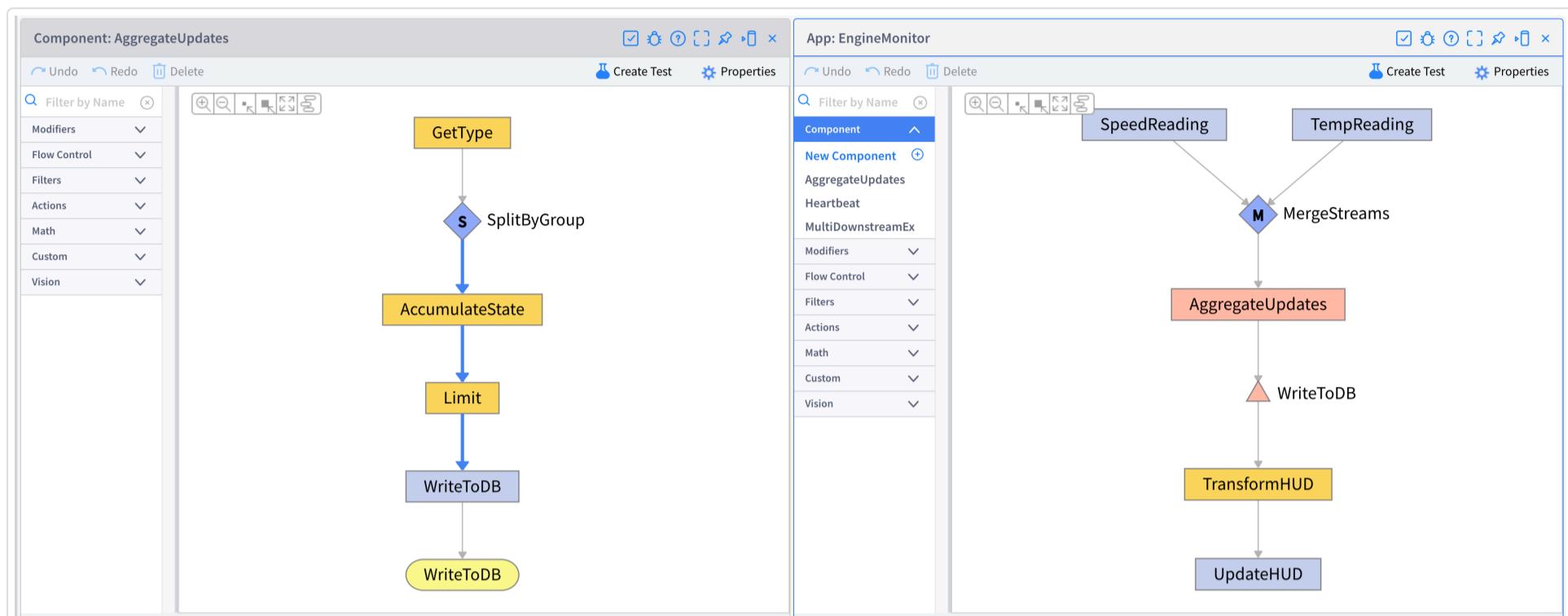


# App Components Guide

Components allow developers to build reusable fragments of Apps that can be shared with other developers and incorporated into their Apps. Components allow developers to quickly build their own customizable Activity Patterns out of the primitive building blocks already at their disposal. These Components then become the unit of re-use in Apps to avoid duplicating chains of tasks that commonly appear together.



## Component Representation

Components can be thought of in the traditional terms of an interface and an implementation. Component authors craft the implementation much like they would craft a normal App. Then, authors may choose how consumers will interact with the Component by providing a configurable interface. Consumers of the Component then use the Component like a black-box, only seeing and adjusting the configuration properties in the interface.

The Component interface is represented as an instance of the `activitypatterns` resource, and a query for all `activitypatterns` will find a record for every Component in addition to one for every built in Activity Pattern. Consumers will interact with a Component interface in the same way they would interact with a built-in Activity Pattern pattern configuration.

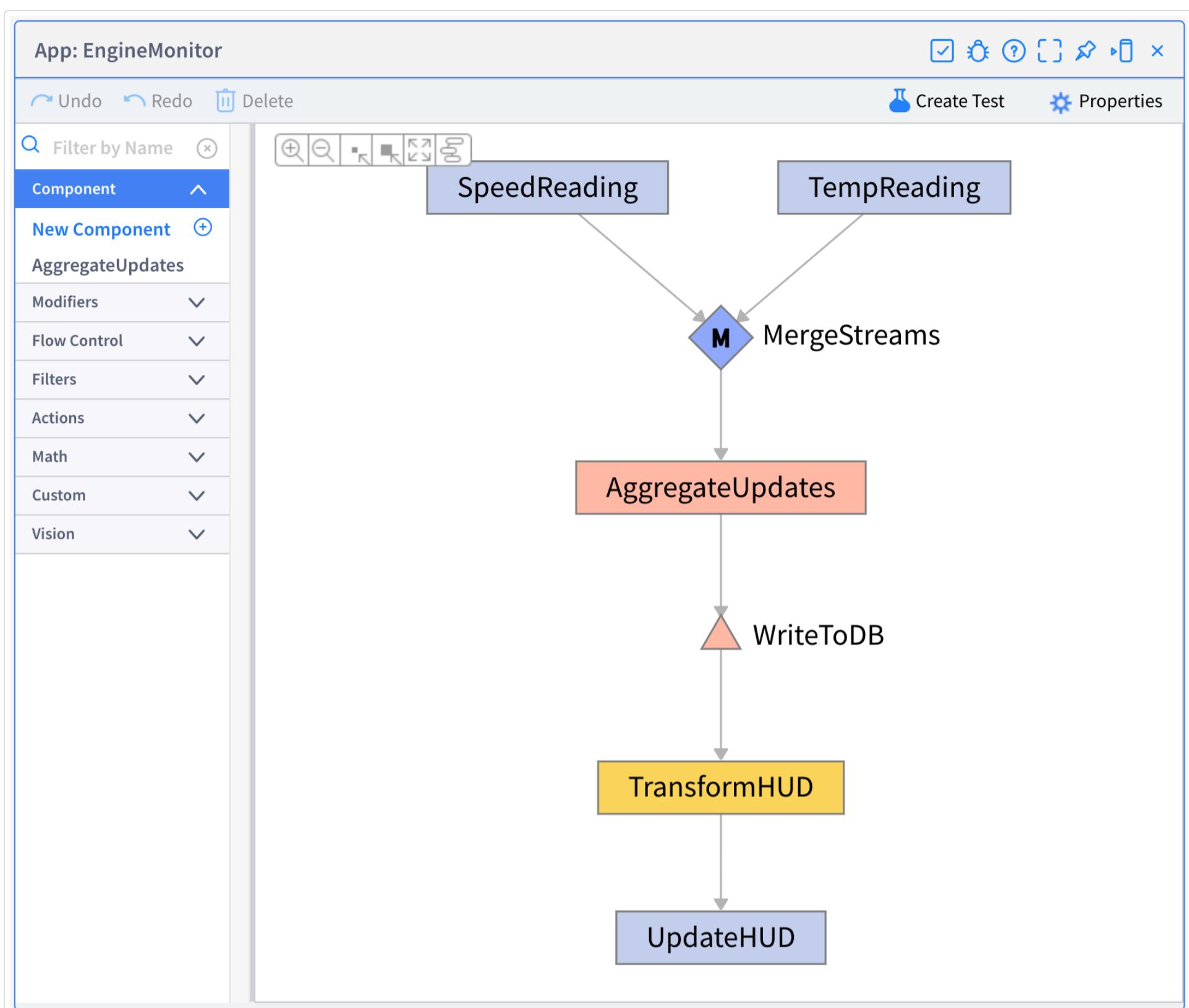
The Component interface also includes a description of its output using *Downstream* events. Each Downstream represents a connection point the consumer can use to trigger other tasks. The combination of a Component's configuration properties and downstream events fully describe the way consumers can interact with the Component.

The Component implementation is represented just like a normal app, as an instance of the `collaborationtypes` resource. A query for all `collaborationtypes` will return both Apps and Components. To distinguish between the two, check if the `isComponent` flag is set to `true`. When an App that references a Component is saved and code generation is performed, the Component implementation is generated into the consumer App *before* code generation is performed on the primitive Activity Patterns.

It is important to consider Components from two different perspectives: developers using existing Components and authoring new Components. These sections are separated below for clarity.

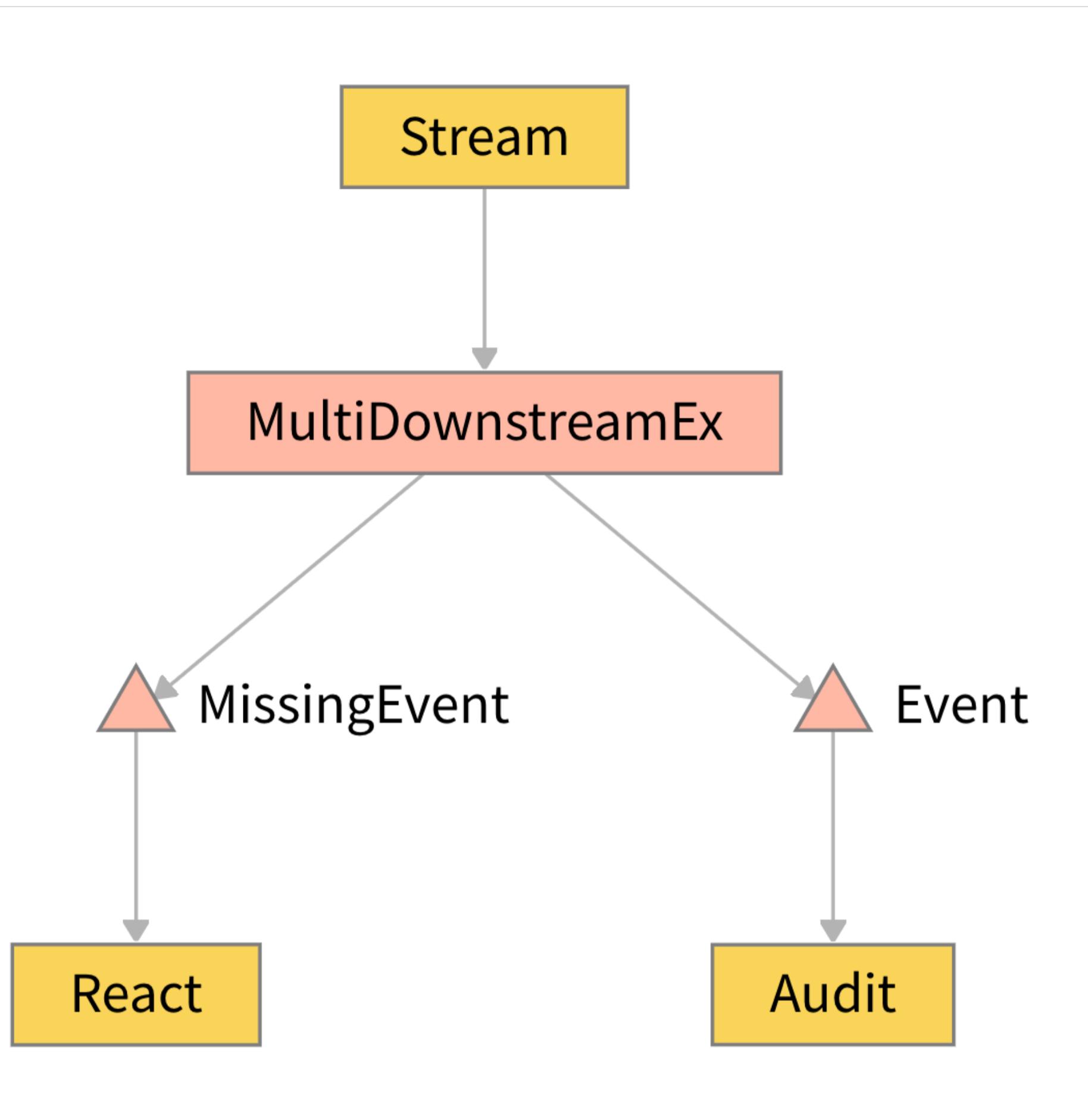
## Using Components

When editing Apps, developers can add Components to their App just like they would add any other activity to the App. The list of available Components can be seen at the top of the Activity Pattern palette on the left side of the pane:



Drag a Component onto the App diagram and connect it by dropping it over an existing task or on a link between two tasks. The Component will be added to the graph as a pink rectangle, along with downstream triangle nodes representing the *downstream* connection points.

In the example above, there is only one exposed downstream (named *WriteToDB*); however, Components can expose multiple downstream events, in which case the App Builder will render a downstream triangle node for each like this:



Once a Component has been added to an app, edit the configuration just like any other task in the App by selecting the node and clicking the “Click to Edit” link to open the configuration modal. Note that clicking any node in an App Component diagram selects all nodes in the Component because the configuration applies to the entire Component.

Note: we sometimes refer to a developer using a Component in their App as a *consumer* of the Component. For instance, the configuration modal described in the previous paragraph may be referred to later as the Consumer Configuration Modal for the Component.

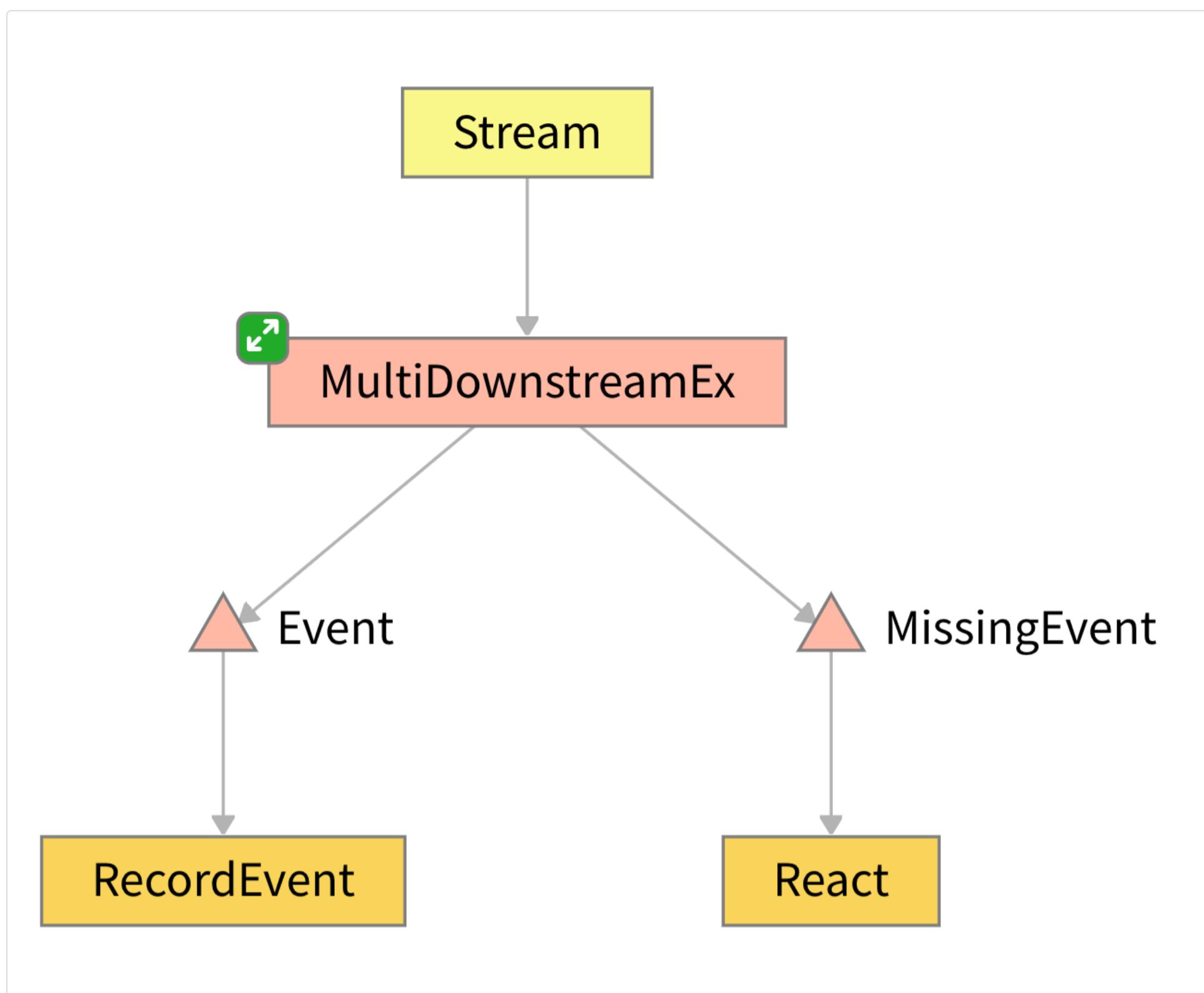
## Importing and Updating Components

App Components can be added to projects the same way Apps can be added to projects. Components are represented as Collaboration Types (just like Apps), so they will appear in the `collaborationtypes` directory of an export. Importing Components works just like importing Apps.

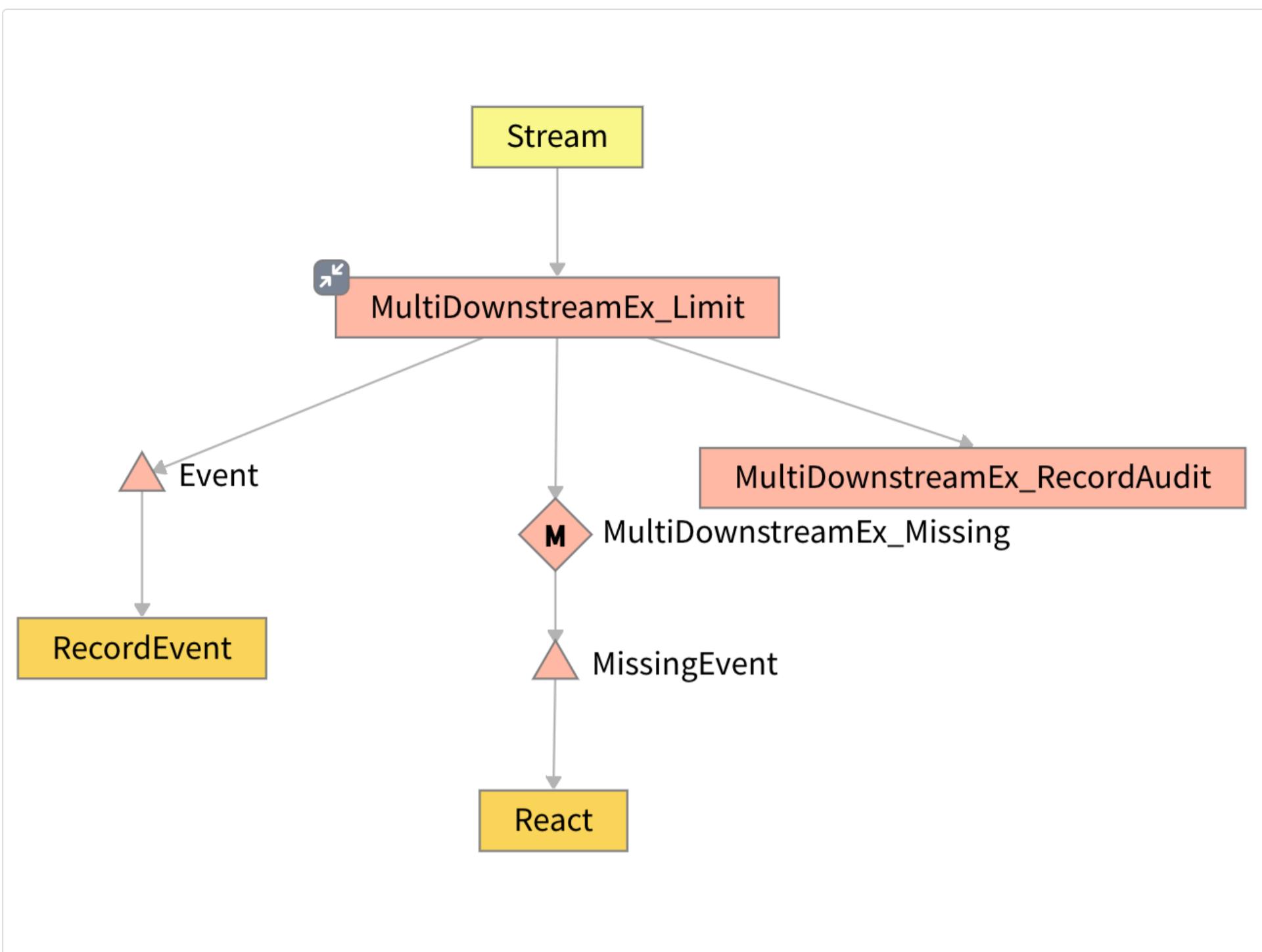
Updating a Component works similar to updates to the built-in Activity Patterns. When a new version is imported it does not immediately have any impact on any running apps that use the Component. The next time an update is made to an App that uses the Component, the updated version of the Component will be used. This gives the App developer a window of time between importing a new Component definition and updating their App to resolve any incompatibilities that the Author may have introduced into the new version. These incompatibilities should be described by the Component Author in the description, but ultimately that is up to the Author.

## Expanding Components

While Components may be edited as single tasks in consuming apps, they are still implemented as a collection of one or more primitive tasks under the covers. After adding a Component to an App and saving it, developers can choose to show the Component as a single node (the way it was originally displayed), or they can expand the Component to show the complete sequence of tasks that implement the Component. To visually expand a Component, hover over the task box, and a green expand icon will appear at the top-left corner of the box like this:



Clicking the expand icon will show all the tasks in the Component implementation, which can be particularly useful on Components with multiple downstream events like the one pictured above. While the collapsed Component shows two downstream events at the same level, if you expand the Component you can see that the downstream events occur at different places in the implementation:

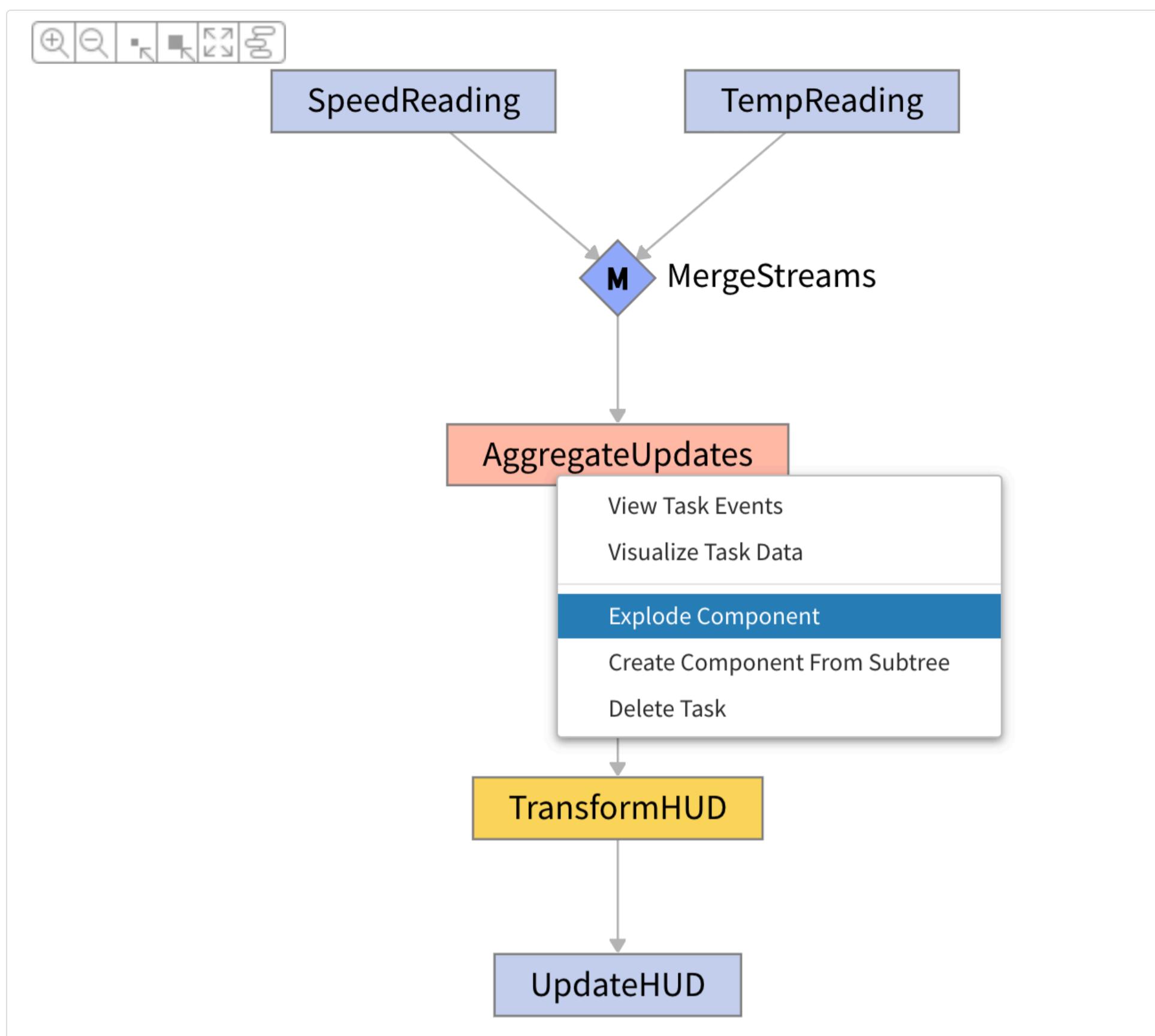


Once the Component is expanded, it can be collapsed by clicking the grey collapse icon in the top-left corner (in the same place as the expand icon).

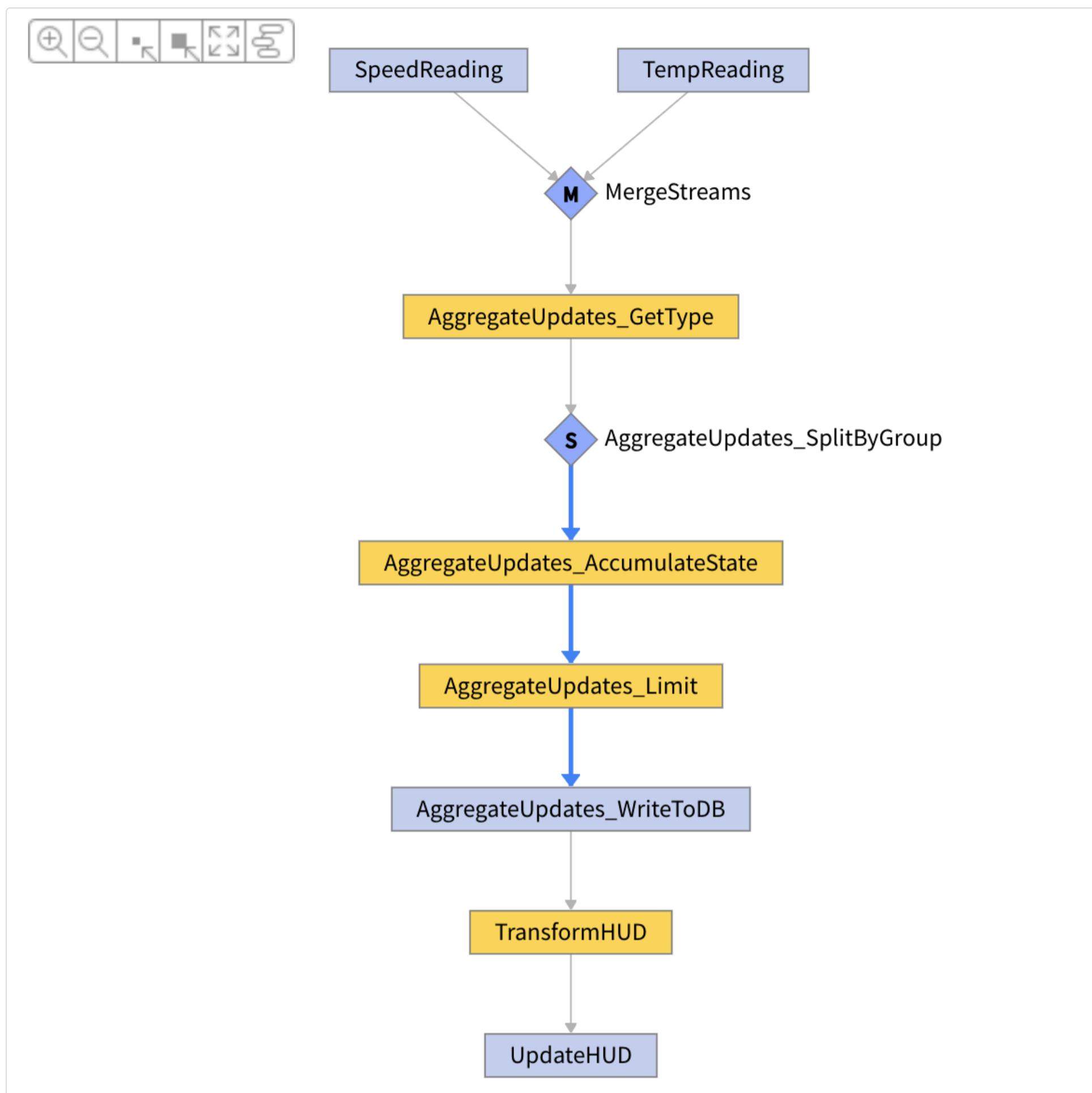
Expanded Components are still treated as a single entity with a single, unified configuration.

## Exploding Components

In some cases you may want to modify configuration properties within the underlying tasks that were not exposed in the Component configuration by the author. The `explode` operation replaces the Component in the App definition with the underlying implementation and removes the read-only restrictions on the Component tasks. The explode operation can be accessed through the context menu by right clicking on a Component node in the App diagram.



After exploding a Component, it is possible to click on the individual tasks separately and modify the configurations individually. The configuration values in the implementation tasks will already be set based on the values generated during the last save of the App.



Exploding a Component is a one-way operation, after which the Component cannot be reconstructed from the decomposed tasks in the App. This means that if a new version of the Component is imported into the namespace, the App will not be able to benefit from any of those changes.

## Authoring Components

There are two ways to start developing a new Component. The first is to start from scratch and add a new Component through the Add popup for Apps:

Apps

+ New Component + New App

Filter by Name

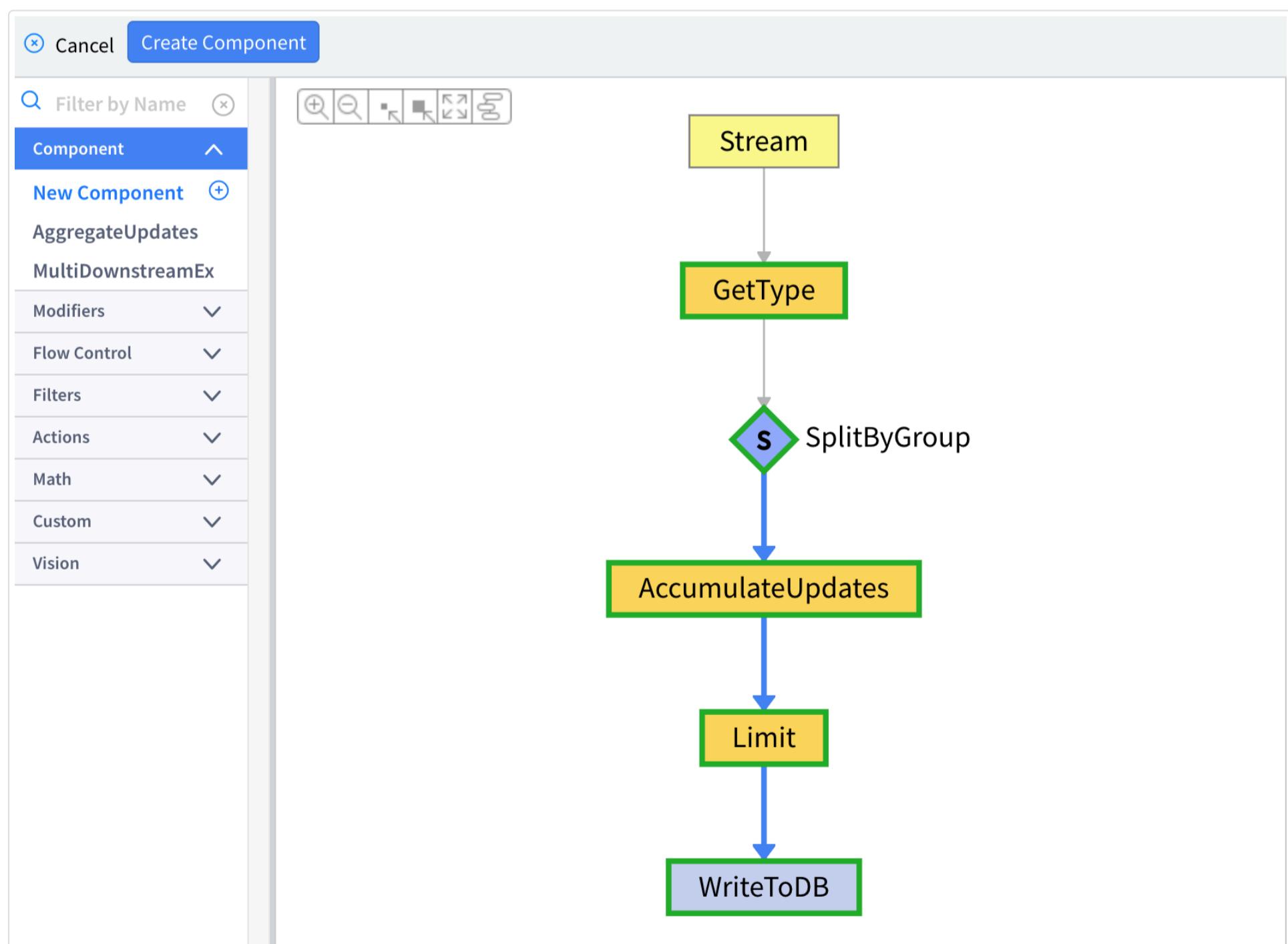
Select All

- AggregateUpdates ^
- EngineMonitor ^
- MultiDownstreamEx ^
- UseMultipleDownstreams ^

Show:  Apps  Components

(x) Cancel

The second way is to start with an existing App, and capture a fragment of the App to serve as the basis for a new Component. To start a new Component capture, either expand the Components section in the palette to the left and click the  button next to **New Component** or shift-click anywhere on the canvas and drag your mouse over the section that you wish to capture:



Once you've selected a segment of the App, you can then select and de-select additional nodes with ctrl-click. Once the desired section of the App has been selected, click the **Create Component** button, give the Component a name, and a new App editor will open with the new Component.

There is only one restriction on the captured App fragment: it cannot contain an Event Stream Activity Pattern.

## Component Interfaces

The Component Interface defines the configuration properties that will be exposed to consumers of the Component. When a consumer configures a Component, the interface is rendered as a configuration modal window with an entry for every configuration property defined on the Component. Each configuration property has a few attributes that determine its behavior:

- name - Must be unique within the Component interface, but can match a config property from an underlying task
- type - The type of value the consumer is expected to supply. This value determines the kind of input field rendered in the consumer's configuration modal.
- required - Is the consumer required to specify a non-null value for this property?
- multi - Is the expected value of this config property an array of values?
- default - A default value to use if none was provided by the consumer. Only allowed for non-required parameters.
- description - An optional explanation to present to the consumer in the Component configuration modal

Add new configuration properties to the Component by clicking the **+ Add Property** button in the top-right corner. Properties in the Component Interface can also be re-ordered with the up and down arrows shown in the screenshot below. To edit the description of a config property and view any advanced options for the specific property type, click the pencil icon at the end of the row:

| Name       | Type    | Req                                 | Multi                    | Default | Actions |
|------------|---------|-------------------------------------|--------------------------|---------|---------|
| interval   | String  | <input type="checkbox"/>            | <input type="checkbox"/> |         | ↑ ↓ ⚒️  |
| maxUpdates | Integer | <input type="checkbox"/>            | <input type="checkbox"/> |         | ↑ ↓ ⚒️  |
| typeName   | Type    | <input checked="" type="checkbox"/> | <input type="checkbox"/> |         | ↑ ↓ ⚒️  |

Properties can also be added to the interface from any task configuration modal in the Component by clicking the plus button to the right of the input field:

| Required Property          | Value      |
|----------------------------|------------|
| interval (Interval String) | 10 seconds |
| maxCount (Integer)         | 1          |

**Expose Property**

Clicking this button will automatically set the value of the config property to  `${propertyName}` which indicates this value will be replaced using the configuration of the Component in the consumer App. More on this in the [Component Implementation](#) section below.

**Cancel** **OK**

Clicking this button will automatically set the value of the config property to  `${propertyName}` which indicates this value will be replaced using the configuration of the Component in the consumer App. More on this in the [Component Implementation](#) section below.

## Configuration Property Types

Configuration properties can take on a variety of different types, including all the primitive VANTIQ [property types](#) as well as a collection of other non-primitive property types that produce unique form fields or drop-lists with specific choices. The property type configured in a Component interface will determine the way the property is presented to consumers of the Component.

The following is a list of supported configuration property types (primitive representation in parentheses):

- VAIL (String) - A nested modal to enter a multi-line VAIL block
- VAIL Expression (String) - A single line input field to enter a single VAIL expression.
- Type (String) - The name of a type selected from a dropdown of types in the current namespace.
- Source (String) - The name of a source selected from a dropdown of sources in the current namespace.
- Procedure (String) - The name of a procedure selected from a dropdown of procedures in the current namespace.
- Event Type (String) - The name of an event type selected from a dropdown of event types accessible through catalogs in the current namespace.
- Catalog (String) - The name of a catalog selected from a dropdown of catalogs accessible to the current namespace.
- Collaboration Type (String) - The name of a collaboration type selected from a dropdown of collaboration types in the current namespace.
- Secret (String) - The name of a secret selected from a dropdown of secrets in the current namespace. NOTE: This cannot be used to fetch the secret value.
- Client (String) - The name of a client selected from a dropdown of clients in the current namespace.
- Imports (String) - A resource and resourceId to be imported into the Component. NOTE: This should generally be used as a multi parameter to match the underlying Activity Patterns that expose an imports parameter.

## Component Implementations

Add new tasks to a Component by dragging items from the palette onto the canvas, just like in an App. If you started with a capture of an existing app, then the existing tasks will already be configured with the values from the original App. Tasks in a Component can be configured just like tasks in an App. The key difference is the ability to refer to configuration properties defined on the Component.

Next to every input field configuration property you should see a dropdown suggestion caret that lists the options that can be referred to for a configuration property. These options will contain a mix of the normal values that could be provided for the input type in a regular app, along with the properties that can be referenced from the Component interface. When values from the Component interface are selected, they will appear within `{} ${}` to indicate they will be replaced with values from the Component configuration during App code generation.

**Specify configuration properties for Limit Activity 'Limit'**

*Limit the number of events that flow through an app by discarding events after the limit has been reached for the configured interval.*

> [AI Configuration Assistant](#)

| Required Property  | Value  |
|--|--|
| interval (Interval String)   | <input type="text" value="\${interval}"/> <span style="float: right;">+</span>   |
| <i>The length of time to apply the maxCount limit. Must be a valid interval string like '1 second' or '3 minutes'.</i> |  |
| maxCount (Integer)   | <input type="text" value="\${maxUpdates}"/> <span style="float: right;">+</span> |
| <i>maxUpdates specified interval should be allowed through this Limit task.</i>  |  |

Cancel
OK

Substitution variables can be nested into any input type supported by the App Builder, including in VAIL properties like this:

```
SELECT * FROM ${typeName} WITH LIMIT = ${maxCount}
```

procedure Type: **VAIL Block**

*A block of VAIL code that may update the value of a variable named 'state' using data in the 'event' variable.*

procedure:

```

1 // Update the value of state using event.
2
3 ▼ if (!state) {
4     state = Utils.getType("${typeName}")
5 }
6
7 event.typeDef = state

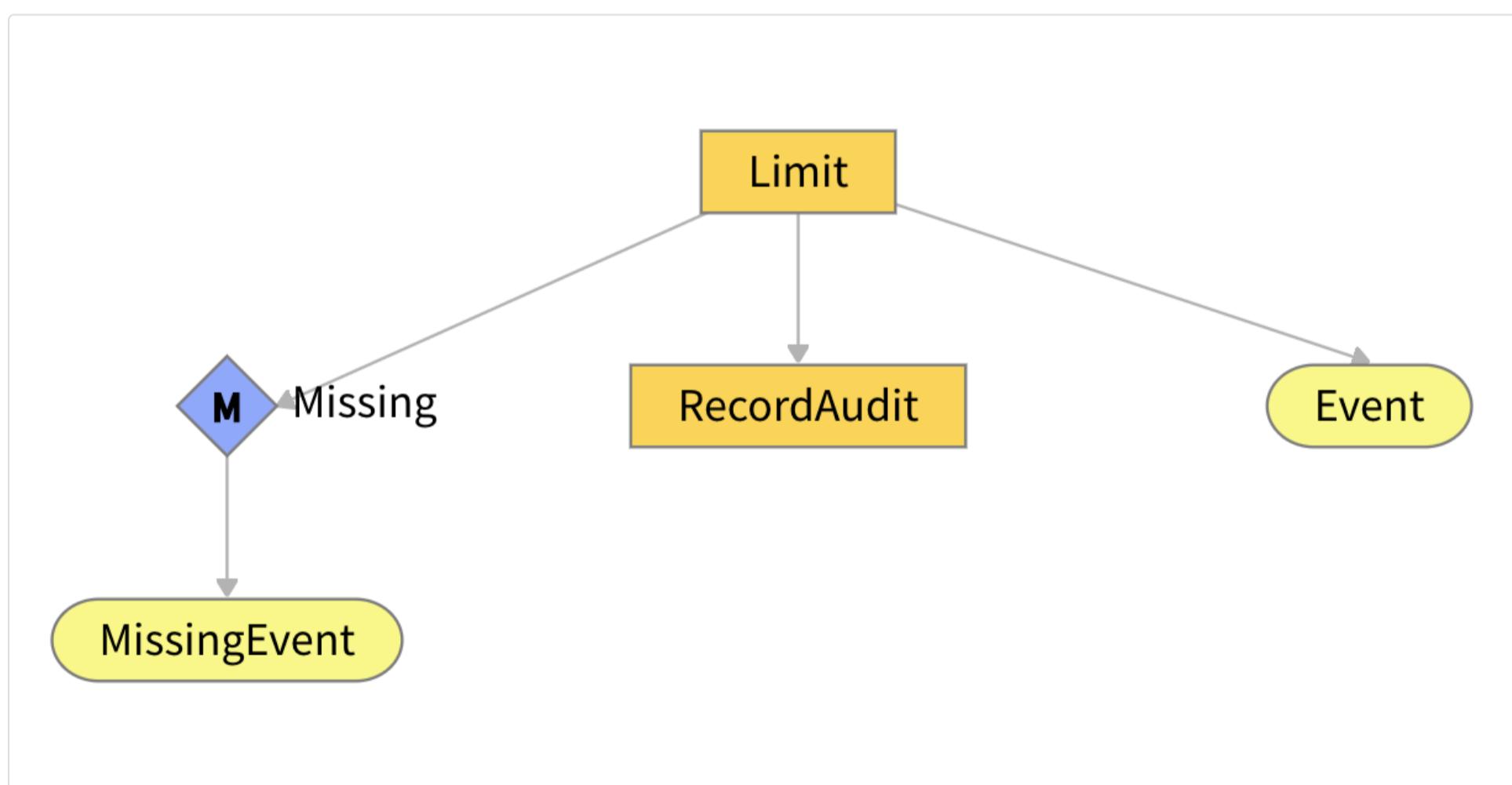
```

Cancel **OK**

When an App that references a Component is saved, code generation occurs in two phases. First, the Component references are resolved by looking up the Component implementation and resolving all of the  `${property}`  references in the Component definition based on the configuration values supplied in the consumer App. If any errors occur during Component resolution, these errors will be reported immediately and code generation will halt. After Component resolution occurs, normal code generation occurs for all tasks in the App in parallel.

## Component Downstreams

While Components must have only one root node, they can contain branches that lead to multiple endpoints. Component authors can then select which endpoints, or intermediary tasks, to expose as downstream connection points to consumers of the Component. For example, the following Component contains three branches, but only two lead to downstream nodes that are usable in consumer apps.



Downstreams are another part of the interface a Component offers to consumers. This extra layer of abstraction allows the Component author to change tasks within the Component implementation without creating an incompatibility with any apps that already use the Component.

## Custom Generator Procedures

It is possible to customize the Component resolution and variable substitution phase for a Component by defining a custom generator procedure. By default, every Component uses the built-in procedure `AppGeneration.generateComponent`, which performs the basic variable substitution from a configured interface into the Component implementation during code generation.

To override the default generation Procedure, click the *Custom Generation Procedure* button in the Component Configuration Modal, and the window will expand to show the procedure definition:

The screenshot shows the 'Component Properties' dialog. On the left is a table with three rows:

| Name       | Type    |
|------------|---------|
| interval   | String  |
| maxUpdates | Integer |

On the right is a large code editor window containing a template for a 'Custom Generation Procedure'. The code is color-coded and numbered from 25 to 47. It includes comments explaining variables like componentDef, assembly, config, and collaborationType.

```

25 // BEGIN CUSTOM COMPONENT GENERATION CODE
26 // referencable variables:
27 // - componentDef: Original definition of this component (an instance of
28 // system.collaborationtypes)
29 // - assembly: The assembly for this component. Changes made to this object will be
30 // reflected in generated code
31 // - config: Object defining substitution keys used to generate component assembly
32 // - collaborationType: Definition of the collaboration type undergoing code
33 // generation that references this component
34 //
35 var typeDef = Utils.getType(config.typeName)
36
37 config.groupByClause = "event."
38
39 if (TypeDef.groupBy) {
40     config.groupByClause += typeDef.groupBy
41 } else if (TypeDef.naturalKey) {
42     config.groupByClause += typeDef.naturalKey[0]
43 } else {
44     exception("AggregateUpdates.invalid.type", "Type {0} does not have a group by or
45     natural key", [config.typeName])
46 }
47 // END CUSTOM COMPONENT GENERATION CODE
48 ..

```

At the bottom right of the dialog are 'Save' and 'Close' buttons. At the very bottom right of the entire interface are 'Cancel' and 'OK' buttons.

A template based on the default generation procedure is provided out of the box, including comments describing the variables available that can be referenced or modified.

Most important, the custom generator Procedure can modify the config object that is used at variable substitution time for all the config properties that contain  `${property}` references. Use the generator procedure to infer or assign values to additional config properties that can be referenced throughout the Component implementation without exposing any new config properties in the Component interface.

Whenever possible, try to infer related configuration values rather than add additional configuration properties that consumers need to configure. For example, if there is a configuration property that references a type, it's usually possible to infer things like the qual used to look up a unique instance, or the expression to use in a Split By Group task from the natural key or unique indexes on the type.

Copyright © 2024 VANTIQ, Inc.