1000 - (input2 * 1000).fractionalPart())`
- Range check: `input1 >= 1 and input1 <= 10`
- String comparison: `input1 = "my value"`
- Larger value: `float.max(input1, float.PI)`

- Remainder of integer division: `input1.round() % input2.round()`
- Left shift of integer value: `input1.round() << 4.round()`
- Right shift of integer value: `input1.round() >> 2.round()`
- Bitwise not of integer value: `not input1.round()`
- Bitwise or of integer values: `input1.round() or input2.round()`
- Bitwise and of integer values: `input1.round() and input2.round()`
- Bitwise xor of integer values: `input1.round() xor input2.round()`
- String and non-string concatenation: `"Current temperature is " + input1 + " degrees Celsius"`

Parameters

Name	Description	Type	Notes
Expression	An expression - a string representation of an EPL expression. Similar to EPL expressions, but with the differences as described above.	string	

Input Port Details

Name	Description	Type
input1	First input, to be used as input1 in the expression.	any
input2	Second input, to be used as input2 in the expression.	any
input3	Third input, to be used as input3 in the expression.	any
input4	Fourth input, to be used as input4 in the expression.	any
input5	Fifth input, to be used as input5 in the expression.	any

Output Port Details

Name	Description	Type
Result	Result of the expression.	any

FROM BASE N

`apama.analyticsbuilder.blocks.FromBaseN`

Converts a base N string to a float.

The input string can be in any integer base from 2 to 36, where letters of the English alphabet are used as digits for bases above 10. Common bases are 2 (binary), 8 (octal), 10 (decimal) and 16 (hexadecimal). The number being converted can contain a radix point.

Conversion between two arbitrary bases can be achieved by chaining this block with the To Base N block.

Parameters

Name	Description	Type	Notes
Input Base	The number base of the input stream, in the range 2 to 36.	integer	Default: 16

Name	Description	Type	Notes
Radix Character	The character to use as the radix point. Expected to be a dot or a comma.	Option - one of: <ul style="list-style-type: none"> • Dot • Comma 	Default: Dot

Input Port Details

Name	Description	Type
Base N	String input in base N.	string

Output Port Details

Name	Description	Type
Float	Numeric output.	float

KPI

`apama.analyticskit.blocks.cumulocity.KPI`

Compares a value against either a KPI (Key Performance Indicator) or the data point of a device, asset or group of devices.

This block uses data from the KPI input port or from the device, asset or group of devices which contains data points. It extracts the units, label, and the red and yellow ranges. The output indicates whether the value is within the red or yellow range specified by the KPI or data point.

The KPI input can provide properties, typically from a KPI-managed object, which include the red and yellow ranges, the unit and the label. If the device contains a data point for the specified fragment and series, then the values from the data point override those from the KPI.

Parameters

Name	Description	Type	Notes
Input Source	The device, group, or asset which contains a data point. If specified, then this source (typically the same as the measurement source) is checked to see if it contains a data point for the specified fragment and series. If it contains a data point, the red and yellow range values from the source object are used in place of the KPI values. For the KPI block, this parameter must not be set to "All Inputs".	any	Optional
Data Point Fragment and Series	This parameter must be specified if the Input Source parameter is specified. It specifies a data point from the source object. This is typically the same as the fragment and series of the measurement source. The data point fragment and series must be specified as fragment.series.	string	Optional
Upper end of yellow range exclusive	If set, the upper end of the yellow range is treated as being exclusive.	boolean	Default: False

Name	Description	Type	Notes
Upper end of red range exclusive	If set, the upper end of the red range is treated as being exclusive.	boolean	Default: False

Input Port Details

Name	Description	Type
Value	Numeric value to compare with the defined ranges.	float
KPI	Object containing the c8y_Kpi property.	pulse

Output Port Details

Name	Description	Type
Red	Is set to true when the value is in the red range (if a red range is defined).	boolean
Yellow	Is set to true when the value is in the yellow range (if a yellow range is defined).	boolean
Unit	The unit name from the data point.	string
Label	The label name from the data point.	string

LIMIT

`apama.analyticsbuilder.blocks.Limit`

Outputs a value that is kept within the defined upper and lower limits.

The input value is limited so that the output does not exceed the boundaries defined by the Lower Limit and Upper Limit parameters. If the input violates either limit, then the output is set to the parameter value, otherwise the value is passed through unchanged.

It is only mandatory to provide one of the limits. If this is the case, then the input is only limited in the direction of the specified parameter.

Parameters

Name	Description	Type	Notes
Upper Limit	Any input above this value results in the output being capped at this value.	float	Optional
Lower Limit	Any input below this value results in the output being capped at this value.	float	Optional

Input Port Details

Name	Description	Type
Value	The input value.	float

Output Port Details

Name	Description	Type
Output	The input value, if it is within the limits defined by the Lower Limit and Upper Limit parameters. If the input value exceeds one of the limit parameters, it is set to the value of that parameter.	float

RANGE

`apama.analyticsbuilder.blocks.Range`

Compares the input value against the defined lower and upper range values to detect whether the input is within or out of the range, or whether it crosses the range.

By default, the range includes the value for the lower range but excludes the value for the upper range. For example, if the lower range is 100 and the upper range is 200, then all values from 100 to 199 are within the range. 200 is considered to be out of the range.

A pulse is sent when the defined range is crossed. That is, when either the lower or upper range is crossed, or if the value goes from below the range to over the range (or vice versa) without ever being within the range.

Parameters

Name	Description	Type	Notes
Lower Range	The lower range value.	float	Optional
Include Lower Range	If selected, an input value equal to the lower range is considered to identify whether it is within or out of the range. If not selected, an input value equal to the lower range is not considered.	boolean	Default: True
Upper Range	The upper range value.	float	Optional
Exclude Upper Range	If selected, an input value equal to the upper range is not considered to identify whether it is within or out of the range. If not selected, such an input value is considered to identify whether it is within or out of the range.	boolean	Default: True

Input Port Details

Name	Description	Type
Value	The input value to validate against the defined range.	float
Reset	Resets the state of the block.	pulse

Output Port Details

Name	Description	Type
Out of Range	Is set to true when the input value is not within the defined range.	boolean
In Range	Is set to true when the input value is within the defined range.	boolean
Crossed	Sends a pulse when the range is crossed.	pulse

RANGE LOOKUP

`apama.analyticskit.blocks.core.RangeLookup`

Finds the range in which the input value lies.

Ranges are defined using a list of unique upper bound values in increasing order. The lower bound of the first range depends on the value of the Minimum Value parameter. The lower bound of each subsequent range is defined by the upper bound of the previous range. The range to which an input value equal to a boundary value belongs depends on the Exclude Upper parameter. The block outputs the mapped value for the range the input lies within, or a failed flag if the input value is not within any of the ranges.

Parameters

Name	Description	Type	Notes
Minimum Value	The lower bound of the first range (first row). If nothing is specified, negative infinity is taken as the minimum value.	float	Optional
Ranges	A boundary and mapped value pair for the upper bound of a range for which to look up the input value and a mapped value that is to be used as the output value if the input value lies within the range.	List of Upper Bound value and Mapped Value	
Exclude Upper	If selected, an input value equal to the upper bound of a row is considered to be part of the range of the next row. If not selected, such an input value is considered to be part of the range of the current row.	boolean	Default: True
Type	The type that is to be used for the output value.	Option - one of: <ul style="list-style-type: none"> • Float • Boolean • String 	Optional

Input Port Details

Name	Description	Type
Value	The input value for which the range is to be found.	float

Output Port Details

Name	Description	Type
Mapped	The Mapped Value for the range that is matched.	any
Failed	Is true when the input does not lie in any range. Otherwise it is false.	boolean

ROUNDING

`apama.analyticskit.blocks.core.Rounding`

Rounds the input to a specified number of decimal points or to an integer, using a selectable rule.

Rounding a numerical value means replacing it by another value that is approximately equal but has a shorter and simpler representation.

The rules available for use are:

- Up (or take the ceiling, or round towards plus infinity) rounds the input up to the nearest target number.
- Down (or take the floor, or round towards minus infinity) rounds the input down to the nearest target number.
- Towards Zero (or truncate, or round away from infinity) rounds the input towards zero to the nearest target number.
- Nearest (or round half up, or round half towards positive infinity) rounds to the nearest target number. Numbers that are equidistant from the two nearest target numbers are always rounded up. For example, value 23.5 gets rounded to 24, but -23.5 gets rounded to -23.
- Even or Nearest rounds to the nearest target number. Numbers that are equidistant from the two nearest target numbers are always rounded to the nearest even target. For example, 0.5 rounds down to 0 and 1.5 rounds up to 2. Also known as Bankers Rounding.

The value is rounded to the nearest 'target number' - this is a whole number (if the number of decimal points is zero), or rounded to the number of decimal points specified. If the number of decimal points is negative, it is rounded to a power of 10. For example, if the number of decimal points is 2, it is rounded to the nearest 0.01 (that is, hundredths). If the number of decimal points is -3, it is rounded to the nearest 1000 (that is, thousands).

Parameters

Name	Description	Type	Notes
Rule	The rounding rule to be applied.	Option - one of: <ul style="list-style-type: none"> • Up • Down • Towards Zero • Nearest • Even or Nearest 	
Number of Decimal Points	The number of decimal points the input is to be rounded to.	integer	Default: 0

Input Port Details

Name	Description	Type
Value	The input value which is to be rounded.	float

Output Port Details

Name	Description	Type
Rounded Value	The input value rounded to a specified number of decimal points or to an integer.	float

THRESHOLD

`apama.analyticskit.blocks.core.Threshold`

Compares the input value against the defined threshold value to detect whether the input breaches the threshold or whether it crosses the threshold.

A breach occurs when the direction has been set to 'Above' and the input value is greater than the defined threshold value, or when the direction has been set to 'Below' and the input value is less than the defined threshold, or when the direction has been set to 'Above or Equal' and the input value is greater than or equal to the defined threshold value, or when the direction has been set to 'Below or Equal' and the input value is less than or equal to the defined threshold value.

A pulse is sent when the defined threshold value is crossed from any direction.

Parameters

Name	Description	Type	Notes
Threshold Value	This value is compared against the input value.	float	
Direction	The direction in which to look: whether the input value is above, below or equal to the defined threshold, or whether it crosses the defined threshold.	Option - one of: <ul style="list-style-type: none"> • Above • Above or Equal • Below • Below or Equal • Crossing 	Default: Above

Input Port Details

Name	Description	Type
Value	The input value to compare against the defined threshold value.	float
Reset	Resets the state of the block.	pulse

Output Port Details

Name	Description	Type
Breached Threshold	Is set to true when the threshold is breached.	boolean
Within Threshold	Is set to true when the threshold is not breached.	boolean
Crossed Threshold	Sends a pulse when the threshold is crossed.	pulse

TO BASE N

`apama.analyticsbuilder.blocks.ToBaseN`

Converts a float to a base N string.

The output string can be in any integer base from 2 to 36, where letters of the English alphabet are used as digits for bases above 10. Common bases are 2 (binary), 8 (octal), 10 (decimal) and 16 (hexadecimal). The number being converted can contain a radix point. The output is calculated to a maximum precision of 16 radix places.

Conversion between two arbitrary bases can be achieved by chaining this block with the From Base N block.

Parameters

Name	Description	Type	Notes
Output Base	The number base of the output stream, in the range 2 to 36.	integer	Default: 16

Name	Description	Type	Notes
Radix Character	The character to use as the radix point. Expected to be a dot or a comma.	Option - one of: <ul style="list-style-type: none"> • Dot • Comma 	Default: Dot

Input Port Details

Name	Description	Type
Float	Numeric input.	float

Output Port Details

Name	Description	Type
Base N	String output in base N.	string

AGGREGATE

This category contains the following blocks:

Block Name	Description
Average (Mean)	Calculates the mean of the values over time.
Counter	Gives a count of the total inputs and repeated inputs.
Discrete Statistics	Generates statistics of sum, count, average (mean), standard deviation, minimum and maximum for discrete input values.
Gradient	Calculates the weighted linear regression gradient for the values.
Group Statistics	Generates periodic aggregate values across all the devices in a group for which the block has received input values.
Integral	Calculates the integral of the input value over time.
Minimum / Maximum	Calculates the minimum and maximum of a value over time.
Standard Deviation	Calculates the standard deviation and variance of the values over time.

AVERAGE (MEAN)

`apama.analyticskit.blocks.core.Mean`

Calculates the mean of the values over time.

This block is suitable for continuous values, even if they are irregularly sampled. The time between inputs or samples is significant, while the number of samples is not. Use this block, for example, if the input is a physical property, such as temperature, that is sampled either

regularly or irregularly (for example, only generating measurement values on a change in temperature). Use the Discrete Statistics block instead of the Average (Mean) block for independent measurements, such as ticket sales, where the number of measurements is significant, but the time between measurements is not.

The mean is defined as the sum of the input's value multiplied by how long the input has stayed at that value, within an optional window, divided by the window duration or the time since the block was started or last reset, whichever is smallest.

The block can operate over a time-bounded window that is specified with the Window Duration parameter. If this parameter is not specified, the block uses an unbounded window. The Reset input port clears the window contents. Output is generated on any new input or, if the Output Threshold parameter is set, only when the output changes by more than the specified output threshold (which includes if no further input occurs, or the value only changes due to old entries expiring). The Sample input port can be used to force re-evaluation and generate the latest value.

See also the "Value types" topic in the Analytics Builder documentation for more details and an example of the frequency of output from this block and how values in windows behave.

If a window is configured, the block uses a set of 20 buckets, so the expired value is an approximation of the average value across a bucket.

Note: The Average (Mean) block generates the mean for an individual device. If the input comes from a range of devices, the mean is generated separately for each device in that group. To calculate and generate aggregate values for the group as a whole (not for individual devices), use the Group Statistics block.

Parameters

Name	Description	Type	Notes
Window Duration (secs)	If present, the amount of time (in seconds) for which values are to be kept in the window. This must be a finite and positive number.	float	Optional
Output Threshold	If present, the output to be sent at the point when it changes by at least this value. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Value	Input for which the mean is to be calculated.	float
Reset	Clears the content of the window.	pulse
Sample	Forces re-evaluation of the current mean value and sends the output.	pulse

Output Port Details

Name	Description	Type
Average	The sum of the value multiplied by how long it stays at that value divided by the total time.	float

COUNTER

`apama.analyticskit.blocks.core.Counter`

Gives a count of the total inputs and repeated inputs.

If two consecutive input values have different types, they are not evaluated as repeat values. For example, a float value of 1.0 will not

evaluate as a repeat of an integer value of 1. All other evaluations for whether two values are equal follow the same rules as EPL. For more information, refer to [Types](#).

You can specify a maximum value for each count independently. By default, each counter returns to one, counting the first input after the maximum has been reached. This is useful, for example, if you want to trigger an action on every n^{th} input, or if the input has repeated more often than expected. You can also set each counter to stop once it reaches the maximum value. In this mode, a pulse on the Reset input port is required to reset the count, although this resets both counters at the same time. If this dual reset is an issue, you must use a separate Counter block to deal with each count independently.

Parameters

Name	Description	Type	Notes
Maximum Count	The maximum value the Count output port can take. A negative or zero value is interpreted as float.MAX (~2e308). This only affects the Count output port. If the counter is looping, this can be useful for triggering an action on every n^{th} input.	float	Default: 0.0
Loop Count	If selected, the Count output port resets to one when the maximum value is exceeded. If not selected, the counter stops counting when the maximum value is reached and must be reset manually. This only affects the Count output port.	boolean	Default: True
Maximum Number Same	The maximum value the Number Same output port can take. A negative or zero value is interpreted as float.MAX (~2e308). This only affects the Number Same output port. This can be useful for assessing whether a value has remained static for an unexpectedly long time, such as positions within a GPS measurement.	float	Default: 0.0
Loop Number Same	If selected, the Number Same output port resets to one when the maximum value is exceeded. If not selected, the counter stops counting when the maximum value is reached and must be reset manually. This only affects the Number Same output port.	boolean	Default: True

Input Port Details

Name	Description	Type
Value	The input to be counted.	any
Reset	Resets both counters.	boolean

Output Port Details

Name	Description	Type
Count	The total count of all the inputs.	float
Number Same	The current count of repeated inputs with the same value. On a change of value, this is reset to one.	float

DISCRETE STATISTICS

`apama.analyticsbuilder.blocks.DiscreteStatistics`

Generates statistics of sum, count, average (mean), standard deviation, minimum and maximum for discrete input values.

This block is suitable for discrete time inputs, where the number of samples (or inputs) is significant, while the time between them is not. The Average (Mean) and Standard Deviation blocks are more suitable for continuous values that may be irregularly sampled, such as temperature readings. Use this block, for example, if each sample represents a transaction such as a ticket being sold.

If the Sample input port is not connected, every value is sampled. If a new value is received during the same activation period as a reset, the value is sampled after the reset, otherwise it is ignored. As every value is used, the standard deviation uses the generic formula: $\sigma^2 = \sum (x - \mu)^2 / N$.

If the Sample input port is connected, the block only samples the data when the Sample input port receives a signal. In this case, the sampling standard deviation uses the formula: $\sigma^2 = \sum (x - \mu)^2 / (N-1)$.

If reset and sample signals are received together, the reset is processed first.

Input Port Details

Name	Description	Type
Value	The current value.	float
Sample	Use the current value in generating statistics. If left unconnected, all values are used.	pulse
Reset	Reset the state of the block.	pulse

Output Port Details

Name	Description	Type
Sum	Sum of the sampled input values, $\sum x$.	float
Count	Count of the number of sampled input values, N .	float
Average	Average (mean) of the sampled input values, $\mu = \sum x / N$.	float
Standard Deviation	Standard deviation of the sampled input values. If all values are being sampled, with the Sample input port disconnected, the generic standard deviation, $\sigma^2 = \sum (x - \mu)^2 / N$, is used. Otherwise the sampling standard deviation, $\sigma^2 = \sum (x - \mu)^2 / (N-1)$, is used.	float
Minimum	Minimum of the sampled input values.	float
Maximum	Maximum of the sampled input values.	float

GRADIENT

`apama.analyticskit.blocks.core.Gradient`

Calculates the weighted linear regression gradient for the values.

A gradient measures the rate of change of a value over time. A positive gradient indicates an increase of the input values, and a negative gradient indicates a decrease of the input values. The magnitude of the gradient signifies the scale of change.

The block can operate over a time-bounded window that is specified with the Window Duration parameter. If this parameter is not specified, it uses an unbounded window and the block re-evaluates for every 1 second, and uses 1-second buckets. If a window is configured, the block uses a set of 20 buckets, so the time of expired values is an approximation to the nearest bucket interval. The first gradient output is generated only when a minimum of two buckets is available for computation.

The Reset input clears the content of the window. Sample input can be used to force re-evaluation and generate the latest value.

Parameters

Name	Description	Type	Notes
Window Duration (secs)	If present, the amount of time (in seconds) for which values are to be kept in the window. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Value	The input value for which the gradient is to be calculated.	float
Reset	Clears the content of the window.	pulse
Sample	Forces re-evaluation of the current value and sends the output.	pulse

Output Port Details

Name	Description	Type
Gradient	The gradient of the input values.	float

GROUP STATISTICS

`apama.analyticskit.blocks.core.GroupStatistics`

Generates periodic aggregate values across all the devices in a group for which the block has received input values.

This block generates the following aggregate values:

- Minimum
- Maximum
- Device Count
- Average
- Standard Deviation
- Variance

The block can operate over a time-bounded window that is specified with the Window Duration parameter. If this parameter is not specified, the block uses an unbounded window. Output is generated periodically as specified by the Output Period parameter.

If a window is configured, the block uses a set of 20 buckets, so the expired value is an approximation to the nearest bucket interval.

Note: The Group Statistics block calculates and generates aggregate values for the group as a whole (not for individual devices). To generate aggregates for an individual device in a group, use the Average (Mean), Standard Deviation, or Minimum/Maximum blocks. The Group Statistics block only considers devices from which it has received input values.

Parameters

Name	Description	Type	Notes
Window Duration (secs)	If present, the amount of time (in seconds) for which values are to be kept in the window. This must be a finite and positive number.	float	Optional

Name	Description	Type	Notes
Output Period (secs)	The amount of time (in seconds) between each output. This must be a finite and positive number.	float	Default: 5.0

Input Port Details

Name	Description	Type
Value	Input value for which the aggregate values are to be calculated.	float

Output Port Details

Name	Description	Type
Minimum	The smallest input value (closest to negative infinity) across all the devices in a group.	float
Maximum	The largest input value (closest to positive infinity) across all the devices in a group.	float
Device Count	The number of devices in a group for which input values have been received so far.	float
Average	The sum of the value multiplied by how long it stays at that value divided by the total time and the device count.	float
Standard Deviation	The standard deviation of the input values across all the devices in a group.	float
Variance	The variance of the input values across all the devices in a group.	float

INTEGRAL

`apama.analyticskit.blocks.core.Integral`

Calculates the integral of the input value over time.

Integral is defined as the sum of the input's value multiplied by how long the input has stayed at that value, within an optional window, since the block was started or last reset.

The block can operate over a time-bounded window that is specified with the Window Duration parameter. If this parameter is not specified, it uses an unbounded window. The Reset input clears the window contents. Output is generated on any new input or, if the Output Threshold parameter is set, only when the output changes by more than the specified output threshold (which includes if no further input occurs, or the value only changes due to old entries expiring). The Sample input can be used to force re-evaluation and generate the latest value.

If a window is configured, the block uses a set of 20 buckets, so the expired value is an approximation of the average value across a bucket.

Parameters

Name	Description	Type	Notes
Window Duration (secs)	If present, the amount of time (in seconds) for which values are to be kept in the window. This must be a finite and positive number.	float	Optional
Output Threshold	If present, the output to be sent at the point when it changes by at least this value. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Value	Input for which the integral is to be calculated.	float
Reset	Clears the content of the window.	pulse
Sample	Forces re-evaluation of the current integral value and sends the output.	pulse

Output Port Details

Name	Description	Type
Integral Value	The sum of the value multiplied by how long it stays at that value.	float

MINIMUM / MAXIMUM

`apama.analyticskit.blocks.core.MinMax`

Calculates the minimum and maximum of a value over time.

The minimum is defined as the smallest value (closest to negative infinity) of the input values in the window, and the maximum is defined as the largest value (closest to positive infinity) of the input values in the window.

The block can operate over a time-bounded window that is specified with the Window Duration parameter. If this parameter is not specified, the block uses an unbounded window. The Reset input clears the window contents. Output is generated on any new input that exceeds the current minimum or maximum, or, if the Window Duration parameter is set, when a previous minimum or maximum expires.

If a window is configured, the block uses a set of 20 buckets, so the time of expired values is an approximation to the nearest bucket interval.

Note: The Minimum/Maximum block generates the minimum and maximum for an individual device. If the input comes from a group of devices, these values are generated separately for each device in that group. To calculate and generate aggregate values for the group as a whole (not for individual devices), use the Group Statistics block.

Parameters

Name	Description	Type	Notes
Window Duration (secs)	If present, the amount of time (in seconds) for which values are to be kept in the window. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Value	Input for which the minimum and maximum is to be calculated.	float
Reset	Clears the content of the window.	pulse

Output Port Details

Name	Description	Type
Minimum	The smallest value in the window (closest to negative infinity).	float
Maximum	The largest value in the window (closest to positive infinity).	float

STANDARD DEVIATION

`apama.analyticskit.blocks.core.StandardDeviation`

Calculates the standard deviation and variance of the values over time.

This block is suitable for continuous values, even if they are irregularly sampled. The time between inputs or samples is significant, while the number of samples is not. Use this block, for example, if the input is a physical property, such as temperature, that is sampled either regularly or irregularly (for example, only generating measurement values on a change in temperature). Use the Discrete Statistics block instead of the Standard Deviation block for independent measurements, such as ticket sales, where the number of measurements is significant, but the time between measurements is not.

Standard deviation is a measure that is used to quantify the amount of variation or dispersion of a set of data values. A low standard deviation indicates that the data points tend to be close to the mean of the set, while a high standard deviation indicates that the data points are spread out over a wider range of values.

The block can operate over a time-bounded window that is specified with the Window Duration parameter. If this parameter is not specified, the block uses an unbounded window. The Reset input port clears the window contents. The Sample input port can be used to force re-evaluation and generate the latest value. Output is generated on any new input or, if the Window Duration parameter is set, output is generated periodically on every new bucket that is added to the window.

If a window is configured, the block uses a set of 20 buckets, so the time of expired values is an approximation to the nearest bucket interval.

Note: The Standard Deviation block generates the standard deviation and variance for an individual device. If the input comes from a group of devices, these values are generated separately for each device in that group. To calculate and generate aggregate values for the group as a whole (not for individual devices), use the Group Statistics block.

Parameters

Name	Description	Type	Notes
Window Duration (secs)	If present, the amount of time (in seconds) for which values are to be kept in the window. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Value	Input for which the standard deviation and variance is to be calculated.	float
Reset	Clears the content of the window.	pulse

Name	Description	Type
Sample	Forces re-evaluation of the current value and sends the output.	pulse

Output Port Details

Name	Description	Type
Standard Deviation	The standard deviation of the input values.	float
Variance	The variance of the input values.	float

FLOW MANIPULATION

This category contains the following blocks:

Block Name	Description
Combiner	Calculates the output based on the selected mode and the connected inputs.
Gate	Blocks the input from going to output unless the gate is open and enabled.
Latch Values	Latches the latest input value received while the block is enabled.
Pulse	Converts a non-pulse input into a pulse output.
Selector	Outputs a parameter value depending on which input port has a true value, lowest number taking precedence.
Switch	Outputs the values from a given input, or acts as a circuit breaker.
Time Delay	Delays the input by the specified amount of time.

COMBINER

`apama.analyticskit.blocks.core.Combiner`

Calculates the output based on the selected mode and the connected inputs.

Available modes are:

- Minimum: Outputs the minimum of the connected inputs which have received a value. All numeric literals are treated as float type.
- Maximum: Outputs the maximum of the connected inputs which have received a value. All numeric literals are treated as float type.
- Average (Mean): Outputs the average (mean) of the connected inputs for which a value has been received. All numeric literals are treated as float type.
- Latest Changed: Outputs the latest changed value. If multiple values change in a single activation, then the input port with the highest number is used. For example, if Value 1 and Value 2 get an updated value, Value 2 is selected for output. Inputs must be of the same type. When using the Latest Changed mode, a value is not considered changed if the actual value does not change. For example, when using the Latest Changed mode with temperature sensors, it outputs the sensor value whose temperature was most recently changed.
- Latest Input: Outputs the latest input value even if the actual value is unchanged. If multiple values update in a single activation, then the input port with the highest number is used. For example, if you have a number of sensors that measure the temperature

periodically at different intervals and you want to get the latest temperature received by the block by all of those sensors, you can use the Combiner block with the Latest Input mode. This provides the latest published temperature, even if a sensor measures and publishes the same temperature.

Parameters

Name	Description	Type	Notes
Mode	The mode to be selected.	Option - one of: <ul style="list-style-type: none"> • Minimum • Maximum • Average (Mean) • Latest Changed • Latest Input 	

Input Port Details

Name	Description	Type
Value 1	First input value to the block.	any
Value 2	Second input value to the block.	any
Value 3	Third input value to the block.	any
Value 4	Fourth input value to the block.	any
Value 5	Fifth input value to the block.	any

Output Port Details

Name	Description	Type
Combined Value	The calculated combined value.	sameAsAll(value1, value2, value3, value4, value5)

GATE

`apama.analyticskit.blocks.core.Gate`

Blocks the input from going to output unless the gate is open and enabled.

The block starts disabled if the Enable input is connected. Otherwise, the block is always enabled. The block starts closed if the Open input is connected. Otherwise, the block starts open.

Parameters

Name	Description	Type	Notes
------	-------------	------	-------

Name	Description	Type	Notes
Null Value	The value to use as output value when the gate is either disabled or closed. The exact type of the value depends on the Null Value Type parameter. If the Null Value Type parameter is specified then the string value is parsed as the specified type. If the Null Value Type parameter is not specified, and the value is parsable into a float or a boolean value then it is parsed into the corresponding value type, otherwise it is used as a string value.	string	Optional
Null Value Type	The type of the value specified by the Null Value parameter. If specified, the null value is parsed into the specified type.	Option - one of: <ul style="list-style-type: none"> • Float • Boolean • String 	Optional
Close Duration (secs)	The amount of time (in seconds) the gate should be closed when a close signal is received. If the parameter is not specified, then after a close signal is received, the gate remains closed until an open signal is received. If the parameter is specified, then the gate automatically opens after the specified number of seconds, unless another open or close signal is received before the time has elapsed. On opening, if the gate is also enabled, then the latest input value is sent out as output. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Value	The value to pass to output.	any
Enable	Enables the gate. If the gate is always to be enabled, then leave this port unconnected.	boolean
Open	Opens the gate. If the gate is to be open at the start, then leave this port unconnected.	pulse
Close	Closes the gate. If the Close Duration parameter is specified, then the gate is closed only for the specified number of seconds.	pulse

Output Port Details

Name	Description	Type
Gated Value	Same as the input value when the gate is open and enabled. Otherwise the value specified by the Null Value parameter.	input(value)
Activated	Output pulse. Generated when the gate becomes active. The gate is active when it is both open and enabled.	pulse
Deactivated	Output pulse. Generated when the gate becomes inactive. The gate is inactive when it is either closed or disabled.	pulse

LATCH VALUES

`apama.analyticskit.blocks.core.Latch`

Latches the latest input value received while the block is enabled.

Only generates an output if the input value changes and the block is enabled. The block starts disabled if the Enable input is connected. Otherwise, the block is always enabled.

Input Port Details

Name	Description	Type
Value	The input value to be monitored.	any
Enable	Enables the block. If the block is always to be enabled, then leave this unconnected.	boolean

Output Port Details

Name	Description	Type
Latched Value	Same as the input value. Generated only if the input has changed while the block was enabled.	input(value)
Changed	Output pulse. Generated only if the input has changed while the block was enabled.	pulse
Enabled	Output pulse. Generated when the block is enabled but was previously disabled.	pulse
Disabled	Output pulse. Generated when the block is disabled but was previously enabled.	pulse

PULSE

`apama.analyticskit.blocks.core.Pulse`

Converts a non-pulse input into a pulse output.

The block can be configured to send a pulse when the value changes which is also the default conversion behavior, on every input, or on every non-zero value.

The default conversion behavior sends a pulse if the input is a string or float and the value changes, or if the input is a boolean and the value changes to true.

This is useful with blocks which consume both pulse and non-pulse values, and where the input value is treated as non-pulse without the explicit conversion.

For example, a numeric value passed to the OR block is treated as true if non-zero (as described in the "Type conversions" topic of the Analytics Builder documentation). However, when passing a numeric value to the Pulse block and then connecting the output of the Pulse block to the OR block, the numeric value is converted to a pulse so the OR block sends a pulse.

Parameters

Name	Description	Type	Notes
------	-------------	------	-------

Name	Description	Type	Notes
Mode	Controls when the block sends a pulse.	Option - one of: <ul style="list-style-type: none"> • On value change (excluding to false) • On every input • On non-zero values 	Default: On value change (excluding to false)

Input Port Details

Name	Description	Type
Value	The input to treat as pulse.	any

Output Port Details

Name	Description	Type
Pulse	The output pulse value converted from the input value.	pulse

SELECTOR

`apama.analyticsbuilder.blocks.Selector`

Outputs a parameter value depending on which input port has a true value, lowest number taking precedence.

You specify the output value that is to be sent using the parameters of this block. Only one of the parameter values is sent in the output. It is sent when the corresponding input port receives a true value. If more than one input port receives a true value, then the input port is used which has the lowest number in its name. For example, Input 1 has a higher priority than Input 2.

If all input values are false, then the value specified with the No Input parameter is sent.

Example: Input 1 has "high", Input 2 has "medium", Input 3 has "low", and the No Input parameter has the value "off". If none of the input ports receives a true value, then "off" is sent as the output value. If both the Input 2 and Input 3 ports receive a true value and Input 1 receives a false value, then "medium" is sent as the output value. This is because Input 2 has a higher priority than Input 3.

Parameters

Name	Description	Type	Notes
Input 1	The output to be sent when the Input 1 port receives a true value.	string	
Input 2	The output to be sent when the Input 2 port receives a true value, but only if this is the port with the lowest number in its name.	string	Optional
Input 3	The output to be sent when the Input 3 port receives a true value, but only if this is the port with the lowest number in its name.	string	Optional
Input 4	The output to be sent when the Input 4 port receives a true value, but only if this is the port with the lowest number in its name.	string	Optional

Name	Description	Type	Notes
Input 5	The output to be sent when the Input 5 port receives a true value, but only if this is the port with the lowest number in its name.	string	Optional
No Input	The output to be sent when there is no input which has a true value.	string	
Type	<p>How to interpret the parameter values and set the output type. If JSON is selected as the type, all input parameter values must be valid JSON values of the same type. This allows the Selector block to output properties which can be used, for example, with the Set Properties block to generate different sets of properties for the output blocks, depending on which input port of the Selector block is enabled.</p> <p>If the Type parameter remains unselected, the type of the output value is set based on the types of all input parameter values. If all values are either true or false, the output is a boolean value. If all values are numbers, the output is a float value. In all other cases, the output is a string value.</p>	Option - one of: <ul style="list-style-type: none"> • Float • Boolean • String • JSON 	Optional

Input Port Details

Name	Description	Type
Input 1	Causes the Input 1 parameter value to be sent if true.	boolean
Input 2	Causes the Input 2 parameter value to be sent if true (and no lower numbered input is true).	boolean
Input 3	Causes the Input 3 parameter value to be sent if true (and no lower numbered input is true).	boolean
Input 4	Causes the Input 4 parameter value to be sent if true (and no lower numbered input is true).	boolean
Input 5	Causes the Input 5 parameter value to be sent if true (and no lower numbered input is true).	boolean

Output Port Details

Name	Description	Type
Output	The output value from one of the parameters.	any

SWITCH

`apama.analyticsbuilder.blocks.Switch`

Outputs the values from a given input, or acts as a circuit breaker.

If the Selected Input parameter is specified, then the given input must exist and the corresponding input port must be connected.

If the Selected Input parameter is not specified, then the block acts as a circuit breaker.

You can use the initial default names for the inputs. However, you can also rename them to be more descriptive. The input names must be unique, so two inputs cannot share the same name. Connected inputs must all be of the same type.

The expected use case is that the Selected Input parameter is set to a template parameter which can then be set individually for each model instance to decide which input is used.

Parameters

Name	Description	Type	Notes
Selected Input	To specify an output, specify one of the five input names. To act as a circuit breaker, leave this parameter empty.	string	Optional
Name for input1	The name of the first input to the block.	string	Default: input1
Name for input2	The name of the second input to the block.	string	Default: input2
Name for input3	The name of the third input to the block.	string	Default: input3
Name for input4	The name of the fourth input to the block.	string	Default: input4
Name for input5	The name of the fifth input to the block.	string	Default: input5

Input Port Details

Name	Description	Type
input1	The first input.	any
input2	The second input.	any
input3	The third input.	any
input4	The fourth input.	any
input5	The fifth input.	any

Output Port Details

Name	Description	Type
Switch	The value from the specified input, or no value (circuit breaker) if the Selected Input parameter is empty.	sameAsAll(input1, input2, input3, input4, input5)

TIME DELAY

`apama.analyticskit.blocks.core.TimeDelay`

Delays the input by the specified amount of time.

Parameters

Name	Description	Type	Notes
Delay (secs)	The amount of time (in seconds) by which the input is delayed. This must be a finite and positive number.	float	

Input Port Details

Name	Description	Type
Value	The input value to be delayed.	any

Output Port Details

Name	Description	Type
Delayed Value	The delayed output value.	input(value)

UTILITY

This category contains the following blocks:

Block Name	Description
Constant Value	Outputs a value, either when the Trigger input port receives a signal or at startup.
Cron Timer	Sends a signal output based on cron-like periodic timer syntax.
Duration	Measures the time elapsed from a set start time.
Extract Property	Extracts the specified property from the input value and converts it to the specified type.
Geofence	Compares the input value against the defined geofence value to detect whether the device is within the geofence, and whether the device entered or exited the geofence.
Missing Data	Generates an output if the input has not occurred for a set amount of time.
Set Properties	Outputs a pulse with properties set from values on the input ports.
Text Substitution	Substitutes identifiers marked with a hash and braces (for example, <code>#{name}</code>) in the text template with corresponding entries from the input values.
Toggle	Converts two pulse inputs to a boolean output based on the set and reset signals, with optional delays.

CONSTANT VALUE

`apama.analyticsbuilder.blocks.ConstantValue`

Outputs a value, either when the Trigger input port receives a signal or at startup.

The Trigger input port can be used to delay the output until a trigger input is received. If the Trigger input port is not connected, then the

block outputs a value when the model is activated, which may trigger further processing.

The Value parameter can be treated as a string, boolean, float or JSON value. If treated as a JSON string, number or boolean, then the output is of this type. If treated as a JSON object, the output is a pulse with the properties from the object. JSON arrays are only permitted within an object.

Parameters

Name	Description	Type	Notes
Value	The value to output.	string	
Type	How to interpret the Value parameter and set the output type. If JSON is selected as the type, the Value parameter must be a valid JSON value. If the Type parameter remains unselected, the type of the output value is set based on the type of the input parameter value. If the value is either true or false, the output is a boolean value. If the value is a number, the output is a float value. If the value is a valid JSON, the output is a pulse with the properties from the object. In all other cases, the output is a string value.	Option - one of: <ul style="list-style-type: none"> • Float • Boolean • String • JSON 	Optional

Input Port Details

Name	Description	Type
Trigger	If connected, send output on this trigger.	boolean

Output Port Details

Name	Description	Type
Output	The output value from the Value parameter.	any

CRON TIMER

`apama.analyticskit.blocks.core.CronTimer`

Sends a signal output based on cron-like periodic timer syntax.

The timer tick output is sent every time the time matches the pattern. The Time Zone parameter is a case-sensitive string that can be set to any supported time zone. The other parameters can be set to an asterisk (*) to match all times, a list of comma-separated numbers to fire at those specific times, or "*" / number" to fire on every multiple of the number.

For example:

To send a signal every Wednesday at 20:30:

- Day of Week: 4
- Hour: 20
- Minute: 30

To send a signal every 5 seconds:

- Seconds: * / 5

To send a signal on Europe/Berlin time zone:

- Time Zone: Europe/Berlin

The Days of Month and Days of Week parameters operate together such that if both are specified, then any match in either parameter triggers an output. For example, if Days of Month is set to 15 and Days of Week is set to 1, then there is an output on every Monday of the month and the 15th regardless of which day that is.

This block is not supported in simulation mode.

Parameters

Name	Description	Type	Notes
Seconds	The seconds at which to trigger the signal. Range: 0 to 59.	string	Default: 0
Minutes	The minutes at which to trigger the signal. Range: 0 to 59.	string	Default: *
Hours	The hours at which to trigger the signal. Range: 0 to 23.	string	Default: *
Days of Month	The days of the month on which to trigger the signal. Range: 1 to 31.	string	Default: *
Months	The months in which to trigger the signal. Range: 1 to 12.	string	Default: *
Days of Week	The days of the week on which to trigger the signal. Range: 0 (Sunday) to 6 (Saturday).	string	Default: *
Time Zone	The time zone to be used. Example: Europe/Berlin. For a full list, see the Supported time zones . The default value is empty which means that the platform's time zone is used.	string	Default:

Output Port Details

Name	Description	Type
Timer Tick	Sent at each trigger time based on the cron parameters.	pulse

DURATION

`apama.analyticskit.blocks.core.Duration`

Measures the time elapsed from a set start time.

The start time is set by a start signal which activates the block. If the block is already active, then a start signal is ignored and the existing measurement remains unaffected. The block is deactivated with a reset signal which also disables any periodic outputs.

The block generates a float output of the time elapsed since the start signal, measured in seconds. If the block is inactive at the time it receives a measure signal, then 0.0 is generated as the output.

If multiple signals are received at the same time, they are processed in the order of measure, reset and start. Thus, for example, if measure and reset signals are received together, the block first generates an output with the current duration and then resets its state. Processing the inputs in this order allows the current state of the block to be output and the block to be restarted within a single unit of time, if desired.

Parameters

Name	Description	Type	Notes
Periodic Output (secs)	The amount of time (in seconds) between automatically generated outputs by an active block. If not set, then an output is only generated when a measure signal is received. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Measure	Triggers an output of the duration (in seconds) since the block was activated. If the block is inactive, then 0.0 is output.	pulse
Reset	Deactivates and resets the state of the block.	pulse
Start	Activates the block.	pulse

Output Port Details

Name	Description	Type
Duration	The time in seconds since a start signal was received.	float

EXTRACT PROPERTY

`apama.analyticskit.blocks.core.ExtractProperty`

Extracts the specified property from the input value and converts it to the specified type.

The value in the Value input named by the Property Path parameter must be a string, number or boolean.

You can specify square brackets as part of the Property Path parameter to extract a specific element from a sequence.

You can also specify a period (.) as part of the Property Path parameter to extract nested values from a dictionary.

If you want to ignore the periods (.) and square brackets in the Property Path parameter and treat them as any other character, then you must select the Ignore Separators In Property Path checkbox.

For example:

- If the input is { "users" : [{ "age" : 40.375 }] } (in JSON form), then you can extract the value of age by specifying users[0].age as the Property Path parameter.
- If the input is { "location" : { "city" : "Cambridge" } } (in JSON form), then you can extract the value of city by specifying location.city as the Property Path parameter.
- If the input is { "location.city" : "Cambridge" } (in JSON form), then you can extract the value of location.city by specifying location.city as the Property Path parameter and selecting the Ignore Separators In Property Path checkbox.
- If the input is { "a.b" : { "c.d" : "foo" } } (in JSON form) and you want to extract the value of c.d, then you must use two Extract Property blocks in a chain, both with the Ignore Separators In Property Path checkbox selected: the first block to extract the value of a.b and the second block to extract the value of c.d.

If the value is an object, then the properties of that object are output as properties on the Extracted Value output port.

In converting a string to a float, this block treats an empty string as a value of 0.0, rather than as not parseable.

Note: To extract a custom property from the Measurement Input block, you must add the prefix measurement_ to the name of the property in the Property Path parameter of the Extract Property block. For example, if the name of the custom property you want to extract is city, you must specify measurement_city as the Property Path parameter.

Parameters

Name	Description	Type	Notes
Property Path	The name or path of the property that is to be extracted from the input value. If not set, all of the properties are output.	string	Optional
Property Type	The type to which the property value is to be converted. If set to Properties, a pulse is output with properties from the extracted value.	Option - one of: <ul style="list-style-type: none"> • String • Boolean • Float • Properties 	Default: String
Clear On Missing	If selected, the default value of the specified type is output if the Property Path parameter was not specified or if the value cannot be converted into the specified type.	boolean	Default: False
Ignore Separators In Property Path	If selected, the value of the Property Path parameter is treated as a property name. For example, if Ignore Separators In Property Path is selected and the Property Path parameter is specified as location.city, then the property name location.city is extracted from the input value.	boolean	Default: False

Input Port Details

Name	Description	Type
Value	The input value from which the property is to be extracted.	any

Output Port Details

Name	Description	Type
Extracted Value	The value that has been extracted from the input value.	any

GEOFENCE

`apama.analyticsbuilder.blocks.Geofence`

Compares the input value against the defined geofence value to detect whether the device is within the geofence, and whether the device entered or exited the geofence.

The boundary line itself is considered to be outside of the geofence area.

Parameters

Name	Description	Type	Notes
Geofence	A polygon describing an area on the Earth's surface. Note that crossing the international date line, namely the longitude of +/- 180 degrees, is not supported.	sequence<apama.analyticsbuilder.LngLat>	

Input Port Details

Name	Description	Type
Position	The input value from which the properties for the longitude (lng) and latitude (lat) are to be extracted.	any

Output Port Details

Name	Description	Type
Within Geofence	Is set to true when the device is within the defined geofence.	boolean
Entered Geofence	Sends a signal when the device enters the defined geofence.	pulse
Exited Geofence	Sends a signal when the device leaves the defined geofence.	pulse

MISSING DATA

`apama.analyticskit.blocks.core.MissingData`

Generates an output if the input has not occurred for a set amount of time.

The block can optionally detect an absence of changes in value if repeated same-value inputs are not counted.

Parameters

Name	Description	Type	Notes
Ignore Repeated Inputs	If selected, same-value inputs are ignored.	boolean	Default: False
Duration (secs)	The amount of time (in seconds) the block waits for an input. This must be a finite and positive number.	float	

Input Port Details

Name	Description	Type
Value	The data whose presence or absence is being detected.	any
Reset	Resets the state of the block - resets any timer and the previous value.	pulse

Output Port Details

Name	Description	Type
Missing Data	Absence of data, true if no input is received within the specified duration, else false.	boolean

SET PROPERTIES

`apama.analyticsbuilder.blocks.SetProperties`

Outputs a pulse with properties set from values on the input ports.

The property names are taken from the parameters and the values from the input ports. New properties are only output if they have been received. An output is sent as soon as at least one of the inputs is provided.

Properties are set on the output in the following order of precedence:

1. Any properties which have been explicitly specified by the use of a parameter. If an input is a pulse, it is treated as an object using the properties of that input. If an input has a primary value (not a pulse), then the primary value is used. To use the properties instead, use the Extract Property block with the Property Path parameter not set and the Property Type parameter set to Properties which replaces the primary value with a pulse (which is ignored) and the properties are used.
2. The properties of any Value object on an input port which does not have the corresponding parameter set. This is a straight merge of the properties dictionary. If two dictionaries have the same property key, then the input port with the lowest identifier has precedence. Thus any shared properties on Input 1 overwrite properties from Input 2 and down.
3. Any properties on a Value object provided to the Merge input port are kept if they are not overwritten by either of the operations above. The optional Merge input port allows chaining or supplementing a set of properties from another block.

Thus, any properties set on an input are overwritten by those with the same name on a higher precedence input, or when an input is configured for the specified property.

Parameters

Name	Description	Type	Notes
Property 1	A property to set in the output, using the value from input port 1.	string	Optional
Property 2	A property to set in the output, using the value from input port 2.	string	Optional
Property 3	A property to set in the output, using the value from input port 3.	string	Optional
Property 4	A property to set in the output, using the value from input port 4.	string	Optional
Property 5	A property to set in the output, using the value from input port 5.	string	Optional

Input Port Details

Name	Description	Type
Input 1	Value to be added using the property name in parameter Property 1.	any
Input 2	Value to be added using the property name in parameter Property 2.	any
Input 3	Value to be added using the property name in parameter Property 3.	any
Input 4	Value to be added using the property name in parameter Property 4.	any
Input 5	Value to be added using the property name in parameter Property 5.	any
Merge	Source to merge with the specified properties. Properties from here not replaced by an input value are passed through to the output.	any

Output Port Details

Name	Description	Type
Output	The output value with extra properties supplied.	pulse

TEXT SUBSTITUTION

`apama.analyticsbuilder.blocks.TextSubstitution`

Substitutes identifiers marked with a hash and braces (for example, `{name}`) in the text template with corresponding entries from the input values.

At least one of the Object or Source input ports must be connected. Identifiers that cannot be resolved are not substituted.

The identifiers prefixed with source. (for example, `{source.name}`) are searched for in the value received on the Source input port. For example, if the value received on the Source input port is `{ "name": "sample_name" }`, then the identifier `{source.name}` is resolved to `sample_name`.

The identifiers not prefixed with source. are searched for in the value received on the Object input port.

Nested identifiers can be specified by separating them with a dot (.). For example, when the Object input port has received the value `{ "address": { "street": { "name": "example_street" } } }`, then the identifier `{address.street.name}` is resolved to `example_street`.

Keys with a dot (.) in them are not supported, so if the Object input port value is of the form `{ "address.street": { "name": "example_street" } }`, then the `example_street` value cannot be resolved because the identifier `{address.street.name}` expects street to be nested inside the address entry.

Primitive values such as integer, float, boolean and string are substituted directly, but complex values are converted to a JSON representation before substitution.

Any identifier with the text time (case-insensitive) in it and value type float is interpreted as a timestamp value and is converted into the format `yyyy-MM-ddTHH:mm:ss.SSSZ` before substitution. This can be modified with optional parameters using the following syntax: `{time:param1="value1",param2="value2"}`. Use the parameter `TZ="time_zone"` to specify a different time zone and/or use the parameter `FORMAT="format_string"` to specify a different format. For example, `{time:TZ="America/New_York",FORMAT="HH:mm:ssZ"}` specifies the time zone for New York and the format to be `HH:mm:ssZ`. The model fails to activate if the time zone is not recognized or the format is invalid.

Note: The format string for the time must not contain quotes (") and braces ({ and }).

A hash (#) can be specified in the text template by escaping it as follows: `{#}`.

For more information, see the [list of time zones](#) and the [list of valid time format strings](#).

Parameters

Name	Description	Type	Notes
Text Template	The text that is used to generate the output by substituting the identifiers in it, such as <code>{name}</code> , with the values from the input ports.	string	

Input Port Details

Name	Description	Type
Object	Used to substitute identifiers that are not prefixed with source.. For example, <code>{name}</code> or <code>{timestamp}</code> .	any
Source	Used to substitute identifiers that are prefixed with source.. For example, <code>{source.name}</code> .	any

Output Port Details

Name	Description	Type
Output	String containing the substitutions from the text template.	string

TOGGLE

`apama.analyticskit.blocks.core.Toggle`

Converts two pulse inputs to a boolean output based on the set and reset signals, with optional delays.

Without delays, the output state is changed to true on a set signal, and changed to false on a reset signal. If both signals are received at the same time, the output state is toggled (to true if signals are received for the first time).

If delay times are specified, the output state is only changed to true or false after the delay has been applied.

The following exception applies if both signals are received at the same time: the output state is only toggled if both delay times are the same, or have not been specified at all.

Parameters

Name	Description	Type	Notes
Set Delay (secs)	The amount of time (in seconds) after which the set signal is processed. If the parameter is not specified, then the signal is immediately processed. This must be a finite and positive number.	float	Optional
Reset Delay (secs)	The amount of time (in seconds) after which the reset signal is processed. If the parameter is not specified, then the signal is immediately processed. This must be a finite and positive number.	float	Optional

Input Port Details

Name	Description	Type
Set	Sets the output to true. Any pending delayed set or reset signals are cancelled on the new input.	pulse
Reset	Sets the output to false. Any pending delayed set or reset signals are cancelled on the new input.	pulse

Output Port Details

Name	Description	Type
Value	The output value generated from the input signals.	boolean

SMART RULES (NEW) PLUGIN

GETTING STARTED WITH THE SMART RULES (NEW) PLUGIN

WHAT IS THE SMART RULES (NEW) PLUGIN

The smart rules plugin extends the Analytics Builder capabilities within Streaming Analytics by enabling users to create and manage Analytics Builder model instances directly from device and group contexts within applications such as Device Management, Cockpit, and Digital Twin Manager.

For more details about Analytics Builder, refer to [Analytics Builder](#).

PREREQUISITES

Before using the smart rules (NEW) plugin, ensure that the following requirements are met:

- The tenant is subscribed to the Streaming Analytics microservice that supports the Analytics Builder capability.
- The smart rules plugin is installed in the desired application, and the **Smart rules (NEW) plugin** is enabled in **Preview feature**.
- The user has the required privileges to access Analytics Builder models and create new instances of Analytics builder templated models. Refer to [Permissions](#).

Managing permissions to smart rules (NEW) instances

The **Smart rule instances** permission under [global roles](#) allows you to manage permissions required to access smart rules (NEW) instances.

Permission	Description
READ	Grants read-only access to smart rule instances. Users with this permission can view existing instances but cannot create, update, or delete them.
ADMIN	Grants permission to create, read, update, and delete (CRUD) smart rule instances.

These permissions ensure fine-grained access control, allowing administrators to manage user privileges based on operational requirements.

The permissions described above grant read-only access to the underlying Analytics Builder model. Administrative actions on the Analytics Builder models (like create, edit, or delete) require the *CEP management* permissions, see [permissions](#).

CREATING YOUR FIRST SMART RULE

This topic provides the basic workflow for creating your first smart rules using the plugin from applications like Device Management. You will create a simple templated analytics model that creates an alarm when the temperature breaches a given threshold value. The steps below require that at least one device is already registered in Cumulocity. Preferably, this device already sends measurement values to Cumulocity.

The model that you add will contain three blocks:

- An input block which receives measurements from devices, groups, or assets.
- A threshold block which verifies the measurement has breached the threshold value.
- An alarm output block which creates an alarm object for specified devices, groups, or assets.

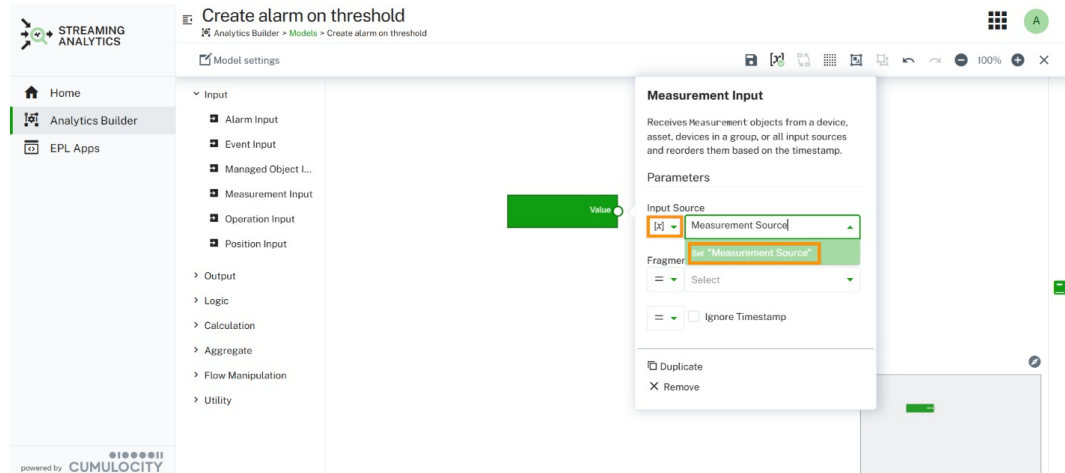
Step 1: Create an Analytics model

This section walks you through creating a simple temperature monitoring model. For more detailed Analytics Builder information, refer to

Understanding models.

1. Open the Streaming Analytics application and navigate to **Analytics Builder > Models**.
2. On the top bar, click **New Model**, enter a model name (for example, "Create alarm on threshold") and click **OK**.
3. From the palette on the left, expand **Input** and drag the **Measurement Input** block onto the canvas. In the block parameter editor:
 - For **Input Source**, select "Template parameter" from the dropdown [x] ▼, then provide a name. For example,

"Measurement Source".



- For **Fragment and Series**, select "Template parameter" from the dropdown, then provide a name. For example: "Input Fragment and Series".
4. Expand **Calculation** and drag the **Threshold** block onto the canvas. In the block parameter editor:
 - For **Threshold Value**, select "Template parameter" from the dropdown, then provide a name. For example: "Threshold Value".
 5. Expand **Output** and drag the **Alarm Output** block onto the canvas. In the block parameter editor:
 - For **Output Destination**, select "Template parameter" from the dropdown, then use the same template parameter given for **Input Source** of the **Measurement Input** block. For example: "Measurement Source".
 - For **Alarm Type**, select "Template parameter" from the dropdown, then provide a name. For example: "Alarm Type".
 - For **Message**, select "Template parameter" from the dropdown, then provide a name. For example: "Alarm Text".
 - For **Severity**, select "Template parameter" from the dropdown, then provide a name. For example: "Alarm Severity".
 6. Connect the blocks by clicking and dragging between their ports (small circles on the sides of blocks):
 - Connect the **Value** output of the **Measurement Input** to the **Value** input of the **Threshold**.
 - Connect the **Breached** output of the **Threshold** to the **Create Alarm** input of the **Alarm Output**.
 7. In the toolbar of the model editor, click the template parameter icon [x] to open the template parameter dialog. Ensure the template parameter entry provided for **Input Source** of the **Measurement Input** block (example : "Measurement Source") has "Source or Destination" updated to include one or more values from the **Restrict to** dropdown (for example, "Device") and set **Value Selection** to "From Context". Click **OK** to save the changes.

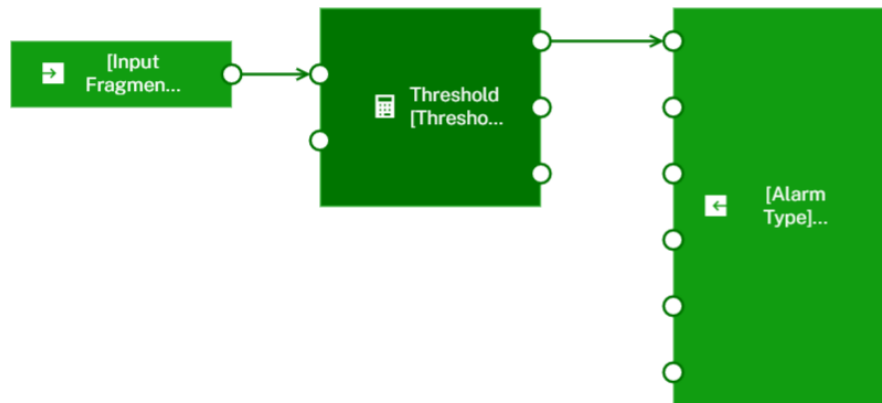
For more details about "From Context" and **Restrict to**, refer to [Define template parameters](#)

Name	Type	Value Selection	Default Value	Usage Count
Measurement Source	Source or Destination	From Context		2
		Required		
		Optional		
Input Fragment and Series	Fragment and Series	From Context		1

For detailed information about creating and managing template parameters in Analytics Builder, refer to [Managing template parameters](#).

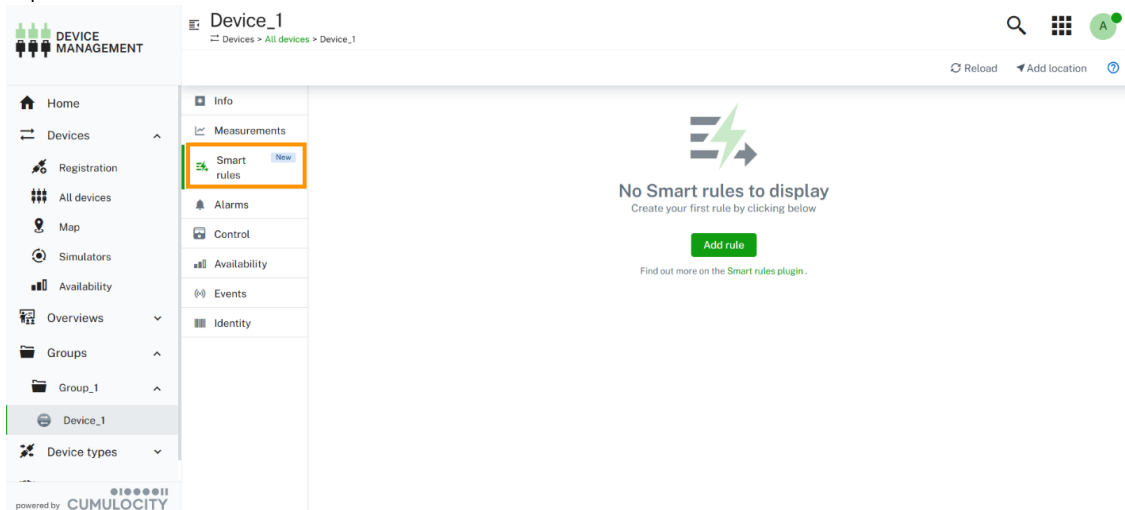
8. In the toolbar of the model editor, click the save icon [Save] to save the model.

When completed, your model will look similar to this:



Step 2: Create a model instance using the smart rules plugin

1. Navigate to the Device Management application.
2. Go to **Devices > All devices** and select a device.
3. Click the **Smart rules (NEW)** tab (embedded as one of the device details tabs). Refer to the [Prerequisites](#) section to ensure all requirements are met.



4. Click **Add rule** to open the **Add rule** dialog. Since you are currently in the **Device** context, only models that have been configured with **From Context** template parameters restricted to **Device** will be visible here. Select your configured model and click **OK**.

+ Add rule

Search name, description or tag



All rules (1)

Create alarm on threshold

Cancel

OK

5. Configure the rule parameters:
 - Optionally update the name or add a note if desired.
 - Populate the template parameter values. The value of the template parameter (for example, "Measurement Source") configured with **From Context** is derived automatically from the current device or group context.
 - Click **Save** to save the rule, or toggle to **Active** to save and deploy immediately.
6. Your rule now appears in the smart rules list, showing its status (Active/Inactive).

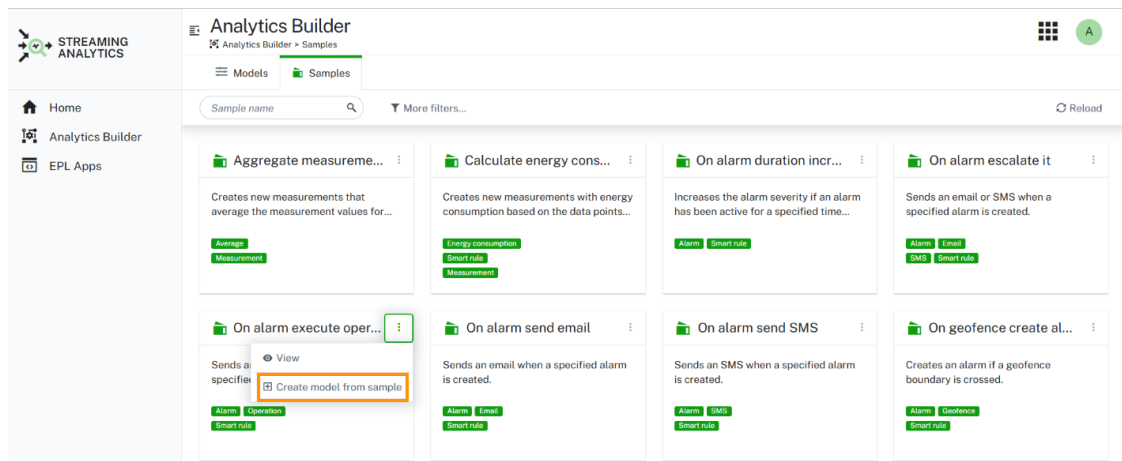
Congratulations! You have successfully created and deployed your first smart rule using the smart rules plugin.

CREATING SMART RULES FROM EXISTING ANALYTICS BUILDER SAMPLES

You can also create smart rules using pre-built sample models from Analytics Builder. The process is similar to creating from scratch but faster since the model logic is already defined.



Step 1: Create a model from a sample

1. Navigate to **Analytics Builder > Samples**.
2. Click the actions menu of your desired sample (for example, "On alarm execute operation") and select **Create model from sample**.



3. The model editor opens with the sample model ready for use.


Step 2: Configure template parameters

1. Click the template parameter icon  in the toolbar.
2. Ensure the template parameter entry provided for **Input Source** of the **Alarm Input** block (for example, "Device or group of devices") has "Source or Destination" updated to include one or more values from the **Restrict to** dropdown (for example, "Device") and set **Value Selection** to "From Context". Click **OK** to save the changes.
3. In the toolbar of the model editor, click the save icon  to save the model.

Step 3: Create a rule from the Device Management application

Follow the same steps as described in [Step 2: Create model instance using the smart rules plugin](#) above.

TROUBLESHOOTING

1. **No models available in Add rule dialog**
 - Ensure that the Analytics Builder models have template parameters configured with the "From Context" value.
 - Verify your current context matches the model's "Source or Destination" type restrictions. For example, when in a device context where the model's **Restrict to** dropdown is configured with "Groups" only, that model will not be available. To appear in device contexts, "Device" must be selected in the **Restrict to** dropdown.
2. **Previously created rules don't appear in the smart rules list**
 - Verify that the Analytics instances are set to **Production mode** in the Analytics Builder instance editor.
 - Check if Analytics instance names are set in the Analytics Builder instance editor.
 - Confirm you're viewing the correct device or group context.
3. **Error in smart rule deployment**
 - Click the runtime error icon  **Runtime error** to view information about the error.

UNDERSTANDING THE SMART RULES (NEW) PLUGIN

ANALYTICS BUILDER WORKFLOW VERSUS SMART RULES PLUGIN

Standard Analytics Builder workflow

In the standard Analytics Builder workflow, to deploy analytics instances, users must switch from their primary workflow to the Analytics Builder:

1. Navigate to the Streaming Analytics application.
2. Go to **Analytics Builder**.
3. Select your model and navigate to the **Instances** page.
4. Create instances/rules for the model in the instance editor.
5. Configure and deploy rules from the instance editor.

This process requires switching between applications and can be time-consuming when managing rules for multiple devices or groups.

For detailed information about the instance editor workflow, refer to [Instance editor](#).

With smart rules plugin

The smart rules plugin streamlines this process by enabling Analytics Builder model instance creation and deployment directly from device and group contexts, eliminating the need for switching to a different application.

UNDERSTANDING THE SMART RULES INTERFACE

A rule has the following fields and properties:

Field	Description
Name	Automatically populated with the model name but can be customized for more descriptive names. When creating multiple rules from the same model without changing the rule name, the plugin automatically appends #1, #2, #3, etc. to differentiate instances. Rules are sorted by instance name.
Note	You can add a note for the rule.
Created	Shows the timestamp when the rule was created.
Last Updated	Shows the timestamp when the rule was last updated.
Model Name	Displays the name of the model associated with the rule. This field is read-only.
Template parameters	Template parameters configured as "From Context" are automatically populated based on your current device/group context. Context-based parameters are read-only as they inherit values from the current context. Non-context template parameters remain editable for user configuration.
Active/Inactive	Deploy the rule by toggling from inactive to active state. Once the rule is deployed, all fields and properties become read-only unless the rule is undeployed.

INTEGRATION WITH ANALYTICS BUILDER INSTANCE EDITOR

The smart rules plugin maintains seamless bidirectional integration with the Analytics Builder instance editor.

Rules created in the Analytics Builder instance editor appear in the smart rules plugin only when the mode is set to **Production**. Non-production modes indicate the rule is under development or testing and not ready for deployment.

Rules created through the smart rules plugin automatically appear in the corresponding model's instance editor. These rules can be edited, modified, or deleted from the instance editor.

To prevent deployment of incomplete rules, the instance editor includes enhanced validation on the instance name. The instance name is required and must be unique within your tenant. The instance is available for deployment only after you provide a unique, non-empty instance name.

INFO

This validation applies only to models configured with template parameters with the value selection set to "From Context".

- When the instance name is empty, a message **Set distinct instance name to select production mode** is displayed below the

mode field.

- If you enter a duplicate instance name, an error message **Instance name already exists** appears below the instance name field and prevents you from proceeding.

	INSTANCE NAME/NOTE	FOO	RUN MODE	STATUS		
1	<input type="text" value="Instance_Name"/>	<input type="text" value="Device_1"/>	<input type="text" value="Draft"/>	<input type="text" value="Inactive"/>		
	Add note					
2	<input type="text" value="Instance_Name"/>	<input type="text" value="Device_1"/>	<input type="text" value="Draft"/>	<input type="text" value="Inactive"/>		
	<div>Instance name already exists</div> Add note		<div>Set distinct instance name to select production mode</div>			

This validation ensures that all production-ready rules have proper identification and prevents deployment conflicts.

EPL APPS

INFO

This documentation assumes basic familiarity with Apama application development. Refer to the [Apama documentation](#) for further details.

USING THE APAMA EVENT PROCESSING LANGUAGE (EPL)

The Apama Event Processing Language has a syntax similar to Java. In addition to simple flow control statements such as `if`, `while`, `for`, users can write listeners with the `on` keyword to react to events.

Apama EPL is documented in the [Apama documentation](#).

As an example, the following statement listens for new temperature sensor readings greater than a particular temperature:

```
on all Measurement(type="c8y_TemperatureMeasurement") as m {
  if m.measurements.getDefault("c8y_TemperatureMeasurement").getDefault("T").value > 100.0 {
    Alarm alarm := new Alarm;
    alarm.type := "c8y_TemperatureAlert";
    alarm.source := m.source;
    alarm.time := currentTime;
    alarm.text := "Temperature too high";
    alarm.status := "ACTIVE";
    alarm.severity := "CRITICAL";
    send alarm to Alarm.SEND_CHANNEL;
  }
}
```

Here, `Measurement` is a pre-defined event containing the measurements. In this example, `m` is the `Measurement` event, the listener is filtering for measurements which are `c8y_TemperatureMeasurement` and the property is `c8y_TemperatureMeasurement.T.value` which is in degrees Celsius of a temperature sensor (see [Fragment library](#)).

Listeners such as the above should be placed in a monitor in the `onload` statement, and the file must contain `using` statements for the types used by the listener - for most of the Cumulocity events, these are in the package `com.apama.cumulocity`. The full list is provided below - for the sake of brevity, we will omit these from further examples:

```

using com.apama.cumulocity.ManagedObject;
using com.apama.cumulocity.Operation;
using com.apama.cumulocity.Event;
using com.apama.cumulocity.Alarm;
using com.apama.cumulocity.Error;
using com.apama.cumulocity.Measurement;
using com.apama.cumulocity.MeasurementValue;
using com.apama.cumulocity.FindAlarm;
using com.apama.cumulocity.FindAlarmResponse;
using com.apama.cumulocity.FindAlarmResponseAck;
using com.apama.cumulocity.FindManagedObject;
using com.apama.cumulocity.FindManagedObjectResponse;
using com.apama.cumulocity.FindManagedObjectResponseAck;
using com.apama.cumulocity.FindMeasurement;
using com.apama.cumulocity.FindMeasurementResponse;
using com.apama.cumulocity.FindMeasurementResponseAck;
using com.apama.cumulocity.FindOperation;
using com.apama.cumulocity.FindOperationResponse;
using com.apama.cumulocity.FindOperationResponseAck;
using com.apama.cumulocity.FindEvent;
using com.apama.cumulocity.FindEventResponse;
using com.apama.cumulocity.FindEventResponseAck;
using com.apama.cumulocity.SendEmail;
using com.apama.cumulocity.SendSMS;
using com.apama.cumulocity.Util;
using com.apama.util.AnyExtractor;
using com.apama.correlator.timeformat.TimeFormat;
using com.softwareag.connectivity.httpclient.HttpOptions;
using com.softwareag.connectivity.httpclient.Request;
using com.softwareag.connectivity.httpclient.RequestType;
using com.softwareag.connectivity.httpclient.Response;

monitor ListenForHighTemperatures {
  action onload() {
    on all Measurement(type="c8y_TemperatureMeasurement") as e {
      if e.measurements.getOrDefault("c8y_TemperatureMeasurement").getOrDefault("T").value > 100.0 {
        // handle the measurement
      }
    }
  }
}

```

HOW CAN I CREATE DERIVED DATA FROM EPL?

To create a new alarm or operation, create an instance of the relevant event type and use the `send` statement to send it to the relevant channel (defined with a constant on the event type). Assume that an alarm should be generated immediately if the temperature of a sensor exceeds a defined value. This is done with the following statement:

```

on all Measurement(type="c8y_TemperatureMeasurement") as m {
  if m.measurements.getOrDefault("c8y_TemperatureMeasurement").getOrDefault("T").value > 100.0 {
    send Alarm("", "c8y_TemperatureAlert", m.source, currentTime, "Temperature too high", "ACTIVE", "CRITICAL", 1, new dictionary<string, any>) to
    Alarm.SEND_CHANNEL;
  }
}

```

Technically, this statement produces a new alarm event each time a temperature sensor reads more than 100 degrees Celsius and sends it to Cumulocity.

HOW CAN I CONTROL DEVICES FROM EPL?

Remote control with EPL is done by sending an operation event. Remote operations are targeted to a specific device. The following example illustrates switching a relay based on temperature readings:

```

on all Measurement(type="c8y_TemperatureMeasurement") as m {
  if m.measurements.getDefault("c8y_TemperatureMeasurement").getDefault("T").value > 100.0 {
    send Operation("",m.source,"PENDING",{ "c8y_Relay":<any>{"relayState":"CLOSED"}}) to Operation.SEND_CHANNEL;
  }
}

```

- `m.source` is a placeholder for the ID of the heating that should be triggered.
- The `params` field (the last field) defines the nested content of the operation. In this example we create a `c8y_Relay` operation and set the `relayState` to `CLOSED`. Note that the top-level fields must be `dictionary<string, any>`, thus the use of the `<any>` cast operation.

HOW CAN I QUERY DATA FROM EPL?

It may be required to query information from the Cumulocity database as part of the ongoing event processing. This is supported by sending events and using listeners to wait for responses. Here is an example that shows how to summarize total sales for vending machines every hour. The sales report data created after a purchase is retrieved from the Cumulocity database.

```

using com.apama.aggregates.count;

monitor SalesReport {
  event SalesReport {
    Event e;
    ManagedObject customer;
  }
  event SalesOutput {
    integer count;
    string customerId;
  }
}

action onload() {

  monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);

  on all Event() as e {
    monitor.subscribe(FindManagedObjectResponse.SUBSCRIBE_CHANNEL);
    integer reqId := integer.getUnique();
    on all FindManagedObjectResponse(reqId=reqId) as mor and not FindManagedObjectResponseAck(reqId=reqId) {
      route SalesReport(e, mor.managedObject);
    }
    on FindManagedObjectResponseAck(reqId=reqId) {
      monitor.unsubscribe(FindManagedObjectResponse.SUBSCRIBE_CHANNEL);
    }
    send FindManagedObject(reqId,"",{ "childAssetId":e.source}) to FindManagedObject.SEND_CHANNEL;
  }

  from sr in all SalesReport() within 3600.0 every 3600.0
  group by sr.customer.id
  select SalesOutput(count(), sr.customer.id) as sales {
    send Measurement("", "total_cust_trx", "customer_trx_counterId", currentTime,
    {
      "total_cust_trx":{
        "total":MeasurementValue(sales.count.toFloat(), "COUNT", new dictionary<string,any>)
      }
    }, {"customer_id":<any> sales.customerId}) to Measurement.SEND_CHANNEL;
  }
}

```

In the above example we start by creating definitions for `SalesReport` and `SaleOutput` events. These hold the `SalesReport` (the `Event` and `ManagedObject` that identifies a sale) and the information we want to derive from a set of sales: the `count` and `customerId`. We listen for `Event` objects, and send a `FindManagedObject` request to look up the `ManagedObject` that the event came from. These `SalesReport` objects are sent, via the route statement, into a stream query. The stream query fires every hour (3,600 seconds) and selects an aggregate of the sales data per customer, and sends a new measurement representing the sales for that

vending machine.

BASIC FUNCTIONALITY

DEVELOPING APPS

An EPL app is a Cumulocity application written in a single EPL (*.mon) file. You can develop EPL apps in two different ways:

- You can use the [Streaming Analytics application](#) which is available from Cumulocity's application switcher and develop your EPL apps within Cumulocity.
- Or you can create a Git repository for your EPL apps and develop it on your local machine with the [Apama Extension](#) for Microsoft Visual Studio Code. This approach also works for larger EPL applications deployed to a custom microservice.

INFO

To be able to develop and deploy EPL apps with the Streaming Analytics application and/or to upload locally-developed EPL files into Cumulocity, your tenant must be subscribed to the Apama-ctrl microservice that supports EPL apps. If you do not see the **EPL Apps** page in the Streaming Analytics application and you wish to use EPL apps, contact [product support](#).

CAUTION


An EPL app has the ability to make nearly arbitrary changes to the objects in a tenant, whether that's inventory, alarms or many other sorts of object. A user who has ADMIN permission for "CEP management" is able to create and activate EPL apps and thus also has almost full control over the current tenant. Therefore, you should be careful about which users on the tenant have this permission.

Developing apps with the Streaming Analytics application

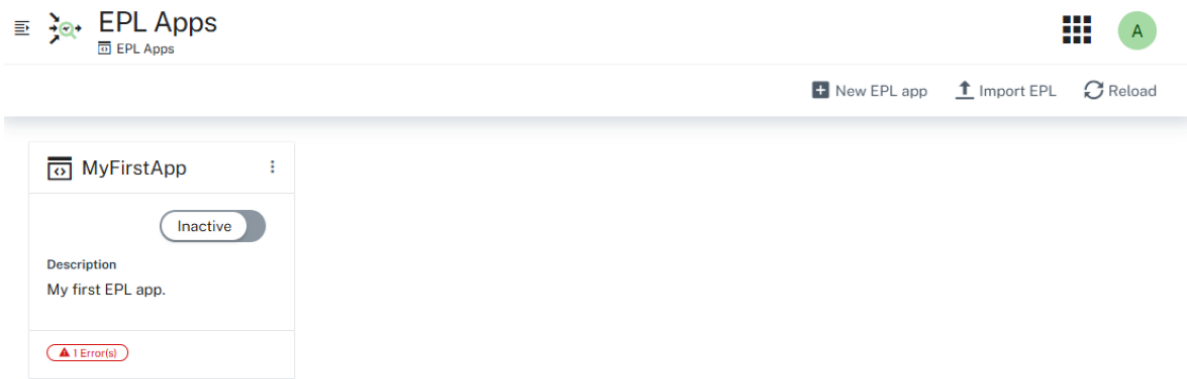
The **EPL Apps** page of the Streaming Analytics application provides an interface for interactively editing new or existing EPL apps (*.mon files) as well as importing and activating (deploying) EPL apps.

Any user on the tenant wishing to use the **EPL Apps** page must be a **CEP Manager**. See [Managing permissions and roles](#).

Step 1 - Invoke the Streaming Analytics application

Open the application switcher and click the  icon for the **Streaming Analytics** application. Then navigate to the **EPL Apps** page.

When you go to the **EPL Apps** page, the EPL app manager is shown first, listing any existing EPL apps. Each app is shown as a card. You can add new EPL apps and manage existing EPL apps from here.



Each card that is shown for an app has an actions menu at the top which allows you to edit, download or delete the app.

From this page, you can:

- Edit existing EPL apps. Either use the **Edit** command from the actions menu or simply click on the card that is shown for the app.
- Create new EPL apps. See below.
- Import EPL apps. If you prefer to develop your apps outside of Cumulocity (for example, using the Apama Extension for VS Code), click **Import EPL** in the top menu bar to upload an Apama EPL (*.mon) file as an app into the Streaming Analytics application.
- Download EPL apps. Use the **Download** command from the actions menu to download the app as a *.mon file.
- Deploy existing EPL apps. On the card that is shown for an app, change the mode from **Inactive** to **Active**. For more information, see [Deploying apps](#). When activating an app, any syntax errors are reported back immediately. The error state is shown on the card, helping you to ensure your app is in good shape. Click on the error to display information on what went wrong. It is not possible to activate an app if it has syntax errors. The errors are shown on the card until they have been fixed and the app has been activated again.
- Reload all EPL apps. Click **Reload** in the top menu bar to refresh the display to show any changes other users have made since the page loaded, including any errors that have been introduced in the meantime.

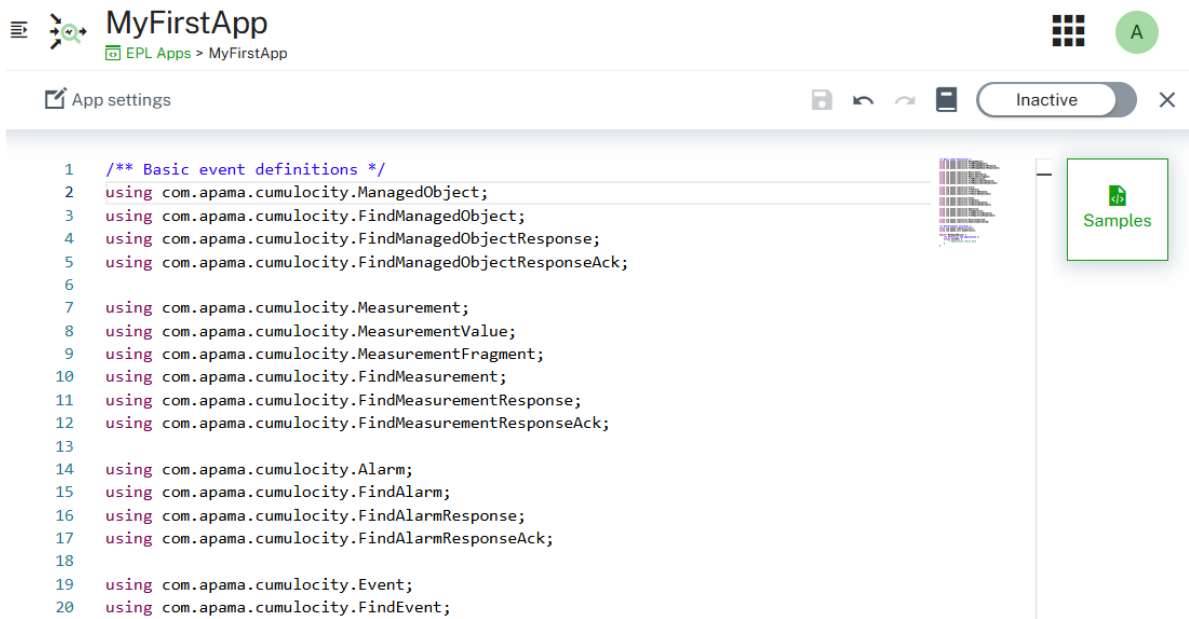
Step 2 - Create an EPL app

Click **New EPL app** in the top menu bar. In the resulting **Create app** dialog box, enter a unique app name. You can also enter a description which will be shown on the card that is created for the new app. Click **OK**.

The EPL editor appears. The EPL code for the new app already contains the typical basic event definitions and utilities that are required for working with Cumulocity. You can adapt them as required for your app. Consult the documentation and samples for more details.

INFO

When you click **Cancel** without specifying an app name, the EPL editor also appears and the default name "New" is then shown in the breadcrumb. You can edit the EPL code, but as long as you do not specify an app name, you will not be able to save the app. Click **App settings** and specify an app name in the resulting dialog box.




To help you get started, several samples are available. To see them, click **Samples** which is shown to the right of the editor. Click on a sample to see a preview of its contents. You can select part of the sample code and copy it over into your own code using the standard key combinations Ctrl+C and Ctrl+V. You can also use the command buttons to copy the entire code to the clipboard and insert it at an appropriate position in your own code, or to replace all of your existing code with the sample code.

Using the buttons in the top menu bar, you can undo/redo your last changes in the current session and you can save your changes.

It is also possible to change the mode from **Inactive** to **Active** (or vice versa) in the EPL editor. Again, when there is an error in your EPL code, it is not possible to activate the app. The errors are highlighted within the code.

INFO

Be aware that the EPL editor makes use of a standard web component. It provides many generic developer functions, some of which are not relevant to EPL, including but not limited to Quick Fix and Show Hover.

Click the close icon  in the top menu bar to leave the EPL editor and thus to return to the list of EPL apps.

CAUTION

All unsaved changes are lost when you navigate to a different URL or close the browser window.

Step 3 - Test the EPL app

Once your app is activated, you should be able to see the results of it running. This may include sending measurements, receiving data, creating alarms, and logging in the Apama-ctrl microservice. For information on how to check the log files of the Apama-ctrl microservice, see [Monitoring microservices](#).

See also [Deploying apps](#).

For information about a more systematic and automated approach to testing EPL apps, see [Testing apps](#).

Developing apps with the Apama Extension for VS Code

The Apama Extension for VS Code provides a full development environment and is the tool of choice when you have a complex EPL application. When your EPL app (that is, the *.mon file) is ready, you must import (deploy) it into Cumulocity.

Step 1 - Install the Apama Extension for VS Code

1. **Install Visual Studio Code:** Download and install [Microsoft Visual Studio Code](#).
2. **Install Apama Extension:** Install the [Apama Extension](#) from the Visual Studio Code Marketplace.
3. Following the steps listed on the extension page to setup WSL (if using Windows) and to install a container engine for running Apama inside a container, or else install Apama locally.

Step 2 - Create a project and repository

The best way to start EPL apps development is to create a new Git repository based on the [Streaming Analytics Sample Repository Template](#). Go to that page in GitHub, and click **Use this template** and then **Create a new repository**. Projects created from the template already have the main bundles you will need for EPL apps.

If you already have a Git repository and Apama project for your application, just copy across the `.devcontainer` directory from the template repository so that it can be opened in a VS Code Dev Container.

Step 3 - Open the repository in VS Code

To open your Git repository in VS Code, open the VS Code command palette (`F1`), run `Dev Containers: Clone Repository in Container Volume` and then enter the `https://` link to your GitHub repository. This assumes you have a Docker-compliant container engine installed; if not see the [Apama Extension](#) documentation for information about how to work with project folders using a local installation of Apama.

Instead of using VS Code you can use a web browser to open the repository in [GitHub Codespaces](#) without installing anything at all.

Step 4 - Create an EPL file for your application

If using the template repository, you will find a sample EPL app in the `src/` directory of your repository. This is a good starting point for your own EPL apps. Feel free to customize that file, or replace it with your own application, perhaps based on the other samples in the [EPL Apps Tools](#) repository. You can also create additional `.mon` files, one for each EPL app.

For more information about interacting with Cumulocity from your application, see [The Cumulocity Transport Connectivity Plug-in](#) in the Apama documentation.

Step 5 - Test the EPL

Before you import the newly created file as an EPL app into Cumulocity and activate it there, you might want to test if the file works as expected from within the Apama Extension for VS Code.

First, check the "Problems" view for any basic errors such as incorrect syntax. Then, create a `test` that runs your application in a local correlator. If you used the template to create your repository you will find a sample test to get started under the `tests/` directory.

Once the EPL app is ready, refer to [Deploying apps](#) to find out how to deploy it to Cumulocity.

DEPLOYING APPS

You can deploy the following to Cumulocity:

- EPL apps. You can [develop or import a single *.mon file with the Streaming Analytics application](#). This is the simplest mechanism for deploying an EPL app.
- Apama applications in a custom microservice. You can upload complex Apama projects (typically developed with the Apama Extension for VS Code) to Cumulocity and [deploy them as custom microservices](#) using the Cumulocity Microservice SDK.

INFO

In the Streaming Analytics application, the term "activate" is used for deploying an app.

Deploying EPL apps as single *.mon files with the Streaming Analytics application

When an EPL app (that is, a *.mon file) is activated in Cumulocity, the *.mon file is assigned a unique package name. This prevents conflicts when multiple modules are activated. For this reason, you should not specify a `package` statement in a *.mon file. If you must share events between different parts of your application, then write the event definitions and monitors that use it in a single *.mon file.

There is a restricted set of utilities and base events available for your EPL app. At the time of writing, these include the **Time Format** and **HTTP Client > JSON with generic request/response event definitions** bundles.

When any EPL app signals a runtime error, this will be raised as an alarm. Runtime errors include uncaught exceptions, as well as any explicit logging of warnings and errors that your EPL app wants to do. Health issues that relate to the Apama runtime in general will also be raised as alarms.

For more detailed diagnostics of the Apama runtime and any active EPL apps, you can look at the logs for the Apama-ctrl microservice. See [Monitoring microservices](#) for more information on log files. However, some familiarity with Apama is necessary to get the most out of an Apama log file.

Deploying Apama applications as custom microservices

Using the Apama Extension for VS Code, you can also develop more complex projects which:

- are spread across multiple *.mon files
- must be isolated from other Apama applications
- use connectivity plug-ins or EPL plug-ins that are not enabled by default

These kinds of applications should be deployed as custom microservices to Cumulocity.

Required settings in the microservice manifest

The microservice manifest provides the required settings to manage microservice instances and the application deployment in Cumulocity. For detailed information, see [Microservice manifest](#).

Apama can be used in either a single-tenant microservice or a multi-tenant microservice. Therefore, the microservice manifest must set the isolation level to either PER_TENANT or MULTI_TENANT. When Apama is used in a multi-tenant microservice, the Apama application must be written to be multi-tenant aware. For more information, see [Working with multi-tenant deployments](#) in the Apama documentation.

The following permissions are required by the microservice in order to start up and use all features in the Cumulocity transport from EPL. These are set with requiredRoles in the microservice manifest.

- ROLE_APPLICATION_MANAGEMENT_READ
- ROLE_INVENTORY_READ
- ROLE_INVENTORY_ADMIN
- ROLE_INVENTORY_CREATE
- ROLE_MEASUREMENT_READ
- ROLE_MEASUREMENT_ADMIN
- ROLE_EVENT_READ
- ROLE_EVENT_ADMIN
- ROLE_ALARM_READ
- ROLE_ALARM_ADMIN
- ROLE_DEVICE_CONTROL_READ
- ROLE_DEVICE_CONTROL_ADMIN
- ROLE_IDENTITY_READ
- ROLE_OPTION_MANAGEMENT_READ
- ROLE_BULK_OPERATION_READ
- ROLE_SMS_ADMIN

INFO

To take advantage of the Cumulocity Notifications 2.0 reliable data forwarding capability to receive notifications, you must also add the following permission to the manifest of the custom microservice and contact [product support](#) to set the the `notification2.streaming-analytics` feature flag.

- ROLE_NOTIFICATION_2_ADMIN

INFO

The above is the minimum list of permissions that a custom Apama microservice needs. If you are developing a custom microservice, you may add more permissions to the microservice manifest.

To deploy an Apama application as a microservice

1. Develop your application in the Apama Extension for VS Code.
2. Add a Dockerfile as described by [Deploying Apama Applications with Docker](#). The sample file in the *Apama/etc/Dockerfile.project* directory of the Apama installation is a good starting point.
3. Add any custom steps to the Dockerfile that might be necessary, for example, building a custom plug-in.
4. Use the Cumulocity microservice utility tool for packaging and deploying the project; for detailed information, see [Microservice utility tool](#). When creating the directory structure for the microservice utility tool to build from, copy your entire project directory inside that directory with the name “docker/”. For example:

```
docker/src/
docker/tests/
docker/Dockerfile
docker/...
cumulocity.json
```

You must create the [microservice manifest](#) manually, but there is no need for anything special in the microservice manifest; no roles or probes are required. However, if you want to configure a liveness or readiness probe, you can configure an `httpGet` probe for the path `/ping` on port 15903 (Apama's default port). Enabling auto-scaling is not recommended, as Apama applications are usually stateful and do not automatically partition their input.

You can pack, deploy and subscribe from this directory, resulting in your Apama application being turned into a running microservice. The behavior of the application when being run outside of Cumulocity (from the Apama Extension for VS Code or your test environment) will be near-identical to its behavior inside Cumulocity. When deployed as a microservice doing requests to the Cumulocity API, Apama will automatically pick up the credentials to connect to the tenant you deployed it to, overwriting any other credentials provided to Apama. However, if you wish to receive real-time events, you must have valid credentials specified in the project configuration as you do when connecting to Cumulocity from an external Apama environment.

5. When you are ready to deploy to Cumulocity, upload the application as a microservice. For details, refer to [Managing microservices](#).

INFO

After December 2024, the location of the Docker images has changed for all supported release trains. They are now available at Amazon ECR Public Gallery instead of at Docker Hub. If you still use the images from the previous location, you must migrate them.

TESTING APPS

The best way to check your EPL applications behave correctly is to create automated tests for them.

The [Apama EPL Apps Tools](#) GitHub repository includes extensions for the PySys test framework that allow you to simply write tests for your EPL apps and to run them automatically, either by uploading the EPL to Cumulocity, or by running inside a local correlator communicating with Cumulocity. See the [EPL Apps Tools documentation](#) for detailed information. For more information on PySys, see the [API Reference for Python](#) in the main Apama documentation.

To provide the credentials and server URL for your Cumulocity tenant, you need to set some environment variables when running the tests:

```
CUMULOCITY_SERVER_URL=<URL>
CUMULOCITY_USERNAME=<USERNAME>
CUMULOCITY_PASSWORD=<PASSWORD>
```

As well as running tests locally during development, you can script deployment and testing of your EPL apps for CI/CD (continuous integration and continuous delivery) purposes.

CONNECTIVITY BUNDLES FOR COMMUNICATING WITH CUMULOCITY

The **Cumulocity IoT > Cumulocity Notifications 2.0** bundle exposes the Cumulocity client to EPL apps using the Notifications 2.0 mechanism.

For general information on how to receive Cumulocity update notifications, see [Receiving update notifications](#) in the Apama documentation.

INFO

The **Cumulocity Notifications 2.0** connectivity bundle has been added for receiving notifications from Cumulocity using the Notifications 2.0 mechanism. The existing **Cumulocity Client** connectivity bundle that uses the legacy long-polling mechanism is deprecated in favor of this new bundle. In addition, it is now possible to add the **Cumulocity REST Support** connectivity bundle if you do not receive any notifications. You must only add one of these three bundles to your project.

SUPPORTED REST SERVICES

EPL apps are designed to listen for REST (Representational State Transfer) services and supports all GET, POST, PUT and DELETE operations. Example requests for the different operations are listed below.

To perform these operations, you must have READ and ADMIN permissions for "CEP management" (see also [Managing permissions and roles](#)).

Request headers for all operations

Each request must be authenticated to Cumulocity.

Name	Description
Accept	"application/json". This is a mandatory parameter.

Common response codes

The following common error response codes can be expected for all requests:

Code	Description
401	Unauthorized.
403	Forbidden. EPL apps are not available with the Apama-ctrl-starter microservice.

Any other response codes that can be expected from a specific request are given below.

Common field descriptions

The following common fields are available with the responses, depending on the operation:

Field	Description
contents	The full contents of the EPL file.
description	A description of the file.

Field	Description
eplPackageName	The package name of the EPL file. If the name contains special characters (including spaces), these characters are escaped to make them valid EPL identifiers and avoid injection errors.
errors	A list of all compilation errors in the file, if any, with line numbers and text.
id	A unique identifier of the file.
name	The name provided for this bit of EPL.
state	Whether the EPL is injected into the correlator and running. This can either be <code>active</code> or <code>inactive</code> .
warnings	A list of all compilation warnings in the file, if any, with line numbers and text.

GET - Retrieve all available EPL files

Endpoint: `/service/cep/eplfiles`

Example request

```
GET /service/cep/eplfiles
```

Responses

Code	Description
200	Successful operation. See also the example value below.
400	Bad request. Header contents has unexpected value.

Example value for response code 200:

```
{
  "eplfiles":[
    {
      "description":"","
      "eplPackageName": "eplfiles.Ordinal1",
      "errors":[

      ],
      "id":"39615",
      "name":"Ordinal1",
      "state":"active",
      "warnings":[

      ]
    }
  ]
}
```

GET - Retrieve all available EPL files with their contents

Endpoint: `/service/cep/eplfiles`

Request parameters

Name	Description
contents	Boolean type. Fetches the EPL files with their contents. This is an optional query parameter.

Example request

```
GET /service/cep/eplfiles?contents=true
```

Responses

Code	Description
200	Successful operation. See also the example value below.

Example value for response code 200:

```
{
  "eplfiles":[
    {
      "contents":"monitor M0 { action onload() { on wait(1.0) { log \"Hello\" at INFO; }}}",
      "description":"",
      "eplPackageName": "eplfiles.Ordinal1",
      "errors":[
      ],
      "id":"39615",
      "name":"Ordinal1",
      "state":"active",
      "warnings":[
      ]
    }
  ]
}
```

GET - Retrieve EPL file by identifier

Endpoint: `/service/cep/eplfiles/{id}`

Request parameters

Name	Description
id	Identifier of the EPL file to be fetched. This is a mandatory parameter.

Example request

```
GET /service/cep/eplfiles/{id}
```

Responses

Code	Description
200	Successful operation. See also the example value below.
404	File with identifier not found. See also the example value for this response code at the end of this section.

Example value for response code 200:

```
{
  "contents": "monitor M0 { action onload() { on wait(1.0) { log \"Hello\" at INFO; }}}",
  "description": "",
  "eplPackageName": "eplfiles.Ordinal1",
  "errors": [
  ],
  "id": "39615",
  "name": "Ordinal1",
  "state": "active",
  "warnings": [
  ]
}
```

POST - Create a new EPL application

Endpoint: `/service/cep/eplfiles`

Example request

`POST /service/cep/eplfiles`

The following is an example of a request body:

```
{
  "name": "Ordinal1",
  "contents": "monitor M1 { action onload() { on wait(1.0) { log \"Hello\" at INFO; }}}",
  "state": "active",
  "description": ""
}
```

Note the following:

- The `name` is used for the package of the file (thus the EPL file must not contain a `package` statement) and must be unique across all EPL files. The name is prefixed and certain characters are escaped. The actual package name used is returned in the `eplPackageName` field for convenience (you can search for this in the microservice log file to find log statements).
- Make sure to provide safely escaped `contents`.
- `description` is optional and can be empty.

Responses

Code	Description
201	Successfully created / Created with errors in file / Created with warnings in file. See also the examples below.
405	Invalid input.

Example for response code 201 when successfully created:

```
{
  "description": "",
  "eplPackageName": "eplfiles.Ordinal1",
  "errors": [
  ],
  "id": "39615",
  "name": "Ordinal1",
  "state": "active",
  "warnings": [
  ]
}
```

Example for response code 201 when created with warnings or errors:

```
{
  "description": "",
  "eplPackageName": "eplfiles.Ordinal1",
  "errors": [
    {
      "line": 5,
      "text": "assigning a float to an integer variable"
    }
  ],
  "id": "39651",
  "name": "Ordinal1",
  "state": "inactive",
  "warnings": [
    {
      "line": 10,
      "text": "\"assert\" may become a reserved word in future versions of EPL"
    }
  ]
}
```

PUT - Update EPL file by identifier

Endpoint: `/service/cep/eplfiles/{id}`

Request parameters

Name	Description
id	Identifier of the EPL file to be updated. The identifier must be included in the path. This is a mandatory parameter.

Example request

PUT `/service/cep/eplfiles/{id}`

The following is an example of a request body:

```
{
  "name": "Ordinal1",
  "contents": "monitor M1 { action onload() { on wait(1.0) { log \"Hello\" at INFO; }}}",
  "state": "active",
  "description": ""
}
```

See also the information given for the POST request.

Responses

Code	Description
200	Successfully updated. See also the example values below.
404	File with identifier not found. See also the example value for this response code at the end of this section.

Example value for response code 200 when successfully updated with no errors:

```
{
  "description": "",
  "eplPackageName": "eplfiles.Ordinal1",
  "errors": [

  ],
  "id": "39615",
  "name": "Ordinal1",
  "state": "active",
  "warnings": [

  ]
}
```

Example value for response code 200 when updated with errors or warnings:

```
{
  "description": "",
  "eplPackageName": "eplfiles.Ordinal1",
  "errors": [
    {
      "line": 5,
      "text": "assigning a float to an integer variable"
    }
  ],
  "id": "39651",
  "name": "Ordinal1",
  "state": "inactive",
  "warnings": [
    {
      "line": 10,
      "text": "\"assert\" may become a reserved word in future versions of EPL"
    }
  ]
}
```

DELETE - Delete EPL file by identifier

Endpoint: `/service/cep/eplfiles/{id}`

Request parameters

Name	Description
id	Identifier of the EPL file to be deleted. The identifier must be included in the path. This is a mandatory parameter.

Example request

```
DELETE /service/cep/eplfiles/{id}
```

Responses

Code	Description
200	Successfully deleted.
404	File with identifier not found. See also the example value for this response code at the end of this section.

Example value for response code 404

The response code 404 indicates that a file with a specific identifier was not found.

```
{
  "error": "Not Found",
  "exception": "com.apama.in_c8y.FileNotFoundException",
  "message": "File with id 39613 not found",
  "path": "/epfiles/39613",
  "status": 404,
  "timestamp": "2020-01-17T12:21:42.457+0000"
}
```

where

- **error** is the error message.
- **exception** specifies the exception that was raised.
- **message** is a description of the exception message.
- **path** is the path that was requested.
- **status** is the status of the application.
- **timestamp** is the timestamp in ISO format.

EVENTS AND CHANNELS

In Apama EPL, interactions with the rest of the Cumulocity ecosystem are done via events. A number of event definitions is provided for accessing Cumulocity data.

INFO

Apama and Cumulocity use different “event” concepts. Apama events are used for all interactions with Cumulocity, such as listening for and creating device measurements, alarms and (Cumulocity) events. For more information on Apama events, see [Defining event types](#) in the Apama documentation. For more information on Cumulocity events, see [Events](#) in the Cumulocity OpenAPI Specification.

Predefined event types

There are some predefined event types to interact with several Cumulocity APIs. Events are sent to Apama applications automatically when a new measurement, alarm or event is created. For interacting with the Cumulocity backend, you can create an event and send it to the relevant channel. Cumulocity will automatically execute either the database query or create the API calls necessary for sending mails, SMS, or similar.

Look at the [data model](#) in the API Reference for EPL (ApamaDoc) to see how the events for each stream are structured.

Sending events to a channel

Sending an event is done by constructing the event, either with `new <type>` followed by assignments to the fields, or with a constructor specifying all of the fields. The `send` statement is then used to send the event to Cumulocity. The `send` statement requires a channel - this is the `SEND_CHANNEL` constant on the event type.

Listening to events

You can trigger your EPL by listening to events on channels. You can subscribe to channels with the `monitor.subscribe("string name")` method. This can be done in the startup of your monitor, or if you only want to receive events some of the time, called as needed, followed by `monitor.unsubscribe("string name")`.

Listen for events using the `on` statement, followed by the event type that you are listening to, open and close parentheses, and `as <identifier>` to name a variable that will hold the event.

By default, a listener will fire once; to make it repeat for all events, use the `all` keyword before the event type.

Filters

Adding filters can be done by specifying one or more fields between the parentheses for a listener. Only top-level fields can be filtered for.

Use `if` statements for more complex filtering, or for filtering on subproperties of events (for example, in dictionaries).

Standard event types and channels

For the standard Cumulocity events, there are constants that contain the channels for sending and receiving events, for example:

```
monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
send msmnt to Measurement.SEND_CHANNEL;
```

The events listed in the following table are part of the `com.apama.cumulocity` package.

Event	Channel for sending	Channel for receiving
Operation	Operation.SEND_CHANNEL	Operation.SUBSCRIBE_CHANNEL
Measurement	Measurement.SEND_CHANNEL	Measurement.SUBSCRIBE_CHANNEL
Event	Event.SEND_CHANNEL	Event.SUBSCRIBE_CHANNEL
Alarm	Alarm.SEND_CHANNEL	Alarm.SUBSCRIBE_CHANNEL
ManagedObject	ManagedObject.SEND_CHANNEL	ManagedObject.SUBSCRIBE_CHANNEL
MeasurementFragment	MeasurementFragment.SEND_CHANNEL	MeasurementFragment.SUBSCRIBE_CHANNEL

Measurement fragments

`Measurement` and `MeasurementFragment` events are always published.

You can generate listeners in EPL that will match on the contents of `MeasurementFragment` events rather than `Measurement` events. For example:

```
on all MeasurementFragment(type="c8y_SpeedMeasurement", valueFragment = "c8y_speed", valueSeries = "speedX", value > SPEED_LIMIT) as mf {
}
```

See also [Measurement fragments](#).

Distinguishing between create and update notifications

When listening for `Alarm`, `Event`, `ManagedObject` or `Operation` events from Cumulocity, you may want to distinguish between create and update operations. Each of these event types have actions named `isCreate()` and `isUpdate()` for this purpose.

Example for listening for new alarms:

```
on all Alarm() as alarm {
  if alarm.isCreate() {
    log "Alarm created: " + alarm.toString() at INFO;
  }
  // else it's an update
}
```

And similarly, only for updated alarms:

```

on all Alarm() as alarm {
  if alarm.isUpdate() {
    log "Alarm updated: " + alarm.toString() at INFO;
  }
  // else it's a create
}

```

For events that have come from Cumulocity, one of `isUpdate()` or `isCreate()` will always return true. Both actions are provided for choice and readability.

For more information, including examples for the different types of objects, see [Receiving update notifications](#) in the Apama documentation.

See also the [API Reference for EPL \(ApamaDoc\)](#) for more information about the `isCreate()` and `isUpdate()` actions.

EXAMPLE

This example listens for new measurements using the `com.apama.cumulocity.MeasurementFragment` API. It filters incoming measurements to find speed values above a given maximum speed and raises an alarm if the limit is breached.

1. Subscribe to the `MeasurementFragment.SUBSCRIBE_CHANNEL` channel.
2. Listen to the measurement fragment and filter on `type`, which is `c8y_SpeedMeasurement`. Ensure that `valueFragment` has the value `c8y_speed` and that `valuesSeries` filters on `speedX` only. Also filter on `value` when it is greater than `SPEED_LIMIT`.
3. Create the event using the constructor specifying all of the fields.
4. Send the event to the correct channel - `Alarm.SEND_CHANNEL`.

The resulting *.mon file can look like this:

```

using com.apama.cumulocity.Alarm;
using com.apama.cumulocity.MeasurementFragment;

monitor TriggerAlarmForSpeedBreach {
  constant float SPEED_LIMIT := 30.0;
  action onload() {
    monitor.subscribe(MeasurementFragment.SUBSCRIBE_CHANNEL);
    // Every time a measurement fragment with the specific details of the match criteria is triggered then we should raise an alarm
    on all MeasurementFragment(type="c8y_SpeedMeasurement", valueFragment = "c8y_speed", valueSeries = "speedX", value > SPEED_LIMIT) as mf {
      send Alarm("", "c8y_SpeedAlarm", mf.source, currentTime,
        "Speed limit breached", "ACTIVE", "CRITICAL", 1,
        new dictionary<string,any>() to Alarm.SEND_CHANNEL;
    }
  }
}

```

BUILT-IN ACTIONS

OVERVIEW

With Apama EPL, it is possible to utilize functions, called “actions”. Every monitor will have at least one action - the `onload` action. This section covers the already built-in actions ready to use.

See also the [API Reference for EPL \(ApamaDoc\)](#) for actions on the built-in types.

QUERYING CUMULOCITY DATA

To interact with your historical data, you can use one of the following request-response event pairs to look up resources.

Example: To look up alarms, you can send a `com.apama.cumulocity.FindAlarm` request event with appropriate query parameters to the `FindAlarm.SEND_CHANNEL` channel. In response, you can expect 0 or more `com.apama.cumulocity.FindAlarmResponse` events (depending on the number of resources that match the lookup request) and a `com.apama.cumulocity.FindAlarmResponseAck` event on the `FindAlarmResponse.SUBSCRIBE_CHANNEL` channel. Similar functionality is also provided for looking up managed objects, events, measurements and operations.

The events listed in the following table are part of the `com.apama.cumulocity` package.

To look up	Request-Response Events	Example
ManagedObject	FindManagedObject FindManagedObjectResponse FindManagedObjectResponseAck	Example
Alarm	FindAlarm FindAlarmResponse FindAlarmResponseAck	Example
Event	FindEvent FindEventResponse FindEventResponseAck	Example
Measurement	FindMeasurement FindMeasurementResponse FindMeasurementResponseAck	Example
Operation	FindOperation FindOperationResponse FindOperationResponseAck	Example
CurrentUser	CurrentUser GetCurrentUser GetCurrentUserResponse	Example
TenantOption	TenantOption FindTenantOptions FindTenantOptionsResponse	Documentation

INVOKING OTHER PARTS OF THE CUMULOCITY REST API

The Cumulocity REST API covers some extra functionality which is not covered with the individual event types. To invoke any other part of the REST API, a generic request-response API is provided which you can use to invoke any part of the Cumulocity API.

You can use the following request-response events:

- `com.apama.cumulocity.GenericRequest`
- `com.apama.cumulocity.GenericResponse`
- `com.apama.cumulocity.GenericResponseComplete`

INFO

The Apama-ctrl microservice, and thus all EPL apps code within it, runs with a number of permissions which permit the EPL to access all objects in the inventory and also read user details.

This includes personal identifiable information, such as username, email address, and so on.

For more information, see [REST implementation](#) in the Cumulocity OpenAPI Specification and [Invoking other parts of the Cumulocity REST API](#) in the Apama documentation.

INVOKING HTTP SERVICES

INFO

The information below is for interacting with *external* HTTP services. For making requests to other parts of the Cumulocity REST API, see [Invoking other parts of the Cumulocity REST API](#). For making requests to anything else on the platform, including other microservices, see [Connecting Apama to other microservices](#).

To interact with HTTP services using REST and JSON, create a `com.softwareag.connectivity.httpclient.HttpTransport` instance using one of the factory methods:

- `HttpTransport.getOrCreate(string host, integer port)` returns `HttpTransport`
- `HttpTransport.getOrCreateWithConfiguration(string host, integer port, dictionary <string, string> configurations)` returns `HttpTransport` (the keys in the configurations dictionary are the constants on `HttpTransport` with the `CONFIG_` prefix)

On the `HttpTransport` object, call one of the create methods, passing a path and payload as needed, to produce a `Request` object.

On the `Request` object, you may set cookies, headers or query parameters as needed, and can then invoke the request with the `execute(action<Response> callback)`. Supply the name of an action in your monitor for the callback, and it will be invoked with the `Response` when the request has completed (or timed out).

In the callback, the `Response` object is supplied with `statusCode` and `payload`. Fields on the payload are accessible via the `com.apama.util.AnyExtractor` object it is supplied in - see the information on [access fragments](#) below.

Refer to the [API Reference for EPL \(ApamaDoc\)](#) for further details.

UTILITY FUNCTIONS

Access fragments

You can access fragments via the `params` dictionary of most events. The `AnyExtractor` object can be constructed to help you extract data from any objects containing multiple subfragments and access:

- action `getInteger(string path)` returns integer
- action `getFloat(string path)` returns float
- action `getString(string path)` returns string
- action `getBoolean(string path)` returns boolean
- action `getSequence(string path)` returns `sequence<any>`
- action `getDictionary(string path)` returns `dictionary<any, any>`

You can use a JSON path to navigate in the object structure. For example:

```
string s := AnyExtractor(measurement.params["fragment"]).getString("sub.fragment.object");
```

Example "fragment": "c8y_TemperatureMeasurement".

Example "sub.fragment.object": "c8y_TemperatureMeasurement.T.Unit".

Casting "any" values

Alternatively, use a cast to convert an `any` to a particular type:

```
string s := <string> measurement.params["strfragment"];
```

Note that a cast operation will throw if the object is of a different type.

currentTime and the TimeFormatter

The read-only variable `currentTime` can be used to obtain the current server time. Apama deals with time using seconds since the Unix Epoch (1 Jan 1970 UTC). You can easily transform it to a human-readable form using the `TimeFormat` object. The `TimeFormat` object can be used for formatting dates and times, and also for parsing them.

Example:

```
using com.apama.correlator.timeformat.TimeFormat;

monitor Example {
  action onload {
    log TimeFormat.format(currentTime, "yyyy.MM.dd 'at' HH:mm:ss") at INFO;
  }
}
```

For more information on `TimeFormat` and its functions, see [Using the TimeFormat Event Library](#) in the Apama documentation and the API Reference for EPL (ApamaDoc).

inMaintenanceMode

The `Util.inMaintenanceMode()` function is a fast way to check if the device is currently in maintenance mode. It takes a managed object as a parameter and returns a boolean which is true if the device is in maintenance mode.

Example:

```
using com.apama.cumulocity.Measurement;
using com.apama.cumulocity.Event;
using com.apama.cumulocity.FindManagedObject;
using com.apama.cumulocity.FindManagedObjectResponse;
using com.apama.cumulocity.FindManagedObjectResponseAck;

using com.apama.cumulocity.Util;

monitor ExampleMonitor {
  action onload() {
    // Subscribe to Measurement.SUBSCRIBE_CHANNEL to receive all measurements
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
    monitor.subscribe(FindManagedObjectResponse.SUBSCRIBE_CHANNEL);
    on all Measurement() as m {
      integer reqId := integer.getUnique();
      send FindManagedObject(reqId, m.source, new dictionary<string,string>) to FindManagedObject.SEND_CHANNEL;
      on FindManagedObjectResponse(reqId = reqId, id = m.source) as d and not FindManagedObjectResponseAck(reqId = reqId) {
        if not Util.inMaintenanceMode(d.managedObject) {
          send Event("", "c8y_Event", m.source, currentTime, "Received measurement from active device", new dictionary<string,any>) to
Event.SEND_CHANNEL;
        }
      }
    }
  }
}
```

replacePlaceholders

To build strings, you can use concatenation as follows:

```
string s:= "An event with the text " + evt.text + " has been created.";
```

If the texts get longer and have more values that are dynamically set from the data, you can use the `Util.replacePlaceholders()` function. In your text string, you mark the placeholders with the field name from the event and surround it by `#{}` . The second parameter to `replacePlaceholders` can be any event type.

`Utils::replacePlaceholders` looks up the field name specified in the event or in the parameters of the event to generate the text replacement. You can use field names of type `#{X.Y}` to access nested structures in the event.

```
myMailText := Util.replacePlaceholders("The device #{source} created an event with the text #{text} at #{time}", alarm);
```

If the replacement string is of a form such as `#{source.name}` where `source.name` is the name of the underlying managed object/device or `#{source.c8y_Hardware.notes}` where `c8y_Hardware` is a fragment on the managed object, then special handling is required to achieve the replacement. After the initial replacement, you must update the placeholder field name and run

`Util::replacePlaceholders` again with the source `managedObject`.

```
myMailText := Util.replacePlaceholders("The device #{source} with the serial number #{source.c8y_Hardware.serialNumber} created an event with the text #{text} at #{time}. The device is located at #{source.c8y_Address.street} in #{source.c8y_Address.city}.", alarm);
myMailText := myMailText.replaceAll("#{source.", "#{");
myMailText := Util.replacePlaceholders(myMailText, managedObject);
```

ADVANCED FEATURES

CUSTOM FRAGMENTS

Cumulocity APIs let you structure your data freely. In Apama EPL, this is done by adding entries to `params`, which is of the type `dictionary<string, any>`. Each Cumulocity event in the `com.apama.cumulocity` package (such as `Alarm`, `Event`, `Measurement` or `Operation`) has a `params` field, which is translated to fragments or optional fields. Thus, when receiving events, your code must look up entries in the `params` field. When sending events, this can be done by defining event types, or you can use the `dictionary<string, any>` type. When receiving events, the EPL type is `dictionary<any, any>`. Note that EPL is strongly typed, so if you are creating an event with no fragments, a `new dictionary<string, any>` expression is required. If you are providing entries inline with a dictionary literal, then EPL determines the type based on the type of the first key-value pair - thus, for `dictionary<string, any>`, cast the first value to an `any` type with the `<any>` cast operator:

```
send Event(..., new dictionary<string,any>) to Event.SEND_CHANNEL;
send Event(..., {"fragment":<any>"value"}) to Event.SEND_CHANNEL;
```

The `MeasurementValue` type is provided for the measurements in the `Measurement` type. `MeasurementValue` has `value` and `unit` fields and `params` for other fragments.

Example 1:

```
send Measurement("", "c8y_TemperatureMeasurement", "12345", currentTime, {
  "c8y_TemperatureMeasurement":{
    "T1":MeasurementValue(1.0, "C", new dictionary<string,any>),
    "T2":MeasurementValue(2.0, "C", new dictionary<string,any>),
    "T3":MeasurementValue(3.0, "C", new dictionary<string,any>),
    "T4":MeasurementValue(4.0, "C", new dictionary<string,any>),
    "T5":MeasurementValue(5.0, "C", new dictionary<string,any>)
  }},
  new dictionary<string,any>) to Measurement.SEND_CHANNEL;
```

This will result in the following JSON structure:

```
{
  "type": "c8y_TemperatureMeasurement",
  "time": "...",
  "source": {
    "id": "12345"
  },
  "c8y_TemperatureMeasurement": {
    "T1": {
      "value": 1,
      "unit": "C"
    },
    "T2": {
      "value": 1,
      "unit": "C"
    },
    "T3": {
      "value": 1,
      "unit": "C"
    },
    "T4": {
      "value": 1,
      "unit": "C"
    },
    "T5": {
      "value": 1,
      "unit": "C"
    },
  },
}
```

MEASUREMENT FRAGMENTS

A measurement can be broken into individual measurement fragments. This can be done for each fragment and series present in the measurement. See [Cumulocity's domain model](#) for more information on measurement fragments.

Listen for events of type `com.apama.cumulocity.MeasurementFragment` when you require filtering based on measurement fragments or series, instead of listening for `com.apama.cumulocity.Measurement` events and looking inside the `measurements` dictionary. For more information, see [Using measurement fragments](#) in the Apama documentation.

LISTENERS

Triggering a statement by an arriving event is not the only possibility. The following sections cover other ways to combine listeners. Refer to [Defining Event Listeners](#) in the Apama documentation for full details.

Filters

Filters enable you to trigger by combinations or sequences of other triggers. If you have a trigger like this

```
on all Event() as e { ... }
```

it is also possible to add filters in the pattern.

```
on all Event(type = "c8y_EntranceEvent") as e { ... }
```

You can listen for more than one event:

```
on Event() as e and Alarm() as a { ... }
```

This will trigger on receiving an Event and an Alarm event - the first of each will be captured.

You can also trigger by sequences:

```
on all (Event() as e -> Alarm() as a) { ... }
```

This will trigger for every pair “Event followed by Alarm”. On receiving an event, it will stop listening for further events and start listening for alarms instead. Once an alarm is received, it will start listening for events again.

Timers

You can also trigger listeners based on time. You can either trigger in a certain interval, for example, fire every 5 minutes (300 seconds):

```
on all wait(300.0) { ... }
```

Or you can have a listener fire at certain times of the day, with similar functionality to Unix’s cron scheduler:

```
// timer:at(minutes, hours, daysOfMonth, month, daysOfWeek, (optional) seconds)
// minutes: 0-59
// hours: 0-23
// daysOfMonth: 1-31
// month: 1-12
// daysOfWeek: 0 (Sunday) - 6 (Saturday)
// seconds: 0-59

on all at(*, *, *, *, *) {} // trigger every minute

on all at(*/*10, *, *, *, *) {} // trigger every 10 minutes
on all at(0, 1, *, *, [1,3,5]) {} // trigger at 1am every monday, wednesday and friday
on all at(0, */2, 1,7, *, *) {} // trigger every 2 hours on every day in the first week of every month
```

You can also combine timer patterns with other patterns. For example, you can check if there was an event within a certain time after another event:

```
on Event() -> wait(600.0) and not Alarm() { ... }
```

This will trigger if there is an event and within 10 minutes (600 seconds) there is no alarm. Note the use of `not` which terminates the listener if the event occurs.

You can use a tenant option to set the time zone used for `on all at` timers. To set the tenant option, specify the `microservice.runtime` category and the `timezone` key. For example:

```
{
  "category": "microservice.runtime",
  "key": "timezone",
  "value": "Europe/Warsaw"
}
```

See also [Timezone variable](#) in this documentation and [Supported time zones](#) in the Apama documentation.

INFO

This tenant option is only read when the microservice starts. If the tenant option is changed, the microservice only picks this up on the next microservice subscription.

Streams - windows

Streams give you the possibility to operate on windows of events. Streams use the `from` keyword instead of `on` and define a window to operate over, and select what output they want from that window using aggregates. Windows can be restricted by two means:

1. Windows for a certain time - use the `within` keyword.

```
from m in all Measurement(type="c8y_TemperatureMeasurement") within 3600.0 select avg(m.measurements["c8y_TemperatureMeasurement"]["T"].value) as avgValue { }
```

2. Windows with a certain amount of events - use the `retain` keyword.

```
from m in all Measurement(type="c8y_TemperatureMeasurement") retain 100 select avg(m.measurements["c8y_TemperatureMeasurement"]["T"].value) as avgValue { }
```

Streams - outputting periodically

Streams can also control how frequently they evaluate, using the `every` specifier.

```
// will output the last measurement arrived every 1 minute
from m in all Measurement(type="c8y_TemperatureMeasurement") within 60.0 every 60.0 select
last(m.measurements["c8y_TemperatureMeasurement"]["T"].value) as lastValue { }

// will output the first of every 20 measurements arriving
from m in all Measurement(type="c8y_TemperatureMeasurement") retain 20 every 20 select first(m.measurements["c8y_TemperatureMeasurement"]
["T"].value) as firstValue { }

// will output the average of all 20 measurements after the 20th arrived
from m in all Measurement(type="c8y_TemperatureMeasurement") retain 20 every 20 select avg(m.measurements["c8y_TemperatureMeasurement"]
["T"].value) as avgValue { }
```

See the Apache documentation for [built-in aggregate functions](#).

CREATING OWN EVENT TYPES

As well as the predefined event types, you can define your own event types. These can be useful to detect patterns of events occurring which trigger other parts of the same module.

```
event MyEvent {
  Measurement m1;
  Measurement m2;
}

...

on Measurement() as m1 -> Measurement() as m2 {
  route MyEvent(m1, m2);
}
```

INFO

Cumulocity deploys each module into its own namespace, so event definitions from one module cannot be used in other modules. This prevents dependencies between modules.

CREATING OWN ACTIONS

Typically, you will structure a monitor using actions (much like functions in Java), as shown in the following examples.

Increasing the given severity:

```

action upgradeSeverity(string old) returns string {
  if old = "WARNING" { return "MINOR"; }
  if old = "MINOR" { return "MAJOR"; }
  if old = "MAJOR" { return "CRITICAL"; }
  return old;
}

```

Calculating the distance between two geo-coordinates:

```

action distance(float lat1, float lon1, float lat2, float lon2) returns float {
  float R := 6371000.0;
  float toRad := float.PI / 180.0;
  float lat1Rad := lat1 * toRad;
  float lat2Rad := lat2 * toRad;
  float deltaLatRad := (lat2-lat1) * toRad;
  float deltaLonRad := (lon2-lon1) * toRad;
  float a := (deltaLatRad/2.0).sin().pow(2.0) * lat1Rad.cos() * lat2Rad.cos() * (deltaLonRad/2.0).sin().pow(2.0);
  float c := 2.0 * a.sqrt().atan2((1.0-a).sqrt());
  return R * c;
}

```

VARIABLES

You can define variables in your modules.

```

string myEmailText := "Hello World";
sequence<string> supportedOperationsList := ["c8y_Restart", "c8y_Relay"];

```

If you define a monitor-scope variable (that is, inside a monitor but not within any actions on that monitor), then that can be used in a listener if you use a colon (:) instead of `as` for the event co-assignment in the listener. Thus, the example below logs the latest event every 10 seconds:

```

monitor MyMonitor {
  // monitor scope:
  Event e;
  action onload() {
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
    on all Event():e {}
    on all wait(10.0) {
      log e.toString();
    }
  }
}

```

When a listener starts, it takes a copy of all of the local variables. The example below thus logs each event after a 10 second delay, even if other events come in between.

```

monitor MyMonitor {
  // monitor scope:
  Event e;
  action onload() {
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
    on all Event():e {
      on all wait(10.0) {
        log e.toString();
      }
    }
  }
}

```

SPAWNING MONITOR INSTANCES AND CONTEXTS

While it is possible to handle multiple devices in a single monitor (for example, using `group by` and `partition by` in streams, or maintaining a dictionary keyed on the device ID for other state), it is often useful to separate processing of different devices into separate monitor instances.

New monitor instances can be created using the `spawn` statement. This takes a copy of the monitor's monitor scope variables and runs the named action in a new monitor instance. No listeners are copied into the new monitor. It is also possible to specify a context to spawn the new monitor instance in. Different contexts can run concurrently with each other, and also help isolate different monitors from each other. When constructing a context, supply a name to identify the context, and a boolean to control if the context is public - that is, it receives the Cumulocity events by default (sent to the default channel).

This pattern is often used with the `unmatched` keyword to identify events that are not matched by any other listeners in that context. By using a separate context for each monitor, the unmatched behavior is scoped to that monitor. For example:

```
monitor PerDeviceMeasurementTracker {
  action onload() {
    spawn factory to context("PerDeviceMeasurementTracker", true);
  }
  action factory() {
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
    on all unmatched Measurement() as m {
      spawn perDevice(m);
    }
  }

  dictionary<string, Measurement> latestMeasurementByType; // measurements for this device

  action perDevice(Measurement m) {
    processMeasurement(m);
    on all Measurement(source = m.source) as m {
      processMeasurement(m);
    }
  }
  action processMeasurement(Measurement m) {
    latestMeasurementByType[m.type] := m;
  }
}
```

BEST PRACTICES AND GUIDELINES

EPL MONITORS

Symptom: Your event processing rules are disabled automatically

If a monitor throws an exception from `onload` or a listener and the exception is not caught, then the monitor will be terminated. Catch exceptions or avoid the reason for them occurring.

Similarly, if a monitor completes processing an event and has no listeners left active, then it cannot be triggered again, and will automatically remove itself.

Avoid excessive memory usage per monitor

Make sure that your event processing rules do not leak listeners. For example, when doing request-response operations, ensure that no listeners are left active after the response is processed, or if a timeout occurs and there is no response.

NUMBER FORMATS

Cumulocity measurements use the float type. Note that the timestamps are stored as floats (seconds since 1 Jan 1970, 00:00 UTC).

SUBSCRIBING TO CHANNELS AND CONTEXTS

A context is a parallel processing unit within Apama. Monitor instances can be deployed to multiple contexts using the `spawn...to` syntax. When subscribing to a channel, all monitor instances within a context will receive events for that subscription. So it is recommended practice to put different subscriptions in different contexts. The use of contexts can prevent part of the application being overloaded from affecting other parts of the application.

Contexts are created with a user-friendly name, and each individual instance of the context object corresponds to different contexts, even if they have the same name.

For example:

```
action onload() {
    context subContext := context("Worker");
    spawn worker() to subContext;
}
action worker() {
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
    on all Measurement() as m {
        ...
    }
}
```

APAMA LIMITATIONS IN CUMULOCITY

Using Apama within the Cumulocity environment necessarily has some restrictions to the capabilities available when Apama is used standalone.

There are a number of ways that assets may be deployed to Apama within Cumulocity and the restrictions vary according to those mechanisms:

- EPL apps - the simplest mechanism to deploy Apama assets into a fully managed Apama correlator, see [Deploying apps](#).
- Custom microservices - where more complex Apama projects can be built using Cumulocity's [Microservice SDK](#).

When designing an Apama solution to be deployed within any form of Cumulocity environment, consider the following points.

General Apama limitations when using EPL apps or a custom microservice

- For scalability, a correlator may move between hosts and therefore does not have access to a persistent file system. It is a standard Cumulocity constraint that all microservices (either provided by the platform, or custom) must be stateless, see [Microservice SDK](#). The Apama features affected by this include:
 - MemoryStore persistence.
- Non-HTTP/REST connections to an external system or process are mostly impractical. Although if a service is available over the internet, then it can be used (for example, an HTTP client inside Apama could connect to publicly accessible HTTP servers). The Apama features affected by this include:
 - Connections between correlators.
- For security and implementing user access control, Cumulocity does not make the correlator port available to external processes, see [Microservice SDK](#). The following capabilities require access to the correlator port and hence are not compatible with this access control:
 - Command line tools such as `engine_connect`, `engine_management`, `engine_send`, `engine_receive`.
 - Correlator REST interface.
- To reduce both the memory usage of an application during startup and the application's startup time, ensure the application is completely initialized before injecting monitors that automatically unload, and before running time-consuming queries.

Specific Apama limitations when using EPL apps

- For ease of use, the correlator startup is controlled by Cumulocity. Thus, features that require you to change configuration files or command line options are not accessible. The Apama features affected by this include:
 - Connectivity plug-ins.
- For security, the file system used by the correlator is not accessible. The Apama features affected by this include:
 - Accessing the input log.
 - Using custom plug-ins.
 - Using file system assets for enrichment.
- For simplicity, it is only possible to make independent EPL injections. Each monitor is managed independently and so

dependencies between different monitors cannot be created. The Apama features affected by this include:

- A *.mon file must not contain a package statement (to do so is an error).
- It is not possible to share event definitions between separate *.mon files.

All of these restrictions are implemented to ensure the smooth and secure operation of EPL apps within Cumulocity.

EXAMPLES

CALCULATING AN HOURLY AVERAGE OF MEASUREMENTS

We are assuming the input data looks like this:

```
{
  "c8y_TemperatureMeasurement": {"T": {"value": "...", "unit": "C"}},
  "time": "...",
  "source": {"id": "..."},
  "type": "c8y_TemperatureMeasurement"
}
```

To create the average (mean), we need the following parts in the module:

- A time window over one hour, grouped by device (source).
- A `select` that returns the average calculation every hour, the source and the unit (as we must use an aggregate over the window contents, we select the last unit - we assume all measurements are of the same unit). Note the `AverageByDevice` event definition to hold these.
- Everything created as a new measurement.

For example:

```
using com.apama.aggregates.avg;
using com.apama.aggregates.last;
using com.apama.cumulocity.Measurement;

monitor HourlyAvgMeasurementDeviceContext {

  event AverageByDevice {
    string source;
    float avgValue;
    string unit;
  }

  action onload() {
    // Subscribe to Measurement.SUBSCRIBE_CHANNEL to receive all measurements
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);

    from m in all Measurement(type="c8y_TemperatureMeasurement") within (3600.0)
    group by m.source select
      AverageByDevice(m.source,
        avg(m.measurements["c8y_TemperatureMeasurement"]["T"].value),
        last(m.measurements["c8y_TemperatureMeasurement"]["T"].unit)) as avgdata {
        send Measurement("", "c8y_AverageTemperatureMeasurement", avgdata.source, currentTime,
          {"c8y_AverageTemperatureMeasurement":
            {
              "T": MeasurementValue(avgdata.avgValue, avgdata.unit, new dictionary<string,any>)
            }
          }, new dictionary<string,any>) to Measurement.SEND_CHANNEL;
      }
  }
}
```

CREATING ALARMS FROM BIT MEASUREMENTS

Devices often keep alarm statuses in registers and cannot interpret the meaning of alarms. In this example, we assume that a device just sends the entire register as a binary value in a measurement. A rule must identify the bits and create the respective alarm.

We create three dictionaries to map alarm text, type and severity for each of the bits, and an action to look up the value. We use -1 to indicate a default value, and replace <position> with the string form of the position.

```
dictionary<integer, string> positionToAlarmType := {
  0 : "c8y_HighTemperatureAlarm",
  1 : "c8y_ProcessingAlarm",
  2 : "c8y_DoorOpenAlarm",
  3 : "c8y_SystemFailureAlarm",
  -1 : "c8y_FaultRegister<position>Alarm"
};

dictionary<integer, string> positionToAlarmSeverity := {
  0 : "MAJOR",
  1 : "WARNING",
  2 : "MINOR",
  3 : "CRITICAL",
  -1 : "MAJOR"
};

dictionary<integer, string> positionToAlarmText := {
  0 : "The machine temperature reached a critical status",
  1 : "There was an error trying to process data",
  2 : "Door was opened",
  3 : "There was a critical system failure",
  -1 : "An undefined alarm was reported on position <position> in the binary fault register"
};

action getText(integer bitPosition, dictionary<integer, string> lookup) returns string {
  string template := lookup.getOr(bitPosition, lookup[-1]);
  return template.replaceAll("<position>", bitPosition.toString());
}
```

To analyze the binary measurement value, we will interpret it as a string value and loop through each character. The `getActiveBits()` function will do that and return a list of the bit positions at where the measurement had a "1". We can then use a `for` loop to iterate through that:

```

action getBitPositions(string binaryAsText) returns sequence<integer> {
  sequence<integer> bitsSet := new sequence<integer>;
  integer i := 0;
  while i < binaryAsText.length() {
    string character := binaryAsText.substring(i, i+1);
    if character = "1" {
      bitsSet.append(binaryAsText.length() - i - 1);
    }
    i:=i+1;
  }
  return bitsSet;
}

action onload() {
  // Subscribe to Measurement.SUBSCRIBE_CHANNEL to receive all measurements
  monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
  on all Measurement(type = "c8y_BinaryFaultRegister") as m {
    string faultRegister := m.measurements.getDefault("c8y_BinaryFaultRegister").getDefault("errors").value.toString();
    integer bitPosition;
    for bitPosition in getBitPositions(faultRegister) {
      Alarm alarm := new Alarm;
      alarm.type := getText(bitPosition, positionToAlarmType);
      alarm.severity := getText(bitPosition, positionToAlarmSeverity);
      alarm.text := getText(bitPosition, positionToAlarmText);
      alarm.source := m.source;
      alarm.time := m.time;
      alarm.status := "ACTIVE";
      send alarm to Alarm.SEND_CHANNEL;
    }
  }
}

```

Creating a measurement like this

```

{
  "c8y_BinaryFaultRegister": {"errors": {"value": 10110}},
  "time": "...",
  "source": {"id": "..."},
  "type": "c8y_BinaryFaultRegister"
}

```

will trigger the last statement three times.

- measurement at bit position 1 - c8y_ProcessingAlarm, WARNING, "There was an error trying to process data"
- measurement at bit position 2 - c8y_DoorOpenAlarm, MINOR, "Door was opened"
- measurement at bit position 4 - c8y_FaultRegister4Alarm, MAJOR, "An undefined alarm was reported on position 4 in the binary fault register"

and therefore create three alarms.

CONSUMPTION MEASUREMENTS

Assuming we have a sensor which measures the current fill level of something and sends the values on a regular basis to Cumulocity, we can easily create additional consumption values. Calculating the absolute difference between two measurements can be useful, but it will only give you a clear view if the measurements are sent always in the same interval. Therefore, we will put the absolute difference in relation to the time difference and calculate as a per hour consumption.

We will compare the value and time difference of two adjacent measurements for a device, using a stream retaining 2 entries, and selecting the first and last timestamp and value.

```

using com.apama.aggregates.last;
using com.apama.aggregates.first;
using com.apama.aggregates.count;

monitor FillLevelMeasurements {

  event FillLevel {
    float firstValue;
    float firstTime;
    float lastValue;
    float lastTime;
    string source;
  }

  action calculateConsumption(FillLevel l) returns float {
    if(l.firstTime = l.lastTime) {
      return 0.0;
    } else {
      return ((l.lastValue - l.firstValue) * 3600.0) / (l.lastTime - l.firstTime);
    }
  }

  action onload() {
    // Subscribe to Measurement.SUBSCRIBE_CHANNEL to receive all measurements
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
    from m in all Measurement(type = "c8y_WaterTankFillLevel") partition by m.source retain 2 group by m.source having count() = 2
    select FillLevel(first(m.measurements["c8y_WaterTankFillLevel"]["level"].value), first(m.time),
      last(m.measurements["c8y_WaterTankFillLevel"]["level"].value), last(m.time), m.source) as fill {

      Measurement m := new Measurement;
      m.type := "c8y_HourlyWaterConsumption";
      m.time := currentTime;
      m.source := fill.source;
      MeasurementValue mv := new MeasurementValue;
      mv.value := calculateConsumption(fill);
      mv.unit := "l/h";
      m.measurements[m.type] := {"consumption":mv};
      send m to Measurement.SEND_CHANNEL;
    }
  }
}

```

MISCELLANEOUS SAMPLE APPS

The EPL editor in the Streaming Analytics application provides several sample apps which demonstrate how to use Apama EPL, for example, to query for Cumulocity objects or to create alarms. You can use these samples to build your own apps.

STUDY - CIRCULAR GEOFENCE ALARMS

OVERVIEW

This section gives an in-depth example how you can create more complex rules. It uses multiple of the features explained before in the other sections of this guide.

If you are just starting with Apama EPL, take a look at [Examples](#).

PREREQUISITES

Goal

We want our tracking devices that are continuously sending location events to automatically generate alarms if they move outside a geofence. This geofence will be a circle and should be configurable for each device separately. The alarm will be created at the moment the device moves outside the geofence. While it is moving outside, it should not create new alarms because the first one will remain active. As soon as the device moves back into the geofence, the alarm will be cleared.

Cumulocity data model

Location event structure (the part we need):

```
{
  "id": "...",
  "source": {"id": "..."},
  "text": "...",
  "time": "...",
  "type": "...",
  "c8y_Position": {"alt": ..., "lng": ..., "lat": ...}
}
```

We store the geofence configuration in the device (the radius will be configured in meters):

```
{
  "c8y_Geofence": {"lat": ..., "lng": ..., "radius": ...}
}
```

Additionally, we want to enable/disable the geofence alarms for each device without removing the configuration entirely. We will do that by adding/removing "c8y_Geofence" to `c8y_SupportedOperations` in the device:

```
{
  "c8y_SupportedOperations": [..., "c8y_Geofence", ...]
}
```

Calculation

The device is outside of the geofence if the distance between the current position and the center is bigger than the configured radius of the geofence. What we need is a function that can calculate the difference between *two* geo-coordinates:

```
action distance(float lat1, float lon1, float lat2, float lon2) returns float {
  float R := 6371000.0;
  float toRad := float.PI / 180.0;
  float lat1Rad := lat1 * toRad;
  float lat2Rad := lat2 * toRad;
  float deltaLatRad := (lat2-lat1) * toRad;
  float deltaLonRad := (lon2-lon1) * toRad;
  float a := (deltaLatRad/2.0).sin().pow(2.0) * lat1Rad.cos() * lat2Rad.cos() * (deltaLonRad/2.0).sin().pow(2.0);
  float c := 2.0 * a.sqrt().atan2((1.0-a).sqrt());
  return R * c;
}
```

The above action will return the distance in meters.

STEP 1: FILTERING THE INPUT

The main input for this module will be events. To discard non-matching events as early as possible, we perform this as the first check in the listener:

```
monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
on all Event() as e {
  if e.params.containsKey("c8y_Position") {
    // we have an event
  }
}
```

STEP 2: COLLECTING NECESSARY DATA

In the next step, we need the configuration of the geofence for the calculation and grab it.

```
monitor.subscribe(FindManagedObjectResponse.SUBSCRIBE_CHANNEL);
...
integer reqId := integer.getUnique();
send FindManagedObject(reqId, e.source, new dictionary<string,string>) to FindManagedObject.SEND_CHANNEL;
on FindManagedObjectResponse(reqId = reqId) as resp
  and not FindManagedObjectResponseAck(reqId) {
    ManagedObject dev := resp.managedObject;
  }
```

STEP 3: CHECKING IF THE DEVICE SUPPORTS C8Y_GEOFENCE

With the device available we will now check if there is a geofence configured for the device and if it is activated (contains "c8y_Geofence" in `supportedOperations`). To check the `c8y_SupportedOperations` array, we can use the `indexOf()` function. This function will loop through all elements and return the index of that entry, or a negative number if the value is not present. For the configuration, we will just check if the device contains the fragment "c8y_Geofence".

Once we have an event and a device, we extract the data from the event's `c8y_Position` and the device's `c8y_Geofence`. These objects are mapped to `dictionary<any, any>` entries in the `params`. As the `params` hold values of type `any`, we must cast to a `dictionary<any, any>`.

```
if(dev.params.containsKey("c8y_Geofence") and dev.supportedOperations.indexOf("c8y_Geofence") >= 0) {
  dictionary<any, any> evtPos := <dictionary<any, any> > e.params["c8y_Position"];
  float eventLat := <float> evtPos["lat"];
  float eventLng := <float> evtPos["lng"];

  dictionary<any,any> devGeofence := <dictionary<any,any> > dev.params["c8y_Geofence"];
  float centerLat := <float> devGeofence["lat"];
  float centerLng := <float> devGeofence["lng"];
  float maxDistance := <float> devGeofence["radius"];
}
```

STEP 4: CREATING THE TRIGGER

As mentioned earlier, the device is outside of the fence if the distance between the current device position and the geofence center is bigger than the configured geofence radius. To trigger the alarm, we need two events so we can check if the device entered or left the geofence within these two events.

In the first step, we calculate the distance with the function mentioned earlier:

```
float d := distance(centerLat, centerLng, eventLat, eventLng);
```

Now we re-route this as an event with:

```

event LocationEventWithDistance {
  string source;
  float distance;
  Event e;
  float maxDistance;
}

...

route LocationEventWithDistance(e.source, d, e, maxDistance);

```

We place the source in the event so we can easily match it in a listener.

We now set up a listener triggered by the event `LocationEventWithDistance`, listening for the next `LocationEventWithDistance` - for the same source:

```

on all LocationEventWithDistance() as firstPos {
  on LocationEventWithDistance(source = firstPos.source) as secondPos {
    // now have two events with distances
  }
}

```

This pair of `LocationEventWithDistance` events now holds all data for checking if we should create the alarm or not. Note that we are filtering the `secondPos` event to be for the same source as the first - there will be an active listener for every device we have received an event from.

STEP 5: CREATING THE ALARM

To create the alarm, we now need two events where the first one has a distance smaller than the radius and the second one has a distance bigger than the radius. This would mean that the device just left the geofence.

```

if firstPos.distance <= firstPos.maxDistance and
secondPos.distance > secondPos.maxDistance {
  send Alarm("", "c8y_GeofenceAlarm", firstPos.source, currentTime,
    "Device moved out of circular geofence", "ACTIVE",
    "MAJOR", 1, new dictionary<string,any>) to Alarm.SEND_CHANNEL;
}

```

STEP 6: CLEARING THE ALARM

To clear the alarm, we must just switch the condition at the bottom and additionally grab the currently active alarm to get its ID. We do not need to care about whether there is an existing alarm at this point. If there is none, the listener will trigger the `and not FindAlarmResponseAck`, terminating the listener:

```

monitor.subscribe(FindAlarmResponse.SUBSCRIBE_CHANNEL);
...
if firstPos.distance > firstPos.maxDistance and
secondPos.distance <= secondPos.maxDistance {
  integer reqId:= integer.getUnique();
  send FindAlarm(reqId, {"source": firstPos.source,
    "status": "ACTIVE", "type": "c8y_GeofenceAlarm"}) to FindAlarm.SEND_CHANNEL;
  on FindAlarmResponse(reqId=reqId) as alarmResponse
    and not FindAlarmResponseAck(reqId=reqId) {
    send Alarm(alarmResponse.id, "c8y_GeofenceAlarm",
      firstPos.source, currentTime, "Device moved back into circular geofence",
      "CLEARED", alarmResponse.alarm.severity, 1, new dictionary<string, any>) to Alarm.SEND_CHANNEL;
  }
}

```

PUTTING EVERYTHING TOGETHER

We can now combine all the parts into one module. The order of the listeners does not matter.

```
using com.apama.cumulocity.ManagedObject;
using com.apama.cumulocity.Measurement;
using com.apama.cumulocity.Event;
using com.apama.cumulocity.Alarm;
using com.apama.cumulocity.FindManagedObject;
using com.apama.cumulocity.FindManagedObjectResponse;
using com.apama.cumulocity.FindManagedObjectResponseAck;
using com.apama.cumulocity.FindAlarm;
using com.apama.cumulocity.FindAlarmResponse;
using com.apama.cumulocity.FindAlarmResponseAck;

monitor MonitorDevicesForCircularGeofence {

  event LocationEventWithDistance {
    string source;
    float distance;
    Event e;
    float maxDistance;
  }

  action onload {
    monitor.subscribe(Measurement.SUBSCRIBE_CHANNEL);
    monitor.subscribe(FindManagedObjectResponse.SUBSCRIBE_CHANNEL);
    monitor.subscribe(FindAlarmResponse.SUBSCRIBE_CHANNEL);
    on all Event() as e {
      if e.params.containsKey("c8y_Position") {
        // we have an event
        integer reqId := integer.getUnique();
        send FindManagedObject(reqId, e.source, new dictionary<string,string> to FindManagedObject.SEND_CHANNEL;
        on FindManagedObjectResponse(reqId = reqId) as resp
        and not FindManagedObjectResponseAck(reqId) {
          ManagedObject dev := resp.managedObject;

          if(dev.params.containsKey("c8y_Geofence") and dev.supportedOperations.indexOf("c8y_Geofence") >= 0) {

            dictionary<any, any> evtPos := <dictionary<any, any> > e.params["c8y_Position"];
            float eventLat := <float> evtPos["lat"];
            float eventLng := <float> evtPos["lng"];

            dictionary<any,any> devGeofence := <dictionary<any,any> > dev.params["c8y_Geofence"];
            float centerLat := <float> devGeofence["lat"];
            float centerLng := <float> devGeofence["lng"];
            float maxDistance := <float> devGeofence["radius"];

            float d := distance(centerLat, centerLng, eventLat, eventLng);

            route LocationEventWithDistance(e.source, d, e, maxDistance);
          }
        }
      }
    }

    on all LocationEventWithDistance() as firstPos {
      on LocationEventWithDistance(source = firstPos.source) as secondPos {
        // now have two events with distances
        if firstPos.distance <= firstPos.maxDistance and
          secondPos.distance > secondPos.maxDistance {
          send Alarm("", "c8y_GeofenceAlarm", firstPos.source, currentTime,
            "Device moved out of circular geofence", "ACTIVE",
            "MAJOR", 1, new dictionary<string,any>) to Alarm.SEND_CHANNEL;
        }

        if firstPos.distance > firstPos.maxDistance and
          secondPos.distance <= secondPos.maxDistance {
```

```

integer reqId:= integer.getUnique();
send FindAlarm(reqId, {"source": firstPos.source,
"status": "ACTIVE", "type": "c8y_GeofenceAlarm"}) to FindAlarm.SEND_CHANNEL;
on FindAlarmResponse(reqId=reqId) as alarmResponse
and not FindAlarmResponseAck(reqId=reqId) {
  send Alarm(alarmResponse.id, "c8y_GeofenceAlarm",
    firstPos.source, currentTime, "Device moved back into circular geofence",
    "CLEARED", alarmResponse.alarm.severity, 1, new dictionary<string, any>) to Alarm.SEND_CHANNEL;
}
}
}
}
}

action distance(float lat1, float lon1, float lat2, float lon2) returns float {
  float R := 6371000.0;
  float toRad := float.PI / 180.0;
  float lat1Rad := lat1 * toRad;
  float lat2Rad := lat2 * toRad;
  float deltaLatRad := (lat2-lat1) * toRad;
  float deltaLonRad := (lon2-lon1) * toRad;
  float a := (deltaLatRad/2.0).sin().pow(2.0) * lat1Rad.cos() * lat2Rad.cos() * (deltaLonRad/2.0).sin().pow(2.0);
  float c := 2.0 * a.sqrt().atan2((1.0-a).sqrt());
  return R * c;
}
}

```

CONNECTING APAMA TO OTHER MICROSERVICES


OVERVIEW

Streaming analytics applications using Apama can make use of applications running in other microservices. This section uses the `/health` endpoint of an Apama-ctrl microservice, but the steps apply to connecting to any other microservice running inside Cumulocity. This section is going to show you how to create a connection to the Cumulocity platform from within Apama EPL which can be used to invoke other microservices directly. It will then show you how to make a request and decode the result.

We will assume that you are developing an EPL app using the EPL editor that is part of the Streaming Analytics application and demonstrate a request to a microservice. The steps in this guide will also work with any other way you could be creating an Apama application and can be used to interact with any microservice.

We will be making use of the `CumulocityRequestInterface` API. For more technical information about this API, see [Invoking microservices](#) in the Apama documentation.

CREATING AN EPL APP

Click the Streaming Analytics icon  in the application switcher. On the resulting home screen, navigate to the **EPL Apps** page and then click **New EPL app**. You will now see an EPL editor window in which to create the app which interacts with another microservice.

CONNECTING TO THE CUMULOCITY PLATFORM

To support making these requests, we provide a helper event with actions to automatically connect to the Cumulocity platform and then create requests which can be used to call other microservices. This helper event is called `CumulocityRequestInterface` and is within the `com.apama.cumulocity` package. This helper event provides a static action which will connect to Cumulocity and return an instance of the event. It can automatically connect either from within a microservice or the Cumulocity platform itself, or from a remote correlator. That instance has an action which will create a request to call a specific microservice.

To create the connection from your own code, simply call the `connectToCumulocity` method and store the result:

```
CumulocityRequestInterface cumulocity := CumulocityRequestInterface.connectToCumulocity();
```

This will automatically create a connection using the credentials and connection details provided to your microservice, or using the configuration for the Cumulocity transport when connecting from an external Apama instance.

MAKING MICROSERVICE REQUESTS

The `CumulocityRequestInterface` instance has an action on it to create a request:

```
/**
 * Allows creation of a request on a transport that
 * has been configured for a Cumulocity connection.
 *
 * @param method The type of HTTP request, for example "GET".
 * @param path A specific path to be appended to the request.
 * @param payload A dictionary of elements to be included in the request.
 */
action createRequest(string method, string path, any payload) returns Request
```

This takes the HTTP method to use (usually GET, PUT or POST), a path including the Cumulocity service prefix (typically something like `/service/serviceName/path/on/service`) and the payload. The payload will be converted to a JSON document before submitting to the microservice. The action returns a `Request` object which is part of the HTTP Client interface, documentation of which can be found in the [API Reference for EPL \(ApamaDoc\)](#).

Requests are executed with a call-back action as an argument which will be invoked when the request is completed with the response as an argument. If you must set any options, query parameters or headers on the request, you can set those on the `Request` object before calling it. For example:

```
action responseCallback(Response resp) {
    string objectId := resp.payload.getString("id");
    ...
}
...
Request req := cumulocity.createRequest("GET", "/service/otherService/data", any());
req.setQueryParameter("type", "object");
req.execute(responseCallback);
```

The response will also be decoded from JSON and the response payload uses the `AnyExtractor` pattern which you can find linked from the `Response` event in the HTTP Client transport documentation. The above example will be equivalent to the REST request `GET http://cumulocity/service/otherService/data?type=object`.

EXAMPLE REQUEST TO A MICROSERVICE ENDPOINT

The following is a very simple application that shows how to query another microservice. We are using the `/health` endpoint of an Apama-ctrl microservice as an example.

We will start with EPL which connects to Cumulocity and calls an action to send the request.

```
using com.apama.cumulocity.CumulocityRequestInterface;
using com.softwareag.connectivity.httpclient.Request;
using com.softwareag.connectivity.httpclient.Response;

monitor CallAnotherMicroservice {

    CumulocityRequestInterface requestiface;

    action onload() {
        requestiface := CumulocityRequestInterface.connectToCumulocity();
        sendHealthRequest();
    }
}
```

Sending the request

First, we create the `Request` with: the request type, the request path, and the payload `any()` because in this example we do not need to put anything in the payload.

We then use `execute` to send the request and provide an action to be called with the response.

```
action sendHealthRequest()
{
  Request healthRequest:=
    requestIface.createRequest("GET", "/service/cep/health", any());
  healthRequest.execute(responseHandler);
}
```

We use an Apama-ctrl microservice for this example, which has the context path of `/cep`. To modify this for another microservice, substitute `/cep` with the context path as defined in the manifest for your microservice. The `/health` endpoint completes the request path for this example, but could be replaced with any valid endpoint of the microservice.

Receiving the response

Here is the defined action that we used when sending the request. This action is called in response to the sent request and is provided with the `Response` object.

For this example, we simply log the status code and the body of the response.

```
action responseHandler(Response healthResponse)
{
  integer statusCode := healthResponse.statusCode;
  string payload := healthResponse.payload.data.toString();
  log "Health response status code = " + statusCode.toString() +
    ", response body = " + payload at INFO;
}
```

OTHER MICROSERVICES

This section was demonstrating talking to an Apama-ctrl microservice. However, you can also access any other microservice through Cumulocity as long as it uses standard REST requests with JSON payloads. You must simply construct the appropriate `/service` URL using the name of your microservice followed by the path of the request within your microservice.

USING CUMULOCITY MQTT SERVICE

The Cumulocity MQTT Service is an MQTT endpoint to Cumulocity which supports publishing and subscribing arbitrary payloads. These need to be processed within Cumulocity into a format that can be consumed by the platform. This connectivity allows for that processing to be done in Streaming Analytics. The MQTT Service allows for messages to be processed by Streaming Analytics before being stored in the platform and for Streaming Analytics to send messages to the device.

INFO

The Cumulocity MQTT Service feature is currently in Public Preview and may be subject to change in the future.

MQTT Service samples are provided to demonstrate how to consume and publish device-specific messages using the DeviceService API. See [Create an EPL App](#) for details.

For further documentation on using the MQTT Service generally, see [MQTT Service](#), and for using it within EPL Apps, see [the Apama documentation](#).

Some specific EPL Apps notes:

- You do not need to add any bundles.
- To use the `BASE64__FORMAT` , you need to [create and upload a custom extension](#) to parse the binary data.

CUSTOMIZATION AND PERMISSIONS

OPTIMIZING REQUESTS WITH CONCURRENT CONNECTIONS

In order to provide better performance for requests to the Cumulocity platform, Streaming Analytics uses multiple client connections to perform requests concurrently. This can provide improved performance, but may also change the ordering in which requests are executed and responses are returned. By default, the Cumulocity transport tries to use multiple connections and restricts ordering to avoid races that may affect your EPL application.

An attempt is made to ensure order is maintained when required. For example, all updates to a single managed object are performed serially in the order they were sent to the transport. For more details see [Optimizing requests to Cumulocity with concurrent connections](#) in the Apama documentation.

You can adjust the default number of client connections with the `client.numClients` tenant option in the `streaminganalytics` category. For example:

```
{
  "category": "streaminganalytics",
  "key": "client.numClients",
  "value": "5"
}
```

If you require a fully serial transport, set the value of `client.numClients` to 1.

INFO

This does not apply to the Apama-ctrl-smartrules, Apama-ctrl-smartrulesmt, and Apama-ctrl-mt microservices. They have a fix value of 1 (that is, fully serial) for this option, which is not configurable.

CONTROLLING ACCESS TO THE STREAMING ANALYTICS APPLICATION

By default, the Streaming Analytics application gives you access to the **Analytics Builder** and **EPL Apps** pages. Administrators may wish to control which of these are shown on different tenants or for different users, or modify the wording of the cards on the home screen, see also [Customizing the home screen of the Streaming Analytics application](#).

Which pages are available also depends on the variant of the Apama-ctrl microservice that is running.

- If the microservice is not running, an error message is shown indicating that the microservice cannot be accessed, and only a card with information about smart rules is shown on the home screen.
- If the Apama-ctrl-starter microservice is running, the **EPL Apps** page is not shown (and cannot be enabled) as the EPL apps functionality is not available in Apama-ctrl-starter.
- If the Apama-ctrl-mt microservice is running, the **Analytics Builder** page is shown for all subscribed tenants. The **EPL Apps** page is shown on the tenant that owns the microservice, but not shown (and cannot be enabled) on the subtenants.
- If the Apama-ctrl-smartrules or Apama-ctrl-smartrulesmt microservice is running, neither the **Analytics Builder** nor the **EPL Apps** page is shown (and cannot be enabled). In this case, only the card with information about smart rules is shown on the home screen.
- For other variants of the Apama-ctrl microservice, both the **Analytics Builder** and **EPL Apps** pages are shown by default.

For an entire tenant, if a "feature application" named `feature-disable-analyticsbuilder` and/or `feature-disable-eplapps` is available within the tenant, then the relevant part is disabled. This can be done by an Enterprise tenant or Management tenant (see also [Managing tenants](#)) and then subscribing to subtenants (the subtenant administrators are then not able to unsubscribe this application if

the parent tenant wishes to restrict access to the functionality). To create such a “feature application”, send a POST request to `/application/applications` (as an administrator with the permission to create applications). For example, to disable Analytics Builder:

```
{
  "name": "feature-disable-analyticsbuilder",
  "contextPath": "feature-disable-analyticsbuilder",
  "type": "HOSTED",
  "resourcesUrl": "/",
  "manifest": {
    "noAppSwitcher": true
  },
  "key": "feature-disable-analyticsbuilder-key"
}
```

Or to disable EPL Apps:

```
{
  "name": "feature-disable-eplapps",
  "contextPath": "feature-disable-eplapps",
  "type": "HOSTED",
  "resourcesUrl": "/",
  "manifest": {
    "noAppSwitcher": true
  },
  "key": "feature-disable-eplapps-key"
}
```

You can also send the POST request using a curl command, for example, to disable Analytics Builder:

```
curl --user username -X POST -H 'Content-Type: application/json' -d '{"name": "feature-disable-analyticsbuilder", "contextPath": "feature-disable-analyticsbuilder", "type": "HOSTED", "resourcesUrl": "/", "manifest": {"noAppSwitcher": true}, "key": "feature-disable-analyticsbuilder-key"}' -k https://{{hostname}}/application/applications/
```

Or to disable EPL Apps:

```
curl --user username -X POST -H 'Content-Type: application/json' -d '{"name": "feature-disable-eplapps", "contextPath": "feature-disable-eplapps", "type": "HOSTED", "resourcesUrl": "/", "manifest": {"noAppSwitcher": true}, "key": "feature-disable-eplapps-key"}' -k https://{{hostname}}/application/applications/
```

By default, all users can see the same set of pages (according to the limitations above). You can also restrict the visibility of the pages to only users who have the permission `ROLE_ANALYTICSBUILDER_READ` or `ROLE_EPLAPPS_READ`, see also [Managing permissions and roles](#). To enable this, set the category of the tenant option to `streaminganalytics` and the `applicationAccess` key to the value “role”, see the [Tenant API](#) in the Cumulocity OpenAPI Specification, or use a curl command as given in the example below:

```
curl --user username -X POST -H 'Content-Type: application/json' -d '{"category": "streaminganalytics", "key": "applicationAccess", "value": "role"}' -k https://mytenant/tenant/options
```

You must replace the username with the name of a user who has ADMIN permission for “Option management”.

Note that this only affects the visibility of the cards and pages in the Streaming Analytics application. The [supported REST services](#) only require READ and ADMIN permissions for “CEP management”.

CUSTOMIZING THE HOME SCREEN OF THE STREAMING ANALYTICS APPLICATION

The cards that are shown on the home screen of the Streaming Analytics application contain text and links which you can customize on a

per-tenant and per-language basis. To do this, download the *documentation.json* file for the language you wish to customize from the URL `/service/cep/apamacorrelator/EN/documentation.json` (this must be authenticated for a user in the Cumulocity tenant). Replace the "EN" within the URL with the language code for the file you want to download.

The *documentation.json* file includes the URLs for the documentation links across the Streaming Analytics application and the text that is shown on the home screen. You can modify this to your requirements.

After you have made all required changes, package the modified copies into a ZIP file containing the following files:

- *files/support/cumulocity/EN/documentation.json*
- *cumulocity.json*

The *cumulocity.json* file contains the following:

```
{
  "contextPath": "streaminganalytics-customization",
  "availability": "MARKET",
  "type": "HOSTED",
  "name": "streaminganalytics-customization",
  "key": "streaminganalytics-customization-key",
  "noAppSwitcher": true
}
```

Then upload the ZIP file using the Administration application: go to **Ecosystem > Applications**, click **Add application**, and select the method **Upload web application**. See [Managing applications](#) for detailed information.

You may need to clear your browser cache for the changes to take effect.

You can include multiple languages in a single ZIP file as needed. You can subscribe this to subtenants as needed from an Enterprise tenant.

On new releases of the platform, it is recommended that you review the source *documentation.json* file for any changes. New entries in this file will be picked up with their default values.

CONFIGURATION REQUIREMENTS FOR NOTIFICATIONS 2.0

Streaming Analytics can use the Cumulocity Notifications 2.0 reliable data forwarding capability to receive notifications for measurements, events, alarms, managed objects and operations that are processed by the Cumulocity platform. For more information, see [About Notifications 2.0](#) in the Cumulocity OpenAPI Specification.

The availability of this feature is governed by two feature flags:

- `notification2.streaming-analytics`
The Cumulocity Notifications 2.0 feature is currently in Private Preview. If you want to have it enabled for your tenant, you must contact [product support](#) to set this feature flag.
- `streaming-analytics.messaging`
In addition to this, if you are using one of the variants of the Apama-ctrl microservice, you must also set this feature flag. You then need to resubscribe the Apama-ctrl microservice to pick up changes to the feature flag. Use the feature toggles REST API to do so; see [Feature toggles API](#) in the Cumulocity OpenAPI Specification.

If you are using a custom microservice, you must also add the `ROLE_NOTIFICATION_2_ADMIN` permission to the microservice manifest once the `notification2.streaming-analytics` feature flag has been activated; see also [Required settings in the microservice manifest](#). For the Apama-ctrl microservices, it is not required to add this permission manually as it is set as the default; see also [Modifying microservice permissions and resource usage](#).

INFO

Keep in mind that you must also resubscribe the microservice to pick up changes to the feature flag.

MODIFYING MICROSERVICE PERMISSIONS AND RESOURCE USAGE

The resource usage and permissions that the Apama-ctrl microservice operates with are defined in the manifest file of the Apama-ctrl microservice. See [Microservice manifest](#) for more information.

If you have access to the microservice image (typically available only to operations), then you are able to extract the microservice image, modify the manifest, rebuild the microservice, and reupload the microservice to Cumulocity as an application in the Administration application.

The manifest specifies CPU and memory resource usage. In some circumstances, these must be changed (different sizes of the microservice image are provided with different configurations).

The manifest also specifies the permissions with which the microservice runs. This is the set of permissions that every request from EPL (or any other code running in the Apama-ctrl microservice) runs with. The Apama-ctrl microservice itself requires the following permissions:

- ROLE_APPLICATION_MANAGEMENT_READ
- ROLE_INVENTORY_READ
- ROLE_INVENTORY_ADMIN
- ROLE_INVENTORY_CREATE
- ROLE_MEASUREMENT_READ
- ROLE_MEASUREMENT_ADMIN
- ROLE_EVENT_READ
- ROLE_EVENT_ADMIN
- ROLE_ALARM_READ
- ROLE_ALARM_ADMIN
- ROLE_DEVICE_CONTROL_READ
- ROLE_DEVICE_CONTROL_ADMIN
- ROLE_IDENTITY_READ
- ROLE_IDENTITY_ADMIN
- ROLE_CEP_MANAGEMENT_READ
- ROLE_CEP_MANAGEMENT_ADMIN
- ROLE_OPTION_MANAGEMENT_READ
- ROLE_SMS_ADMIN
- ROLE_AUDIT_ADMIN
- ROLE_AUDIT_READ
- ROLE_USER_MANAGEMENT_READ
- ROLE_USER_MANAGEMENT_OWN_READ
- ROLE_TENANT_MANAGEMENT_READ
- ROLE_BULK_OPERATION_ADMIN
- ROLE_BULK_OPERATION_READ
- ROLE_NOTIFICATION_2_ADMIN

You can add other permissions to this list (or remove them from it) to grant (or remove) permissions to EPL code.

TROUBLESHOOTING AND DIAGNOSTICS

If you have the Apama-ctrl-smartrules or Apama-ctrl-smartrulesmt microservice, most of the functionality described in this topic does not apply.

DOWNLOADING DIAGNOSTICS AND LOGS

INFO

Diagnostics are not available for the Apama-ctrl-smartrules and Apama-ctrl-smartrulesmt microservices.

To download diagnostics information, you need READ permission for “CEP management”. See [Managing permissions and roles](#) for more information.

INFO

ADMIN permission for “CEP management” does not include READ permission.

If you have READ permission for “CEP management”, links for downloading diagnostics information are available when you click the **User** button in the Streaming Analytics application. This opens the right drawer which contains a **Diagnostics** section with the following links:

- A **Basic diagnostics (ZIP)** link for downloading basic diagnostics information. This should typically be a few megabytes and take about 5 seconds to generate.
- An **Enhanced diagnostics (ZIP)** link for downloading enhanced, more resource-intensive diagnostics information.

It may be useful to capture this diagnostics information when experiencing problems, or for debugging EPL apps. It is also useful to provide to [product support](#) if you are filing a support ticket. You can find the tenant ID in the **Platform info** section of the right drawer. If you want to find out the version numbers of the different components on your tenant, click the **Download platform details** button in the **Platform info** section and then open the downloaded JSON file. See [User options and settings](#) for more details.

Basic diagnostics information is provided in a ZIP file named *diagnostic-overview<timestamp>.zip* and includes the following information:

- The microservice log file contents, if available, including a record of the correlator’s startup logging and the last hour or maximum of 20,000 lines of logging.
- Apama-internal diagnostics information (similar to the `engine_watch` and `engine_inspect` command-line tools available in Apama).
- A copy of all EPL apps, smart rules and analytic models.
- A copy of any alarms that the Apama-ctrl microservice has raised.
- CPU profiling (over a duration of 5 seconds).
- EPL memory profiler snapshots.
- Some information from the environment (tenant details, environment variables).
- Version information for the components.

Enhanced diagnostics information is provided in a ZIP file named *diagnostic-enhanced<timestamp>.zip* and includes the following information:

- Contains what is in the above-mentioned *diagnostic-overview<timestamp>.zip* file.
- In addition, it includes requests that are more resource-intensive and may significantly slow down the correlator. These include the contents of the queues, CPU usage, and so on.

What you can see or do depends on your permissions:

- If you have only READ permission for “CEP management”, you have read-only access to EPL apps and analytic models and you can access the diagnostics information.
- Without ADMIN permission for “CEP management”, you are not able to activate or edit EPL apps or analytic models.
- If you have both READ and ADMIN permissions for “CEP management”, you have read-write access and you can access the diagnostics information.
- If you have only ADMIN permission for “CEP management” and no READ permission, you are able to load, edit and deploy EPL apps and analytic models, but you are not able to see or access the diagnostics information.

LOG FILES OF THE APAMA-CTRL MICROSERVICE

There are two ways to get the logs of the Apama-ctrl microservice:

- You can download diagnostics information from the Streaming Analytics application as described in [Downloading diagnostics and logs](#).
- In some cases, it is useful to view the log file of the Apama-ctrl microservice directly in Cumulocity. The log file is accessible via the Administration application. You can find it on the **Logs** tab of the Apama-ctrl microservice. You must subscribe to the microservice so that you can see the logs. For more information on microservices and log files, see [Managing microservices](#) and [Monitoring microservices](#).

The correlator log is embedded in the log file of the Apama-ctrl microservice. See also [Descriptions of correlator status log fields](#) in the Apama documentation.

Contact [product support](#) if needed.

AUDIT LOGS FOR STREAMING ANALYTICS

Activation and deactivation of analytic models and EPL apps is shown in the audit logs. The audit logs are accessible via the Administration application and the audit API. See [Audit logs](#) and [Audit API](#) in the Cumulocity OpenAPI Specification for details of accessing the audit logs.

Audit log entries include the current action and the name of the user performing that action. For example:

The screenshot shows the 'Audit logs' interface in the Cumulocity Administration application. At the top, there's a header with a gear icon, the title 'Audit logs', and a breadcrumb 'Accounts > Audit logs'. Below the header is a filter bar with 'All types' (dropdown), 'Date from' and 'Date to' (calendar icons), 'Who' (text input), 'Apply filters' (button), and a 'Reload' button with a refresh icon. The main content is a table with three columns: 'Device time', 'Event', and 'Description'. Two log entries are shown:

Device time	Event	Description
8 Jan 2025, 14:37:08	Updated EPL app BY apamabld SERVER TIME 8 Jan 2025, 14:37:08	MyFirstApp(64160205): state=active
8 Jan 2025, 14:36:43	Created EPL app BY apamabld SERVER TIME 8 Jan 2025, 14:36:43	MyFirstApp(64160205) state=inactive

DIAGNOSTICS REST ENDPOINTS

INFO

These endpoints are not available for the Apama-ctrl-smartrules and Apama-ctrl-smartrulesmt microservices.

The following diagnostics endpoints are available for REST requests. These require authentication as a user with READ permission for “CEP management”:

- `/service/cep/diagnostics/metrics`
GET only. Plain text format.
Prometheus metrics from the correlator. For details, see [Monitoring with Prometheus](#) in the Apama documentation.
- `/service/cep/diagnostics/overview`
GET only. ZIP file download.
Obtains the *diagnostic-overview<timestamp>.zip* file as described above.
- `/service/cep/diagnostics/enhanced`
GET only. ZIP file download.
Obtains the *diagnostic-enhanced<timestamp>.zip* file as described above.
- `/service/cep/diagnostics/request`
PUT only. JSON.
Gives access to generic management requests against the correlator. For details, see [Shutting down and managing components](#) in the Apama documentation.
- `/service/cep/diagnostics/correlator/info`
GET only. JSON.
Obtains the `engine_inspect` information.

MONITORING REST ENDPOINTS

The following monitoring endpoints are available for REST requests. These require authentication as a valid user, but do not require any special roles.

- `/service/cep/health`
GET only. JSON.
Obtains information on whether the Apama-ctrl microservice is up or not.

INFO

For Apama-ctrl-smartrulesmt (that is, the multi-tenant variant of the Apama-ctrl-smartrules microservice), only basic microservice status values are provided.

- `/service/cep/prometheus`
GET only. Plain text format.
Prometheus metrics from the correlator and microservice. For details, see [Monitoring with Prometheus](#) in the Apama documentation.

ALARMS GENERATED BY THE APAMA-CTRL MICROSERVICE

Alarms are created by user applications in the Cumulocity tenant (for example, by an analytic model, an activated EPL file, or a smart rule). To learn about alarms in general, refer to [Working with alarms](#). The Apama-ctrl microservice also generates alarms because it has encountered some problem, so that the user is notified about the situation. The information below is about alarms that are generated by the Apama-ctrl microservice, their causes, consequences and possible ways to resolve them.

You can view alarms in the following ways:

1. In the Cockpit application. See [Cockpit](#) for detailed information.
2. In the Administration application, under **Ecosystem > Microservices**. Click the Apama-ctrl microservice and then click **Status**. See [Monitoring microservices](#) for detailed information.
3. From the Streaming Analytics application. Click the **User** button to display the right drawer and then click one of the links in the **Diagnostics** section to download a ZIP file that contains alarms information under `/alarm/alarms_apama-ctrl-object.json`. See [Downloading diagnostics and logs](#) for detailed information.

ALARM SEVERITIES

Severity	Description
CRITICAL	Apama-ctrl was unable to continue running the user's applications and will require corrective action.
MAJOR	Apama-ctrl has encountered a situation that will result in some loss of service (for example, due to a restart).
MINOR	Apama-ctrl has a problem that you might want to fix.
WARNING	There is a warning.

ALARMS CREATED BY THE APAMA-CTRL MICROSERVICE

Apama-ctrl can create alarms to notify users in scenarios such as the correlator running out of memory, uncaught exceptions in activated EPL files, and so on. Once you see an alarm in the Cumulocity tenant, you should diagnose it and resolve it depending on the severity level of the raised alarm. Each alarm has details such as title, text, type, date, and count (represents the number of times the alarm has been raised).

The following is a list of the alarms. The information further down below explains when these alarms will occur, their consequences, and how to resolve them.

- [Change in tenant options and restart of Apama-ctrl](#)
- [Safe mode on startup](#)
- [Deactivating models in the Apama-ctrl-starter microservice](#)
- [High memory usage](#)
- [Warning or higher level logging from an EPL file](#)
- [An EPL file throws an uncaught exception](#)
- [An EPL app is running in an infinite or long-running loop](#)
- [EPL app restore timeout on restart of Apama-ctrl](#)
- [Multiple extensions with the same name](#)
- [Smart rule configuration failed](#)
- [Smart rule restore failed](#)
- [Connection to correlator lost](#)
- [Performance alarms](#)
- [Parent tenant not subscribed](#)
- [Analytics Builder dropped events](#)

Once the cause of an alarm is resolved, you must acknowledge and clear the alarm in the Cumulocity tenant. Otherwise, you will continue to see the alarm until a further restart of the Apama-ctrl microservice.

INFO

The alarm texts for the alarms below may undergo minor changes in the future.

[Change in tenant options and restart of Apama-ctrl](#)

This alarm is raised when a tenant option changes in the `analytics.builder` or `streaminganalytics` category. For details on the tenant options, refer to the [Tenant API](#) in the Cumulocity OpenAPI Specification for more details.

- Alarm type: `tenant_option_change`
- Alarm text: Detected changes in tenant option. Apama-ctrl will restart in order to use it.
- Alarm severity: MAJOR

Analytics Builder allows you to configure its settings by changing the tenant options, using key names such as `numWorkerThreads` or `status_device_name`. For example, if you want to process things in parallel, you can set `numWorkerThreads` to 3 by sending a REST request to Cumulocity, which will update the tenant option. Such a change automatically restarts the Apama-ctrl microservice. To notify the users about the restart, Apama-ctrl raises an alarm, saying that changes have been detected in a tenant option and that Apama-ctrl will restart in order to use it.

Once you see this alarm, you can be sure that your change is effective.

Safe mode on startup

This alarm is raised whenever the Apama-ctrl microservice switches to safe mode.

- Alarm type: `apama_safe_mode`
- Alarm text: Apama-ctrl appears to be repeatedly restarting. As a precaution, user-provided EPL, analytic models and extensions that might have caused this have been disabled. Refer to the audit log for more details. Please check any recent alarms, or contact support or your administrator.
- Alarm severity: CRITICAL

Apama-ctrl detects if it has been repeatedly restarting and if user assets (EPL apps, analytic models, extensions) have been modified recently. Apama-ctrl disables all user assets as a precaution. Potential causes are, for example, an EPL app that consumes more memory than is available or an extension containing bugs.

You can check the mode of the microservice (either normal or safe mode) by making a REST request to `service/cep/diagnostics/apamaCtrlStatus`, which contains a `safe_mode` flag in its response.

To diagnose the cause of an unexpected restart, you can try the following:

- Check the EPL apps memory profiler by making a REST request to `/service/cep/diagnostics/eplMemoryProfiler` for any memory leaks. Note that you must re-activate the EPL apps that were active before as the Apama-ctrl microservice loses information about the previous microservice instance when it restarts due to safe mode. To replicate the previous scenario, run the EPL apps and process some events to trigger a leak and then use the memory profiler to check for any memory leaks.
- Check the microservice logs for any exceptions by downloading the basic diagnostics ZIP file as described in [Downloading diagnostics and logs](#). In the downloaded ZIP file, you can find the logs under `/diagnostics/`. As mentioned in the above point, re-activate the EPL apps and analytic models that were active before and then check the logs.
- Check the audit logs. The audit logs are accessible via the Administration application and the audit API. See [Audit logs](#) and [Audit API](#) in the Cumulocity OpenAPI Specification for details of accessing the audit logs.

In safe mode, all previously active analytic models and EPL apps are deactivated and must be manually re-activated.

Deactivating models in the Apama-ctrl-starter microservice

This alarm is raised when Apama-ctrl switches from the fully capable microservice to the Apama-ctrl-starter microservice with more than 3 active models.

- Alarm type: `apama_ctrl_starter`
- Alarm text: The following models were de-activated as Analytics Builder is restricted to <activate limit> active models: (<models>).
- Alarm severity: MINOR

With the Apama-ctrl-starter microservice, a user can have a maximum of 3 active models. For example, a user is working with the fully capable Apama-ctrl microservice and has 5 active models, and then switches to Apama-ctrl-starter. Since Apama-ctrl-starter does not allow more than 3 active models, it deactivates all the active models (5) and raises an alarm to notify the user.

High memory usage

This alarm is raised whenever the Apama-ctrl microservice consumes 90% of the maximum memory permitted for the microservice container. During this time, the Apama-ctrl microservice automatically generates the basic diagnostics ZIP file which contains diagnostics information used for identifying the most likely cause for memory consumption.

There are 3 variants of this alarm, depending on the time and count restrictions of the generated basic diagnostics ZIP file.

First variant:

- Alarm type: `apama_highmemoryusage`
- Alarm text: Streaming Analytics is using 90% of available memory (<totalMemory>). Your apps will be in danger of crashing. Diagnostics file is located at <URL-to-ZIP-file> You can also download the file by navigating to Administration > Management > Files Repository
- Alarm severity: WARNING

Second variant:

- Alarm type: `apama_highmemoryusage`
- Alarm text: Streaming Analytics is using 90% of available memory (<totalMemory>). Your apps will be in danger of crashing. Have recently created diagnostics snapshot (within last hour).
- Alarm severity: WARNING

Third variant:

- Alarm type: `apama_highmemoryusage`
- Alarm text: Streaming Analytics is using 90% of available memory (<totalMemory>). Your apps will be in danger of crashing. Have created 5 diagnostics snapshots, not creating any more, refer to past alarms.
- Alarm severity: WARNING

Running EPL apps (and to a lesser extent, smart rules and analytic models) consumes memory, the amount will depend a lot on the nature of the app running. The memory usage should be approximately constant for a given set of apps, but it is possible to create a “memory leak”, particularly in an EPL file or a custom block. The Apama-ctrl microservice monitors memory and raises an alarm with WARNING severity if the 90% memory limit is reached along with the basic diagnostics ZIP file and saves it to the files repository (as mentioned in the alarm text).

Apama-ctrl generates the basic diagnostics ZIP files with the following conditions:

- Only if it has not been generated in the last 1 hour.
- A maximum of 5 basic diagnostics ZIP files from its start time until it stops.
- Overall, it can generate a maximum of 20 ZIP files per Cumulocity tenant, beyond which it keeps deleting the oldest ZIP files during its startup process.

To diagnose high-memory-consuming models and EPL apps, you can try the following (it could be listener leaks, excessive state being stored or spawned monitors leaking, and so on):

- Download the automatically generated basic diagnostics ZIP file (refer to the alarm text for its location) and look at `correlator/inspect.json` and `correlator/status.json` for the number of listeners. This number may be large in the case of a listener leak. Note that this ZIP file includes the EPL memory profiler snapshots.
- Download the diagnostics information from the Streaming Analytics application. Click the **User** button to display the right drawer and then click one of the links in the **Diagnostics** section (as described in [Downloading diagnostics and logs](#)). The EPL memory profiler from the **Basic diagnostics (ZIP)** link in `/diagnostics/eplMemoryProfiler.csv` gives the memory consumed by each monitor along with details such as the number of listeners or the number of monitor instances running something like shown in the snippet below. This can help you to understand which monitor is consuming more memory and try to reduce it.

Monitor	Monitor instances	EPL objects	Listeners	Bytes	Overhead bytes
mon1	1	5384	4	1073908	383240
mon2	1	2	2	696	2280
mon3	1	4	1	840	752

- When using the **Enhanced diagnostics (ZIP)** link in the Streaming Analytics application, the diagnostics information includes, in addition to the information that you get with the **Basic diagnostics (ZIP)** link, requests that are more resource-intensive and may significantly slow down the correlator. This includes the contents of the queues. So when diagnosing the cause for the first time, it is recommended to use the ZIP file from the **Basic diagnostics (ZIP)** link, unless additional information is required.
- Also check for memory usage on all the input and output queues available from the **Enhanced diagnostics (ZIP)** link in `/diagnostics/toStringQueues.txt`.

If the memory continues to grow, then when it reaches the limit, the correlator will run out of memory and Apama-ctrl will shut down. To prevent the microservice from going down, you must fix this as a priority.

See also [Diagnostic tools for Apama in Cumulocity](#) in the [Cumulocity Tech Community](#).

Warning or higher level logging from an EPL file

This alarm is raised whenever messages are logged by EPL files with specific log levels (including CRITICAL, FATAL, ERROR and WARNING).

The Streaming Analytics application allows you to deploy EPL files to the correlator. The Apama-ctrl microservice analyzes logged content in the EPL files and raises an alarm for specific log levels with details such as monitor name, log text and alarm type (either of WARNING or MAJOR), based on the log level.

For example, the following is a simple monitor which prints a sequence and logs some texts at different EPL log levels.

```
monitor Sample{
  action onload() {
    log "Info"; // default log level is now INFO
    log "Fatal Error" at FATAL; // log level is FATAL
    log "Critical Error" at CRIT; // log level is CRITICAL
    log "Warning" at WARN; // log level is WARNING
  }
}
```

Apama-ctrl analyzes all the log messages, filters out only certain log messages, and raises an alarm for the identified ones. Thus, Apama-ctrl generates the following three alarms for the above example:

First alarm:

- Alarm type: `APAMA_CTRL_FATAL_<HASHCODE>`
- Alarm text: <Monitor name>-Fatal Error.
- Alarm severity: MAJOR

Second alarm:

- Alarm type: `APAMA_CTRL_CRIT_<HASHCODE>`
- Alarm text: <Monitor name>-Critical Error.
- Alarm severity: MAJOR

Third alarm:

- Alarm type: `APAMA_CTRL_WARN_<HASHCODE>`
- Alarm text: <Monitor name>-Warning.
- Alarm severity: WARNING

An EPL file throws an uncaught exception

You have seen that the Apama-ctrl microservice raises alarms for logged messages. In addition, there can also be uncaught exceptions (during runtime). Apama-ctrl identifies such exceptions and raises alarms so that you can identify and fix the problem.

For example, the following monitor throws `IndexOutOfBoundsException` during runtime:

```
monitor Sample{
  sequence<string> values := ["10", "20", "30"];
  action onload() {
    // IndexOutOfBoundsException (runtime error)
    log "Value = " + values[10] at ERROR;
  }
}
```

Apama-ctrl generates the following alarm for the above example:

- Alarm type: `APAMA_CTRL_ERROR_<HASHCODE>`
- Alarm text: <Monitor name>-Error on line <x> of monitor : `IndexOutOfBoundsException` - Out of bounds index passed to sequence [] operator - Sample
- Alarm severity: MAJOR

You can diagnose the issue by the monitor name and line number given in the alarm.

For more details, you can also check the log files of the Apama-ctrl microservice if the tenant has the “microservice hosting” feature enabled. Alarms of this type should be fixed as a priority as these uncaught exceptions will terminate the execution of that monitor instance, which will typically mean that your app is not going to function correctly. This might even lead to a correlator crash if not handled properly. See also [Log files of the Apama-ctrl microservice](#).

An EPL app is running in an infinite or long-running loop

If an EPL app has an infinite or long-running loop, it may block the correlator context for too long, not letting any other apps run in the same context or, even worse, causes excessive memory usage (as the correlator is unable to perform any garbage collection cycles) leading to the app running out of memory. The Apama-ctrl microservice identifies such scenarios (the correlator logs warning messages if an app is blocking a context for too long) and raises alarms, so that the user can identify and fix the problem.

For example, the following monitor blocks the correlator main context:

```
event MyEvent {
}

monitor Sample{
  action onload() {
    while true {
      // do something
      send MyEvent() to "foo";
    }
  }
}
```

Apama-ctrl generates the following alarm for the above example:

- Alarm type: `APAMA_CTRL_WARN_<HASHCODE>`
- Alarm text: `<EPLAppName>.<monitorName>` - This EPL app is probably in an infinite or long-running loop and impedes the operation of your other apps. It is recommended that you deactivate and diagnose this app.
- Alarm severity: WARNING

You can diagnose the issue by the monitor name given in the alarm.

For more details, you can also check the log files of the Apama-ctrl microservice if the tenant has the “microservice hosting” feature enabled. Alarms of this type should be fixed as a priority as these scenarios may lead to the microservice and correlator running out of memory. See also [Log files of the Apama-ctrl microservice](#).

EPL app restore timeout on restart of Apama-ctrl

If restoring an EPL app on a restart of the Apama-ctrl microservice takes a long time and exceeds the time limit specified by the `recovery.timeoutSecs` tenant option (in the `streaminganalytics` category) or a default of 60 seconds, the Apama-ctrl microservice times out and raises an alarm, indicating that it will restart and reattempt to restore the EPL app. The alarm text includes the names of any EPL apps that are considered to be the reason for the timeout.

- Alarm type: `eplapp_restore_timeout`
- Alarm text: Restoring EPL apps after Apama-ctrl microservice restart has timed out. The EPL app `<app name>` could not be restored. The following EPL apps may be the cause of this: `<comma-separated list of app names>`. The Apama-ctrl microservice will restart now, and restoring will be reattempted. If this continues to fail, the Apama-ctrl microservice will enter safe mode, disabling all EPL apps.
- Alarm severity: MAJOR

The following information is only included in the alarm text if the Apama-ctrl microservice detects that the timeout is due to some EPL apps: “The following EPL apps may be the cause of this: `<comma-separated list of app names>`.”. If no such apps are detected, this information is omitted from the alarm text.

Multiple extensions with the same name

This alarm is raised when the Apama-ctrl microservice tries to activate the deployed extensions during its startup process and there are multiple extensions with the same name.

- Alarm type: `extension_error`
- Alarm text: Multiple extensions with the same name have been found: `<list of all duplicate extension names>`

- Alarm severity: CRITICAL

This disables all extensions that were deployed to Apama-ctrl. In order to use the deployed extensions, the user must decide which extensions to keep and then delete the duplicate ones.

INFO

In case of multiple duplicates, this alarm is only listed once.

Smart rule configuration failed

This alarm is raised if a smart rule contains an invalid configuration.

- Alarm type: `smartrule_configuration_error`
- Alarm text: <Smart rule identifier>: Smart rule create/edit failed. One or more fields are invalid, please check smart rule configuration.
- Alarm severity: MAJOR

To diagnose the cause, download the basic diagnostics ZIP file as described in [Downloading diagnostics and logs](#). Or, if that fails, log on as an administrator and look at the result of a GET request to `/service/smartrule/smartrules?withPrivateRules=true`. Review the smart rules JSON and look for invalid smart rule configurations. Such smart rules must be corrected.

The Apama microservice log contains more details on the reason for the smart rule configuration failure. For example, it is invalid to configure an “On measurement threshold create alarm” smart rule with a data point that does not exist.

Smart rule restore failed

This alarm is raised if a corrupt smart rule is present in the inventory and the correlator therefore fails to recover it correctly during startup.

- Alarm type: `smartrule_restore_failed`
- Alarm text: Smart rule restore failed. Contact support.
- Alarm severity: MAJOR

To diagnose the cause, download the basic diagnostics ZIP file as described in [Downloading diagnostics and logs](#). Or, if that fails, log on as an administrator and look at the result of a GET request to `/service/smartrule/smartrules?withPrivateRules=true`. Review the smart rules JSON and look for invalid smart rule configurations. Such smart rules may need to be deleted or corrected.

Connection to correlator lost

This alarm is raised in certain cases when the connection between the Apama-ctrl microservice and the correlator is lost. This should not happen, but can be triggered by high load situations.

- Alarm type: `lost_correlator_connection`
- Alarm text: Unable to ping correlator: <message>, Apama-ctrl will restart.
- Alarm severity: MAJOR

Apama-ctrl will automatically restart. Report this to [product support](#) if this is happening frequently.

Performance alarms

Input or output queues that are filling up are a symptom of a serious performance degradation, suggesting that events or requests are being produced by Apama or Cumulocity faster than they can be processed by Apama or Cumulocity.

The performance of the correlator's input and output queues is periodically monitored. Different types of alarms can be raised, where the alarm text contains a snapshot of the correlator status at the time of raising the alarm.

This alarm is raised for the input queues:

- Alarm type: `input_queues_filling`
- Alarm text: Correlator input queues are filling. If this alarm is being regularly raised, there is a chance that the correlator cannot process the requests at the rate at which they are arriving. Slowest receiver name: <name>, Slowest receiver queue size: <size>.

Slowest context name: <name>, Slowest context queue size: <size>.

- Alarm severity: WARNING

This alarm is raised for the output queues:

- Alarm type: `output_queues_filling`
- Alarm text: Correlator output queues are filling. If this alarm is being regularly raised, there is a chance that Cumulocity is not able to process the requests at the rate the correlator is sending them. Slowest receiver name: <name>, Slowest receiver queue size: <size>, Slowest context name: <name>, Slowest context queue size: <size>.
- Alarm severity: WARNING

This alarm is raised for both the input and output queues:

- Alarm type: `input_output_queues_filling`
- Alarm text: Correlator input and output queues are filling. If this alarm is being regularly raised, there is a chance that Cumulocity is not able to process the requests at the rate the correlator is sending them, causing the slowest output queue to fill up. This might have also caused the slowest input queue to fill up. Slowest receiver name: <name>, Slowest receiver queue size: <size>, Slowest context name: <name>, Slowest context queue size: <size>.
- Alarm severity: MAJOR

See also [List of correlator status statistics](#) in the Apama documentation.

Check the text from the above alarms to get an indication of which queue is blocking. A problem is likely to trigger these alarms, followed by this alarm:

- Alarm text: Real-time event processing is currently overloaded and may stop processing your events. Please contact support.
- Alarm severity: CRITICAL

This alarm is raised whenever the CEP queue for the respective tenant is full. It is coming from Cumulocity Core, but concerns Apama-ctrl.

Core java application nodes that send events to the CEP engine maintain per-tenant queues for the incoming events. This data gets processed by the CEP engine for the hosted CEP rules. For various reasons, these queues can become full and cannot accommodate newly arriving data. In such cases, an alarm is sent to the platform so that the end users are notified about the situation.

If the CEP queue is full, older events are removed to handle new incoming events. To avoid this, you must diagnose the cause of the queue being full and resolve it as soon as possible.

The CEP queue size is based on the number of CEP events, not raw bytes.

To diagnose the cause, you can try the following. It may be that the Apama-ctrl microservice is running slow because of time-consuming smart rules, analytic models or EPL apps, or the microservice is deprived of resources, or code is not optimized, and so on. Check the correlator input and output queues from the above alarms (or from the microservice logs or from the basic diagnostics ZIP file under `/correlator/status.json`).

- If both input and output queues are full, this suggests a slow receiver, possibly EPL sending too many requests (or too expensive a request) to Cumulocity.
- Else, if only the input queue is full, EPL is probably running in a tight loop. Try analyzing the `cpuProfile.csv` output in the basic diagnostics ZIP file, especially the monitor name and CPU time. The data collected in the profiler may also help in identifying other possible bottlenecks. For details, refer to [Using the CPU profiler](#) in the Apama documentation.
- Else, the cause may be some issue with connectivity or in Cumulocity Core.

Parent tenant not subscribed

This alarm is raised for a subtenant that was subscribed before the parent tenant was subscribed.

- Alarm type: `parent_tenant_not_subscribed`
- Alarm text: The microservice cannot function fully until the parent tenant is also subscribed to the microservice. Please contact the administrator.
- Alarm severity: MAJOR

The Apama-ctrl microservice allows you to subscribe to tenants in any order. However, as long as the parent tenant is not subscribed, the microservice functionality will not work on the subtenant.

This alarm is cleared once the parent tenant is subscribed.

Analytics Builder dropped events

This alarm is raised when an Analytics Builder model drops an event because it is delayed beyond the reorder buffer duration.

- Alarm type: `analyticsbuilder_dropped_events`
- Alarm text: Analytics Builder dropped <number> events because they were delayed beyond the reorder buffer duration. The last dropped event was received at <system time> (<number> seconds old): '<last dropped event string>'.
- Alarm severity: WARNING

Analytics Builder models use buffers to reorder incoming events by their source timestamp and process them in order. By default, input blocks reorder events with a delay of up to 1 second. If an event is received after a delay of more than the reorder buffer duration, the event may be dropped without processing. See [Input blocks and event timing](#) for detailed information.

To resolve the issue, you can either disable the reordering of input events or increase the duration of the reorder buffer. You can disable the reordering of input events by enabling the **Ignore Timestamp** parameter of the input blocks. When the **Ignore Timestamp** parameter is enabled, the input events are processed as soon as possible without reordering. You can increase the duration of the reorder buffer by changing the `timedelay_secs` tenant option in the `analytics.builder` category. For more information on `timedelay_secs`, see [Keys for model timeouts](#).