



SAS[®] Event Stream Processing: Overview

2026.08

This document might apply to additional versions of the software. Open this document in [SAS Help Center](#) and click on the version in the banner to see all available versions.

<i>Data in Motion versus Data at Rest</i>	2
<i>What Is SAS Event Stream Processing?</i>	2
Product Overview	2
Product Details	3
Planning Your Event Stream Processing Project	4
<i>What is an Event Stream Processing Model?</i>	5
Components of an Event Stream Processing Model	5
Understanding Event Streams	5
Using Connectors to Publish and Subscribe Event Streams	5
Understanding Events	5
Understanding Event Blocks	7
Understanding Projects	8
Understanding Continuous Queries	9
Understanding Windows	10
<i>Streaming Events through a Continuous Query</i>	12
Overview	12
Processing the First Event	14
Processing the Second Event	15
Processing the Third Event	16
Processing the Fourth Event	16
Processing the Fifth Event	17
<i>Getting Started in a Kubernetes Environment</i>	18
Create and Experiment with Projects	18

Publishing and Deploying Projects	19
Managing Client Behavior in a Kubernetes Environment	20
Accessing the REST API in a Kubernetes Environment	20
Understanding SAS Event Stream Manager	20
What Is SAS Event Stream Manager?	20
SAS Event Stream Manager Instances	21
Overview of Using SAS Event Stream Manager to Deploy Projects	22
Production Assets	25
Integration with Grafana and Monitoring Tools	26
Visualizing Event Streams	26
Monitoring Projects	26
Using SAS Event Stream Processing Web-based Clients as Progressive Web Apps	26
Examples	27

Data in Motion versus Data at Rest

[View the slideshow in SAS Help Center.](#)

What Is SAS Event Stream Processing?

Product Overview

SAS Event Stream Processing enables you to quickly process and analyze streaming data. It works with various SAS solutions to provide real-time data analysis for manufacturing, health-care, energy and utilities, financial services, cyber security, and fraud detection.

Real-time, high throughput, low latency data flows are called *event streams*. Those event streams are published in applications that use the following:

- connector classes or adapter executables
- the Python, JAVA, or C publish/subscribe APIs
- SAS Event Stream Processing Studio

Event stream processing projects can perform real-time analytics on event streams. Typical use cases for event stream processing include but are not limited to the following:

- sensor data monitoring and management

- operational systems condition monitoring and management
- object detection and classification
- cyber security analytics
- capital markets trading systems
- fraud detection and prevention
- personalized marketing

Product Details

SAS Event Stream Processing

SAS Event Stream Processing provides the following components. Obtain pre-built Docker images of these components and deploy them in a Kubernetes cluster.

Table 1 *Product Components*

Component	Description	Docker Image
ESP server	An executable that instantiates and executes projects.	<code>sas-event-stream-processing-server-app</code>
SAS Event Stream Processing Studio	A web-based client that enables you to create, edit, upload, and test event stream processing projects. For more information, see SAS Event Stream Processing: Using SAS Event Stream Processing Studio.	<code>sas-event-stream-processing-studio-app</code>
SAS Event Stream Manager	A web-based client that enables you to manage your SAS Event Stream Processing environment. For more information, see “Understanding SAS Event Stream Manager”.	<code>sas-event-stream-manager-app</code>

You can use the web-based clients to interact with ESP servers that run in a Kubernetes environment.

SAS Event Stream Processing provides advanced analytics and machine learning techniques that you can use within an event stream processing project. For more information, see [SAS Event Stream Processing: Using Streaming Analytics](#).

You can deploy SAS Event Stream Processing one of two ways.

- With other SAS products. For more information, see [SAS Viya Platform: Deployment Guide](#).
- As a standalone offering of the product. For more information, see “Deploying a Standalone SAS Event Stream Processing Environment” in [SAS Event Stream Processing: Using SAS Event Stream Processing in a Kubernetes Environment](#).

For more information, see [SAS Event Stream Processing: Using SAS Event Stream Processing in a Kubernetes Environment](#).

SAS Event Stream Processing for Edge Computing

SAS Event Stream Processing for Edge Computing enables deployment of simplified components to edge systems. You can deploy one of the following Docker images on an x86_64 compatible processor.

- One that provides basic event stream processing capabilities (`sas-esp-server-edge`).
- One that provides basic event stream processing capabilities plus the analytical algorithms packaged with SAS Event Stream Processing (`sas-esp-server-edge-analytics`). Libraries and extensions are required for execution of offline models contained in analytic store files.
- One that provides basic event stream processing capabilities, the analytical algorithms packaged with SAS Event Stream Processing, and GPU support (`sas-event-stream-processing-server-app`).

Alternatively, you can deploy SAS Event Stream Processing for Edge Computing on premises on a system with 64-bit ARMv8 chipsets.

For more information, see [SAS Event Stream Processing: Using the ESP Server in an Edge Environment](#).

Planning Your Event Stream Processing Project

As you plan an event stream processing project, answer the following questions:

- Is the project to be deployed at the [edge](#) or in the [cloud](#)?
- What specific event streams are [published](#) into a project, and with what protocol and format?
- What happens to the data? That is, how are event streams transformed and [analyzed](#)?
- Are any third-party tools or applications required to process event streams? How are you going to make them [available](#) to your project?
- What are the resulting event streams of interest? Do other projects or applications subscribe to these event streams, and in what format and protocol?

Your answers to these questions guide the planning of the structure of your event stream processing project.

What is an Event Stream Processing Model?

Components of an Event Stream Processing Model

[View the slideshow in SAS Help Center.](#)

Understanding Event Streams

[View the slideshow in SAS Help Center.](#)

Using Connectors to Publish and Subscribe Event Streams

[View the slideshow in SAS Help Center.](#)

Understanding Events

What is an Event?

[View the slideshow in SAS Help Center.](#)

Processing Data in an Event

Data in an event object is stored in an internal format as described in the schema object . All key values are contiguous and packed at the front of the event. An event object maintains internal hash values based on the key with which it was built. In addition, there are functions in the

`dfESPeventcomp` namespace for a comparison of events that were created using the same underlying schema.

An event supports the following opcodes as it processes data.

Table 2 *Opcodes Supported by SAS Event Stream Processing*

Opcode	Description
Delete (D)	Removes event data from a window.
Insert (I)	Adds event data to a window.
Update (U)	Changes event data in a window.
Upsert (P)	Updates event data if the key field already exists. Otherwise, the opcode adds event data to a window.
Safe Delete (SD)	Removes event data from a window without generating an error if the event does not exist.

When publishing, when you do not know whether an event needs an Update or Insert opcode, use Upsert. The Source window where the event is injected determines whether it is handled as an Insert or an Update. The Source window then propagates the correct event and opcode to the next set of connected windows in the model or to subscribers.

Data Types in Events

Events support the following data types:

- INT32
- INT64
- DOUBLE
- STRING
- DATE (granularity to the second)
- STAMP (granularity to the microsecond)
- MONEY (192-bit fixed decimal)
- BINARY (binary large object or blob)
- RUTF8STR (reference-counted string or rstring)
- ARRAY (32-bit integers, 64-bit integers, double)

With the base data types (INT32, INT64, DOUBLE, STRING, DATE, STAMP, and MONEY), data is stored inline in the event. Inline storage enables fast indexing and serialization.

With the BINARY, RUTF8STR, and ARRAY data types, data is not stored inline. Instead, data is referenced in an event, which means that the event holds a pointer to the actual data. These data types cannot be used as key fields for an event. They are reference-counted at the object level. This

Event Block	Description
Normal	Processing through the project is not atomic. Events are packaged together for efficiency, but are individually treated once they are injected into a source window.

A unique transaction ID is propagated through transformed event blocks as the blocks work their way through a model. This persistence enables event stream subscribers to correlate a group of subscribed events back to a specific group of published events through the ID.

Typically, you set the size of an event block with the `blocksize` parameter of the connector or adapter used in your project. To optimize performance, the following is recommended.

- For small events (≥ 1 and ≤ 6 fields, contains no blobs), set `blocksize` to 256.
- For medium events (≥ 7 and ≤ 12 fields, contains no blobs), set it to 32.
- For large events (≥ 13 fields, contains no blobs), set it to 4.
- For huge events (contains blobs), set it to 1.

There is rarely justification to set `blocksize` larger than 16000. That value is practical only when small events arrive very quickly. Whenever events include blob data for images, set `blocksize` to 1.

Understanding Projects

Overview

A *project* specifies a container that holds one or more continuous queries and is backed by a thread pool of user-defined size. The level of determinism for a project's incremental computations is, by default, full concurrency. You can change this with the `use-tagged-token` attribute of the `project` element, which enables a project to use tagged token data flow semantics. You can also specify an optional port for publish/subscribe scalability.

Data Flow Model within a Project

The data flow model within a project is always computationally deterministic.

When a model is multi-threaded, intermediate calculations can occur at different times across different project runs; there is a higher level of parallelism. Therefore, when a project watches every incremental computation, the increments could vary across runs even though the unification of the incremental computation is always the same.

When a model is single-threaded, there is a higher level of determinism.

Note: Regardless of the determinism level or of the number of threads used, each window always processes all data in order. Therefore, data received by a window is never rearranged and processed out of order.

Understanding Project Packages

A project usually references other resources such as input files, analytic stores, custom connectors, and so on. These additional resources must be physically available to the ESP server that runs the project.

To make those resources available, you can set up a project package that implements a standard set of folders. You can place resources that the project references into those folders. These folders are created on the persistent volume (PV) that is set up when you deploy SAS Event Stream Processing.

For information about how to set up project packages, see [“Project Package” in SAS Event Stream Processing: Using SAS Event Stream Processing Studio](#)

Understanding Continuous Queries

A continuous query specifies a container that holds one or more directed graphs of windows and that enables you to specify the connectivity between windows. The windows within a continuous query can transform or analyze data, detect patterns, or perform computations. Query containers provide functional modularity for large projects. Typically, each container holds a single directed graph.

Continuous query processing follows these steps:

- 1 An event block (with or without atomic properties) that contains one or more events is injected into a Source window.
- 2 The event block flows to any derived window that is directly connected to the Source window. If transactional properties are set, then the event block of one or more events is handled atomically as it makes its way to each connected derived window. That is, all events must be performed in their entirety.

If any event in the event block with transactional properties fails, then all of the events in the event block fail. Failed events are logged. They are written to a bad records file for you to review, fix, and republish when you enable this feature.

- 3 Derived windows transform events into zero or more new events that are based on the properties of each derived window. After new events are computed by derived windows, they flow farther down the model to the next level of connected derived windows, where new events are potentially computed.
- 4 This process ends for each active path down the model for a given event block when either of the following occurs:
 - There are no more connected derived windows to which generated events can be passed.
 - A derived window along the path has produced zero resulting events for that event block. Therefore, it has nothing to pass to the next set of connected derived windows.

Because you can use continuous queries as a mechanism of modularity, the number of queries that you implement depends on how compartmentalized your windows are. Within a continuous query, you can instantiate and define as many windows as you need. Any given window can flow data to one or more windows. Loop-back conditions are not permitted within continuous queries. You can loop back across continuous queries using the project connector.

Event streams must be published or injected into Source windows through one of the following:

- the publish/subscribe API
- connectors
- adapters
- HTTP clients
- SAS Event Stream Processing Studio

Within a continuous query, you can define a data flow model using all of the available window types.

Understanding Windows

A continuous query contains one or more *Source windows* and one or more *derived windows*. All event streams must enter continuous queries by being published or injected into a Source window. Event streams cannot be published or injected into any other window type.

Windows are connected by *edges*, which have an associated direction.

SAS Event Stream Processing supports the following derived window types:

Table 4 *Derived Window Types*

Window Type	Description
Compute window	Enables a one-to-one transformation of input events into output events through the computational manipulation of the input event stream fields.
Aggregate window	Similar to a Compute window in that non-key fields are computed. An Aggregate window uses the key field or fields for the group-by condition. All unique key field combinations form their own group within the Aggregate window. All events with the same key combination are part of the same group.
Calculate window	Transform data events using a variety of analytical algorithms.
Copy window	Makes a copy of the parent window. Making a copy can be useful to set new event state retention policies. Retention policies can be set only in source and Copy windows. You can set event state retention for a Copy window only when the window is not specified to be Insert-only and when the window index is not set to <code>pi_EMPTY</code>. All subsequent sibling windows are affected by retention management. Events are deleted when they exceed the windows retention policy.
Counter window	Enables you to see how many events are streaming through your model and the rate at which they are being processed.

Window Type	Description
Filter window	Uses a registered Boolean filter function or expression. This function or expression determines what input events are allowed into the Filter window.
Functional window	Enables you to use different types of functions to manipulate or transform the data in events. Fields in a Functional window can be hierarchical, which can be useful for applications such as web analytics.
Geofence window	Enables you to create a window to determine whether the location of an event stream is inside or near an area of interest.
Join window	Takes two input windows and a join type. Supports equijoins that are one to many, many to one, or many to many. Both inner and outer joins are supported.
Lua window	Enables you to generate SAS Event Stream Processing events with Lua programs.
Model Reader window	Read models brought into SAS Event Stream Processing as analytic store files or as a combination of non-binary files.
Model Supervisor window	Manage models received from Model Reader windows.
Notification window	Enables you to send notifications through email, text, or multimedia message. You can create any number of delivery channels to send the notifications. A Notification window uses the same underlying language and functions as the Functional window.
Object Tracking window	Enables you to perform multi-object tracking (MOT) in real time.
Pattern window	Enables the detection of events of interest (EOI). A pattern defined in this window type is an expression that logically connects declared events of interest. To define a pattern window, you need to define events of interests and then connect these events of interest using operators. The supported operators are "AND", "OR", "FBY", "NOT", "NOTOCCUR", and "IS". The operators can accept optional temporal conditions.
Procedural window	Enables the specification of an arbitrary number of input windows and input-handler functions for each input window (that is, event stream).
Python window	Enables you to generate SAS Event Stream Processing events with Python programs.
Remove State window	Facilitates the transition of a stateful part of a model to a stateless part of a model.

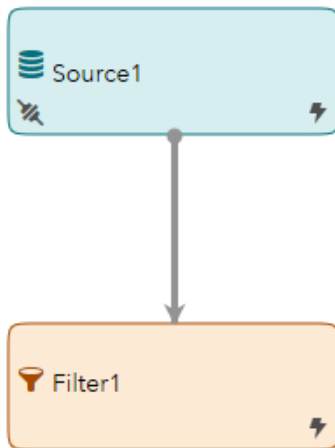
Window Type	Description
Score window	Score events with online analytical algorithms that are packaged with SAS Event Stream Processing or with algorithms in offline models.
StateDB Reader	Query or fetch data from a defined external database based on the incoming event and specified criteria. These criteria are based in part on the indexes you create in the StateDB Writer window.
StateDB Writer	Write an incoming event to the defined external database, create required indexes for the event, and pass the input event unchanged to the next window of a project.
Train window	Refine the algorithm parameters of online models based on streaming event data.
Transpose window	Enables you to interchange an event's rows as columns, or columns as rows.
Union window	Combines multiple event streams with the same schema into a single stream, similar to an SQL union operation.

Streaming Events through a Continuous Query

Overview

The following example demonstrates what happens when events stream through the windows in a continuous query:

Figure 1 Continuous Query with a Source Window and a Filter Window



In this query, a Source window publishes events to a single Filter window. The Source window contains the following schema:

ID*: int32, symbol: string, quantity: int32, price: double

This schema consists of four fields:

Table 5 Source Window Schema

Field	Type
ID	32-bit integer
symbol	string
quantity	32-bit integer
price	double precision floating-point

Key fields in a schema identify an event for operations such as Insert, Update, Delete, or Upsert. Key fields must be unique. If you think of the event stream as a database, then you can think of the key fields as lookup keys.

In the schema, the ID field has the * designator to indicate that this field is part of the key. No other field in the schema has this designator, so the ID field completely forms the key.

Here is XML code for the Source window:

```
<window-source name="Source1" pubsub="true">
  <schema>
    <fields>
      <field type="int32" name="ID" key="true" />
      <field type="string" name="symbol" />
      <field type="int32" name="quantity" />
      <field type="double" name="price" />
    </fields>
  </schema>
</window-source>
```

```

</schema>
<connectors>
  <connector class="fs" name="New_Connector_1">
    <properties>
      <property name="type">pub</property>
      <property name="fsname">events.csv</property>
      <property name="fstype">csv</property>
      <property name="transactional">true</property>
      <property name="blocksize">1</property>
    </properties>
  </connector>
</connectors>
</window-source>

```

A file and socket connector publishes a CSV file in the current directory that contains the events to be processed into the Source window.

The Filter window applies the expression `quantity > 1000`. Thus, events are passed only when the **quantity** field in the event exceeds the value of 1000.

Here is XML code for the Filter window:

```

<window-filter name="Filter1" pubsub="true">
  <expression>quantity > 1000</expression>
</window-filter>

```

Here is the XML for the edge that connects the Source window to the Filter window:

```

<edges>
  <edge target="Filter1" source="Source1" />
</edges>

```

For more information about available filter conditions, see [“Overview to Expressions” in SAS Event Stream Processing: Using Source and Derived Windows](#). For information about the XML language that you use to code continuous queries, see [SAS Event Stream Processing: XML Language Reference for Event Stream Processing Models](#).

The following sections provide detailed information about what happens when five events stream through this model.

Processing the First Event

Suppose that the first event streaming through the query is as follows:

```
e1: [i,n,10,IBM,2000,164.1]
```

- 1 The Source window receives e1 as an Input event. It stores the event and passes it to the Filter window.
- 2 The Filter window receives e1 as an Input event, as designated by the “i” in the first field. The second field in this and all subsequent events designates “normal.”
- 3 The **quantity** field has a value of 2000. Because the filter expression is `quantity > 1000`, the Filter window stores the input. Typically, a Filter window would pass e1 forward. However, because the Filter window has no dependent windows, there is no additional data flow for the event.

The window contents are now as follows:

Table 6 Contents of Source Window after First Event

ID	Symbol	Qty	Price
10	IBM	2000	164.10

Table 7 Contents of Filter Window after First Event

ID	Symbol	Qty	Price
10	IBM	2000	164.10

Processing the Second Event

The second event is as follows:

e2: [p,n,20,MSFT,1000,114.22]

- 1 The Source window receives e2 as an Upsert event. It checks whether the window has a stored event with a key (ID) of 20.
- 2 An ID of 20 is not stored, so the Source window creates a new event e2a: [I, 20, "MSFT", 1000, 114.22]. It stores this new event and passes it to the Filter window.
- 3 The Filter window receives e2a as an Input event.
- 4 The value in the **Quantity** field of e2 equals 1000, which does not meet the condition set by the filter expression in the schema. Thus, this event is not stored or passed to any dependent windows.

The window contents are now as follows:

Table 8 Contents of Source Window after Second Event

ID	Symbol	Qty	Price
10	IBM	2000	164.10
20	MSFT	1000	114.22

Table 9 Contents of Filter Window after Second Event

ID	Symbol	Qty	Price
10	IBM	2000	164.10

Processing the Third Event

The third event is as follows:

```
e3: [d,n,10, , , , ]
```

Note: For a Delete event, you need only specify key fields. Remember that in this example, only the ID field is key.

- 1 The Source window receives e3 as a Delete event.
- 2 The Source window looks up the event that is stored with the same key. The Delete opcode removes the event from the Source window.
- 3 The Source window passes the found record to the Filter window with the Delete opcode specified. In this case, the record that is passed to the Filter window is as follows:
e3a: [d,n,10,IBM,2000,164.1]
- 4 The Filter window receives e3a as an Input event.
- 5 The value in the **quantity** field of e3a equals 2000. This old event that was previously stored makes it through the filter, so it is removed.

The window contents are now as follows:

Table 10 Contents of Source Window after Third Event

ID	Symbol	Qty	Price
20	MSFT	1000	114.22

The Filter window is empty.

Processing the Fourth Event

The fourth event is as follows:

```
e4: [u,n,20,MSFT,3000,114.25]
```

- 1 The Source window receives e4 as an Update event.
- 2 The Source window looks up the event stored with the same key and modifies it.
- 3 The Source window constructs an update block that consists of the following:
 - the new record with updated values marked as an update block
 - the old record that was updated
- 4 The block is marked as a Delete event. The new event Update block that is passed to the Filter window looks like this:

e4a: [ub,n,20,MSFT,3000,114.22] , [d,n,20,MSFT,1000,114.25]

Note: Both the old and new records are supplied because derived windows often require the current and previous state of an event. They need these states in order to compute any incremental change caused by an Update.

- 5 The Filter window receives e4a as an Input event.
 - 6 The value in the **Quantity** field of e4a > 1000, but previously it was <= 1000. The input did not pass the previous filter condition, but now it does pass. Because the input is not present in the filter window, the Filter window generates an Insert event of the following form:
- e4b: [i,n,20,MSFT,3000,114.25]
- 7 The Insert event is stored. The Filter window would pass e4b. However, because there are no dependent windows, this input does not pass. There is no further data flow for this event.

The window contents are now as follows:

Table 11 Contents of Source Window after Fourth Event

ID	Symbol	Qty	Price
20	MSFT	3000	114.25

Table 12 Contents of Filter Window after Fourth Event

ID	Symbol	Qty	Price
20	MSFT	3000	114.25

Processing the Fifth Event

The fifth event is as follows:

e5: [i,n,30,APPL,2000,225.06]

- 1 The Source window receives e5 as an Insert event, stores it, and passes e1 to the Filter window.
- 2 The Filter window receives e5 as an Input event. Because the value in the **quantity** field > 1000, the Filter window stores the input. Because the filter window has no dependent windows, there is no further data flow.

The window contents are now as follows:

Table 13 Contents of Source Window after Fifth Event

ID	Symbol	Qty	Price
20	MSFT	3000	114.25
30	APPL	2000	225.06

Table 14 Contents of Filter Window after Fifth Event

ID	Symbol	Qty	Price
20	MSFT	3000	114.25
30	APPL	2000	225.06

Getting Started in a Kubernetes Environment

Create and Experiment with Projects

Use SAS Event Stream Processing Studio to create and experiment with projects.

When you are using another SAS Viya product, access SAS Event Stream Processing Studio by

clicking  and selecting **Design Streaming Projects**. The presence of the **Applications** menu and the list of applications within it depends on how your system was installed and on your access permissions.

When you have deployed the standalone version of SAS Event Stream Processing, use a web browser to open the following URL:

`https://name-of-ingress-host/SASEventStreamProcessingStudio`

For *name-of-ingress-host*, specify the host that is configured for your Kubernetes cluster Ingress.

Note: The Sign In to SAS window appears only when your deployment has been configured to enable users to log on to SAS Event Stream Processing Studio. If your deployment has not been configured in this way, you are not required to enter a user ID and password to access the application.

After you have opened SAS Event Stream Processing Studio, follow these steps:

- 1 Design a project in the [modeler](#), upload a file that contains [XML model code](#), or [install an example project](#).
- 2 [Test](#) and validate the event stream processing model within the project.
- 3 Publish one or more event streams into a project in one of the following ways:
 - through connectors (in-process classes). A single instance of a connector reads from or writes to a single window of an event stream processing model. For more information, see [“What Are Connectors and Adapters?” in SAS Event Stream Processing: Connectors and Adapters](#).
 - through the Java, C, or Python publish/subscribe API.
For more information, see [“Overview to the Publish/Subscribe APIs” in SAS Event Stream Processing: Publish/Subscribe API Reference](#).
- 4 Subscribe to relevant window event streams within continuous queries by using connectors or the publish/subscribe API.

A project XML file often references resources such as input files, analytic stores, custom connectors, and plug-ins. These resources must be available to the ESP server on which the project runs. A project package contains a project XML file and a standard set of folders that contain the files that a project references. For more information about working with project packages, see [“Project Package” in SAS Event Stream Processing: Using SAS Event Stream Processing Studio](#).

Publishing and Deploying Projects

You can use SAS Event Stream Processing Studio to publish a version of your project to SAS Event Stream Manager.

Then you can use SAS Event Stream Manager to deploy the project to a non-production deployment (for testing purposes) or to a production deployment. For more information, see [“Overview of Using SAS Event Stream Manager to Deploy Projects”](#).

After that, you can manage the operation of the project with SAS Event Stream Manager. For more information, see [“Load and Start a Project on an ESP Server That Runs in a Kubernetes Cluster” in SAS Event Stream Processing: Using SAS Event Stream Manager](#).

Managing Client Behavior in a Kubernetes Environment

You can use the **Settings** page of SAS Event Stream Processing Studio or SAS Event Stream Manager to adjust how certain features work. For example, you can set or change environment variables and properties.

Note: The changes that you make in the **User Session** sections on the **Settings** pages of SAS Event Stream Processing Studio and SAS Event Stream Manager are independent of one other. To use the same settings in both applications, you must configure the settings in both applications.

Accessing the REST API in a Kubernetes Environment

The REST API that is provided with SAS Event Stream Processing is based on resource-oriented URLs. It uses HTTP authentication and HTTP verbs and returns HTTP response codes to indicate API errors.

When you deploy SAS Event Stream Processing in a Kubernetes environment, you must use HTTP authentication in order to access its REST API. You must register clients, obtain an access token, and use the access token to make REST calls.

When you deploy SAS Event Stream Processing with other SAS products, follow the steps in [this blog post](#) to obtain the access token. For more information about SAS Viya platform REST APIs, see [the documentation](#).

Understanding SAS Event Stream Manager

What Is SAS Event Stream Manager?

SAS Event Stream Manager is a web-based client that enables you to manage your SAS Event Stream Processing environment.

You can use SAS Event Stream Manager to perform the following tasks:

- deploy SAS Event Stream Processing projects into production environments and test environments
- monitor the health of your deployments, ESP servers, and running projects
- administer your deployments and manage change
- monitor your SAS Event Stream Processing metering servers

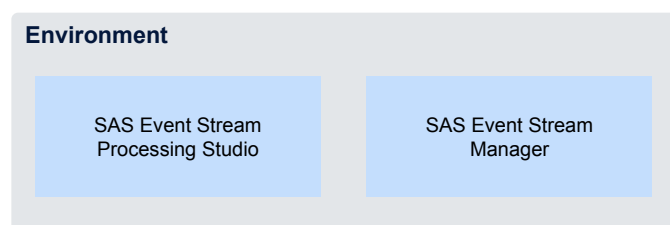
You can use SAS Event Stream Processing Studio to create the projects that you deploy to ESP servers using SAS Event Stream Manager.

SAS Event Stream Manager Instances

By default, a single instance of SAS Event Stream Manager is deployed. This instance supports the management of test and production environments in the same physical environment.

The following example shows a SAS Event Stream Processing environment that contains an instance of SAS Event Stream Processing Studio and an instance of SAS Event Stream Manager.

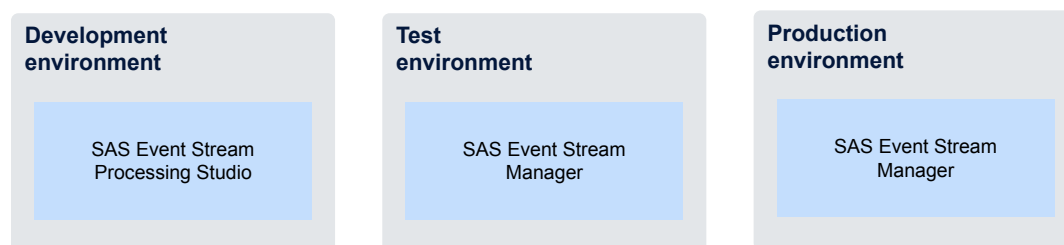
Figure 2 *An Example of an Environment with a Single SAS Event Stream Manager Instance*



However, it is recommended that a separated system with a minimum of two instances of SAS Event Stream Manager is deployed. A *separated system* contains separate test and production environments, with logical separation (separate namespaces) or physical separation (separate clusters) between the environments. Each environment has its own SAS Event Stream Manager instance.

The following example shows a development environment with an instance of SAS Event Stream Processing Studio, a test environment with an instance of SAS Event Stream Manager, and a production environment with an instance of SAS Event Stream Manager.

Figure 3 *An Example of a Separated System*



Here are some benefits of using a separated system:

- Reduced operational risk. In a separated system, it can be easier to prevent untested or experimental projects from affecting live services.
- Protection of data and access. You can limit who can do what in each environment. For example, you give developers access to work in the development and test environments, and allow only designated users to access the production environment. You can keep real customer data out of development and test, and use only masked or synthetic data in those environments.
- Support for reliable promotion and recovery. You promote changes in one direction: from development to test, and then to production. If a test fails, the failure does not affect production users. Rolling back a change is faster and safer when environments are separate.

When you are working with project containers, in addition to promoting project containers to higher promotion levels, you can demote project containers to a lower promotion level. A *project container* is a lightweight container image that complies with the Open Container Initiative (OCI) and is designed to execute projects in a self-contained, immutable container without external software dependencies. For more information, see [“Working with Project Containers” in SAS Event Stream Processing: Using SAS Event Stream Manager](#).

- Alignment with good security practice. Separation of development, test, and production environments is a commonly recognized control in information security frameworks. This separation supports clear boundaries, least-privilege access, and change control.

Here are some examples of typical use cases for separation of development, test, and production environments:

- Regulated workloads. In industries such as financial services or health care, teams use separated environments to meet audit expectations for access control, change management, and data protection.
- Shared platforms. Central IT hosts multiple business projects in SAS Event Stream Processing. Separate environments keep team experiments from affecting other lines of business.
- Deploying the same assets to edge and cloud. Use a project container to facilitate the deployment of assets to multiple environments. You validate project containers in test, and then promote the same versioned artifacts to production in Kubernetes clusters or on edge devices.

The creation of additional SAS Event Stream Manager instances takes place when SAS Event Stream Processing is deployed. For more information about how SAS Event Stream Manager was deployed in your organization, contact your system administrator.

For more information about how SAS Event Stream Manager instances are created when SAS Event Stream Processing is deployed with other SAS Viya products, see [“Creating a Separated System” in SAS Event Stream Processing: Using SAS Event Stream Processing in a Kubernetes Environment](#). For more information about how SAS Event Stream Manager instances are created when standalone SAS Event Stream Processing is deployed, see [“Configure a Separated System for SAS Event Stream Manager” in SAS Event Stream Processing: Using SAS Event Stream Processing in a Kubernetes Environment](#).

Overview of Using SAS Event Stream Manager to Deploy Projects

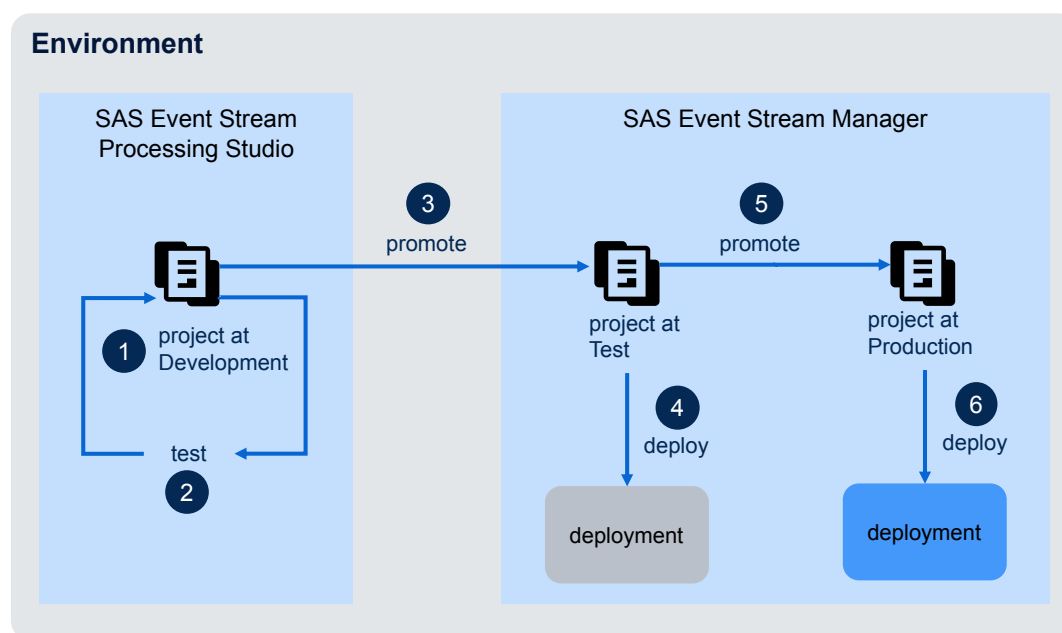
One of the main tasks that you can perform in SAS Event Stream Manager is to deploy SAS Event Stream Processing projects to SAS Event Stream Processing environments. This topic provides an

outline of how a project that is created and tested in SAS Event Stream Processing Studio is made available to SAS Event Stream Manager and then deployed to deployments. In SAS Event Stream Manager, a *deployment* is a logical grouping of assets as a single unit for life cycle management and monitoring.

Deploying a Project When There Is a Single SAS Event Stream Manager Instance

The following diagram shows how a project is deployed when there is a single SAS Event Stream Manager instance.

Figure 4 *Deploying a Project When There Is a Single SAS Event Stream Manager Instance*



- 1 Use SAS Event Stream Processing Studio to create a project, import a project, or install an example project.
- 2 Run the project in test mode to verify that the model works as intended.
- 3 Promote the project from the Development promotion level to the Test promotion level. When SAS Event Stream Processing Studio and SAS Event Stream Manager are running in the same environment, promoting the project to Test also makes the project available in SAS Event Stream Manager.

Promotion levels help prevent projects that are meant for testing from accidentally being used for production (that is, being used in a live environment).

- 4 In SAS Event Stream Manager, deploy the project to a deployment that you are using for testing purposes, to check how the project performs. In SAS Event Stream Manager, a deployment is a logical grouping of assets as a single unit for life cycle management and monitoring.

For example, you can check metrics about consumption of resources in the Kubernetes cluster. If required, at this point you could return to SAS Event Stream Processing Studio to adjust the project and promote another project version to Test.

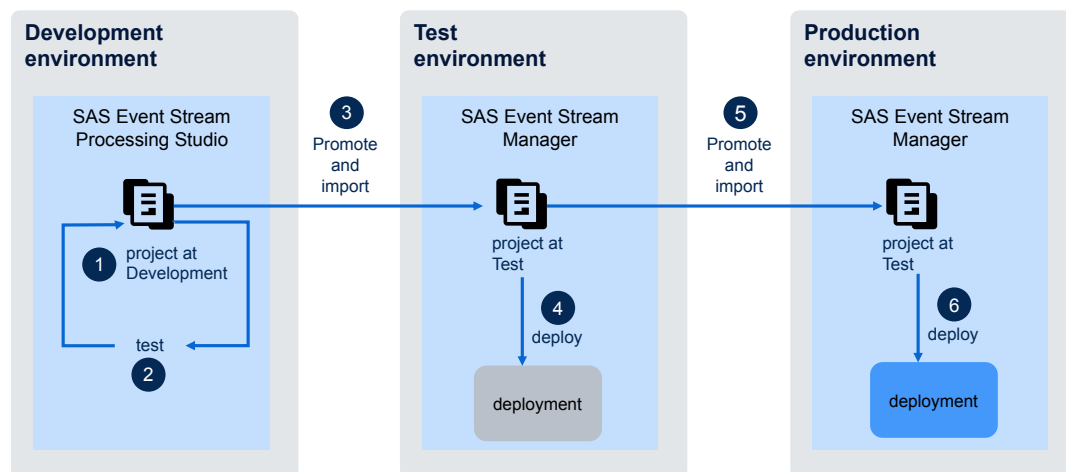
- 5 Promote the project from the Test promotion level to the Production promotion level.
- 6 Deploy the project to a deployment that you are using for production purposes.

For more information about the steps in SAS Event Stream Processing Studio, see [“Example of Promoting a Project Version from Development to Test” in SAS Event Stream Processing: Using SAS Event Stream Processing Studio](#). For more information about the steps in SAS Event Stream Manager, see [“Example of Promoting a Project Version from Test to Production” in SAS Event Stream Processing: Using SAS Event Stream Manager](#).

Deploying a Project in a Separated System

The following diagram shows an example of deploying a project in a separated system. The number of environments and SAS Event Stream Manager instances depends on how SAS Event Stream Manager was deployed in your organization. For more information, contact your system administrator.

Figure 5 Example of Deploying a Project in a Separated System



- 1 In your development environment, use SAS Event Stream Processing Studio to create a project, import a project, or install an example project.
- 2 Run the project in test mode to verify that the model works as intended.
- 3 Promote the project from the Development promotion level to the Test promotion level, and then import the project to the SAS Event Stream Manager instance that is running in your test environment.
- 4 In SAS Event Stream Manager, deploy the project to check how it performs.
For example, you can check metrics about consumption of resources in the Kubernetes cluster. If required, at this point you could return to SAS Event Stream Processing Studio to adjust the project, promote another project version to Test, and import that project version to SAS Event Stream Manager.
- 5 Promote the project from the Test promotion level to the Production promotion level, and then import the project to the SAS Event Stream Manager instance that is running in your production environment.

- 6 Deploy the project. The project is now running in your production environment.

IMPORTANT By default, project versions are stored in a Git repository that is internal to SAS Event Stream Processing. To pass a project from development to test and then to production in a separated system, the project must be stored in a repository on a remote Git server. For more information, see [“File-Based Versioning and Git” in SAS Event Stream Processing: Using SAS Event Stream Processing Studio](#).

If you use SAS Event Stream Manager to deploy a project container in a separated system, the SAS Event Stream Manager instances in the test and production environments must have a connection to the same container registry so that both instances can access the container image. For more information, see [“Add Container Registries” in SAS Event Stream Processing: Using SAS Event Stream Manager](#).

Production Assets

To prevent assets meant for testing from accidentally being used for production (that is, being used in a live environment), certain assets can be designated as production assets.

For example, when a project has been promoted to the Test promotion level, SAS Event Stream Manager does not regard that project as a production asset. In a separated system, you cannot use a SAS Event Stream Manager instance that is designated as a production instance to deploy that project.

In most cases, SAS Event Stream Manager automatically determines whether an asset is a production asset. For example, if a project has been promoted to the Production promotion level, SAS Event Stream Manager normally regards that project as a production asset.

In the following cases, you manually designate assets as production assets:

- a project XML file that is not part of a project package. For more information, see [“Change the Production Status of a Project” in SAS Event Stream Processing: Using SAS Event Stream Manager](#).
- a deployment whose type is Edge. A deployment's type can be Cluster or Edge. When you deploy a project to a deployment whose type is Cluster, an ESP server is automatically created in the cloud. When you deploy a project to a deployment whose type is Edge, you are deploying a project to an ESP server that runs on an edge server (an edge device). SAS Event Stream Manager cannot infer whether the edge device is suitable for production use, so you must supply this information. For more information, see [“Change the Production Status of a Deployment” in SAS Event Stream Processing: Using SAS Event Stream Manager](#).

Integration with Grafana and Monitoring Tools

Visualizing Event Streams

The SAS Event Stream Processing Data Source Plug-in for Grafana enables you to discover and visualize event streams from ESP servers.

For more information, see <https://github.com/sassoftware/grafana-esp-plugin>.

Monitoring Projects

SAS Event Stream Processing Monitoring for Kubernetes uses Grafana to enable project monitoring. It uses Grafana Loki for log aggregation. The tools provided generate metrics to accurately measure CPU usage and memory consumption. Real-time and historical log information is made available at the project level to enable timely debugging.

For more information, see <https://github.com/sassoftware/iot-kubernetes-monitoring-esp/>.

Using SAS Event Stream Processing Web-based Clients as Progressive Web Apps

You can use SAS Event Stream Processing web-based clients (SAS Event Stream Processing Studio and SAS Event Stream Manager) as Progressive Web Apps (PWA). The benefits of using SAS Event Stream Processing as a PWA include the following:

- Application persistence – By default, your session will never time out, so you can restart your work more quickly. Your administrator can enable a time-out for the PWA using a configuration property in SAS Environment Manager. For more information, see [sas.htmlcommons](#) in *SAS Viya Platform: Configuration Properties*.
- Performance – When installed as a PWA, SAS Event Stream Processing typically starts faster than when accessed in the browser.

- Desktop experience – Installing SAS Event Stream Processing as a PWA confers all the benefits of a traditional installation. You can launch SAS Event Stream Processing from the Start menu or Taskbar, and you do not need to sort through countless tabs to find the correct instance of SAS Event Stream Processing.
- Renaming – You can rename each PWA instance to quickly access different environments, such as development, test, or production servers.

For information about installing the web-based clients as PWAs, see the following:

- [“Installing SAS Event Stream Processing Studio as a Progressive Web App” in SAS Event Stream Processing: Using SAS Event Stream Processing Studio.](#)
- [“Install SAS Event Stream Manager as a Progressive Web App” in SAS Event Stream Processing: Using SAS Event Stream Manager.](#)

Examples

Example projects are available through [a GitHub repository](#). You can use SAS Event Stream Processing Studio to install and run these examples. For more information, see [“Install Example Projects” in SAS Event Stream Processing: Using SAS Event Stream Processing Studio.](#)

Tutorial videos for SAS Event Stream Processing are available on the [SAS Event Stream Processing product support page](#).