App (aka Visual Event Handler) Builder Guide

Introduction

Modern Real-Time Business Applications often have, at their core, a collection of disparate data streams that contain inter-related information. Value can only be extracted from this data after it has been massaged, filtered, and combined with other data flowing through separate parts of the business. Vantiq models the flow of this data as a sequence of "events". These events may arrive from outside the Vantiq system (using a variety of protocols) or be generated from within the Vantiq platform.

Events in a Vantiq Application are received and processed by Services via event handlers. This document covers the construction of *visual* event handlers (aka Apps or VEH). VEH are built graphically using the App Builder, which allows users to layout the general flow of data through their system. The App Builder allows users to capture, transform, and make decisions on streams of events occurring within the Vantiq system without writing much, if any, code. The App Builder goes beyond traditional ETL pipelines by not just transforming data, but also identifying important business situations within the event streams which can automatically drive collaborations. Vantiq's mission is to help businesses Sense, Analyze, and Act on real-time data, and the App Builder helps complete that mission by dramatically reducing the amount of code (and development time) that goes into sensing and analyzing the data.

Some of the high level capabilities of the App Builder include:

- Ingesting raw streams of data. The data can come from events on a source like Kafka or MQTT, inserts or updates on a Type, publishes to topic, or from a service event type.
- Transforming events in a stream of data via a visual interface or through a VAIL procedure.
- Joining multiple streams of events on specific conditions like correlated ids or events that occur within a constrained time window.
- Filter out events that don't match specific criteria.
- Updating in-memory state
- Automatically update persistent storage (types) for new events captured on a stream.
- Log events for debugging purposes.
- Initiating human-computer collaboration.

Tutorial

To walk through building your first service, complete with visual event handlers, check out the tutorial.

App Builder Overview

The App Builder is launched when creating/editing a visual Service Event Handler and presents an event handler diagram. This diagram consists of boxes, representing **tasks**, and lines, representing **streams** of events flowing between tasks. Tasks perform an operation on **inbound** events and then emit **outbound** events for the next task to consume. Each task in an event handler consumes one or more inbound streams of events, and produces one or more outbound streams of events. Tasks with more than one outbound event depict their primary output as a line, directly between the producing task and its consumers. Any additional events are depicted using triangles between the tasks. The events in all handlers flow downwards, starting with the root **Event Stream** tasks at the top of the diagram.

VEH are, under the covers, completely implemented in VAIL. When a VEH is created or updated, a collection of rules, event streams, and procedures is generated to implement each task.

Visual event handlers are assembled from a collection of predefined Activity Patterns. The current set includes:

- Accumulate State – Aggregate state over a series of events by calling a procedure that takes the existing state object and the new event and produces the new state object.
- Analytics – Compute a collection of aggregate statistics for events in a stream.
- Answer Question – Answer a question using a specified semantic index and return the generated answer.
- Assign – Assigns the inbound event to the specified entity or collaborator role.
- Branch – Create a conditional branch in the flow of a service event handler.
- Build and Predict Path – Assemble the path from the tracked motion of objects with locations. Predict the location of the object when it is removed from tracking.
- Cached Enrich – Enrich and cache the results for future events. The cached association is refreshed at a configurable interval.
- Chat - Initiate a Chatroom that uses a Chatbot to process chat message and react appropriately.
- Close Collaboration – Close the current collaboration and set its status to either 'failed' or 'completed'.
- Compute Statistics – Compute a collection of aggregate statistics for events in a stream.
- Convert Coordinates – Convert coordinates (typically from image analysis) to another coordinate space.
- DBScan – Applies a density based clustering algorithm to determine if there are one or many clusters
- Delay – Delay the emission of the inbound event for a configurable length of time.
- Dwell – Detect a sequence of events with a consistent state across over a period of time.
- Enrich – Augment events with associated data from a persistent Type.
- Establish Collaboration – Ensures that only one collaboration is created for the entity role specified by the roleName.
- Event Stream – Identifies a stream of events to listen for and serves as the root task in an event handler.
- Filter – Defines a condition that restricts which events flow on to downstream tasks.
- GenAI Flow – Defines an embedded GenAI Flow to be executed by the task.
- Get Collaboration – Return the current collaboration
- Join – Join pairs of events on two parallel streams.
- K-Means Cluster – Applies a K-Means Clustering algorithm to find the specified number of clusters in the data

- Limit – Limits the flow of events through a task to a configurable maximum throughput.
- Linear Regression – Applies linear regression to the data and generates a prediction procedure to predict future data points.
- Loop While – While the specified condition is true, loop through the tasks that are part of the *loop body*.
- Log Stream – Log all inbound events
- Merge – Join multiple independent streams into a single stream.
- Missing – Detect the absence of an event for a time interval.
- Notify – Send a notification to a Vantiq user using the Vantiq Mobile application.
- Polynomial Fitter – Applies a polynomial fit algorithm to the data and generates a prediction, derivative, and integration procedures.
- Predict Paths by Age – Predict positions for paths not recently updated.
- Publish To Service – Send an event to an Inbound Event Type of a Service
- Publish to Source – Send inbound data out to a source.
- Publish to Topic – Send inbound data out to a topic.
- Procedure – Execute a procedure and emit the results.
- Rate – Emit the frequency of inbound events at a regular interval.
- Recommend – Generate a list of recommendations based on the incoming event and and the match directives.
- Record Event – Record an instance of an Event Type in the Event Ledger.
- Reply – Send a reply to the caller of **Event.request** responsible for triggering the current event (if any).
- Run TensorFlow Model on Document – Use a TensorFlow model to analyze a Vantiq Document (typically an image).
- Run TensorFlow Model on Image – Use a TensorFlow model to analyze a Vantiq Image.
- Run TensorFlow Model on Temp Image – Use a TensorFlow model to analyze a Vantiq Image
- Run TensorFlow Model on Tensors – Use a TensorFlow model to analyze data.
- Sample – Sample events from the inbound stream to produce fewer events to the downstream tasks.
- Save to Type – Save inbound data to a type.
- Split By Group – Split the upstream events into sub-streams for the downstream tasks by a group key.
- Submit Prompt – Submit a prompt to the specified LLM and return the result.
- Threshold – Detect the crossing of a threshold across sequential events.
- Time Difference – Measure the time between two consecutive events.
- Track Motion – Assemble the movement of objects with locations (typically YOLO image analysis output).
- Track – Begin tracking one or more Vantiq users until they arrive at a specified location or the collaboration ends.
- Transformation – Defines a procedure or mapping that modifies an event in the stream before being passed to downstream tasks.
- Unwind – Splits an array property on an event into multiple separate events that can be processed independently.
- VAIL – Run an arbitrary block of VAIL code for every event.
- VisionScript – Enables the user to build a *Vision Script* in order to manipulate images.
- Window – Buffer inbound events to emit batches of events. Overlapping and non-overlapping windows are supported.
- WithinTrackingRegion – Filter locations by presence in particular tracking regions.
- YOLO from Documents – Use a YOLO model for image (stored in a document) analysis.
- YOLO from Images – Use a YOLO model for image analysis.
- YOLO from Temp Images – Use a YOLO model for image (stored as a temp image, returned from a video source) analysis.

From these built-in Activity Patterns larger *Components* can be developed that encapsulate collections of tasks. Components can then be shared between developers and used just like any other built-in Activity Pattern.

Creating a VEH

When creating a new VEH, the initial diagram consists of a single Event Stream task. From there, more event streams and downstream tasks can be added by right clicking on an existing task and selecting one of the options from the contextual menu. Each task added to the VEH diagram needs to be properly configured in order for the handler to compile. VEH are compiled whenever they are saved, and if no errors are found during compilation, the VEH will generate a collection of **hidden** VAIL artifacts that implement each of the tasks. If errors are detected during compilation of the app, it will still be saved. When this happens any task in which an error was detected will be highlighted in red.

The App Builder is designed to facilitate an iterative development process. For example: Start with an initial Event Stream and a log stream task to see what raw events are being received. Next add a transformation task that processes the outbound events from the raw event stream and then log the output of the transformation. Then add a filter to the output of the transformation and log the filtered event stream. Once you've tuned the filter and transformation activities to produce the output you expect, you can add a Notify task to the output of the filter task, and you've built an entire handler.

Reliable Apps

When the source of the events for a VEH is configured to be reliable, the Vantiq platform will ensure that the reliability guarantee is extended to the VEH as a whole. This means that if the VEH terminates due to a system failure (such as a Vantiq server instance being shut down), processing will be restarted at the appropriate point such that the semantics of the handler are preserved. In keeping with the low-code nature of the App Builder, this is done automatically and does not require that the developer explicitly manage event acknowledgment or redelivery.

> To be fully reliable, the service **must** also be marked as "replicated" (for reasons that we discuss in the next section).

Tracking Progress

As a reliable event is processed, the system keeps track of the event's progress through the event handler. This is done by recording "checkpoints" as the event (and those that it "produces") are processed by the various tasks. Checkpoints are stored in the owning service's state and are partitioned using the event UUID values. As a result any service with a reliable event handler is automatically stateful. It also means that to be fully reliable, the service must be replicated. That way the checkpoint state is also reliable and can survive various system failures.

When a task completes, it will update the checkpoint to indicate the "child" tasks that will be triggered next. Note that this is only done for children of the *primary* event output. Secondary event outputs (those denoted by a "triangle" in the connector) are not part of the reliable processing. In addition, the Rate and Missing Activity Patterns are considered "terminal" when it comes to reliable processing since the events they produce are not direct dependents of a single source event.

Once there are no more child tasks to be executed, the initiating reliable event will be acknowledged.

Event Redelivery

Should a system failure occur before an event is fully processed, it will be redelivered (per the at least once delivery semantics). When this occurs, the VEH will use the stored checkpoint information to determine where it should continue processing. In general this avoids rerunning tasks that have already been completed, with one important caveat. Due to the way task execution works, there is a small window between when the task's "work" completes and when the checkpoint is updated. If a failure occurs during that window, then the checkpoint will still indicate that the task has not yet completed. This means that the task may be repeated in this case. The implications of this depend on the task's Activity Pattern.

Activity Patterns that operate on service state have been implemented such that should this repetition occur, any attempts to repeat a state change will be ignored. This includes all of the stateful patterns such as AccumulateState and Dwell (see the pattern descriptions for details). Tasks based on any other pattern have no option but to be repeated since there is no internal indication of whether or not they have been done. As a result the developer must be aware that this may occur and be prepared to mitigate as necessary. For example, ensuring that a given procedure can be executed multiple times without adverse side-effects mitigates against the repetition of a Procedure task.

Error Handling

Unless otherwise specified, errors that occur during the execution of a task are considered by be "application" errors and thus will *not* trigger redelivery of the initiating event. The reason for this is that such errors are typically triggered by the contents of the event and thus redelivery would only result in repeated generation of the same error. Encountering a non-transient error triggers acknowledgment of the event and clears any associated checkpoint.

The exceptions to this behavior are:

- Errors which are known to be due to "cross-cluster" communication failures. These errors are typically the result of some transient condition in the cluster (a member being restarted, temporary network issues, etc…).
- Errors flagged as transient by a handler provided by the following Activity Patterns:
    - Procedure
    - VAIL
    - Publish to Source

Transient errors are recorded, but do not trigger either acknowledgment or a checkpoint update. Processing will continue from the point just before the error when the event is eventually redelivered.

Collaboration Management

The following activity patterns are "collaboration aware" and will create tasks which manage the service's collaborations:

- Answer Question
- Assign
- Chat
- Close Collaboration
- GenAI Flow
- Notify
- Recommend
- Submit Prompt
- Track

Adding any of these to a VEH enables collaboration management. As a result, when an event reaches the first Collaboration task on its path from root to leaf, a new collaboration is started and is added to the Service State. As the event is processed between the first Collaboration task and the leaf node for the handler, the collaboration is updated and managed in the Service State.

Activity Patterns

Tasks in a VEH are configurations of predefined Activity Patterns. Each Activity Pattern represents a generalized process that can be expressed with just a few configuration properties. Given an Activity Pattern and a configuration of the pattern, we can produce VAIL code that implements the specifics of the task in the App Builder.

Below are descriptions of the existing Activity Patterns and the configuration properties that each contains. Unless otherwise specified, each Activity Pattern produces a single outbound event stream.

Accumulate State

Keep track of state across a series of events with Accumulate State. Initially, the state value is *null*. For each inbound event the state object is updated by calling the configured accumulator procedure (or VAIL block) with the existing state object as the first parameter, and the new event as the second parameter. The updated state can then be ignored (recommended), assigned to a property in the outbound event, or it can replace the outbound event completely. The first option is recommended because it is much more efficient and because the state is accessible from the handler's service.

For example, consider the following accumulator procedure that keeps track of the number of events with temperatures above and below 100 degrees:

```
PROCEDURE accumulateHighLowTempCounts(existingState, event)

if (!existingState) {
    existingState = {
        lowCount: 0,
        highCount: 0
    }
}

if (event.temp > 100) {
    existingState.highCount++
} else {
    existingState.lowCount++
}

return existingState
```

And consider the following sequence of three events:

```
{
    "temp": 88,
    "sensorId": "XYZ"
}
```

```
{
    "temp": 111,
    "sensorId": "XYZ"
}
```

```
{
    "temp": 109,
    "sensorId": "XYZ"
}
```

The return from the state getter after each event would be:

```
{
    "lowCount": 1,
    "highCount": 0
}
```

```
{
    "lowCount": 1,
    "highCount": 1
}
```

```
{
    "lowCount": 1,
    "highCount": 2
}
```

AccumulateState has the following optional configuration properties:

- **procedure** – two options:
  - *Accumulator Procedure*: The name of the accumulator procedure to call for each event to update the accumulated state. The procedure should take two parameters, both objects, as described above.
  - *VAIL Block*: A block of VAIL code to run that uses `event` to modify `state`. No return value should be used in this case.

If the procedure property is not specified, it will default to `state = event`, allowing AccumulateState to store the most recent event in Service state.

The code specified for the `Accumulate State` task may be executed more than once for a given event (due to retries necessary to implement optimistic concurrency control (OCC)). If/when this occurs, only the result of the final execution will be used to update the state. However, it means that the code block should not have any external side-effects or affect state outside of that being managed by the Accumulate task.

- **outboundBehavior** – Chosen from a list of three options:
  - *Emit original event* - Emit the original inbound event as the outbound event.
    This is the default behavior if no option is selected. In this case, the state code executes asynchronously, so may not be finished by the time any following app tasks run. This works well for applications that only need to show the stats and later tasks in the app do not depend on the exact results of this task.

  - *Attach state value to outboundProperty* - Attach the state value to the configured outboundProperty.
  - *Replace outbound event with state value* - Emit the state value instead of the inbound event.
- **outboundProperty** – The property that will contain the state on event emission when the `Attach state value to outboundProperty` is chosen.
- **imports** – A list of resources referenced by the VAIL block procedure that must be imported.
- **stateProperty** – The name of the state property (global of partitioned) that this accumulator will update. If not specified, a unique state property will be generated.
- **resetFrequency** – Frequency with which the Reset Procedure should be automatically called. This will reset the state to null and emit a *reset* event.
- **schema** – The name of a type that defines the schema of the output of the outbound state. Specifying the schema here will simplify the configuration of downstream tasks.

Accumulate State produces the following event streams:

- **event** (default) – The event emitted depending on the outboundBehavior. This can either be the original event, the original event + state, or the state value.
- **reset** – If a *resetFrequency* is specified, the task will automatically emit a *reset* event with the value of the state just before the state is automatically reset to null.

Analytics

The Analytics activity computes one or many of the following statistical operations: *mean, median, count, min, max, standard deviation, geometric mean, variance, skewness,* and *kurtosis*.

The Analytics activity contains the following required configuration properties:

- **inboundProperties** – The list of property names (e.g. “speed”, “temperature”, etc.) on the inbound events to compute statistics for. Each property must be numeric.
- **reservoirType** – The type of reservoir used to store data points. The type will determine which algorithm is used for computing aggregate statistics and when to purge old data points.
  - *Sliding Time Window* – Computes statistics using `Real` values. All events that occured within the specified time window will be used to calculate statistics (e.g. all events in the last 5 minutes)
  - *Sliding Count* – Computes statistics using `Real` values. The last *n* events will be used to calculate the statistics
  - *Fixed Time Window* – Computes statistics using `Real` values. A fixed time window will add events to the window until the time of the next reset. For example, a fixed time window of 1 hour will calculate analytics for all events within the hour, starting at the top of the hour and ending at the top of the new hour.
  - *Fixed Count* – Computes statistics using `Real` values. Calculate analytics on all of the events until the specified number of events has been reached. Then the analytics will be reset back to null.
- **windowLength** – How long events should stay in the window. For time based reservoirs this is an interval string (e.g.: “10 minutes”). For count based reservoirs this should be the number of events.
- **operations** – A selection of which statistics operations should be applied to the data

The Analytics activity contains the following optional configuration property:

- **dimension** – The hierarchical dimensions that the statistics will be applied. The data will automatically be partitioned at the highest dimension. Statistics will be calculated at the lowest dimension. Each subsequent dimension will return an aggregate of the lower level aggregates. For example, a two level hierarchy “warehouse”, “machine” will first calculate averages on each machine, then calculate the average of averages for all machines in each warehouse to produce per warehouse analytics.

Analytics activity pattern produces the following event streams:

- **event** (default) – Emits the original incoming event.
- **reset** – If the reservoirType is set to *Fixed Time Window* or *Fixed Count*, the Event Handler will automatically emit a *reset* event with the value of the analytics just before the state is automatically reset to null.

The Analytics activity pattern generates three Procedures: Get, Update, and Reset. The Get procedure can be used to retrieve the current result of the analytics. An example return value is shown below:

```
{
   "speed": {
      "count": 3,
      "min": 10,
      "max": 28,
      "mean": 19.333333333333332,
      "median": 20
   },
   "temp": {
      "count": 3,
      "min": 20,
      "max": 60,
      "mean": 40,
      "median": 40
   }
}
```

For more information, refer to the Apache Math Descriptive Statistics Documentation.

Answer Question

The Answer Question activity uses a specified semantic index to answer a question derived from an inbound event. The result is an Object with the properties:

- *answer* (String) –the answer to the submitted question.
- *metadata* (Object) – the metadata associated with the entry(ies) used to provide context when answering the question. The semantic index can be configured to not include this.
- *rephrasedQuestion* (String) – the question rephrased by the LLM using the relevant context. If **rephraseQuestion** is set to `false` in the semantic index this will still contain the rephrased question, but it will not be the question asked to the LLM. This property is only included if the semantic index is configured to include it.

This value is returned in the specified property of the outbound event. The activity pattern uses the built-in SemanticSearch service.

The Answer Question activity has the following required configuration properties:

- **semanticIndex** – the name of the semantic index used to answer the question.
- **question** – the VAIL expression used to compute the question. The expression result must be a `String` value.

The use of an expression to produce the question provides a lot of flexibility. For example, while it could be used to obtain the question directly from the event contents, it can also be used to construct it dynamically. For example, consider a VEH that processes sensor temperature readings from machine. Each sensor reading looks like this:

```
{
    "sensorId": "sensor12345",
    "machineId": "machineXYZ",
    "temperature": 67.25
}
```

We could use the Template service to fold the event data into a question using a template like:

```
What corrective actions should be taken when the machine temperature is ${temperature}?
```

The Answer Question activity has the following optional configuration properties:

- **returnProperty** – the name of a property of the outbound event in which to store the response. Defaults to `indexResponse` if not specified.
- **qaLLM** – the name of the generative LLM to use to synthesize the answer. If none is provided, then the index's default Q&A LLM will be used (if set).
- **minSimilarity** – a number between 0 and 1 denoting the minimum similarity score that must be achieved for a document in the semantic index to be considered relevant. If not specified, the index's default minimum similarity will be used (if set). If neither is set, then answer question leverages the most relevant documents it finds in the index for context in answering the question regardless of the score. Should a minimum similarity specification result in no relevant documents, then answer question will return "I don't know" without consulting the qaLLM.
- **useConversation** – a Boolean indicating whether the task should participate in a conversation when asking the question. Use of this option requires an active collaboration. By default, the collaboration ID is used as the id of the conversation meaning that multiple AnswerQuestion tasks can participate in the same conversation (as long as they are operating on the same collaboration instance).
- **conversationContextPrompt** – This property can be used to help focus the LLM on particular context in the conversation to be referenced when rephrasing a follow-up question from the ongoing conversation. For example, if the conversation has details on a data payload and the follow-up question has new data points, the context prompt could focus the LLM on the differences between those payloads as they relate to the follow-up.
- **conversationName** – a String indicating the name of a conversation in which the task should participate. If not specified, then the collaboration's default conversation will be used. Use of this option allows two or more tasks to share conversation state within the collaboration instance that is separate from the default collaboration-wide conversation. Conversation names must be declared in the collaboration's properties. If an AnswerQuestion task requires its own separate conversation state, it can specify the conversation name `self`. This will create a conversation that is private to the task.
- **answerProperties** – a VAIL expression used to produce a "properties" `Object` which will be attached to the answer when it is recorded in a conversation. Ignored when `useConversation` is `false`.
- **runtimeConfig** - A VAIL expression that results in an Object representing an LLM configuration. When provided, this configuration is combined with the current LLM configuration and then applied to the SubmitPrompt invocation.

The Answer Question activity pattern produces the following event streams:

- **event** (default) – the results of all successful requests.
- **onError** – an event providing context for any failed request. The properties of this event are:
  - **errorCode** (String) – a unique "code" associated with the generated exception.
  - **errorMessage** (String) – the text of the error message for the generated exception.
  - **errorParams** (String[]) – the parameters used to replace substitution values in the error message.
  - **triggeringEvent** (Object) – the event that triggered the exception.

Assign

The Assign activity updates the collaboration by setting the specified collaborator or entity role to the incoming *event*. The Assign activity emits the original event. The Assignment can later be fetched by using the GetCollaboration task or the generated *Get* Procedure in the Service.

The Assign activity contains the following required configuration properties:

- **roleType** – either *entity*, *collaborator*, or *result*.
- **roleName** – the name of the entity or collaborator role (when using *entity* or *collaborator* roleType). When using the *result* roleType, this is the name of the result property being assigned. By default this is the name of the current task, but it can be changed to any legal property name. Multiple Assign tasks in the same app may refer to the same result property.

The Assign activity contains the following optional configuration properties:

- **entityId** – an optional VAIL expression used to determine the id of the entity being assigned. i.e: `event.machineId or event.farmId + event.barnId`.
- **assignedValue** – an optional VAIL expression used to determine the value being assigned. The expression must evaluate to an `Object`. Only used when assigning to the result role type. By default, the entire event will be used as the assigned value. The current value of the result property (if any) will be available via the predefined variable `currentValue`.
- **imports** – A list of resources referenced by the `assignedValue` expression.

For example, if the following event is processed by an Assign task configured with *entity* as the roleType and *machine* as the roleName:

```
{
    "temp": 88,
    "sensorId": "XYZ"
}
```

The collaboration instance is updated to the following:

```
{
    "id": "<collaboration id>",
    "status": "active",
    "name": "<app name>",
    "entities": {
      "machine": {
        "temp": 88,
        "sensorId": "XYZ"
      }
    }
}
```

If the same event is processed by an Assign task named `RecordTemp` configured with *result* as the **roleType**, `currentTemp` as the **roleName** and the expression `event.temp` as the **assignedValue** expression, then the collaboration instance is updated to the following:

```
{
    "id": "<collaboration id>",
    "status": "active",
    "name": "<app name>",
    "results": {
      "currentTemp": 88
    }
}
```

Branch

The Branch Activity Pattern is used to create a conditional branch in the flow of a service event handler. It contains the following configuration property:

- **conditions** – A list of expressions with labels that determines which path the event will take. Each condition can be any VAIL expression that evaluates to a boolean value. For example, `event.temperature > 100` or `event.status == "critical"`. The outbound event generated by the Branch task is determined by the **first** condition that evaluates to true. If none of the conditions evaluate to true, the event will be emitted on the `default` stream.

The Branch activity pattern has a dynamic number of outbound events. The set of events is determined by the conditions specified in the configuration. There is an event named after each of the branch condition labels. If labels are edited or removed, the associated downstream tasks will detach from the Branch task and require manual reattachment.

Build and Predict Path

The BuildAndPredictPath activity

- Constructs path information about objects tracked with `TrackMotion` activities,
- Tags Path members' locations with the region in which they are found if TrackingRegions are defined.
- Adds velocity information to the path steps if the `TrackingRegions` contain distance and/or direction information,
- If requested, predict the location of objects when they are dropped from tracking.

The BuildAndPredictPath activity is generally found *after* a Track Motion activity. BuildAndPredictPath constructs a path from the TrackMotion output. This path is assembled incrementally (*i.e.* over a number of calls) and can grow to a specified maximum size. In the event that the maximum size is exceeded, the latest entries are saved. That is, locations are removed from the oldest part of the path.

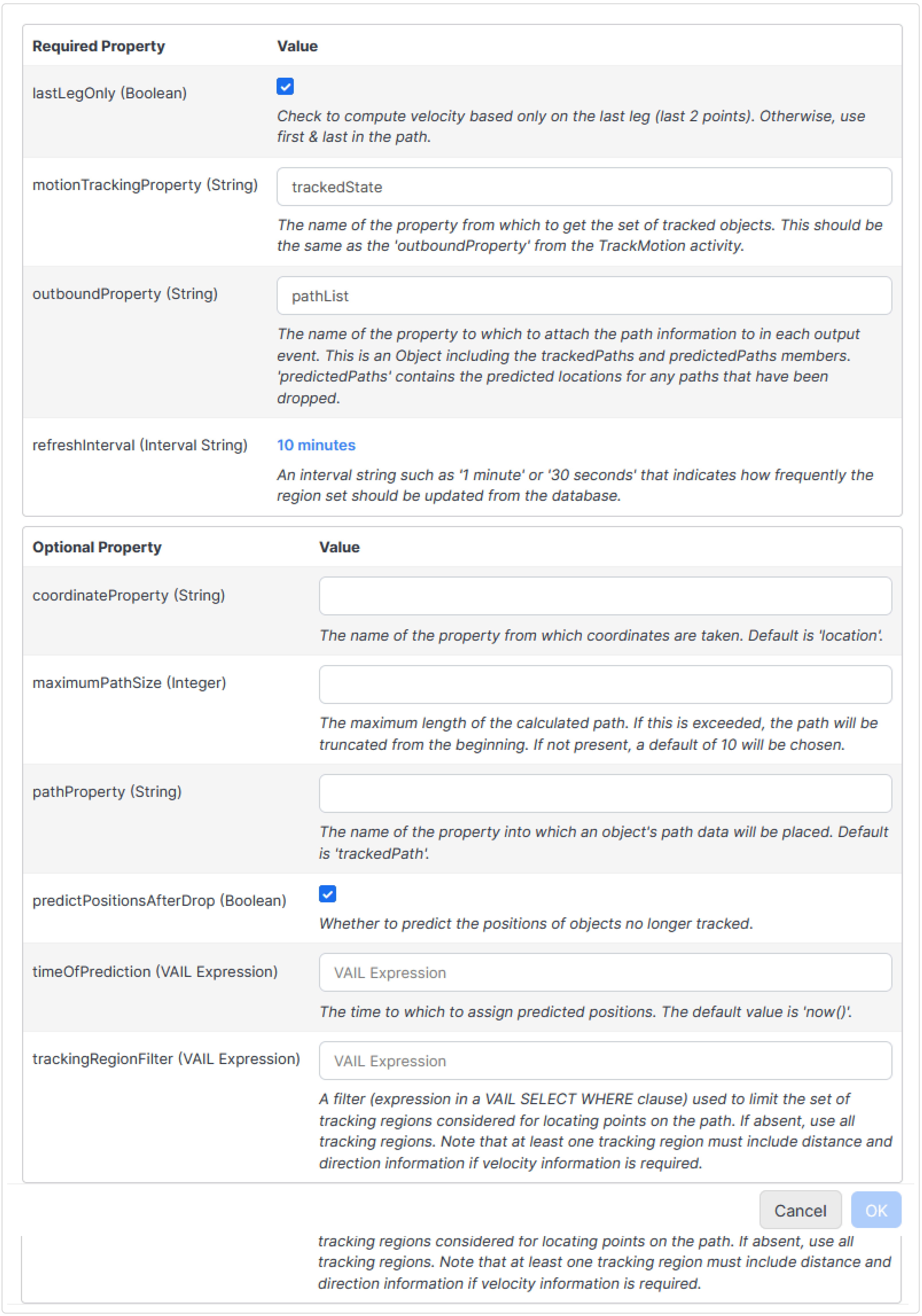
Once the activity is complete, emit the current list of paths in the property defined by the `outboundProperty` property.

The BuildAndPredictPath Activity Pattern requires the following configuration properties:

- **motionTrackingProperty** – the name of the property from which to gather the set of tracked objects. This is generally the same as the `outboundProperty` of the `TrackMotion` activity.
- **outboundProperty** – the name of the property to which to attach the list of paths.
- **refreshInterval** – an interval string (*e.g.* 10 minutes) specifying how often the list of TrackingRegions should be refreshed. In a mature application where region definitions do not change, a large interval may be appropriate.
- **lastLegOnly** – a boolean indicating that, when true, velocity calculations should be performed based on the last two items in the path. If false, the velocity is based on first and last items in the path.

Additionally, one can provide these optional propertys:

- **maximumPathSize** – the maximum length of the calculated path. If unspecified, the default is 10. Care should be taken here as if exceeded, the “first” object may change with each path addition. This will change the velocity calculations (if `lastLegOnly` is not checked).
- **coordinateProperty** – the name of the property from which to glean the coordinates from the tracked objects. The default value is `location`.
- **pathProperty** – the name of the property into which each object’s tracked path will be placed. The default is `trackedPath`.
- **predictPositionsAfterDrop** – whether to add a predicted position to a path when it is dropped.
- **timeOfPrediction** – if *predictPositionsAfterDrop* is true, the time to give for the predicted position. The default value is `now()`.
- **trackingRegionFilter** – an expression that limits the set of `TrackingRegions` considered for the position’s tracking region. This expression will be added to the WHERE clause used to select the regions to consider. If absent or empty, all `TrackingRegions` in the namespace are considered. Please see the tracking regions information for more information.



Required Property	Value
lastLegOnly (Boolean)	✓ *Check to compute velocity based only on the last leg (last 2 points). Otherwise, use first & last in the path.*
motionTrackingProperty (String)	trackedState *The name of the property from which to get the set of tracked objects. This should be the same as the 'outboundProperty' from the TrackMotion activity.*
outboundProperty (String)	pathList *The name of the property to which to attach the path information to in each output event. This is an Object including the trackedPaths and predictedPaths members. 'predictedPaths' contains the predicted locations for any paths that have been dropped.*
refreshInterval (Interval String)	**10 minutes** *An interval string such as '1 minute' or '30 seconds' that indicates how frequently the region set should be updated from the database.*

Optional Property	Value
coordinateProperty (String)	*The name of the property from which coordinates are taken. Default is 'location'.*
maximumPathSize (Integer)	*The maximum length of the calculated path. If this is exceeded, the path will be truncated from the beginning. If not present, a default of 10 will be chosen.*
pathProperty (String)	*The name of the property into which an object's path data will be placed. Default is 'trackedPath'.*
predictPositionsAfterDrop (Boolean)	✓ *Whether to predict the positions of objects no longer tracked.*
timeOfPrediction (VAIL Expression)	VAIL Expression *The time to which to assign predicted positions. The default value is 'now()'.*
trackingRegionFilter (VAIL Expression)	VAIL Expression *A filter (expression in a VAIL SELECT WHERE clause) used to limit the set of tracking regions considered for locating points on the path. If absent, use all tracking regions. Note that at least one tracking region must include distance and direction information if velocity information is required.*

tracking regions considered for locating points on the path. If absent, use all tracking regions. Note that at least one tracking region must include distance and direction information if velocity information is required.

In the example above, we'll use only the last leg for velocity calculations for the motion of objects found in the `trackedState` property, producing the results in the `pathList` property. The task will update our list of regions every 10 minutes.

The BuildAndPredictPath activity produces an object containing

- `trackedPaths` – a set of objects with paths, and
- `droppedPaths` – a set of objects dropped in this pass.

Objects are dropped when no information about that object is presented. This is normally the case when the Track Motion task's `maximumAbsentBeforeMissing` threshold has been met.

The property defined by `outboundProperty` contains this object, with each object within `trackedPaths` containing a set of path elements (named by the `pathProperty`, defaulting to `trackedPath`). Each path element contains location information (bounding box, center, *etc.*), and the `trackingId` and `timeOfObservation` (see Track Motion) properties.

If present, the `droppedPaths` property contains the same information. However, if the `predictPositionsAfterDrop` was set to `true`, then there will be a final predicted position. This predicted position will based on the last two observed points for the object and the observation time between them. Using the distance traveled and the time between those observations, the predicted position will be calculated, based on the `timeOfPrediction` presented. Predicted positions will have an additional property `isPredicted` set to `true`.

If `TrackingRegions` are defined in the namespace, the `regions` property will be present. The `regions` property is a list of the regions in which this step's center point is found. Regions are not mutually exclusive, nor must they cover all possible locations, so the `regions` property can be a set of regions or empty.

If the `trackingRegionFilter` is provided, the set of regions considered will be limited. For example, if `trackingRegionFilter` contains `name in ["mainStreet", "bikePath"]`, then the only regions considered for each position will be the `TrackingRegions` with names *mainStreet* and *bikePath*.

When using the `trackingRegionFilter` value `name in [name in ["mainStreet", "bikePath"]`, that expression is added to a SELECT query on the `TrackingRegions` in the namespace. If no `TrackingRegions` qualify, then the set of `TrackingRegions` considered will be empty.

Path elements will contain velocity information if any of the `TrackingRegions` selected contain `distance` or `direction` definitions. The `velocity` property is an object containing `speed` and `direction` properties. Either, neither, or both may be present depending upon the `TrackingRegions` definitions and selection (based on the expression present in the `trackingRegionFilter` property). Please see the Tracking Regions section of the Resource Guide and the Motion Tracking section of the Image Processing Guide for more details.

The following is an example of output from a `BuildAndPredictPath` activity. Here, we are observing a traffic intersection. A set of `TrackingRegions` is defined, including one region for an intersection, one for the main street, and a bike path. (Note – the example is long. To reduce the length, we've remove repeated common structures. These are shown as *…position info….*)

In this example, this is the output after the third image is examined. You will see that a car has been tracked for all three images, and a truck was visible in only the first and third images (based on the `timeOfObservation` values).

```
{
...
    "pathList":
        "trackedPaths":
            [
                {
                    "confidence": 0.9738035,
                    "location": {
                        "centerY": 698.32983,
                        "top": 613.84503,
                        "left": 248.22499,
                        "centerX": 449.8205,
                        "bottom": 782.8147,
                        "right": 651.41595,
                        "trackingId": "2c2a450a-d011-438b-bf59-505ab069ff36",
                        "timeOfObservation": "2020-07-09T19:25:50.951Z",
                        "regions": [],
                        "velocity": {}
                    },
                    "label": "car",
                    "trackedState": [
                        {
                            "centerY": 698.32983,
                            "top": 613.84503,
                            "left": 248.22499,
                            "centerX": 449.8205,
                            "bottom": 782.8147,
                            "right": 651.41595,
                            "trackingId": "2c2a450a-d011-438b-bf59-505ab069ff36",
                            "timeOfObservation": "2020-07-09T19:25:50.951Z",
                            "regions": [],
                            "velocity": {}
                        },
                        {
                            ...position info...
                            "trackingId": "2c2a450a-d011-438b-bf59-505ab069ff36",
                            "delta": 227.29758889770193,
                            "timeOfObservation": "2020-07-09T19:26:20.714Z",
                            "regions": ["intersection"],
                            "velocity": {
                                "speed": 0.6386117789508797,
                                "direction": 178.31846108202308
                            }
                        },
                        {
                            ...position info...
                            "trackingId": "2c2a450a-d011-438b-bf59-505ab069ff36",
                            "delta": 573.5204100376408,
                            "timeOfObservation": "2020-07-09T19:26:22.558Z",
                            "regions": ["mainStreet"],
                            "velocity": {
                                "speed": 26.00798831086019,
                                "direction": 190.53958118873743
                            }
                        }
                    ]
                },
                {
                    "confidence": 0.56739765,
                    "location": {
                        ...position info...
                        "trackingId": "77557464-56ac-42a2-a550-9653a5a360bc",
                        "timeOfObservation": "2020-07-09T19:25:50.951Z",
                        "regions": [],
                        "velocity": {}
                    },
                    "label": "truck",
                    "trackedState": [
                        {
                            ...position info...
                            "trackingId": "77557464-56ac-42a2-a550-9653a5a360bc",
                            "timeOfObservation": "2020-07-09T19:25:50.951Z",
                            "regions": [],
                            "velocity": {}
                        },
                        {
                            ...position info...
                           "trackingId": "77557464-56ac-42a2-a550-9653a5a360bc",
                            "delta": 166.20815098458465,
                            "timeOfObservation": "2020-07-09T19:26:22.558Z",
                            "regions": ["intersection"],
                            "velocity": {
                                "speed": 0.43973176788635465,
                                "direction": 38.12289240559642
                            }
                        }
                    ]
                }
            ],
        "droppedPaths":
            [
                {
                    "confidence": 0.56739765,
                    "location": {
                        ...position info...
                        "trackingId": "77557464-56ac-42a2-a550-9653a5a376ab",
                        "timeOfObservation": "2020-07-09T19:21:02.974Z",
                        "regions": [],
                        "velocity": {}
                    },
                    "label": "van",
                    "trackedState": [
                        {
                            ...position info...
                            "trackingId": "77557464-56ac-42a2-a550-9653a5a376ab",
                            "timeOfObservation": "2020-07-09T19:21:02.974Z",
                            "regions": [],
                            "velocity": {}
                        },
                        {
                            ...position info...
                           "trackingId": "77557464-56ac-42a2-a550-9653a5a376ab",
                            "delta": 100.000,
                            "timeOfObservation": "2020-07-09T19:21:22.558Z",
                            "regions": ["intersection"],
                            "velocity": {
                                "speed": 0.53978476734534675,
                                "direction": 46.356892976559642
                            }
                        },
                        {
                            ...position info...
                           "trackingId": "77557464-56ac-42a2-a550-9653a5a376ab",
                            "delta": 500.000,
                            "isPredicted": true,
                            "timeOfObservation": "2020-07-09T19:26:22.558Z",
                            "regions": ["intersection"],
                            "velocity": {
                                "speed": 0.43973176788635465,
                                "direction": 38.12289240559642
                            }
                        }
                    ]
                }
            ]
...
}
```

Velocity information is interpreted as follows:

- speed – Units/second where *units* is the same unit used to specify the `TrackingRegions` `distance` property. The time (in number of seconds) is based on the `timeOfObservation` entries for the points in question. This value is provided via the `TrackMotion` activity.
- direction – Compass direction, 0 is north. This is based on the specification of the `TrackingRegions` `direction` property.

It is possible to have a partial velocity. If there is only a `TrackingRegions` `distance` property, the velocity will only contain the `speed` property. Similarly, if the the `TrackingRegions` contains `direction` but no `distance`, only the `direction` property will be present.

Cached Enrich

Associate the inbound event with data from a persistent type and cache the association for reuse to avoid extra database queries. The cached association is refreshed from the database after a configurable interval. CachedEnrich is useful for situations where there is a high volume of events, but the associated data changes relatively infrequently so a separate database lookup is not required. This will typically yield better performance than the normal Enrich activity pattern.

For example, consider an event handler that processes sensor temperature readings from machine. Each sensor reading looks like this:

```
{
    "sensorId": "sensor12345",
    "machineId": "machineXYZ",
    "temperature": 67.25
}
```

That machineId value may specify an associated record in the Machine type, which has information about the acceptable temperatures for that specific machine:

```
{
    "machineId": "machineXYZ",
    "maxTemp": 90,
    "minTemp": 60
}
```

Use CachedEnrich to find the associated Machine record with a matching machineId property. The outbound event would look like:

```
{
    "sensorId": "sensor12345",
    "machineId": "machineXYZ",
    "temperature": 67.25,
    "Machine": {
           "machineId": "machineXYZ",
           "maxTemp": 90,
           "minTemp": 60
   }
}
```

The Machine record will be looked up once per specified refreshInterval, and the result will be cached and used for all subsequent events until the refreshInterval passes, at which point the cache will be updated by querying the Machine type again.

Since CachedEnrich only stores a single record at a time, it is best to use it after a SplitByGroup. For each group created by a split, CachedEnrich will separately query the database for the associated Machine record and will be refreshed independently.

The CachedEnrich Activity Pattern requires three configuration properties:

- **associatedType** – The type in which the associated data is defined. In the above example the associatedType would be Machine.
- **foreignKeys** – An array of property names that exist in both the inbound event and the associatedType that can be used to uniquely identify an instance of the associatedType. In the above example there would be only one foreign key: “machineId”.
- **refreshInterval** – An interval string such as `1 minute` or `2 hours` that expresses how often the associated Machine record should be fetched and cached from the database.

The CachedEnrich pattern does not create the state “update” procedure. Instead it creates the procedure `<taskIdentifier>StateGetForEvent` which does the work necessary to attach the cached data with the incoming event. The signature of this procedure corresponds to the standard “update” procedure.

Chat

The Chat activity pattern is used to start a new chatroom in the Vantiq Mobile App.

The Chat activity contains the following required configuration properties:

- **users** – an array of usernames that will be added to the chatroom. This may either be an actual array of vail expressions or a single vail expression which will evaluate to an array (for example: *event.userList*).

The Chat activity also contains the following optional configuration properties:

- **chatbotSourceName** – The name of the Chatbot source that should be used to listen for new messages in the chatroom. This is not required if there is only one CHATBOT source in the namespace or if no messageBehaviors are defined.
- **imports** – A list of resources referenced by the activity pattern.

The Chat activity produces the following outbound events:

- **message** (default) – The event received when a user sends a message to the Chatroom.
  - *type*: The type of chat event produced. This should always be ‘message’.
  - *id*: The id for the message generated by the Microsoft Azure Chatbot.
  - *timestamp*: The timestamp for the message.
  - *serviceUrl*: The Microsoft Azure Bot Services URL for the service that acts as an intermediary between the chatroom and Vantiq.
  - *channelId*: A String representing the service on which the chat message was sent. For messages in the Vantiq Mobile App this should always be ‘directline’.
  - *conversation*: An object containing a unique *id* the Microsoft Azure Bot Service has assigned to the chatroom.
  - *from*: An object containing the *name* of the user who sent this message in the chatroom.
  - *recipient*: An object containing the *name* and *id* of the bot that detected the message in the Microsoft Azure Bot Services.
  - *text*: The text of the actual message sent in the chatroom.
  - *channelData*: An object containing a unique *id* for the chatroom assigned by the Microsoft Azure Bot Services and the *name* of the chatroom.
  - *authorization*: A String containing a bearer token used to authorize the chatroom message with the Microsoft Azure Bot Services.
- **event** – Emits the original incoming event.

For example, if the a user in the Chatroom asked: “What time is it?” the following *message* event would occur:

```
{
  "type": "message",
  "id": "<id>",
  "timestamp": "2022-05-31T23:58:48.8469026Z",
  "serviceUrl": "https://directline.botframework.com/",
  "channelId": "directline",
  "from": {
    "id": "<Vantiq username>",
    "name": "<Vantiq username>"
  },
  "conversation": {
    "id": "<chat room id>"
  },
  "recipient": {
    "id": "<bot id>",
    "name": "<Bot name>"
  },
  "text": "What time is it?",
  "channelData": {
    "id": "<unique channel id>",
    "name": "<app name>_<task name>: <collaboration id>"
  },
  "authorization": "Bearer <token>"
}
```

The collaboration instance is updated to the following:

```
{
  "id": "<collaboration id>",
  "status": "active",
  "name": "<app name>",
  "results": {
    "<chat task name>": {
      "chatId": "<chatroom id>"
    }
  }
}
```

The Chatroom id may be used to send messages or delete the chatroom using the Chat Service Procedures (Chat.sendMessage and Chat.deleteChatroom).

Close Collaboration

The CloseCollaboration activity pattern is used to update the status of a collaboration when it has finished.

The CloseCollaboration activity pattern has one required configuration property:

- **status**: The status to set on the collaboration instance. This must be either *completed* or *failed*.

The task is always a leaf task within a VEH as it can have no children.

Compute Statistics

The ComputeStatistics Activity Pattern computes the min, max, mean, median, standard deviation and count for a single property of events that pass through the task. There are 4 types of reservoirs that can be used to contain the events used to compute the aggregate statistics (all based on HDR Histogram):

- **Exponentially Weighted** – computes statistics using `Integer` values. More recent events contribute more to the aggregate than older events.
- **Sliding Time Window** – computes statistics using `Integer` values. Events stop being used in the aggregate statistics after a configurable interval of time.
- **Double Histogram** – computes statistics using `Real` values. All events are equally weighted in the histogram.
- **Double Histogram With Reset** – computes statistics using `Real` values. Events are only counted within the window before the aggregate stats are emitted. After the statistics are emitted for the configured outputFrequency the histogram is reset.

The ComputeStatistics Activity Pattern has two required configuration properties:

- **inboundProperties** – The properties for which stats will be computed. Each property must be a numeric property.
- **reservoirType** – Choose between different implementations that determine how and for how long events are counted towards aggregate statistics.

It also has two optional properties:

- **windowLength** – Used to specify how long events should count towards aggregate statistics. Only applies to Sliding Time Window reservoirs.
- **resetFrequency** – How often should the histogram be reset (as an interval string). Only applies to 'Double Histogram With Reset'.

Compute Statistics produces the following event streams:

- event (default) – Emits the original incoming event.

- reset – If a *resetFrequency* is specified, the task will automatically emit a *reset* event with the value of the state just before the state is automatically reset.

The ComputeStatistics pattern has no impact on the structure of the events that pass through it.

The ComputeStatistics activity pattern generates three Procedures: Get, Update, and Reset. The Get procedure can be used to retrieve the current result of the analytics. An example return value is shown below:

```
{
   "mean": 250.87244897959184,
   "min": 200,
   "max": 313.125,
   "median": 244,
   "stdDeviation": 36.683330907259204
}
```

Convert Coordinates

Convert the coordinates resulting from YOLO image analysis according to the calibration information provided. Such an analysis may be the result of a YOLO From Images or YOLO From Documents activity. A YOLO neural net model reports on objects found in an image. For each object found, it reports, among other things, the label for the object and the coordinates within the image. The ConvertCoordinates Activity Pattern augments the images results with coordinates in an different coordinate system. This might be used if an image is scaled (made smaller or larger), or to convert the image coordinates into a completely different coordinate space (*e.g.* GPS).

The result of a ConvertCoordinates activity is a copy of the input event where each object's `location` is replaced with the converted coordinates (as specified by the `calibrationKey`).

The ConvertCoordinates activity has the following optional configuration properties:

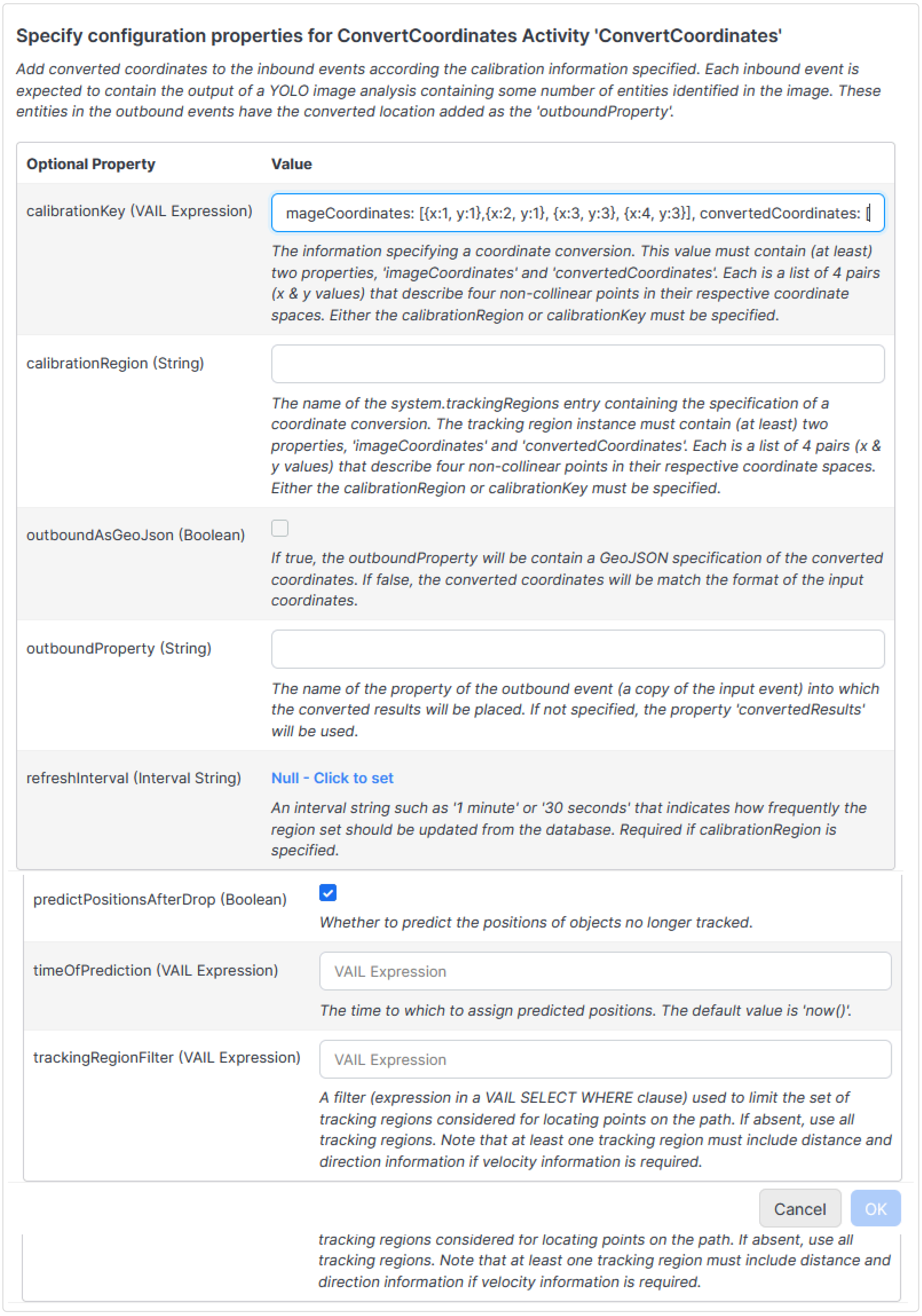
- **calibrationKey** – This is a VAIL expression that contains two (2) properties: `imageCoordinates` and `convertedCoordinates`. Each of these is a list of four (4) pairs (x & y values) that describe 4 non-collinear points in their respective coordinate spaces. That is, they describe a quadrilateral where each point in the `imageCoordinates` list corresponds to the same listed point in the `convertedCoordinates` space. It is important that no three (3) of the points are collinear. When the points are collinear, the conversion can produce unexpected results. The system will attempt to check for collinearity, but developers should be aware of this restriction.
- **calibrationRegion** – The name of the system.trackingRegions entry containing the specification of a coordinate conversion. The tracking region instance must contain (at least) two properties, 'imageCoordinates' and 'convertedCoordinates'. Each is a list of 4 pairs (x & y values) that describe four non-collinear points in their respective coordinate spaces. Either the calibrationRegion or calibrationKey must be specified.
- **outboundAsGeoJson** – a boolean indicating that the converted coordinates should be expressed as GeoJSON. This is most appropriate (though not restricted to) cases where the converted coordinates are intended to represent GPS coordinates. The default value is `false`.
- **outboundProperty** – The name of the property into which to place the converted coordinates. The default value is `convertedResults`.
- **refreshInterval** – An interval string such as '1 minute' or '30 seconds' that indicates how frequently the region set should be updated from the database. Required if calibrationRegion is specified.

As an example, consider a Convert Coordinates activity with an identity `calibrationKey`. That is, a calibration key where the input & output coordinates are the same. Such a key would be specified as a VAIL expression:

```
{
      imageCoordinates: [ {x:1, y:1}, {x:2, y:1}, {x:3, y:3}, {x:4, y:3} ],
  convertedCoordinates: [ {x:1, y:1}, {x:2, y:1}, {x:3, y:3}, {x:4, y:3} ]
}
```

Here, you can see that the `imageCoordinates` and `convertedCoordinates` are the same and neither is collinear.

In the App Builder, this would appear as follows:



Specify configuration properties for ConvertCoordinates Activity 'ConvertCoordinates'

Add converted coordinates to the inbound events according the calibration information specified. Each inbound event is expected to contain the output of a YOLO image analysis containing some number of entities identified in the image. These entities in the outbound events have the converted location added as the 'outboundProperty'.

Optional Property	Value
calibrationKey (VAIL Expression)	mageCoordinates: [{x:1, y:1},{x:2, y:1}, {x:3, y:3}, {x:4, y:3}], convertedCoordinates: [*The information specifying a coordinate conversion. This value must contain (at least) two properties, 'imageCoordinates' and 'convertedCoordinates'. Each is a list of 4 pairs (x & y values) that describe four non-collinear points in their respective coordinate spaces. Either the calibrationRegion or calibrationKey must be specified.*
calibrationRegion (String)	*The name of the system.trackingRegions entry containing the specification of a coordinate conversion. The tracking region instance must contain (at least) two properties, 'imageCoordinates' and 'convertedCoordinates'. Each is a list of 4 pairs (x & y values) that describe four non-collinear points in their respective coordinate spaces. Either the calibrationRegion or calibrationKey must be specified.*
outboundAsGeoJson (Boolean)	☐ *If true, the outboundProperty will be contain a GeoJSON specification of the converted coordinates. If false, the converted coordinates will be match the format of the input coordinates.*
outboundProperty (String)	*The name of the property of the outbound event (a copy of the input event) into which the converted results will be placed. If not specified, the property 'convertedResults' will be used.*
refreshInterval (Interval String)	Null - Click to set *An interval string such as '1 minute' or '30 seconds' that indicates how frequently the region set should be updated from the database. Required if calibrationRegion is specified.*
predictPositionsAfterDrop (Boolean)	☑ *Whether to predict the positions of objects no longer tracked.*
timeOfPrediction (VAIL Expression)	VAIL Expression *The time to which to assign predicted positions. The default value is 'now()'.*
trackingRegionFilter (VAIL Expression)	VAIL Expression *A filter (expression in a VAIL SELECT WHERE clause) used to limit the set of tracking regions considered for locating points on the path. If absent, use all tracking regions. Note that at least one tracking region must include distance and direction information if velocity information is required.*

tracking regions considered for locating points on the path. If absent, use all tracking regions. Note that at least one tracking region must include distance and direction information if velocity information is required.

If we feed the YOLO output from an image containing a car to this activity, we will get the result shown below.

```
{
    "target": {"name": "car.jpg", "fileType": "image/jpeg", ... },
    "results": [{
        "confidence": 0.9961914,
        "location"  : {
                  "centerY": 324.67153930664062,
                  "top"   : 153.53375244140625,
                  "left"  : 71.24849700927734375,
                  "centerX": 497.45350265502929375,
                  "bottom": 495.809326171875,
                  "right" : 923.65850830078125
              },
          "label"     : "car"]
         }],
    "convertedResults": [{
        "confidence": 0.9961914,
        "location"  : {
                  "centerY": 324.67153930664062,
                  "top"   : 153.53375244140625,
                  "left"  : 71.24849700927734375,
                  "centerX": 497.45350265502929375,
                  "bottom": 495.809326171875,
                  "right" : 923.65850830078125
              },
          "label"     : "car"]
         }]
}
```

If we change that activity so that we specify the `outputAsGeoJSON` as true and the `outboundProperty` as "geoJSONResults", our conversion results in the following event:

```
{
    "target": {"name": "car.jpg", "fileType": "image/jpeg", ... },
    "results": [{
        "confidence": 0.9961914,
        "location"  : {
                  "centerY": 324.67153930664062,
                  "top"   : 153.53375244140625,
                  "left"  : 71.24849700927734375,
                  "centerX": 497.45350265502929375,
                  "bottom": 495.809326171875,
                  "right" : 923.65850830078125
              },
          "label": "car"]
         }],
    "geoJSONResults": [{
        "confidence": 0.9961914,
        "location"  : {
           "bottomRight": {
              "coordinates": [495.809326171875, 923.65850830078125],
                    "type": "Point"},
           "topLeft": {
              "coordinates": [153.53375244140625, 71.24849700927734375],
                    "type": "Point"},
            "center": {
              "coordinates": [324.67153930664062, 497.45350265502929375],
                    "type": "Point"}
        },
        "label": "car"
    }]
}
```

DBScan

The DBScan activity uses a DBScan (density based clustering) algorithm to cluster the data using the specified distance measurement.

The DBScan activity contains the following required configuration properties:

- **XYProperties** – The property names (e.g. "speed", "temperature", etc.) on the inbound events to compute clusters for. Users may specify either X and Y properties for 2 dimensional analysis or just one property for one dimensional analysis. Each property must be numeric.
- **reservoirType** – The type of reservoir used to store data points. The type will determine which algorithm is used for computing aggregate statistics and when to purge old data points.
    - *Sliding Time Window* – Computes statistics using `Integer` values. All events that occured within the specified time window will be used to calculate statistics (e.g. all events in the last 5 minutes)
    - *Sliding Count* – Computes statistics using `Integer` values. The last *n* events will be used to calculate the statistics
    - *Fixed Time Window* – Computes statistics using `Real` values. A fixed time window will add events to the window until the time of the next reset. For example, a fixed time window of 1 hour will calculate analytics for all events within the hour, starting at the top of the hour and ending at the top of the new hour.

    - *Fixed Count* – Computes statistics using `Real` values. Calculate analytics on all of the events until the specified number of events has been reached. Then the analytics will be reset back to null.
- **DistanceMeasure** – The distance measuring algorithm to use. This may be: *Euclidean Distance, Manhattan Distance, Canberra Distance Earth Mover's distance, Chebyshev Distance*. For more information, refer to the Apache Math Distance Measure Documentation.
- **minPoints** – The minimum number of points needed for a cluster
- **maxRadius** – The maximum radius of the neighborhood to be considered
- **windowLength** – How long events should stay in the window. For time based reservoirs this is an interval string (e.g.: "10 minutes"). For count based reservoirs this should be the number of events.

DBScan activity pattern produces the following event streams:

- **event** (default) – Emits the original incoming event.
- **reset** – If the reservoirType is set to *Fixed Time Window* or *Fixed Count*, the Event Handler will automatically emit a *reset* event with the value of the analytics just before the state is automatically reset to null.

The DBScan activity pattern generates three Procedures: Get, Update, and Reset. The Get procedure can be used to retrieve the current result of the analytics. An example return value is shown below:

```
{
  "speed": [
    [
      [0, 50],
      [1, 51],
      [2, 52]
    ],
    [
      [0, 20],
      [1, 21],
      [2, 22]
    ]
  ]
}
```

For more information, refer to the Apache Math DBScan Documentation.

Delay

Delay the emission of an inbound event for some duration of time. The delay interval is expressed as a VAIL expression that evaluates to the number of milliseconds to delay the event. For example, if the delay configuration is set to `event.delayLength` and the inbound event looks like:

```
{
    "name": "machine1",
    "temperature": 55.1,
    "delayLength": 5000
}
```

the event will only be sent to the downstream task after 5 seconds have passed. It's also possible to set the delay configuration property to an interval string that is constant for all events, like `5 seconds` or `2 hours`.

The Delay Activity Pattern has only one required property:

- **delay** – An expression evaluating to the number of milliseconds to delay the event.

The Delay Activity Pattern also supports one optional property:

- **imports** – A list of resources referenced by the *delay* expression that must be imported.

Dwell

Dwell can be used to identify a state that hasn't changed for a duration of time across sequential events. For instance, a stream of events might contain temperature readings from a sensor on a machine, and it's dangerous for the machine to have a temperature greater than 100 degrees for more than a minute. If the temperature jumps over 100 degrees temporarily, that's not necessarily dangerous. This is an ideal use case for Dwell, which can be configured to only emit an event when the condition `event.temperature > 100` is true for 60 seconds.

Once a Dwell task has emitted an event because the condition was true over the specified duration, no more events are emitted as long as the condition holds true. If the condition evaluates to false, this resets the dwell detection. Then when the condition evaluates to true again for the specified duration, another event is emitted. The emitted event is the last event that the Dwell task observed matching the condition.

The Dwell Activity Pattern contains the following configuration properties:

- **condition** – two options:
- *VAIL Conditional Expression*: A VAIL boolean expression that expresses the state which must be maintained to trigger the dwell task. The condition can reference any property of the inbound event like `event.propX > event.propY AND event.id == "XYZ"`.
- *Visual Filter*: A visual interface that allows users to create a condition using a series of dropdowns and text fields. The visual filter is a more user-friendly way to create a condition.

- **duration** – How long the condition must hold true across sequential events to trigger output from the dwell task.

The Dwell Activity Pattern contains one optional configuration property:

- **imports** – A list of resources referenced by the *condition* that must be imported.

The Dwell activity pattern produces the following event streams:

- **event** (default) – the event on which the "dwell" condition is detected.
- **reset** – the event that triggers a 'reset' of dwell detection.

Establish Collaboration

The Establish Collaboration activity pattern allows an event handler to run in the context of an existing active collaboration for the same entity.

The Establish Collaboration activity contains the following required configuration properties:

- **entityId** – The vail expression describing the unique identifier for the entity (e.g. `event.machineId` or `event.farmId + event.barnId`).
- **roleName** – The name of the entity role
- **behavior** – The behavior for the task if it finds a matching collaboration for the entity.
  - *Do not open multiple collaborations for one entity* – If an active collaboration for the matching entity exists, EstablishCollaboration will act as a Filter and the event will not flow to any downstream tasks
  - *Establish existing collaboration* – Find a matching collaboration for the entityId if one exists and execute any downstream tasks in the context for that collaboration.
  - *Establish existing collaboration or create new collaboration*: Behaves the same as "Establish existing collaboration", however if a matching collaboration is not found, a new one will be opened.

Event Stream

The root tasks in any VEH are Event Streams. Event Streams identify a resource on which events occur, and an optional constraint that restricts which events to include in the stream. The currently supported resources are *types*, *sources*, and *topics*, so any insert on a type, message arriving on a source, or publish to a topic can be the root of a VEH. VEH can have multiple event streams which identify completely different pieces of data, and these can be joined downstream to detect more complex situations in parallel streams.

The Event Stream Activity Pattern contains the following configuration properties:

- **inboundResource** – One of *types*, *sources*, or *topics*. This describes the resource on which raw events that enter the Event Stream occur.
- **inboundResourceId** – Specifies which type, source or topic the events occur on for this Event Stream. For instance, if your App needs to process all messages coming in from the Kafka source named MyKafkaSource, the inboundResource would be *sources*, and the inboundResourceId would be MyKafkaSource.
- **op** – One of INSERT, UPDATE, or BOTH. Only applies if inboundResource is *types*, in which case the op restricts whether the event stream should listen for only inserts or updates on the type, or if both should be processed.
- **schema** – The name of a type that defines the schema of the inbound events. Specifying the schema here will simplify the configuration of downstream tasks.

Enrich

The Enrich task is useful for attaching persistent data to a stream of events. For instance, an event stream may produce temperature readings for a motor that look like:

```
{
    "engineId": "XYZ",
    "temp": 165
}
```

And there could be an existing Engine Type that contains data associated with each engine that looks like:

```
{
    "engineId": "XYZ",
    "manufacturer": "Ford",
    "modelYear": 2010,
    "optimalTemp": 155,
    ...
}
```

The enrich task would automatically join the Engine data to the event to produce outbound events that look like:

```
{
    "engineId": "XYZ",
    "temp": 165,
    "Engine": {
        "engineId": "XYZ",
        "manufacturer": "Ford",
        "modelYear": 2010,
        "optimalTemp": 155,
        ...
    }
}
```

The Enrich Activity Pattern requires two configuration properties:

- **associatedType** – The type in which the associated data is defined. In the above example the associatedType would be Engine.
- **foreignKeys** – An array of property names that exist in both the inbound event and the associatedType that can be used to uniquely identify an instance of the associatedType. In the above example there would be only one foreign key: “engineId”.

Escalate

Escalations trigger an event after a certain amount of time if they are not cancelled first. Escalations are a useful way to verify something happens within a set amount of time and triggering some additional behaviors if the thing doesn’t happen. Escalations can be cancelled or triggered early using the EscalateState activity pattern. Escalate contains the following configuration properties:

- **escalationTime** - The amount of time, in milliseconds, before the escalation event is triggered. May be an integer, a duration string, or a VAIL expression.
- **outboundBehavior** - Optional. What the output event will look like. Chosen from a list of three options:
  - *Replace outbound event with Escalate results* - Send the default event for Escalate tasks.
    This is the default behavior if no option is selected.
  - *Emit original event* - Send the event that began the escalation. Note that this option will store the entire event in the collaboration, so if the events are large it may cause memory or database pressure.
  - *Attach results to outboundProperty* - Add the normal Escalate event to a property in the original event, then send that. Note that this option will store the entire event in the collaboration, so if the events are large it may cause memory or database pressure.
- **outboundProperty** - Optional. The property to attach the results to if **outboundBehavior** is “Attach results to outboundProperty”. Does nothing for other settings of **outboundBehavior**.

When an Escalate Activity Type is referenced as a behavior associated with another activity, the only required parameter is the collaboration instance.

Escalate activities produce the following event streams:

- **escalate** - This event is generated if the escalation has not been cancelled within escalationTime of the escalation beginning. This event will only occur once, and will not occur if the escalation is cancelled.
- **escalateCancelled** - This event is generated if the escalation is cancelled. This event will only occur once, and will not occur if the escalation has already occurred.

Escalate activities produce the following results in the collaboration:

- **state** - The state of the escalation. Can be one of “waiting”, “cancelled” or “expired”.
- **updateTime** - The time that the escalation was created, or the time that it was updated to be in the “cancelled” or “expired” state, whichever happened most recently.
- **event** - The event that started the escalation. Only present if **outboundBehavior** is “Emit original event” or “Attach results to outboundProperty”.

Escalate activities add or produce the following event when **outboundBehavior** is *Attach results to outboundProperty* or *Replace outbound event with Escalate results*, respectively.

- **collaborationType** - The name of the collaborationType that the task is in.
- **collaborationId** - The id of the collaboration that the escalation is a part of.
- **activityType** - The name of the Escalate task.
- **escalationDeadline** - The time at which the escalation should expire.
- **escalationUpdateTime** - The time that the escalation was created, or the time that it was updated to be in the “cancelled” or “expired” state, whichever happened most recently.
- **state** - The state of the escalation. Can be one of “waiting”, “cancelled” or “expired”.

EscalateState

The EscalateState task exists to manually either trigger or cancel an Escalate task. It will throw an error if the escalation has already expired or has been updated by another EscalateState task.

- **escalateTask** - The name of the task to be triggered or cancelled.
- **state** - The state to set for the task. Chosen from a list of two options:
  - *cancelled* - Cancels the escalation, triggering the original task’s “escalateCancelled” event.
  - *expired* - Triggers the escalation immediately as if the escalation’s deadline had expired.

Filter

A simple task that restricts which events flow through to downstream tasks. The condition is specified as a VAIL conditional expression and can operate on any of the properties in the inbound event.

The Filter Activity Pattern contains only one required configuration property:

- **condition** – two options:
  - *VAIL Conditional Expression*: The VAIL expression that restricts which events flow through the filter to downstream tasks. Ex. `event.netWeight > 1000`
  - *Visual Filter*: A visual interface that allows users to create a condition using a series of dropdowns and text fields. The visual filter is a more user-friendly way to create a condition.

In either case, only events which satisfy the condition will continue onward.

The Filter Activity Pattern contains one optional configuration property:

- **imports** – A list of resources referenced by the *condition* that must be imported.

Filter produces the following event streams:

- **event** (default) – The event emitted when the event passes the condition.
- **rejected** – The event emitted when the event fails the condition. This event cannot be used when the Filter is a child of a LoopWhile task's **whileTrue** event. Tasks that use this path will lose reliability if the original event was reliable.

GenAI Flow

The GenAI Flow activity pattern is used to add custom GenAI functionality to an event handler via an embedded GenAI Flow. The purpose of this activity pattern is to complement Submit Prompt and Answer Question, not to replace them. Enabling the use of GenAI algorithms that differ from the ones used by these patterns.

The GenAI Flow activity has the following required configuration properties:

- **aiprocedure** – a reference to the resource representing the embedded GenAI Flow. This property is read-only.
- **input** – a VAIL expression used to produce the input value passed to the GenAI Flow.

The GenAI Flow activity has the following optional configuration properties:

- **returnBehavior** – chosen from a list of three options:
  - *Use Return Value as Outbound event* - Emit the result of the procedure call as the outbound event. This is the default behavior if no option is selected.
  - *Attach Return Value to returnProperty* - Emit the original event, with an extra property containing the procedure results.
  - *Ignore Return Value* - Emit the original inbound event as the outbound event (unchanged)
- **returnProperty** – the name of a property to assign the result of the GenAI Flow to on the outbound event. NOTE: This property is only used when "Attach Return Value to returnProperty" is selected for returnBehavior.
- **runtimeConfiguration** – specifies the VAIL expression used to produce runtime configuration properties provided to the GenAI Flow. The expression should evaluate to an Object containing the property names and values.
- **useConversation** – a Boolean indicating whether the task should participate in a conversation when submitting the prompt. Use of this option requires an active collaboration. By default, the collaboration ID is used as the id of the conversation meaning that multiple tasks can participate in the same conversation (as long as they are operating on the same collaboration instance).
- **conversationName** – a String indicating the name of a conversation in which the task should participate. If not specified, then the collaboration's default conversation will be used. Use of this option allows two or more tasks to share conversation state within the collaboration instance that is separate from the default collaboration-wide conversation. Conversation names must be declared in the collaboration's properties. If a GenAI Flow task requires its own separate conversation state, use the conversation name `self`. This will create a conversation that is private to the task.

Get Collaboration

The Get Collaboration activity pattern returns the results for the current collaboration. The GetCollaboration activity pattern will first look in the Service State for a collaboration matching the current collaborationId (the collaboration id for this root to leaf path). If no collaboration is found, the database will be queried for such a collaboration.

Get Collaboration tasks may only appear as children of a task that creates a collaboration.

The Get Collaboration activity pattern defines the following optional configuration properties:

- **taskName** – The name of the collaboration task, entity role, or collaboration role. If no task name is provided, the full collaboration instance will be returned.
- **returnBehavior** – Chosen from a list of two options:
  - *Use Return Value as Outbound event* - Emit the result of the procedure call as the outbound event. This is the default behavior if no option is selected.
  - *Attach Return Value to returnProperty* - Emit the original event, with an extra property containing the procedure results.
- **returnProperty** – The name of a property to assign the result of the procedure to on the outbound event. NOTE: This property is only used when "Attach Return Value to returnProperty" is selected for returnBehavior.

Join

The Join Activity Pattern allows users to identify conditions that occur across multiple event streams and merge these into singular events for the consumption of downstream tasks.

For an example, consider the engine monitoring demo application, which detects when the engine enters a bad state. There are two independent streams in the demo, one containing engine speed readings and another with the engine temperature. Each reading is associated with a specific engineId, so to identify an engine in a bad state we need to be able to identify readings on the engine speed stream which have a speed over 1000 and readings on the engine temperature stream over 140, with the same engineId, where both events occur within 30 seconds of each other.

This “Bad Engine” situation can only be captured by identifying conditions on two parallel streams that occur within a constrained time window.

The Join Activity Pattern includes the following configuration properties:

- **joinOrder** – Specify an ordering for the parent tasks. The join will then be applied left to right given the parent ordering.
- **order** – One of BEFORE, AFTER, or EITHER. The order in which the parent events should occur for the join task to trigger, when ordering the parent tasks from left to right. If order doesn’t matter, select EITHER.
- **withinDuration** – A duration string like `30 seconds` which describes the time window in which the events must occur on all parent streams to trigger the Join task. Keep in mind that wider windows will perform worse, as they require the system to keep track of more historical data.
- **constraints** – An array of constraints that need to be true for the join condition to be satisfied. Constraints can be specified on each stream individually or on both events together. The event emitted by each parent is identified by the parent task name, so if a join task has two parents: engineSpeedStream and engineTempStream, one constraint could be `engineSpeedStream.engineId == engineTempStream.engineId`. Another constraint might be `engineSpeedStream.rpm > 1000`. All of the conditions must be satisfied for the join task to execute.
- **imports** – A list of resources referenced by the *constraints* that must be imported.

An important note about joins is the order of the upstream tasks matters. The leftmost stream above the join determines the maximum frequency with which joins can be made. The output from a join task is unique for each unique event from the leftmost parent task. If the leftmost parent of the join produces events once every 10 seconds, the join will emit events at most once every 10 seconds, regardless of the frequency of the other parent tasks.

Note: A Join task cannot be used beneath a SplitByGroup task.

The Join activity pattern produces the following event streams:

- **event** (default) – the combined event containing the joined input events
- **timeout** – a partially combined event containing any partial matches (starting with the leftmost stream) that fail to find a subsequent matching event.

K-Means Cluster

The K-Means Cluster activity uses a K-Means Clustering algorithm to cluster the data into the specified number of clusters using the specified distance measurement.

The K-Means Cluster activity contains the following required configuration properties:

- **XYProperties** – The property names (e.g. “speed”, “temperature”, etc.) on the inbound events to compute clusters for. Users may specify either X and Y properties for 2 dimensional analysis or just one property for one dimensional analysis. Each property must be numeric.
- **reservoirType** – The type of reservoir used to store data points. The type will determine which algorithm is used for computing aggregate statistics and when to purge old data points.
  - *Sliding Time Window* – Computes statistics using `Integer` values. All events that occured within the specified time window will be used to calculate statistics (e.g. all events in the last 5 minutes)
  - *Sliding Count* – Computes statistics using `Integer` values. The last *n* events will be used to calculate the statistics
  - *Fixed Time Window* – Computes statistics using `Real` values. A fixed time window will add events to the window until the time of the next reset. For example, a fixed time window of 1 hour will calculate analytics for all events within the hour, starting at the top of the hour and ending at the top of the new hour.
  - *Fixed Count* – Computes statistics using `Real` values. Calculate analytics on all of the events until the specified number of events has been reached. Then the analytics will be reset back to null.
- **windowLength** – How long events should stay in the window. For time based reservoirs this is an interval string (e.g.: “10 minutes”). For count based reservoirs this should be the number of events.
- **DistanceMeasure** – The distance measuring algorithm to use. This may be: *Euclidean Distance, Manhattan Distance, Canberra Distance Earth Mover’s distance, Chebyshev Distance*. For more information, refer to the Apache Math Distance Measure Documentation.
- **numClusters** – The number of clusters to split the data into.

K-Means Cluster activity pattern produces the following event streams:

- **event** (default) – Emits the original incoming event.
- **reset** – If the reservoirType is set to *Fixed Time Window* or *Fixed Count*, the Event Handler will automatically emit a *reset* event with the value of the analytics just before the state is automatically reset to null.

The K-Means Cluster activity pattern generates four Procedures: Get, Update, Reset, and Predict. The Get procedure can be used to retrieve the current result of the analytics. An example return value is shown below:

```
{
  "speed": [
    {
      "points": [
        {"point": [20]},
        {"point": [21]},
        {"point": [22]}
      ],
      "center": [21]
    },
    {
      "points": [
        {"point": [80]},
        {"point": [81]},
        {"point": [82]}
      ],
      "center": [81]
    },
    {
      "points": [
        {"point": [50]},
        {"point": [51]},
        {"point": [52]}
      ],
      "center": [51]
    }
  ]
}
```

The Predict Procedure can be used to predict which of the existing clusters the test point would best fit into. Using the clusters above, executing the Predict Procedure with the parameters: `value=25` and `yPropName=speed` the return value would be:

```
{
   "center": [21],
   "points": [
      {"point": [20]},
      {"point": [21]},
      {"point": [22]}
   ],
   "distance": 4
}
```

For more information, refer to the Apache Math K-Means Clusters Documentation.

Linear Regression

The Linear Regression activity uses a linear regression algorithm to determine the slope and intercept given recorded x/y pairs as well as predict future values.

The Linear Regression activity contains the following required configuration properties:

- **XYProperties** – The property names (e.g. “speed”, “temperature”, etc.) on the inbound events to compute clusters for. Each property must be numeric.
- **reservoirType** – The type of reservoir used to store data points. The type will determine which algorithm is used for computing aggregate statistics and when to purge old data points.
  - *Sliding Time Window* – Computes statistics using `Integer` values. All events that occured within the specified time window will be used to calculate statistics (e.g. all events in the last 5 minutes)
  - *Sliding Count* – Computes statistics using `Integer` values. The last *n* events will be used to calculate the statistics
  - *Fixed Time Window* – Computes statistics using `Real` values. A fixed time window will add events to the window until the time of the next reset. For example, a fixed time window of 1 hour will calculate analytics for all events within the hour, starting at the top of the hour and ending at the top of the new hour.
  - *Fixed Count* – Computes statistics using `Real` values. Calculate analytics on all of the events until the specified number of events has been reached. Then the analytics will be reset back to null.
- **windowLength** – How long events should stay in the window. For time based reservoirs this is an interval string (e.g.: “10 minutes”). For count based reservoirs this should be the number of events.

Linear Regression activity pattern produces the following event streams:

- **event** (default) – Emits the original incoming event.
- **reset** – If the reservoirType is set to *Fixed Time Window* or *Fixed Count*, the Event Handler will automatically emit a *reset* event with the value of the analytics just before the state is automatically reset to null.

The Linear Regression activity pattern generates four Procedures: Get, Update, Reset, and Predict. The Get procedure can be used to retrieve the current result of the analytics. An example return value is shown below:

```
{
  "intercept": 0,
  "slope": 10,
  "meanSquaredError": 0,
  "r": 1
}
```

The Predict Procedure can be used to predict the y-value for a given input. Using the linear regression above, executing the Predict Procedure with the parameters: `value=25` and `yPropName=speed` returns the value `250`.

For more information, refer to the Apache Math Linear Regression Documentation.

Limit

The limit Activity Pattern allows users to define a max number of events, and an interval over which that maximum will be applied. For every interval, events will be allowed to flow through the limit task until the max is hit, at which point all future events until the end of the interval are discarded.

The required configuration properties include:

- **maxCount** – The maximum number of events that are allowed through the limit within the interval.
- **interval** – an interval string such as `1 second` or `10 minutes` that defines the timeframe in which the maxCount limit should be applied.

Log Stream

The Log Stream task is useful for debugging purposes. It takes no configuration properties, and simply logs each event that reaches the log stream task at the INFO level. When developing an app, it's often useful to attach a log stream to each task as you experiment with its configuration in order to check the output of a task. Then, once the task is properly configured, remove the log stream, begin adding a new task and attach a new log stream to the new task.

Loop While

The Loop While activity pattern acts like a visual while loop within the App Builder.

The Loop While activity pattern has the following required configuration properties:

- **condition** (VAIL Expression) – while the condition holds true, the loop will continue. The condition may refer to the *counterProperty* directly (for example: *counter < event.list.size()* or *retryCount < 10*)

The Loop While activity pattern has the following optional configuration properties:

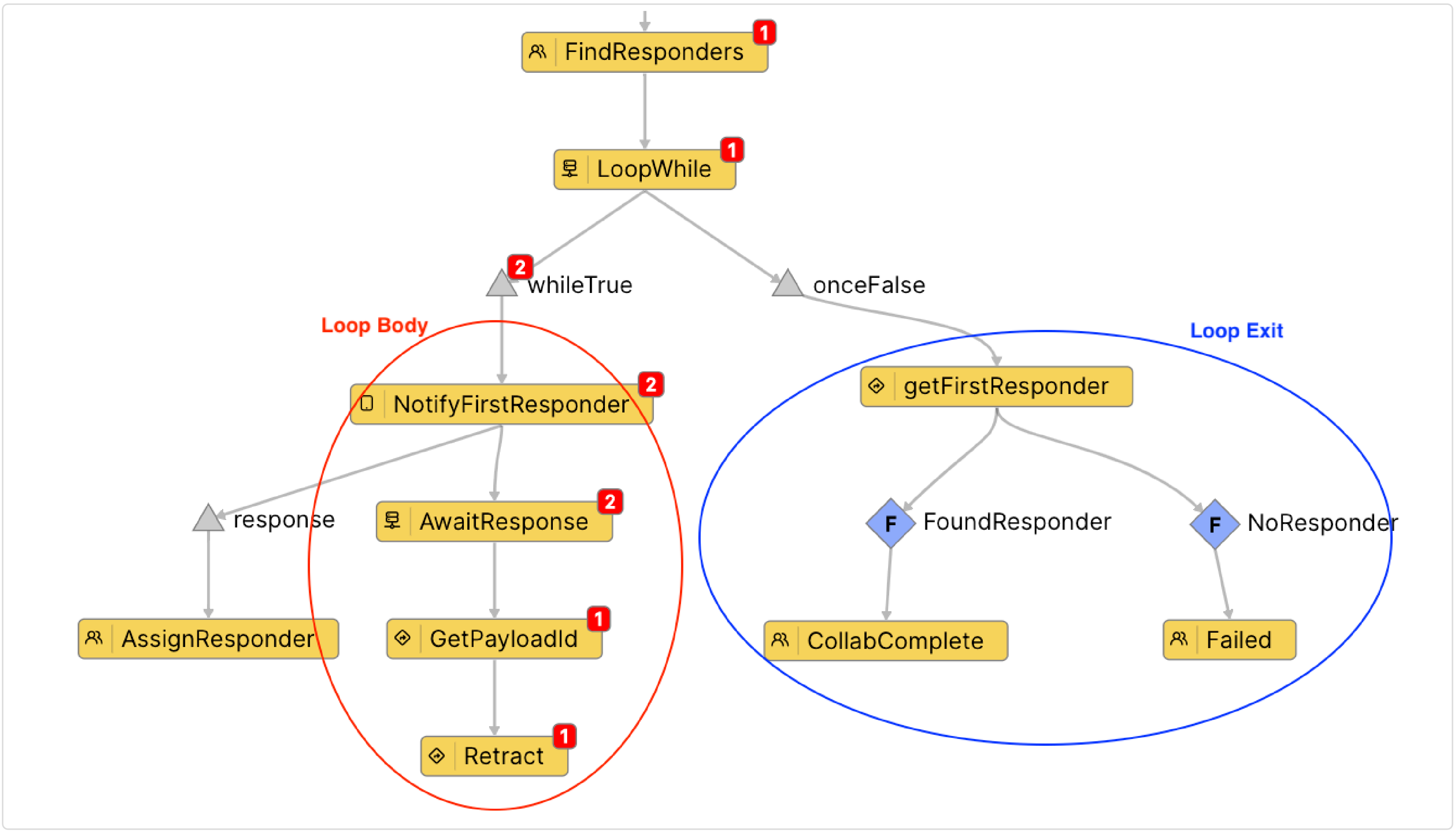
- **imports** – A list of resources referenced by the condition property
- **maxRetryCount** (Integer) – The maximum number of times this loop can occur. Even if the condition still evaluates to true, if the loop counter exceeds the maxRetryCount, the loop will terminate and emit a *maxRetriesExceeded* event
- **counterProperty** (String) – the variable that represents the loop counter (the number of times the loop has been executed). If not set, this defaults to *retryCount*. The counterProperty may be referenced directly in the *condition* and will be attached to the *event* object in any task that is part of the *loop body*.

Loop While produces the following event streams:

- **whileTrue** (default) – The event emitted when the condition evaluates to *true*. This event will contain the *counterProperty* and its current value. Any tasks connected beneath the *whileTrue* event are part of the *loop body* meaning they will be repeatedly executed as part of the loop.
- **onceFalse** – The event that occurs when the condition is evaluated to false.
- **maxRetriesExceeded** – The event that occurs if the success condition has not been met and the maxRetryCount has been exceeded.
- **event** – Emits the original incoming event.

The *loop body* will repeatedly execute the tasks from the *whileTrue* event to its leaf. The leaf is considered the end of the loop and the loop is restarted again at the *top*. The only exception to this is that if a Filter task somewhere between the LoopWhile task and a leaf evaluates to *false*, the loop will continue to the next iteration without terminating. The *whileTrue* event to leaf path is a path of *synchronous* tasks. This does not include any asynchronous events (represented by triangles). The *whileTrue* branch may only contain one *leaf* task.

The example below is a piece of the Advanced Collaboration Tutorial. The loop will iterate through the recommended first responders. A responder is sent a Notification and must respond within 1 minute. If the responder does not reply within 1 minute, the notification is retracted and the next iteration of the loop begins notifying the next responder. The loop condition is: *counter < event.recommendedResponders.size()*. Once the condition is false, the loop exit tasks are executed. If the assignment has been made, the collaboration completes successfully. Otherwise, it is marked as *failed*.



The *loop body* is the path from *NotifyFirstResponder* -> *AwaitResponse* -> *GetPayloadId* -> *Retract*. Notice that the notification response and *AssignResponder* are not part of the loop body because the response event is asynchronous.

Merge

Sometimes it is possible to end up in a situation where multiple streams are producing identical data, and in these cases it is often helpful to merge them into a single stream that can be consumed by downstream tasks. The Merge task does this for any number of streams. For best results, the inbound data on each of the independent parent streams should adhere to the same schema so that downstream tasks can all process the output of the merge in a consistent way.

The Merge task takes no configuration properties.

Missing

The Missing Activity Pattern detects the absence of an event for a fixed time duration. This can be useful for monitoring the heartbeat or pulse of a system (or person). The Missing task takes inbound events and produces no output as long as events occur within the specified window; but if no event is detected for the specified duration, then an output event is produced indicating no inbound event was received. As long as no new inbound event is received, an output event will be produced for every interval that passes without an inbound event.

The Missing Activity Pattern accepts two additional configuration properties:

- **duration** – An interval string like `3 minutes` or `90 seconds` that expresses how frequently inbound events should arrive. An absence of an event for this duration triggers the emission of an outbound event. By default the minimum legal value is `1 minute`. This is controlled by the `minimumScheduledProcedureInterval` quota value (contact your system administrator if you require a lower value).

The output of the missing task looks like:

```
{
    "missingAt": "2017-10-09T21:45:28.231Z",
    "task": "MissingMyTopic",
    "lastEvent": {
        ... // Copy of last event received
    }
}
```

Missing produces the following event streams:

- **event** (default) – The event emitted when no event has appeared during the configured interval.
- **restart** – The event emitted on the first event received after a missing event was sent. It has the following fields:
    - *newEvent* – The event that triggered the restart.
    - *lastEvent* – The last event received before the task was considered missing.
    - *timeElapsed* – The number of seconds between the last two events.
    - *lastReceived* – The time at which lastEvent was received.

Notify

The Notify activity pattern sends a Notification to one more Vantiq users using the Vantiq Mobile Client. The contents of this notification are defined by the title, body, and a Vantiq Client.

The Notify activity pattern has the following required configuration properties:

- **title** (VAIL Expression) – The title of the notification.
- **body** (VAIL Expression) – Body text of this notification.
- **users** (Union) – List of usernames to send this notification to. This may either be an actual array of VAIL expressions or a single VAIL expression which will evaluate to an array (for example: *event.userList*).
- **clientName** (Enumerated) – The name of the Client that will be displayed when the user opens and interacts with this notification.

Compatibility Note: The Notify pattern does not support `${variable}` notation for substitution. Instead, use `"my message... " + <variable> + "..."` syntax.

The Notify activity pattern has the following optional configuration properties:

- **imports** – A list of resources to be imported for this activity.
- **maxResponseTime** (VAIL Expression) – The maximum amount of time (ie: "20 minutes") that the notified users have to reply. Otherwise, a responseTimeout event is raised.
- **pushSourceName** (Enumerated) – The name of the push notification source used to send out this notification. If not specified, one will be auto-generated.
- **firstResponseFilter** (VAIL Expression) – WHERE clause for the firstResponse event. If specified, the firstResponse event will only be raised if the response passes the filter.
- **disableTemplateSubstitution** (Boolean) – If true, the Client payload will not be subjected to template substitution prior to sending the notification. Instead, the Client will be sent as-is. This can be helpful if your Client contains Javascript code that uses String interpolation (e.g. `${variable}`) which will otherwise be likely to result in errors in the Notification task. The system will still apply template substitution to the title and body.

Notify produces the following event streams:

- **response** (default) – The event that occurs when a response to the notification is received.
  - *value*: The response object containing the values the user selected in the input fields of the notification.
  - *submitValue*: The submit value the user selected.
  - *responseTopic*: The topic the response was published to which triggered this response event.
  - *username*: The Vantiq user who responded to the notification.
  - *state*: An object containing the *collaborationId* for the collaboration instance which sent this notification. May also contain any other data specified in payload.state in the configuration.
  - *arsInfo*: An object containing metadata collected by the device which published this response, including the deviceName, deviceId, and location (in GeoJSON format) from which the response was sent.
- **firstResponse** – The event arrives when the first notified user responds to this notification.
  - *value*: The response object containing the values the user selected in the input fields of the notification.
  - *submitValue*: The submit value the user selected.
  - *responseTopic*: The topic the response was published to which triggered this response event.
  - *username*: The Vantiq user who responded to the notification.
  - *state*: An object containing the *collaborationId* for the collaboration instance which sent this notification. May also contain any other data specified in payload.state in the configuration.
  - *arsInfo*: An object containing metadata collected by the device which published this response, including the deviceName, deviceId, and location (in GeoJSON format) from which the response was sent.
- **responseTimeout** – The event that is emitted a notification does not receive a response within the configured "maxResponseTime".
  - *payloadId*: _id for the ArsPayloadMessage instance.
  - *users*: the array of notified users.
- **event** – Emits the original incoming event.

An example *response* event is detailed below:

```
{
  "values": {},
  "submitValue": 0,
  "arsInfo": {
    "deviceId": "9BC12C06-134A-40B2-9271-F6B121E1D663",
    "localeLanguage": "en",
    "responseTimestamp": "2022-06-01T17:53:47.949Z",
    "localeVariant": "*",
    "clientName": "FirstResponderResponse",
    "deviceName": "<device name>",
    "responseLocation": {
      "altitudeMeters": 60.94035339355469,
      "longitude": -122.06644349836851,
      "horizontalAccuracyMeters": 35,
      "speedMetersPerSecond": -1,
      "bearingDegrees": -1,
      "latitude": 37.90660164348477,
      "verticalAccuracyMeters": 8.852736473083496
    },
    "username": "<username>",
    "localeCountry": "us"
  },
  "deviceName": "<device name>",
  "deviceId": "<device id>",
  "responseTopic": "/patient.monitoring.HeartRate.MonitorReading/tasks/NotifyFirstResponder/response/trigger",
  "responseResource": "topics",
  "username": "<username>",
  "responseResourceId": "/patient.monitoring.HeartRate.MonitorReading/tasks/NotifyFirstResponder/response/trigger",
  "state": {
    "collaborationId": "<collaborationId>"
  }
}
```

The collaboration instance is updated to the following:

```
{
  "id": "<collaboration id>",
  "status": "active",
  "name": "<app name>",
  "results": {
    "<notify task name>": {
      "payloadId": "<_id for the ArsPayloadMessage instance>",
      "responses": "<array of response events>"
    }
  }
}
```

The *payloadId* may be used to retract messages using the Notification Service Procedure Notification.retractPayload.

Polynomial Fitter

The Polynomial Fitter activity uses a Polynomial Curve fitter to determine the coefficients and residuals for a given set of points. This Activity Pattern will generate Predict and Derivative Procedures that may be used to predict a value along the curve and the rate of change at a certain point, respectively.

The Polynomial Fitter activity contains the following required configuration properties:

- **XYProperties** – The property names (e.g. “speed”, “temperature”, etc.) on the inbound events to compute clusters for. Each property must be numeric.
- **reservoirType** – The type of reservoir used to store data points. The type will determine which algorithm is used for computing aggregate statistics and when to purge old data points.
    - *Sliding Time Window* – Computes statistics using `Integer` values. All events that occured within the specified time window will be used to calculate statistics (e.g. all events in the last 5 minutes)
    - *Sliding Count* – Computes statistics using `Integer` values. The last *n* events will be used to calculate the statistics
    - *Fixed Time Window* – Computes statistics using `Real` values. A fixed time window will add events to the window until the time of the next reset. For example, a fixed time window of 1 hour will calculate analytics for all events within the hour, starting at the top of the hour and ending at the top of the new hour.
    - *Fixed Count* – Computes statistics using `Real` values. Calculate analytics on all of the events until the specified number of events has been reached. Then the analytics will be reset back to null.
- **windowLength** – How long events should stay in the window. For time based reservoirs this is an interval string (e.g.: “10 minutes”). For count based reservoirs this should be the number of events.
- **degree** – The degree of the polynomial.

Polynomial Fitter activity pattern produces the following event streams:

- **event** (default) – Emits the original incoming event.
- **reset** – If the reservoirType is set to *Fixed Time Window* or *Fixed Count*, the Event Handler will automatically emit a *reset* event with the value of the analytics just before the state is automatically reset to null.

The Polynomial Fitter activity pattern generates six Procedures: Get, Update, Reset, Predict, Derivative, and Integral.

As an example below, say the following events are processed by a Polynomial Fitter task configured with degree *2*.

```
{
  "time": 1,
  "temperature": 38
}
```

```
{
  "time": 2,
  "temperature": 68
}
```

```
{
  "time": 3,
  "temperature": 120
}
```

The Get procedure can be used to retrieve the current result of the polynomial fit. An example return value is shown below:

```
{
  "coefficients": [
    30.000000000000004,
    -3.000000000000007,
    11.000000000000002
  ],
  "RMS": 0,
  "cost": 0,
  "residuals": {
    "l1Norm": 0,
    "linfNorm": 0,
    "norm": 0,
    "dataRef": [0, 0, 0],
    "dimension": 3,
    "naN": false,
    "infinite": false,
    "minIndex": 2,
    "maxIndex": 2,
    "minValue": 0,
    "maxValue": 0
  }
}
```

The Predict Procedure can be used to predict the y-value for a given input. Using the example above, executing the Predict Procedure with the parameters: `value=4` and `yPropName=temperature` the return value is `194`.

The Derivative Procedure can be used to calculate the rate of change at a particular point. Using the example above, executing the Derivative Procedure with the parameters: `value=2` and `yPropName=temperature` the return value is `41`.

The Integral Procedure can be used to calculate the area under the curve between two points. Using the example above, executing the Integral Procedure with the parameters: `from=1`, `to=3`, and `yPropName=temperature` the return value is `143.33333333333331`.

For more information, refer to the Apache Math Polynomial Documentation.

Predict Paths By Age

The PredictPathsByAge Activity Pattern predicts the next location in paths. This is generally used for paths that have not recently been updated. This Activity Pattern processes a set of paths, producing a prediction for paths not updated since some *expiration* time.

The PredictPathsByAge Activity Pattern has the following required configuration properties:

- **expirationTime** – the latest time for which a prediction will be made.
- **lastLegOnly** – a boolean indicating that, when true, velocity calculations should be performed based on the last two items in the path. If false, the velocity is based on first and last items in the path.
- **outboundProperty** – the name of the property into which to place the output information.
- **pathHistory** – VAIL expression containing the path histories from which to predict positions.
- **refreshInterval** – interval at which to refresh the set of regions (*e.g.* 30 seconds).

In addition, the following optional configuration properties may be provided:

- **pathProperty** – the name of the property at which tracked paths are found. Default is `trackedPath`.
- **timeOfPrediction** – the time to give for the predicted position(s). The default value is `now()`.
- **trackingRegionFilter** – an expression that limits the set of `TrackingRegions` considered for the predicted positions' tracking regions. This expression will be added to the WHERE clause used to select the regions to consider. If absent or empty, all `TrackingRegions` in the namespace

are considered. Please see the tracking regions information in the *BuildAndPredictPath* Activity Pattern for more information.

The PredictPositionsByAge Activity Pattern will often be found after a Missing Activity Pattern, acting here to predict positions when no updates have happened for some period of time. It behaves much like Build and Predict Path Activity Pattern, invoked by something other than a position update (typically, the passage of time via the Missing Activity Pattern).

Procedure

The Procedure Activity Pattern executes a procedure and emits the results.

The Procedure Activity Pattern has only one required configuration property:

- **procedure** – The name of the procedure to execute.

If only the procedure name is specified, then the procedure must take exactly one parameter (the inbound event), and the return value from the procedure will be used as the outbound event. This behavior can be changed in a couple ways using the optional parameters described below. These optional parameters allow you to specify additional parameters to the procedure and override the return behavior so the returned value as attached to the inbound event for output purposes, rather than replacing the inbound event entirely.

The Procedure Activity Pattern has the following optional configuration properties:

- **parameters** – An array of VAIL expressions evaluated against the inbound event to produce the parameters to the procedure. When not specified, the procedure will be passed the inbound event as its only parameter.
- **returnBehavior** – Chosen from a list of three options:
  - *Use Return Value as Outbound event* - Emit the result of the procedure call as the outbound event. This is the default behavior if no option is selected.
  - *Attach Return Value to returnProperty* - Emit the original event, with an extra property containing the procedure results.
  - *Ignore Return Value* - Emit the original inbound event as the outbound event (unchanged)
- **returnProperty** – The name of a property to assign the result of the procedure to on the outbound event. NOTE: This property is only used when “Attach Return Value to returnProperty” is selected for returnBehavior.
- **schema** – The name of a type that defines the schema of the output of the procedure. This is used to populate downstream event property suggestions.
- **imports** – A list of resources referenced by the *parameters* that must be imported.
- **partitionKey** – The VAIL expression used to compute the partition key used to route the procedure invocation. This is only required when invoking a private procedure for a stateful service.
- **redeliverOnError** – The name of a procedure used to determine if a reliable event should be redelivered after an error. If procedure returns `true`, then the error is considered transient and will be redelivered. The procedure will be provided with 2 parameters, the original event and the exception that occurred.

The Procedure activity pattern produces the following event streams:

- **event** (default) – the results of all successful requests.
- **onError** – an event providing context for any failed request. The properties of this event are:
  - **errorCode** (String) – a unique “code” associated with the generated exception.
  - **errorMessage** (String) – the text of the error message for the generated exception.
  - **errorParams** (String[]) – the parameters used to replace substitution values in the error message.
  - **triggeringEvent** (Object) – the event that triggered the exception.

PublishToService

The PublishToService Activity Pattern publishes events to an Inbound Event Type of a Service.

- **service** – The Service to which inbound events will be sent.
- **eventTypeName** – The name of the Inbound Event Type of the service which will process this event

PublishToService also contains the optional configuration property:

- **processedByClause** – The Processed By Clause used to describe the remote targets for this publish.

PublishToSource

The PublishToSource Activity Pattern publishes inbound events to a source.

The PublishToSource Activity Pattern contains the following configuration properties:

- **source** – The name of the source to which inbound events will be sent.
- **sourceConfig** – A configuration object used when publishing to the specified source. This translates to the USING clause of a PUBLISH TO SOURCE in VAIL.

The PublishToSource Activity Pattern has the following optional configuration properties:

- **redeliverOnError** – The name of a procedure used to determine if a reliable event should be redelivered after an error. If procedure returns `true`, then the error is considered transient and will be redelivered. The procedure will be provided with 2 parameters, the original event and the exception that occurred.
- **retries** – The number of times to try to publish to the source before failing. Every retry attempt is immediate. Defaults to 1.

The PublishToSource activity pattern produces the following event streams:

- **event** (default) – the results of all successful requests.
- **onError** – an event providing context for any failed request. The properties of this event are:
    - **errorCode** (String) – a unique “code” associated with the generated exception.
    - **errorMessage** (String) – the text of the error message for the generated exception.
    - **errorParams** (String[]) – the parameters used to replace substitution values in the error message.
    - **triggeringEvent** (Object) – the event that triggered the exception.

PublishToTopic

The PublishToTopic Activity Pattern publishes inbound events to a topic.

- **topic** – The topic to which inbound events will be sent. Topic strings must be prefixed by a forward slash (/) character.

PublishToTopic also contains the following optional configuration properties that allow a user to become a publisher of a Catalog event:

- **catalog** – The name that identifies the catalog which defines the event for which the task will become a publisher.
- **event** – The name of the event type to which this task will publish events.
- **processedByClause** – The Processed By Clause used to describe the remote targets for this publish.

Rate

The Rate Activity Pattern keeps count of all events it sees in a time window. At the end of the time window, Rate emits the rate per second of events received in the window. The rate is the count of events seen since the last emission, divided by the number of seconds in the outputFrequency.

The Rate Activity Pattern has one required property:

- **outputFrequency** – An interval string indicating how often the rate should be emitted.

The Rate Activity Pattern has one optional property:

- **rateInterval** – Specifies the interval for which the event rate should be measured, e.g. events per 1 minute or events per 10 seconds. Defaults to 1 second.

Recommend

The Recommend activity pattern generates a list of recommendations based on some input criteria. Recommendations use a procedure to determine how similar each candidate is to the target of the recommendation, and then the results are a limited subset of the most similar candidates.

The Recommend activity pattern contains the following required configuration properties:

- **candidateType** – The type of the results that are output by the recommendation activity.

The Recommend activity pattern also contains the following optional configuration properties:

- **imports** – A list of resources to be imported for this activity
- **recommendationProcedure** – The name of the procedure to execute to produce recommendations. If this is not specified the default recommender (**Recommend.defaultRecommendations**) will be used. This procedure must have the following signature: `procedureName(matchDirectives Object, pattern Object, candidateType String, matchType String)`
- **matchDirectives** – An object containing extra information required by the recommendation procedure. This includes:
    - *excludeProperties*: properties to be ignored by the recommender.
    - *maxDistance*: the maximum distance a recommended match may be from the candidate.
    - *maxRecommendations* the maximum number of recommendations to be returned by this task. This defaults to 10.
- **matchType** – The type of the input object against which recommendations will be evaluated. If not specified it is assumed this is the same as the candidateType.
- **returnBehavior** – Chosen from a list of three options:
    - *Use Return Value as Outbound event* - Emit the result of the recommendation as the outbound event. This is the default behavior if no option is selected.
    - *Attach Return Value to returnProperty* - Emit the original event, with an extra property containing the recommendation results.
- **returnProperty** – The name of a property to assign the result of the procedure to on the outbound event. NOTE: This property is only used when “Attach Return Value to returnProperty” is selected for returnBehavior.

The collaboration instance is updated to the following:

```
{
  "id": "<collaboration id>",
  "status": "active",
  "name": "<app name>",
  "results": {
    "<recommend task name>": {
      "recommendations": "<array of computed recommendations>"
    }
  }
}
```

RecordEvent

The RecordEvent Activity Pattern records an instance of an Event Type in the Event Ledger as part of Vantiq Advanced Event Broker. Attach a RecordEvent activity to an Event Stream to record all of the events seen by a subscriber, or attach RecordEvent to an activity that publishes its events to the catalog to record all events published by the local namespace.

The RecordEvent Activity Pattern has the following required configuration properties:

- **catalogName** – The name that identifies the catalog which defines the event.
- **eventName** – The name of an Event Type in the specified catalog for which the event should be recorded.

To view the ledger in a namespace for a specific Event Type, open the catalog pane and click the book icon next to the Event Type.

Reply

The Reply Activity Pattern is used to send a response to the caller of `Event.request` which triggered processing of the current event. If the event being processed was not sent in this manner, then this is a no-op.

The Reply Activity Pattern has the following required configuration properties:

- **response** – A VAIL expression whose result will be sent as the response value.

The Reply Activity Pattern also contains the following optional configuration properties:

- **imports** – A list of resources referenced by the *response* expression that must be imported.

Run TensorFlow Model On Document

TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

> This Activity Pattern may be constrained by installation-configurable resource consumption limits. See Operational Restrictions for details.

See Run TensorFlow Model On Temp Image. This task is appropriate when the input event is a Document.

Run TensorFlow Model On Image

> This Activity Pattern may be constrained by installation-configurable resource consumption limits. See Operational Restrictions for details.

Run TensorFlow Model on Temp Image

> This Activity Pattern may be constrained by installation-configurable resource consumption limits. See Operational Restrictions for details.

RunTensorFlowModelOnImage, *RunTensorFlowModelOnTempImage*, and *RunTensorFlowModelOnDocument* tasks are used to perform analysis via a TensorFlow model. RunTensorFlowModelOnImage is used when the input event represents a `system.images` resource, RunTensorFlowModelOnDocument is used when the input event represents a `document` resource, and RunTensorFlowModelOnTempImage is used when the input event represents the output of a video source. They are otherwise identical. This section will refer to RunTensorFlowModelOnImage.

TensorFlow models can be used for many purposes. For more details on how to use TensorFlow models in the Vantiq system, please see the Image Processing Guide, specifically the Image Analysis section.

The RunTensorFlowModelOnImage and RunTensorFlowModelOnDocument Activity Patterns are designed to run TensorFlow models of type `tensorflow/plain`. These are models where the semantics of the model are not understood by Vantiq. Because Vantiq is running without the benefit of that understanding, the use of such models requires a deeper understanding of the model's input and output requirements, as well as lower-level interaction with the model by the calling application. This is described in the "Plain" TensorFlow Models section. Users of this Activity Pattern are assumed to be familiar with that section.

By contrast, when running, for example, YOLO models, Vantiq can take advantage of its understanding of those models to pre- and post-process the parameters, providing a semantically relevant interface. Other types of TensorFlow models can be invoked by using the *Run TensorFlow Model* Activity Patterns; in such cases, the callers do not gain the benefit of Vantiq's understanding of these model type semantics.

The *Run TensorFlow Model* Activity Patterns will require input tensors to the model to be specified, and the output event will include a list of any output tensors. These input and output tensors will vary according to the model used, but callers should be aware that they can be quite large. Moreover, they will often require further *intepretation* – walking the output tensors to turn that *raw material* into a more consumable form.

The RunTensorFlowModelOnImage and RunTensorFlowModelOnDocument Activity Patterns contain the following configuration properties:

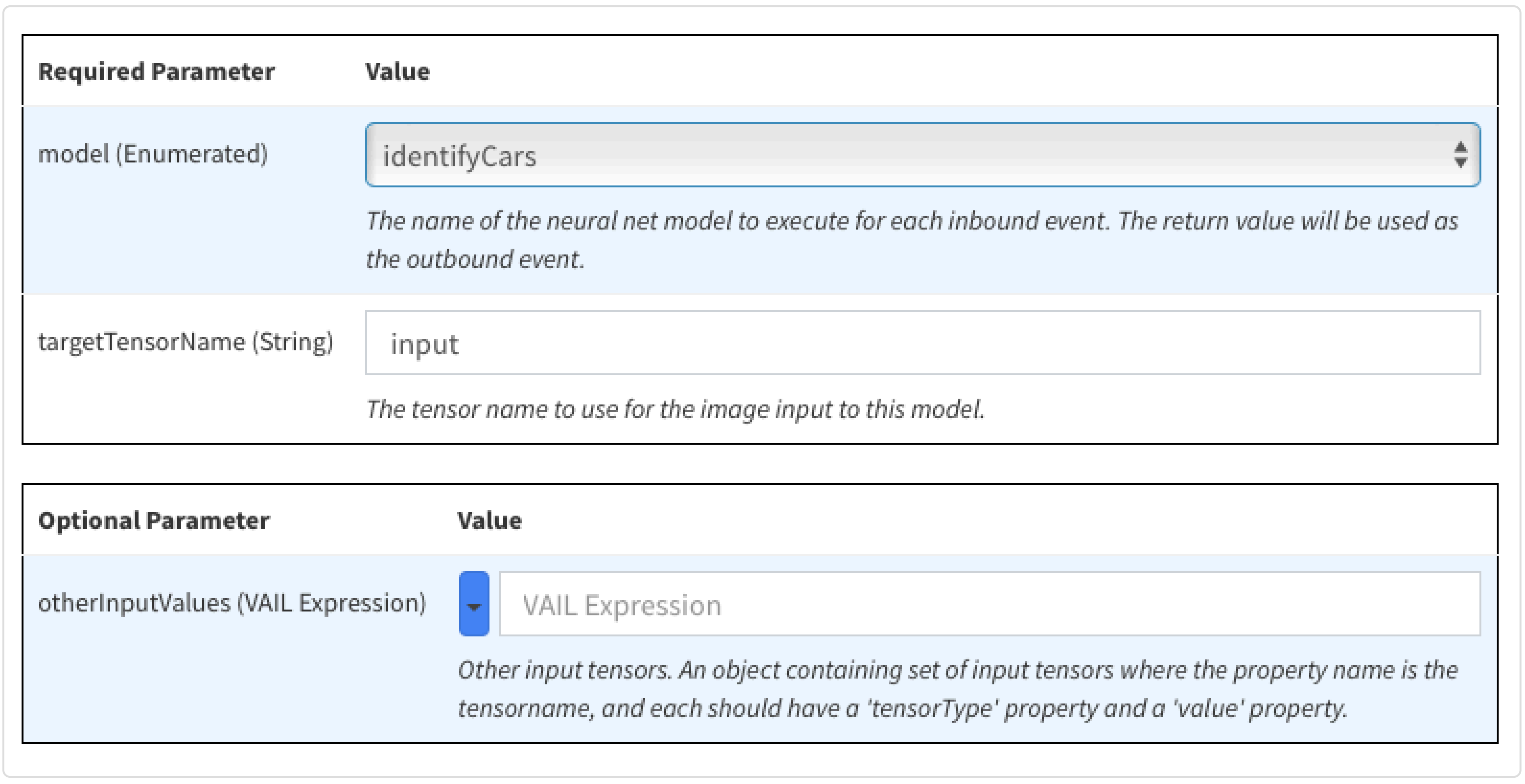
- **inputTensorName** – the name of the tensor to associate with the Image (or Document) comprising. the input event
- **model** – the model to run.

The two properties above are required. There is one additional, optional property:

- **otherInputValues** – a VAIL Object where each property describes an additional input tensor to the model. The property name is the tensor name, and each such property value must contain the following two properties:
  - tensorType – the type of tensor to be constructed

- value – the value to be provided.

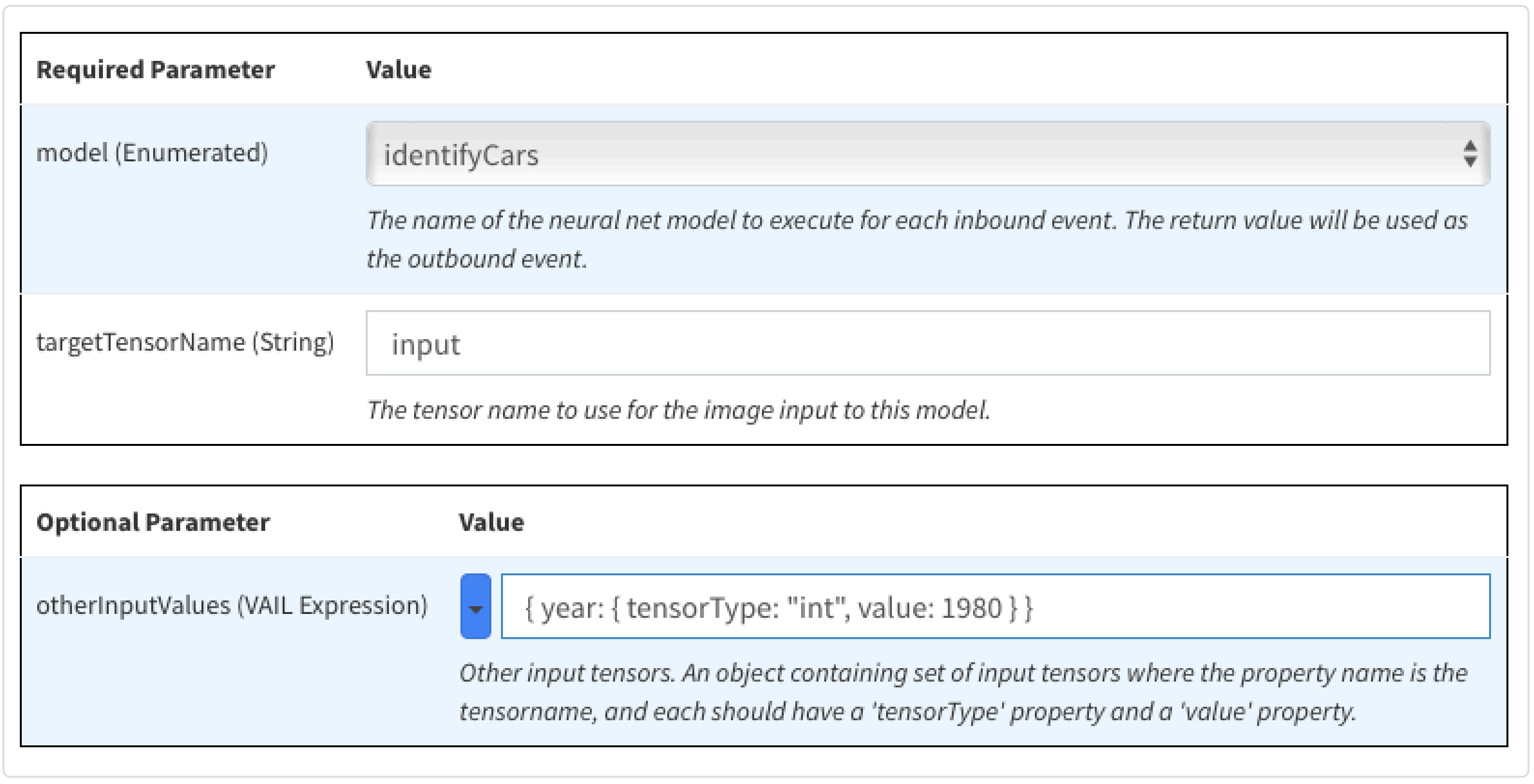
Using only the required properties, a task specification might look like this.



Required Parameter	Value
model (Enumerated)	identifyCars *The name of the neural net model to execute for each inbound event. The return value will be used as the outbound event.*
targetTensorName (String)	input *The tensor name to use for the image input to this model.*

Optional Parameter	Value
otherInputValues (VAIL Expression)	VAIL Expression *Other input tensors. An object containing set of input tensors where the property name is the tensorname, and each should have a 'tensorType' property and a 'value' property.*

If we choose to fill in the optional property `otherInputValues`, we might provide something like the following.



Required Parameter	Value
model (Enumerated)	identifyCars *The name of the neural net model to execute for each inbound event. The return value will be used as the outbound event.*
targetTensorName (String)	input *The tensor name to use for the image input to this model.*

Optional Parameter	Value
otherInputValues (VAIL Expression)	{ year: { tensorType: "int", value: 1980 } } *Other input tensors. An object containing set of input tensors where the property name is the tensorname, and each should have a 'tensorType' property and a 'value' property.*

(These examples parallel those provided in the Image Processing Guide. Please see “Plain” TensorFlow Models for further details.)

The output event from the task contains two properties:

- target – the image that was the input to the task
- results – the results of the analysis. The `results` property will contain an Object analogous to the `otherInputValues` property, an Object whose properties are the output tensor name, each of whose value contains two properties: `tensorType` and `value`.

For example, running an image of a car through our *identifyCars* model might produce the following results.

```
{
    "target":{
        "name": "mycar",
        "fileType": "image/jpeg",
        "contentSize": 71631,
        ...
    },
    "results": {
        modelName: { tensorType: "string", value: "Fusion" },
        manufacturer: { tensorType: "string", value: "Ford" }
    }
}
```

As noted elsewhere, `results` with such a simple and compact structure are atypical. A more common result would be a large array of REAL numbers from which a post-processing step would synthesize results. Taking a YOLO v3 model as an example, if it were run on an image as if it were a `tensorflow/plain` model, the result would be a `float` tensor of over 10,000 predictions, each of which would an array of REAL numbers.

> TensorFlow models are not provided as part of the Vantiq product.

Run TensorFlow Model On Tensors

> This Activity Pattern may be constrained by installation-configurable resource consumption limits. See Operational Restrictions for details.

The RunTensorFlowModelOnTensors task is used to perform analysis via a TensorFlow model. RunTensorFlowModelOnTensors is used when the input event consists of non-image data. This data is generally some form of numerical data from a sensor (*e.g.*, to look for specific conditions) or from other inputs.

TensorFlow models can be used for many purposes. For more details on how to use TensorFlow models in the Vantiq system, please see the Image Processing Guide, specifically the Image Analysis section.

The RunTensorFlowModelOnTensors Activity Patterns are designed to run TensorFlow models of type `tensorflow/plain`. These are models where the semantics of the model are not understood by Vantiq. Because Vantiq is running without the benefit of that understanding, the use of such models requires a deeper understanding of the model's input and output requirements, as well as lower-level interaction with the model by the calling application. This is described in the "Plain" TensorFlow Models section. Users of this Activity Pattern are assumed to be familiar with that section.

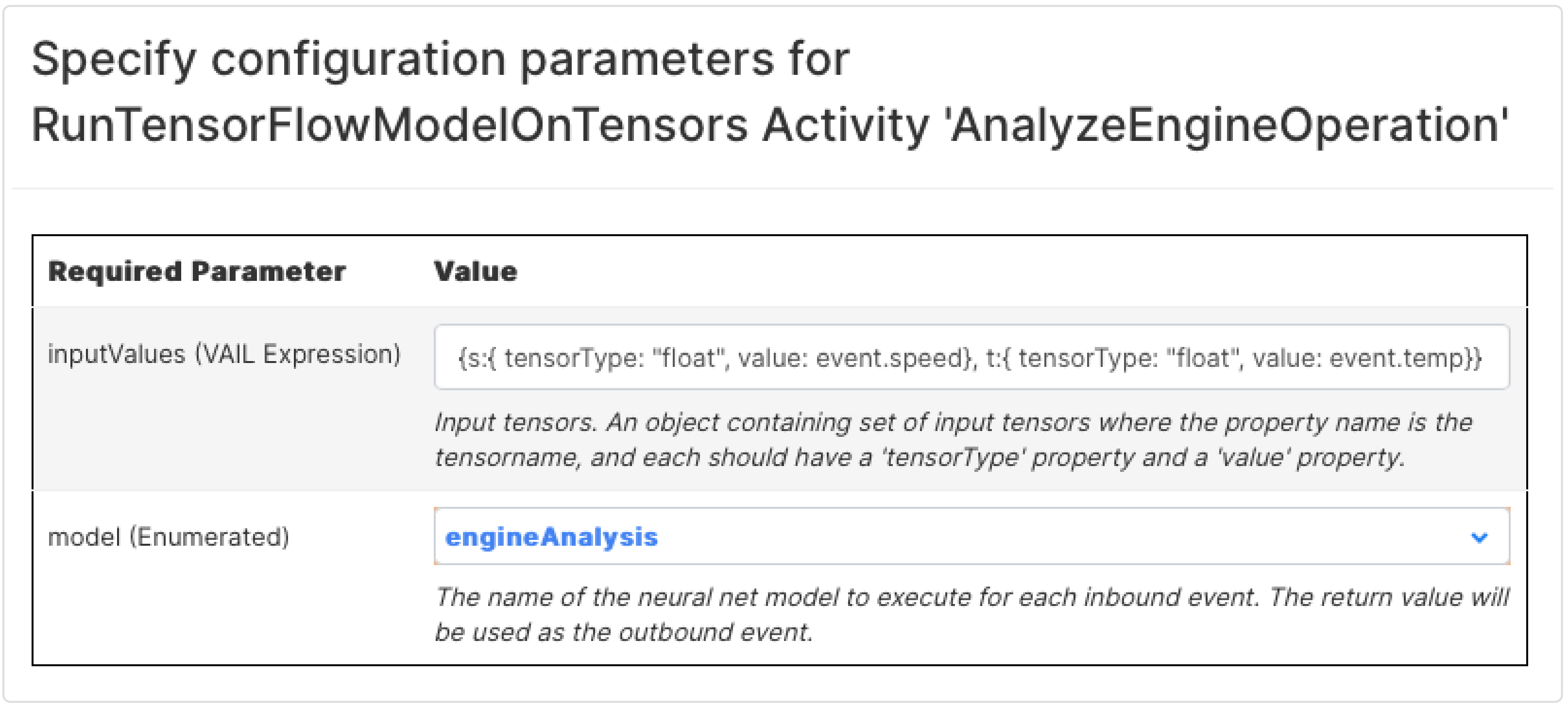
By contrast, when running, for example, YOLO models, Vantiq can take advantage of its understanding of those models to pre- and post-process the parameters, providing a semantically relevant interface. Other types of TensorFlow models can be invoked by using the *Run TensorFlow Model* Activity Patterns; in such cases, the callers do not gain the benefit of Vantiq's understanding of these model type semantics.

The *Run TensorFlow Model* Activity Patterns will require input tensors to the model to be specified, and the output event will include a list of any output tensors. These input and output tensors will vary according to the model used, but callers should be aware that they can be quite large. Moreover, they will often require further *intepretation* – walking the output tensors to turn that *raw material* into a more consumable form.

The RunTensorFlowModelOnTensors Activity Pattern contains the following configuration properties:

- **model** – the name model to run.
- **inputValues** – a VAIL Object where each property describes an additional input tensor value to the model. The property name is the tensor name, and each such property value must contain the following two properties:
    - tensorType – the type of tensor to be constructed
    - value – the value to be provided.

Using these properties, a task specification might look like this.



Specify configuration parameters for RunTensorFlowModelOnTensors Activity 'AnalyzeEngineOperation'

Required Parameter	Value
inputValues (VAIL Expression)	{s:{ tensorType: "float", value: event.speed}, t:{ tensorType: "float", value: event.temp}} *Input tensors. An object containing set of input tensors where the property name is the tensorname, and each should have a 'tensorType' property and a 'value' property.*
model (Enumerated)	engineAnalysis *The name of the neural net model to execute for each inbound event. The return value will be used as the outbound event.*

In this case, we are using a model named *engineAnalysis* that evaluates events from some sensor monitoring an engine. The input tensors for this model are two *float* tensors, named *s* and *t* (for speed and temperature). Thus, the `inputValues` property has properties named *s* and *t*, each with a `tensorType` of *float*, with values taken from the incoming event. The model will report any faults detected or behavioral abnormalities.

The output event from the task contains the `results` property:

- results – the results of the analysis. The `results` property will contain an Object analogous to the `inputValues` property, an Object whose properties are the output tensor name(s), each of whose value contains two properties: `tensorType` and `value`.

For example, running some sensor reports through an *engine analysis* model might produce the following results.

```
{
    ...
    "results": {
        faultsDetected: { tensorType: "integer", value: 0 },
        behavior: { tensorType: "integer", value: [100, 127] }
    }
}
```

In this case, our engine analysis model has returned a fault analysis for a fault of type 0, and some behavior reports with values 100 & 127. This type of report is not uncommon; it is up to the app developer to understand the model semantics for determining the meaning of these values.

As noted elsewhere, `results` with such compact structure may not be typical. For scoring models, a relatively small set of outputs may be normal as these are generally evaluating specific transactions (*e.g.*, our engine's behavior, a particular credit transaction for fraud detection), so large amounts of data is not necessary. However, other models may return larger amounts of data.

> TensorFlow models are not provided as part of the Vantiq product.

Sample

The Sample Activity Pattern is used to produce fewer events to the downstream tasks

The Sample Activity Pattern has the following required configuration property:

- **probability** – probability of emitting any event. This probability is independently computed for each event. Must be a real number between 0 and 1.

The Sample Activity Pattern has the following optional configuration properties:

- **interval** – an interval string such as `1 second` or `10 minutes` that defines the timeframe in which the maxEventsPerInterval limit should be applied.
- **maxEventsPerInterval** – The maximum number of events that are allowed through the task within the interval.
- **emitAtLeastOnce** – When true, guarantees at least one event is emitted per interval, even if no events are chosen by random probability.

SaveToType

The SaveToType Activity Pattern is used to save the inbound event to a type. Each inbound event is written to the database via an insert or upsert depending on the configuration.

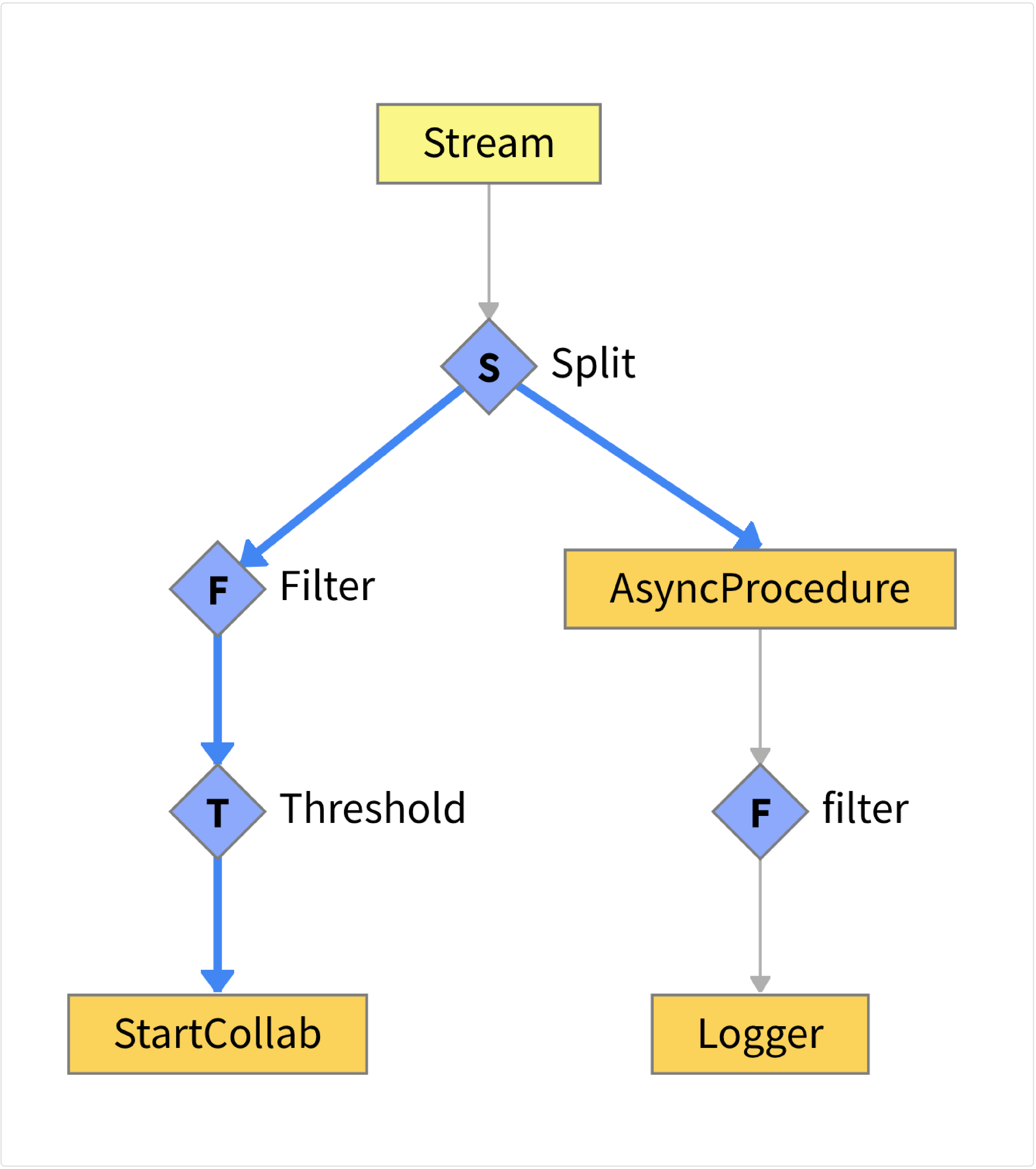
The SaveToType Activity Pattern contains the following configuration properties:

- **type** – The name of the type in which the data will be stored. It's expected that the inbound event will have a schema matching the schema of the specified type.
- **upsert** – When checked, upsert dictates that the operation to use when saving the event should be upsert rather than insert. Upsert can only be used when the specified type has a natural key.

The SaveToType Activity Pattern produces either an insert or update event on the configured type, which is used to trigger the next downstream task(s) in the app. Note that if you have other resources that also insert or update the configured type those operations will also produce events that trigger the downstream tasks in the app.

Split By Group

Split By Group partitions the service state into independent "groups", where the group is keyed off of an expression evaluated for each inbound event. All stateful, downstream tasks operate independently on each partition. The split task has no impact on stateless tasks.



In this example, the “Threshold” task will be partitioned and maintain independent state for each unique group key. All other tasks will ignore the split and operate as they normally do.

> A SplitByGroup task cannot be used beneath another SplitByGroup task. If a second SplitByGroup is needed, the app with the first SplitByGroup needs to publish an event to another app or stream that contains the second SplitByGroup.

There is a single required configuration property for the Split By Group Activity Pattern:

- groupBy – A VAIL expression evaluated for each inbound event that determines the group key for the event.

Submit Prompt

The Submit Prompt activity is used to submit a prompt to the specified LLM. The result is a `String` value returned in the specified property of the outbound event. It uses the built-in LLM service.

The Submit Prompt activity has the following required configuration properties:

- **llm** – the name of the *generative* LLM used to process the prompt.

- **prompt** – the VAIL expression used to compute the prompt. The expression should evaluate to either a String or an `io.vantiq.ai.ChatMessage` Array.

The prompt produced by the given expression will be submitted directly to the specified LLM for processing. No additional context or processing will be performed. The result will be stored in the specified property of the outbound event.

The Submit Prompt activity has the following optional configuration properties:

- **returnProperty** – the name of a property of the outbound event in which to store the response. Defaults to `llmResponse` if not specified.
- **useConversation** – a Boolean indicating whether the task should participate in a conversation when submitting the prompt. Use of this option requires an active collaboration. By default, the collaboration ID is used as the id of the conversation meaning that multiple SubmitPrompt tasks can participate in the same conversation (as long as they are operating on the same collaboration instance).
- **conversationName** – a String indicating the name of a conversation in which the task should participate. If not specified, then the collaboration's default conversation will be used. Use of this option allows two or more tasks to share conversation state within the collaboration instance that is separate from the default collaboration-wide conversation. Conversation names must be declared in the collaboration's properties. If a SubmitPrompt task requires its own separate conversation state, use the conversation name `self`. This will create a conversation that is private to the task.
- **responseProperties** – a VAIL expression used to produce a "properties" `Object` which will be attached to the response when it is recorded in a conversation. Ignore when `useConversation` if `false`.
- **tools** – a VAIL expression used to produce an Array of tools, available to the LLM when responding to the prompt. Each entry in the array should either be a ResourceReference to a VAIL procedure or service (e.g. `"services/com.mycompany.MyService"`) or an instance of the `io.vantiq.ai.FunctionDescriptor` schema type. See the LLM Service for more details. Note that tools are also sometimes referred to as functions.
- **runtimeConfig** - A VAIL expression that results in an Object representing an LLM configuration. When provided, this configuration is combined with the current LLM configuration and then applied to the SubmitPrompt invocation.
- **functionAuthorizer** - The name of a procedure used to authorize execution of LLM tools. The procedure signature must be *(String name, Object arguments)* and return a boolean if execution of the named tool should be either granted (true) or denied (false). See the LLM Reference Guide for more details.

The Submit Prompt activity pattern produces the following event streams:

- **event** (default) – the results of all successful requests.
- **onError** – an event providing context for any failed request. The properties of this event are:
  - **errorCode** (String) – a unique "code" associated with the generated exception.
  - **errorMessage** (String) – the text of the error message for the generated exception.
  - **errorParams** (String[]) – the parameters used to replace substitution values in the error message.
  - **triggeringEvent** (Object) – the event that triggered the exception.

Threshold

Threshold can be used to detect sequential events crossing a threshold. The threshold is expressed as a condition, and crossing the threshold is determined by the condition evaluating to different values for the sequential events. For instance, if we have an inbound stream of temperature readings from sensors, and we want to detect when the temperature goes over 100 degrees, the threshold condition would be `event.temp > 100`. If one event came in with a temp value of 90, then the next came in with a value of 101, the threshold would trigger. Likewise it would also trigger if initially the temperature was first 101, then a new reading came in at 90 degrees.

The threshold Activity Pattern contains only one required configuration property:

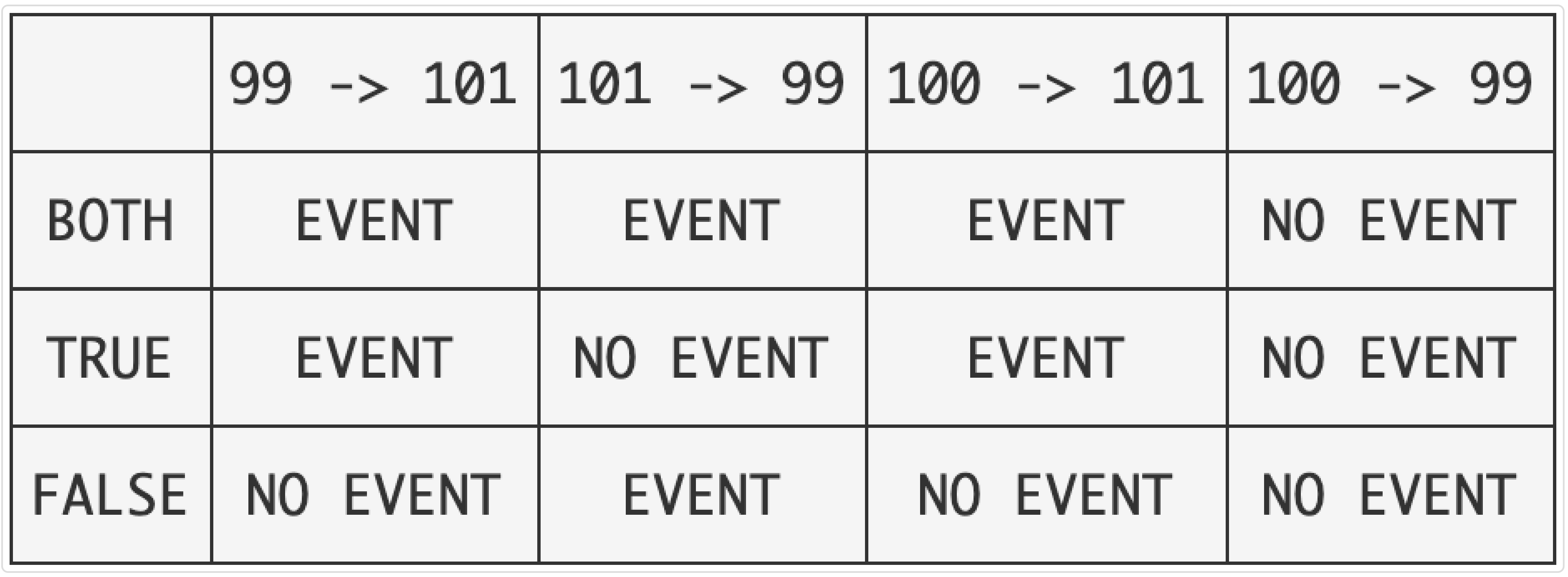
- **condition** – A VAIL boolean expression that expresses the threshold condition. The condition can reference any property of the inbound event like `event.propA > event.someOtherProp`.

Threshold also contains the following optional configuration properties:

- **initializeCondition** – two options
  - *VAIL Conditional Expression*: A VAIL expression, which evaluates to a boolean, that will be run the first time the Threshold task executes to determine initial state.
  - *Visual Filter*: A visual interface that allows users to create a condition using a series of dropdowns and text fields. The visual filter is a more user-friendly way to create a condition.

If the initializeCondition is not specified, the first event will never trigger a threshold crossing.

- **withinDuration** – A duration that indicates how far apart two sequential events can be and still trigger the threshold. If two events occur further apart, they will not trigger the threshold.
- **direction** – One of these values: *true*, *false*, or *both*. Used to specify the boolean value that the condition must evaluate to after crossing the threshold in order to trigger a threshold event. For instance, if it only matters when a temperature reading goes over 100, but not when it goes back under, use a direction value of *true* along with the condition `event.temp > 100`. If the temp was 101 in the previous event, and a new event occurred with a temp value of 90, the evaluation of the condition has changed from true to false, which is the false direction, so the new event would not trip the threshold. A direction value of *both* indicates that crossing the threshold in either direction will emit an event. For some more examples given the condition `event.temp > 100`, consider the following table, where the temperature changes from x -> y over two sequential events:



	99 -> 101	101 -> 99	100 -> 101	100 -> 99
BOTH	EVENT	EVENT	EVENT	NO EVENT
TRUE	EVENT	NO EVENT	EVENT	NO EVENT
FALSE	NO EVENT	EVENT	NO EVENT	NO EVENT

- **overConsecutiveReadings** – Optional property used to indicate a number of sequential reading which all must evaluate to the same condition value in order to trip the threshold. For instance (using the same temp condition), if overConsecutiveReadings is 4, and the temp reading goes from 90, to 101, to 104, and then back to 98, that would not trip the threshold because only two readings were above 100.
- **imports** – A list of resources referenced by the *condition* or *initializeCondition* expressions that must be imported.

The Threshold activity pattern produces the following event streams:

- **event** (default) – the event on which the “threshold” condition is crossed.
- **reset** – the event that triggers a ‘reset’ of threshold checking.

Time Difference

TimeDifference keeps track of the time between two events. It adds the time since the previous event as well as the entire previous event to the current event. No event is output for the very first inbound event. The output would appear as follows, assuming no previous events.

Inbound:

```
{
    "property": "someValue",
    "order": 1
}
```

Outbound:

Nothing, since there was no previous event.

3 seconds later:

Inbound:

```
{
    "property": "anotherValue",
    "order": 2
}
```

Outbound:

```
{
    "property": "anotherValue",
    "order": 2,
    "timingData": {
        "timeDifference": 3000,
        "prevEvent": {
            "property": "someValue",
            "order": 1
        }
    }
}
```

1.5 seconds later:

Inbound:

```
{
    "order": 3,
    "key": "lock"
}
```

Outbound:

```
{
    "order": 3,
    "key": "lock",
    "timingData": {
        "timeDifference": 1500,
        "prevEvent": {
            "property": "anotherValue",
            "order": 2
        }
    }
}
```

The TimeDifference activity has only one required property:

- **outboundProperty** – The property name into which the time difference will be placed. The time difference in milliseconds is placed in `<outboundProperty>.timeDifference`, and the previous event is placed in `<outboundProperty>.prevEvent`.

Track

The Track activity pattern tracks the location of one or more Vantiq users using the Vantiq Mobile App.

The Track activity pattern has the following required configuration properties:

- **users** (Union) – An array of valid Vantiq usernames that will be tracked. This may either be an actual array of vail expressions or a single vail expression which will evaluate to an array (for example: *event.userList*).

The Track activity pattern has the following optional configuration properties:

- **imports** – A list of resources to be imported for this activity.
- **destination** (Union: GeoJSON or VAIL Expression) – The location to track the user to. Once the user arrives at the destination, the user tracking will terminate and produce an *arrival* event. If not specified the users will be tracked until the collaboration ends.
- **level** (Enumerated) – A string defining the granularity of location measurements. Can be either 'fine' or 'coarse'. Default is 'fine', which is more precise than 'coarse'.
- **distanceFilter** (Real) – A number in meters representing the minimum distance traveled to produce a new location reading. Default is 200 meters.
- **desiredAccuracy** (Real) – A number in meters representing the accuracy of location measurements. Default is 100 meters.
- **destinationRadius** (Real) – The maximum distance the tracked user can be from the destination to trigger the arrival event. Default is 500 meters.
- **disclosure** (String) – A message to display on the target user's phone to describe why location tracking features must be enabled on their mobile device.
- **pingInterval** – An interval constant that indicates how often to ping the tracked users phone for a location update if none is received. For example: '30 minutes' or '1 hour'.
- **waypoints** (Union: Array of ArsWaypoints or VAIL Expression) – Places to note during the journey. When the user arrives within *radius* of the waypoint a *waypointArrival* event is emitted. If the user was previously at a waypoint and is no longer with *radius* of the waypoint, a *waypointDeparture* event is emitted.
    - *name* – the ame of the waypoint.
    - *radius* – the radius around the waypoint that may considered arriving at the waypoint.
    - *waypoint* – The GeoJSON location of the waypoint.

Track produces the following event streams:

- **arrival** (default) – The event that occurs when a user arrives at the *destination*.
    - *username*: Vantiq username for the tracked user.
    - *destination*: GeoJSON location for the destination.
- **firstArrival** – The event arrives when the first tracked user arrives at the destination.
    - *username*: name of the user that arrived at the destination.
    - *destination*: GeoJSON location for the destination.
- **waypointArrival** – The event that is emitted when a user enters within the configured *radius* of a particular *waypoint*.
    - *username*: Vantiq username for the tracked user.
    - *waypointName*: name of the waypoint.
    - *waypoint*: GeoJSON location for the waypoint.
    - *collaboration*: the unique id for this collaboration instance.
- **waypointDeparture** – event that is emitted when a user that was previous at a *waypoint* is no longer within the *radius* for that waypoint.
    - *username*: Vantiq username for the tracked user.
    - *waypointName*: name of the waypoint.
    - *waypoint*: GeoJSON location for the waypoint.
    - *collaboration*: the unique id for this collaboration instance.
- **update** – The event that is emitted any time there is a location update.

    - *username*: Vantiq username for the tracked user.
    - *location*: GeoJSON location of the user.
    - *lastActive*: the timestamp for this location update.
    - *collaborationId*: the unique id for this collaboration instance.
- **event** – Emits the original incoming event.

The collaboration instance is updated to the following:

```
{
  "id": "<collaboration id>",
  "status": "active",
  "name": "<app name>",
  "results": {
    "<track task name>": {
      "arrivedUsers": "<array of username of users that have arrived at the destination>",
      "firstUser": "<username of the first user to arrive at the destination>"
    }
  }
}
```

Track Motion

Track the motion of *things with locations*. Here, *things with locations* means events whose content looks similar to the output of YOLO From Images or YOLO From Documents activities (or, since it is copied through, that of the Convert Coordinates activity). Specifically, events which have something that can be considered a `label`, and a `location` property that contains the information about the bounding box surrounding the object. The term *things with locations* emphasizes that the output could come from some different activity.

Tracking motion maintains a collection of tracked objects. For each new object coming in, it finds the best match based on the matching algorithm and motion threshold chosen. To match, the two objects must have the same label (*e.g.* ‘car’). Candidates are chosen based on label match and best fit, based on the motion tracking algorithm chosen. If no suitable match is found, add a new object to the set of tracked objects. Objects missing for a while are droppped.

Motion can be determined by either the *centroid* or *bounding box* algorithm. As noted, the labels of the two objects must be the same for either algorithm.

The *centroid* algorithm compares two positions using distance from the center of the object’s old position to the center of the object’s new position to determine distance. For this algorithm, the *motion tracking threshold* is the maximum distance permitted. For example, if the threshold is 10 and a candidate object has a distance of 20, then that new object cannot be considered movement for the first object. Objects under the threshold are possible matches, and smallest distance is considered the best match.

The *bounding box* algorithm compares the percentage of overlap between the two bounding boxes. For this algorithm, the *motion tracking threshold* is the minimum percentage overlap required to be considered motion. Objects over the threshold are considered possible matches, and the best match is that with the largest overlap percentage.

The TrackMotion Activity Pattern emits an object consisting of the set of tracked objects (`trackedObjects`) and the set of objects dropped (`droppedObjects`).

For each tracked object, TrackMotion updates *location* with the following properties:

- `trackingId` – the identity of the tracked object (used in Build Path)
- `delta` – the value used to determine motion. The value depends on the algorithm employed. This value may be missing when an object first appears. Since it has no previous position, no `delta` is applicable.
- `timeOfObservation` – the time ascribed to this observation of position.

For more information, please see the Motion Tracking section of the Image Processing Guide.

The TrackMotion Activity Pattern requires the following configuration properties:

- **maximumAbsentBeforeMissing** – The interval (*e.g.* ‘20 seconds’) after which a tracked object is dropped.
- **motionDetectionAlgorithm** – The algorithm to use for motion determination. The choices are ‘centroid’ and ‘bounding box’.
- **motionThreshold** – A VAIL expression specifying the threshold value used for determining whether one position may be considered “motion” from another.
- **outboundProperty** – The name of the property into which to place the output of this activity.

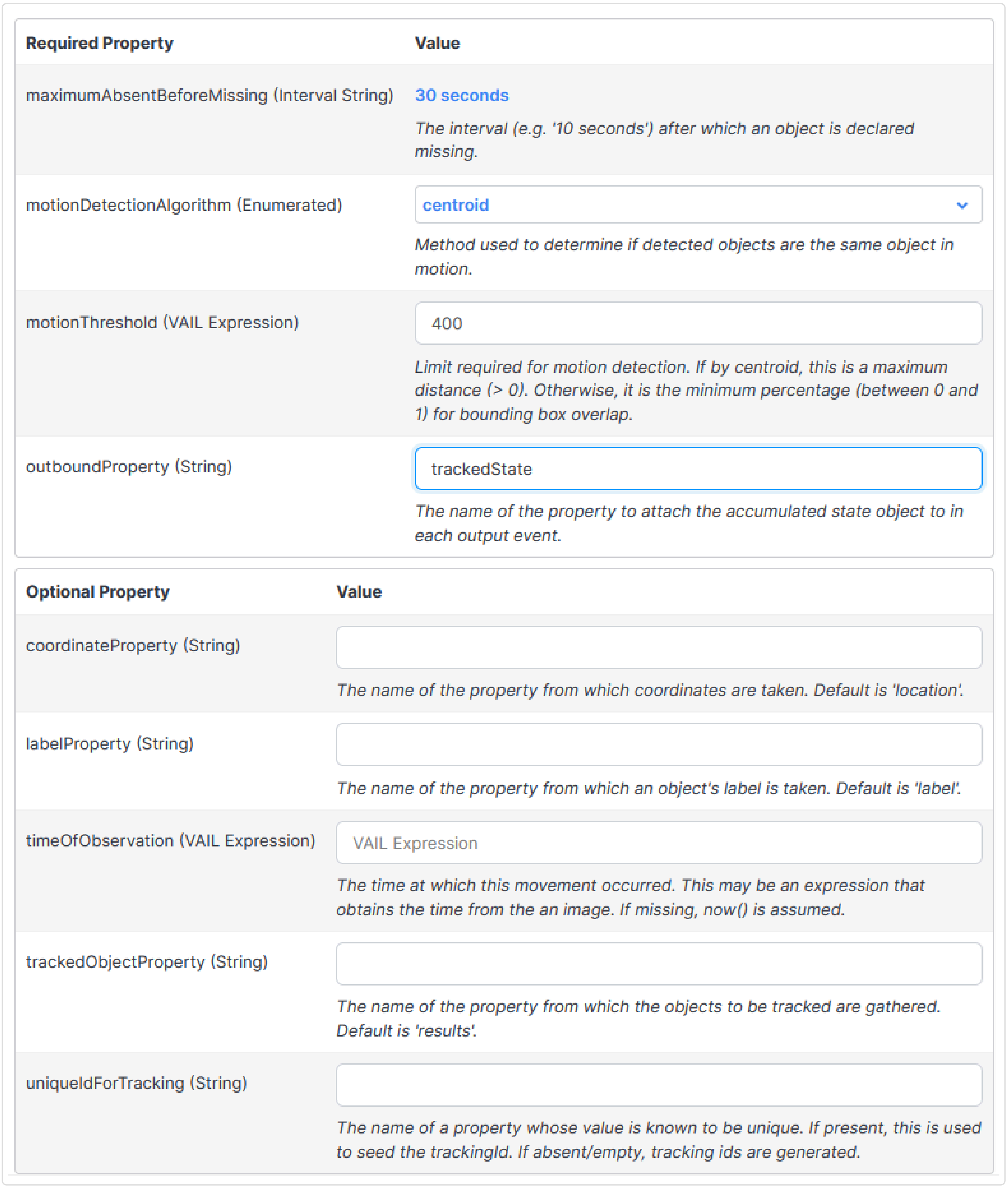
The following optional properties can also be provided:

- **coordinateProperty** – The name of the property from which to glean location. The default value is `location`.
- **labelProperty** – The name of the property from which to determine the object’s label. The default value is `label`.
- **timeOfObservation** – A VAIL expression to determine the time that tracking this object took place. The default value is `now()`.
- **trackedObjectProperty** – The name of the property from which to gather the list of objects to be tracked. The default value is `results`.
- **uniqueIdForTracking** – The name of a property whose value is known to be unique. If provided, the value from this property is used for the tracking id. If not provided (or is missing in some instance), the tracking id is generated.

The following specification adds a TrackMotion activity that

- drops objects missing for 30 seconds,

- tracks using the *centroid* algorithm,
- has a motion threshold of 400, and
- places the output in a property called `trackedState`.



Required Property	Value
maximumAbsentBeforeMissing (Interval String)	30 seconds *The interval (e.g. '10 seconds') after which an object is declared missing.*
motionDetectionAlgorithm (Enumerated)	centroid *Method used to determine if detected objects are the same object in motion.*
motionThreshold (VAIL Expression)	400 *Limit required for motion detection. If by centroid, this is a maximum distance (> 0). Otherwise, it is the minimum percentage (between 0 and 1) for bounding box overlap.*
outboundProperty (String)	trackedState *The name of the property to attach the accumulated state object to in each output event.*

Optional Property	Value
coordinateProperty (String)	*The name of the property from which coordinates are taken. Default is 'location'.*
labelProperty (String)	*The name of the property from which an object's label is taken. Default is 'label'.*
timeOfObservation (VAIL Expression)	VAIL Expression *The time at which this movement occurred. This may be an expression that obtains the time from the an image. If missing, now() is assumed.*
trackedObjectProperty (String)	*The name of the property from which the objects to be tracked are gathered. Default is 'results'.*
uniqueIdForTracking (String)	*The name of a property whose value is known to be unique. If present, this is used to seed the trackingId. If absent/empty, tracking ids are generated.*

Assuming this activity has operated on input containing some traffic, the output would look something like the following:

```
{
...
"trackedState":
        {
            "trackedObjects":
                [
                    {
                        "confidence": 0.72626966,
                        "location": {
                            "centerY": 704.9997,
                            "top": 616.2423,
                            "left": 481.8101,
                            "centerX": 677.0202,
                            "bottom": 793.7571,
                            "right": 872.23035,
                            "trackingId": "b78122a9-5bb9-46d3-bc22-79d39b6593e6",
                            "delta": 227.29758889770193,
                            "timeOfObservation": "2020-07-08T22:56:36.407Z"
                        },
                        "label": "car"
                    },
                    {
                        "confidence": 0.835676,
                        "location": {
                            "centerY": 571.4264,
                            "top": 560.4043,
                            "left": 1072.1838,
                            "centerX": 1084.5463,
                            "bottom": 582.4484,
                            "right": 1096.9087,
                            "trackingId": "ae0a620c-1e51-4023-acea-676bfd122128",
                            "delta": 2.9236888100580316,
                            "timeOfObservation": "2020-07-08T22:56:36.407Z"
                        },
                        "label": "car"
                    },
                    {
                        "confidence": 0.73007876,
                            "location": {
                                "centerY": 617.5767,
                                "top": 589.7673,
                                "left": 1228.4099,
                                "centerX": 1270.2803,
                                "bottom": 645.3862,
                                "right": 1312.1504,
                                "trackingId": "806967c6-b71e-463f-a60c-f7014b0dc33f",
                                "delta": 108.06990195156179,
                                "timeOfObservation": "2020-07-08T22:56:36.407Z"
                            },
                            "label": "car"
                        },
                        {
                            "confidence": 0.56739765,
                            "location": {
                                "centerY": 613.61835,
                                "top": 586.95667,
                                "left": 1224.4358,
                                "centerX": 1260.2947,
                                "bottom": 640.28,
                                "right": 1296.1533,
                                "trackingId": "56976308-83aa-4ed0-8ae6-38f4deddcb67",
                                "timeOfObservation": "2020-07-08T22:55:55.356Z"
                            },
                            "label": "truck"
                        },
                        {
                            "confidence": 0.7020074,
                            "location": {
                                "centerY": 585.1692,
                                "top": 575.26105,
                                "left": 1148.3091,
                                "centerX": 1167.5013,
                                "bottom": 595.0774,
                                "right": 1186.6938,
                                "trackingId": "266f50dc-cd4d-4655-bdf1-daf10e6d4526",
                                "timeOfObservation": "2020-07-08T22:56:36.407Z"
                            },
                            "label": "car"
                        },
                        {
                            "confidence": 0.5984039,
                            "location": {
                                "centerY": 590.17426,
                                "top": 580.5335,
                                "left": 1186.8556,
                                "centerX": 1200.7151,
                                "bottom": 599.81506,
                                "right": 1214.5747,
                                "trackingId": "ba2161d9-2cd7-4f16-af32-d3967091c596",
                                "timeOfObservation": "2020-07-08T22:56:36.407Z"
                            },
                        "label": "car"
                }
            ],
            "droppedObjects": []
        }
...
}
```

In this example, we see that there are three cars (the first three objects) that have been tracked before (they have a `delta` value), and three more (the last three – a truck and two cars) that have just been added (no `delta` value).

We do not see any dropped objects. Were any dropped objects present, their format would be the same as those in the `trackedObjects` list. Once an object is dropped, it will no longer appear in subsequent TrackMotion output.

Transformation

The Transformation task allows users to inject or create their own procedure in a VEH. The transformation procedure is expected to take a single object (the inbound event) as input, and return the mutated outbound event. In between the procedure can do whatever the user wants, including produce events that trigger other event streams in the handler (for instance by publishing to a topic). Returning *null* from the transformation procedure indicates the inbound event should be filtered out, in which case the transformation task will produce no outbound event.

The Transformation Activity Pattern contains only one required configuration property:

- **transformation** – three options:
    - *Transformation Procedure*: The procedure to execute as the transformation procedure. As mentioned above, the procedure must take one parameter and should return the transformed event.
    - *Visual Transformation*: A visual interface that allows users to map inputs to outputs without having to write their own Transformation Procedure. The visual transformation is a more user-friendly way to map inputs to outputs.
    - *Projection*: A VAIL expression that projects a property from inbound event into the outbound event. This is a simple way to transform a property from the inbound event into the outbound event without any additional logic.

The Transformation Activity Pattern has the following optional configuration properties:

- **imports** – A list of resources referenced by the *Visual Transformation* that must be imported.
- **transformInPlace** – if true, start with the inbound event and apply the transformation only to the properties specified by the transformation object. The remaining properties will be unchanged.
- **schema** – The name of a type that defines the schema of the output of the outbound state. If none is provided, the output schema will be assumed to match the input schema. Specifying the schema here will simplify the configuration of downstream tasks.

Unwind

When an inbound event contains an array property or is itself an array, it is sometimes helpful to process the entries within the array independently. The Unwind Activity Pattern allows users to do just that. For example, the inbound event:

```
{
    "name": "joe",
    "heartrates": [
        {"ts": "2017-09-07T23:45:13.573Z", "value": 60},
        {"ts": "2017-09-07T23:46:14.512Z", "value": 65},
        {"ts": "2017-09-07T23:51:11.513Z", "value": 90}
    ]
}
```

contains an array of heartrates that were recorded at different times. Using the unwind activity, it would be possible to turn this one event into three sequential events like:

```
{
    "name": "joe",
    "rate": {"ts": "2017-09-07T23:45:13.573Z", "value": 60}
},
{
    "name": "joe",
    "rate": {"ts": "2017-09-07T23:46:14.512Z", "value": 65}
},
{
    "name": "joe",
    "rate": {"ts": "2017-09-07T23:51:11.513Z", "value": 90}
}
```

each of which would be processed by downstream tasks independently.

Sometimes it is better to place the properties of each object in the unwound array directly into the top level of the outbound object. For the same inbound event, an unspecified outbound property would produce the three sequential events:

```
{
    "name": "joe",
    "ts": "2017-09-07T23:45:13.573Z",
    "value": 60
},
{
    "name": "joe",
    "ts": "2017-09-07T23:46:14.512Z"
    "value": 65
},
{
    "name": "joe",
    "ts": "2017-09-07T23:51:11.513Z"
    "value": 90
}
```

For arrays, the inbound event:

```
[
    {
        "user": "john",
        "message": "I need a little help."
    },
    {
        "user": "jane",
        "message": "Sure, what do you need?"
    }
]
```

contains a few messages from a conversation. The event would be separated into two sequential events:

```
{
    "user": "john",
    "message": "I need a little help."
},
{
    "user": "jane",
    "message": "Sure, what do you need?"
}
```

The Unwind Activity Pattern contains the following configuration properties:

- **unwindProperty** – The name of the property on the inbound event that contains the array of values to be unwound. If unspecified, the entire inbound event will be assumed to be the array to be unwound. In the first and second examples the unwindProperty would be "heartrates", and in the third example it would be unspecified.
- **outboundProperty** – The name of the property in the outbound event which will contain the unwound value. If unspecified, an inbound array will turn each element into an outbound event, and an inbound object will move the properties of each object in the unwound array directly into the top level of the outbound event. In the first example the outboundProperty would be "rate", and in the second and third examples it would be unspecified.

VAIL

The VAIL Activity Pattern runs an arbitrary snippet of VAIL code for every event, making just about any transformation possible. The VAIL snippet will find the inbound event data inside of `event.value` and any modifications made to `event.value` will be seen by downstream tasks.

It's possible to publish events to topics or sources, or even run database queries inside a VAIL block, but keep in mind that this can come at a cost to performance.

The VAIL Activity Pattern has only one required configuration property:

- **VAIL** – A block of VAIL code to execute for every inbound event.

The VAIL Activity Pattern has the following optional configuration properties:

- **imports** – A list of resources referenced by the VAIL block that must be imported.
- **schema** – The name of a type that describes the schema of the output event. If none is provided, the output schema will be assumed to match the input schema. Specifying the schema will simplify the configuration of downstream tasks.
- **useEventValue** – when true, the *event* object will contain the value published by the upstream task. This is equivalent to setting `event = event.value`. If false, the *event* object will contain the Vantiq Event object with its additional event metadata. This option is useful when you only need simple operations such as updating a property. On the other hand, when you need complicated operations like replacing the entire object, you should set it to false. The default value is `true`.

- **redeliverOnError** – The name of a procedure used to determine if a reliable event should be redelivered after an error. If procedure returns `true`, then the error is considered transient and will be redelivered. The procedure will be provided with 2 parameters, the original event and the exception that occurred.

The VAIL activity pattern produces the following event streams:

- **event** (default) – the results of all successful requests.
- **onError** – an event providing context for any failed request. The properties of this event are:
  - **errorCode** (String) – a unique "code" associated with the generated exception.
  - **errorMessage** (String) – the text of the error message for the generated exception.
  - **errorParams** (String[]) – the parameters used to replace substitution values in the error message.
  - **triggeringEvent** (Object) – the event that triggered the exception.

VisionScript

The VisionScript Activity Pattern allows users to easily build *Vision Scripts*, which can be used to manipulate images. *Vision Scripts* are explained in more detail as part of the Image Processing Guide, including a full list of the possible script actions and their respective configuration properties.

The script comprises a sequence of *actions*, where each action has a name, optionally a tag, and optionally some parameters. The *name* identifies the action to be performed, and the *parameters*, if any, provide details about the action to be performed. The *tag* allows a reference to that action by subsequent actions in the same script.

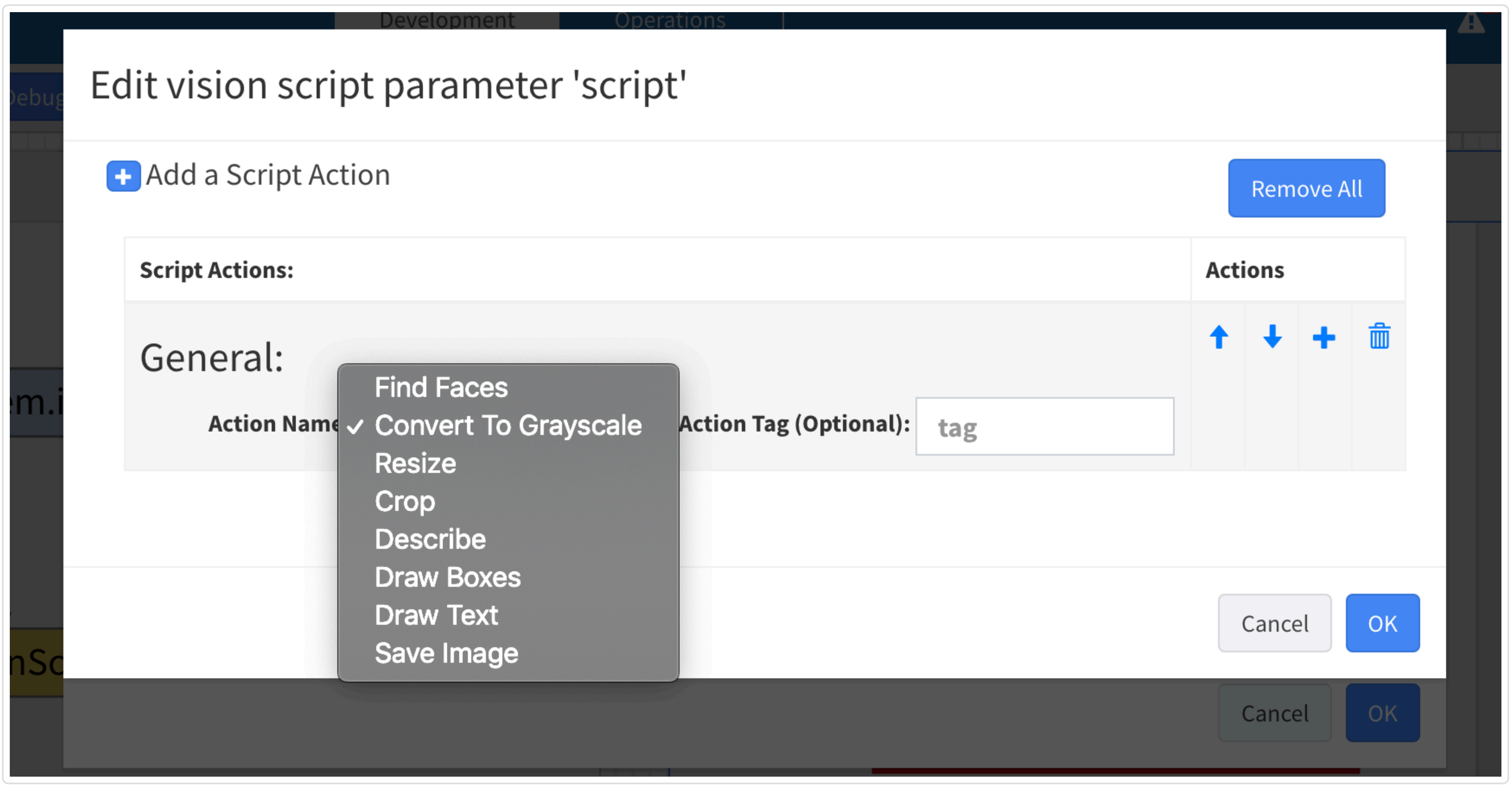
There are two configuration properties for this task:

- **script** – Required. The array of actions that compromise the *Vision Script*.
- **scriptName** – Optional. The name of the script to be run, (possibly useful for debugging).

This Activity Pattern employs a unique interface in order to select and configure the actions that will make up the *Vision Script*.

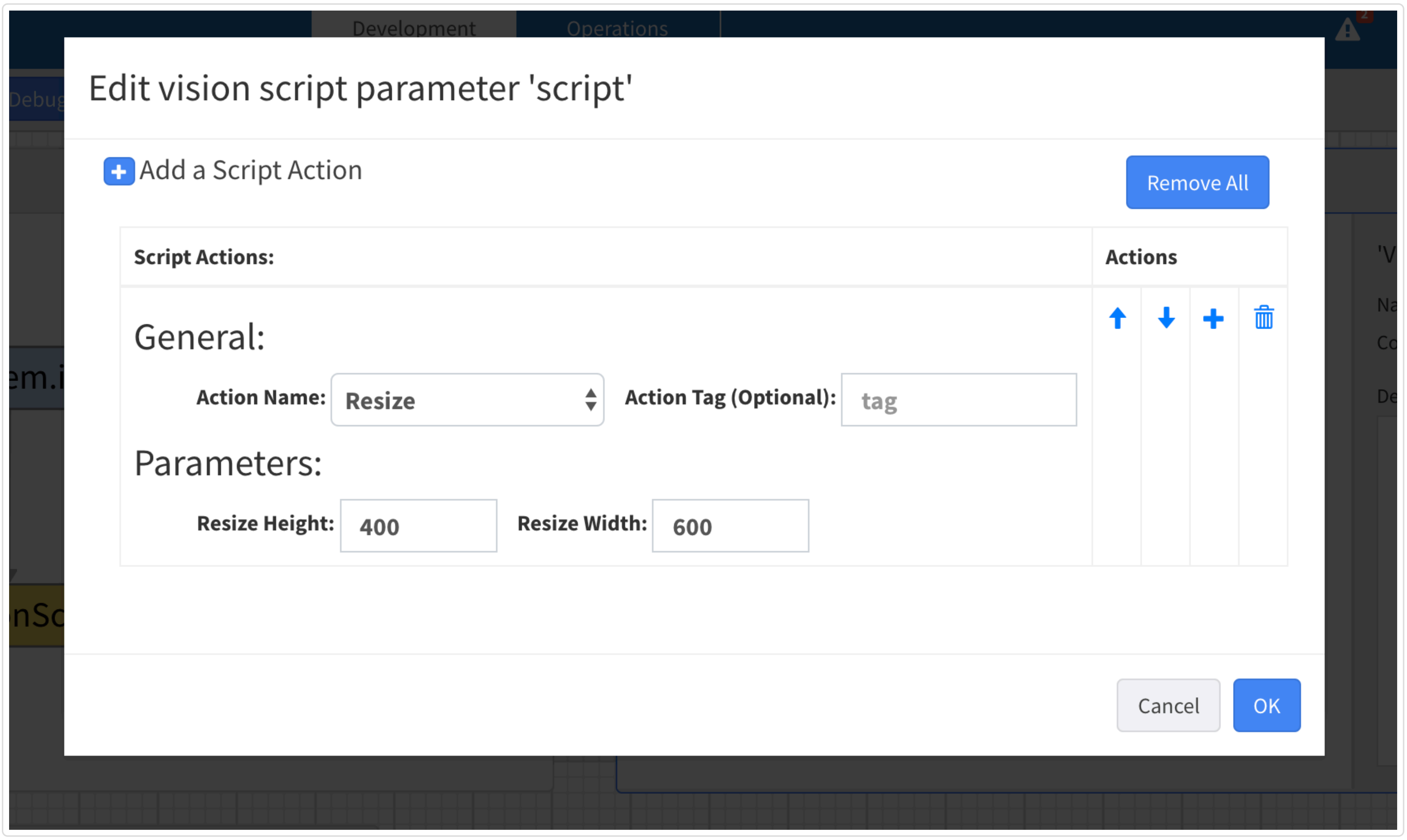
The following screenshots detail the custom interface used to add actions to script.

A dropdown menu is used to select which action to add.



Once the action is selected, a pane will appear allowing you to provide the appropriate parameters (if any), and a tag if desired.

The following image shows how to configure the `Resize` action. Some actions will not have any additional parameters, such as `Convert To Grayscale`, `Find Faces` and `Describe`. However, **all of the actions** will offer the option to add an Action Tag.


Edit vision script parameter 'script'

Add a Script Action

Script Actions:

General:

Action Name: Resize

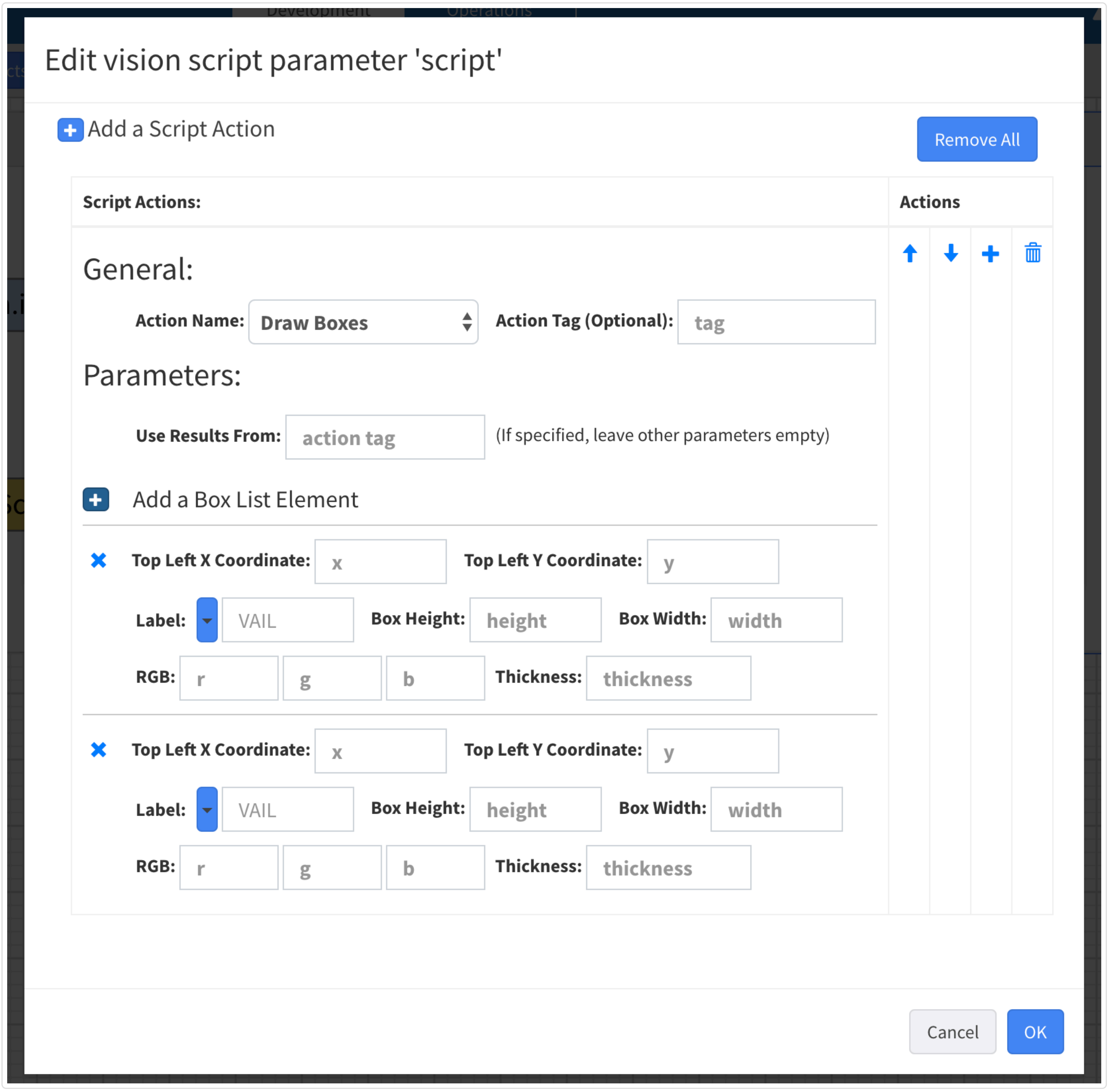
Action Tag (Optional): tag

Parameters:

Resize Height: 400

Resize Width: 600

The `Draw Boxes` action can be configured using a “Box List”, which is made up of multiple “Box Elements”, or by specifying the `Use Results From` field. It should be noted that you **cannot** configure this action using both options. The `Use Results From` field accepts the `Action Tag` of a previous action, (typically a `Find Faces` action), and will use that action’s results to draw the boxes. You can also manually configure each box to be drawn, instead of using results from a previous action. After pressing the “Add a Box List Element” button, you will be prompted to fill out the additional parameters for each box. The following image shows two “Box Elements” that need to be configured.



Edit vision script parameter 'script'

Add a Script Action

Remove All

Script Actions:	Actions

General:

Action Name: Draw Boxes **Action Tag (Optional):** tag

Parameters:

Use Results From: action tag (If specified, leave other parameters empty)

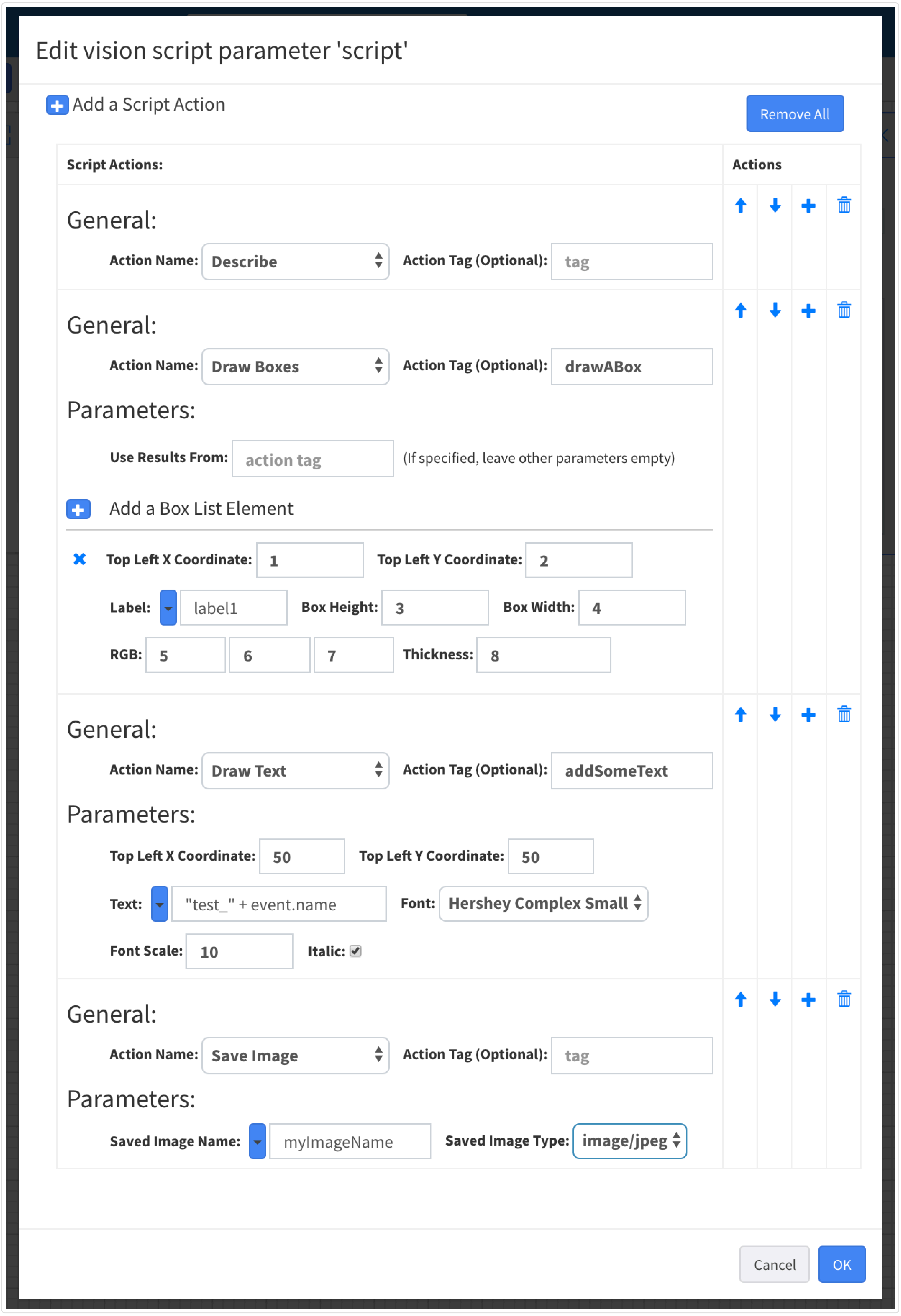
Add a Box List Element

Top Left X Coordinate: x **Top Left Y Coordinate:** y
Label: VAIL **Box Height:** height **Box Width:** width
RGB: r g b **Thickness:** thickness

Top Left X Coordinate: x **Top Left Y Coordinate:** y
Label: VAIL **Box Height:** height **Box Width:** width
RGB: r g b **Thickness:** thickness

Cancel OK

The following image shows what a list of configured actions might look like.



Edit vision script parameter 'script'

Add a Script Action

Remove All

Script Actions:	Actions
General: Action Name: Describe; Action Tag (Optional): tag	
General: Action Name: Draw Boxes; Action Tag (Optional): drawABox. **Parameters:** Use Results From: action tag (If specified, leave other parameters empty). Add a Box List Element. Top Left X Coordinate: 1; Top Left Y Coordinate: 2; Label: label1; Box Height: 3; Box Width: 4; RGB: 5, 6, 7; Thickness: 8	
General: Action Name: Draw Text; Action Tag (Optional): addSomeText. **Parameters:** Top Left X Coordinate: 50; Top Left Y Coordinate: 50; Text: "test_" + event.name; Font: Hershey Complex Small; Font Scale: 10; Italic: ✓	
General: Action Name: Save Image; Action Tag (Optional): tag. **Parameters:** Saved Image Name: myImageName; Saved Image Type: image/jpeg	

Cancel OK

Once the action list has been configured, you can save it by pressing the “OK” button. Using the configuration in the previous image would generate an array of actions that looks similar to the following:

```
[
  {
    "name": "describe"
  },
  {
    "name": "drawBoxes",
    "parameters": {
      "boxList": [
        {
          "x": 1,
          "y": 2,
          "height": 3,
          "width": 4,
          "label": "label1",
          "color": {
            "red": 5,
            "green": 6,
            "blue": 7
          },
          "thickness": 8
        }
      ]
    },
    "tag": "drawABox"
  },
  {
    "name": "drawText",
    "parameters": {
      "text": "\"test_\" + event.name",
      "x": 50,
      "y": 50,
      "font": 6,
      "fontScale": 10,
      "isItalic": true
    },
    "tag": "addSomeText"
  },

  {
    "name": "save",
    "parameters": {
      "saveName": "myImageName",
      "fileType": "image/jpeg"
    }
  }
]
```

After defining the action list and saving the app, the *Vision Script* will automatically be generated.

Window

Window collects inbound events and produces outbound events that contain a list of the previous inbound events. The size of the window is configurable, within the range of 2-10 events. There are two variants: Discrete and Sliding. Discrete windows are the default, where each list of events in an outbound event contains no events from the previous window.

The Window Activity Pattern has one required configuration property:

- **count** – the number of events to emit in each window. No emissions will occur until there are at least count events.

There are also the following optional configuration properties:

- **slidingWindow** – An optional boolean flag. When `true`, indicates the output should be overlapping windows. When `false`, indicates each window should contain a unique batch of events, and the window should reset after each emission. See below for examples.

For example, consider the following 6 inbound events:

```
{
    "name": "joe",
    "temperature": 98
},
{
    "name": "joe",
    "temperature": 99
},
{
    "name": "joe",
    "temperature": 100
},
{
    "name": "joe",
    "temperature": 101
},
{
    "name": "joe",
    "temperature": 96
},
{
    "name": "joe",
    "temperature": 98
}
```

If the window count is set to 3, there would only be two events emitted:

```
{
    "values": [
        {
            "event": {
                "name": "joe",
                "temperature": 98
            },
            "timestamp": "2018-09-07T23:51:11.513Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 99
            },
            "timestamp": "2018-09-07T23:51:13.253Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 100
            },
            "timestamp": "2018-09-07T23:51:17.109Z"
        }
    ]
},
{
    "values": [
        {
            "event": {
                "name": "joe",
                "temperature": 101
            },
            "timestamp": "2018-09-07T23:51:19.464Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 96
            },
            "timestamp": "2018-09-07T23:51:22.143Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 98
            },
            "timestamp": "2018-09-07T23:51:26.112Z"
        }
    ]
}
```

Whereas if the task was configured to use a sliding window you would instead get 4 overlapping events:

```
{
    "values": [
        {
            "event": {
                "name": "joe",
                "temperature": 98
            },
            "timestamp": "2018-09-07T23:51:11.513Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 99
            },
            "timestamp": "2018-09-07T23:51:13.253Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 100
            },
            "timestamp": "2018-09-07T23:51:17.109Z"
        }
    ]
},
{
    "values": [
        {
            "event": {
                "name": "joe",
                "temperature": 99
            },
            "timestamp": "2018-09-07T23:51:13.253Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 100
            },
            "timestamp": "2018-09-07T23:51:17.109Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 101
            },
            "timestamp": "2018-09-07T23:51:19.464Z"
        }
    ]
},
{
    "values": [
        {
            "event": {
                "name": "joe",
                "temperature": 100
            },
            "timestamp": "2018-09-07T23:51:17.109Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 101
            },
            "timestamp": "2018-09-07T23:51:19.464Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 96
            },
            "timestamp": "2018-09-07T23:51:22.143Z"
        }
    ]
},
{
    "values": [
        {
            "event": {
                "name": "joe",
                "temperature": 101
            },
            "timestamp": "2018-09-07T23:51:19.464Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 96
            },
            "timestamp": "2018-09-07T23:51:22.143Z"
        },
        {
            "event": {
                "name": "joe",
                "temperature": 98
            },
            "timestamp": "2018-09-07T23:51:26.112Z"
        }
    ]
}
```

Within Tracking Region

The WithinTrackingRegion task filters the locations coming into it by their presence in a list of regions (any region in the list is sufficient). The filtering can be further specialized by saying to filter only locations with particular labels.

The region associated with a location can be found by either the centroid (coordinate of the center point of the bounding box) or by the bounding box itself (based on the percentage of the bounding box found in a region).

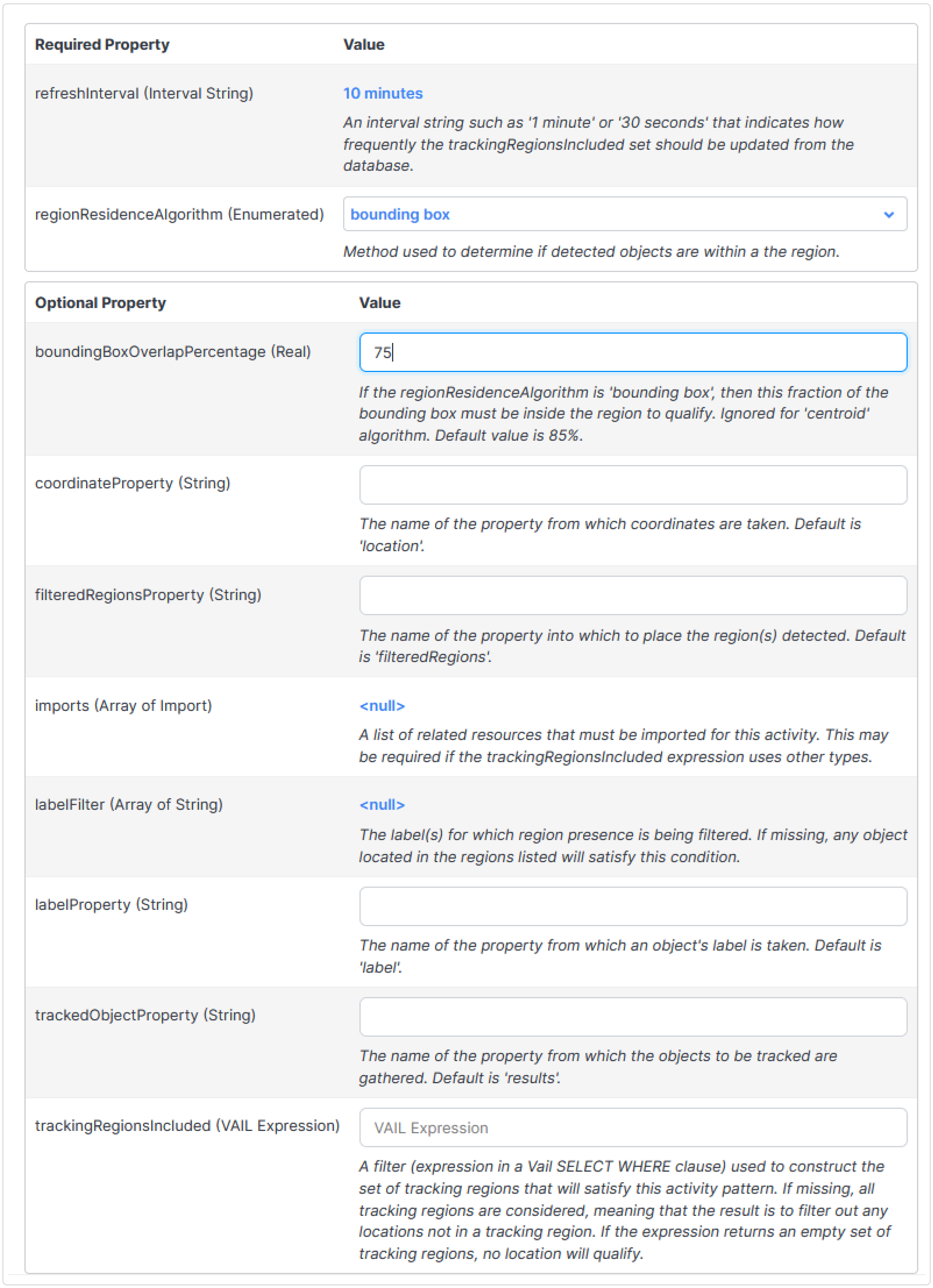
The combination of these allows the construction of apps or event handlers that can restrict the things they report to particular areas (and only objects of certain types (labels)). The set of regions to be considered can be specified and may be different than those reported by path prediction. This allows one to define, say, some set of regions of high security, then, in path prediction, report more detailed areas of (say) the secure building. Extending this with the labels in question, one could only look for, say, people in those areas.

The WithinTrackingRegion activity pattern requires the following configuration properties:

- **regionResidenceAlgorithm** – Method used to determine if an object is within a particular region. Values are `centroid` and `bounding box`.
- **refreshInterval** – an interval string (*e.g.*, 10 minutes) specifying how often the list of TrackingRegions should be refreshed. In a mature application where region definitions do not change, a large interval may be appropriate.

In addition, there are the following optional configuration properties:

- **trackingRegionsIncluded** – an expression that limits the set of `TrackingRegions` considered for the position’s tracking region. This expression will be added to the WHERE clause used to select the regions to consider. If absent or empty, all `TrackingRegions` in the namespace are considered. If the resulting query returns an empty set of `TrackingRegions`, no location will qualify. Please see the tracking regions information for more information.
- **boundingBoxOverlapPercentage** – if the `regionResidenceAlgorithm` is `bounding box`, this is the fraction of the bounding box that must be inside a region to qualify. The default value us 85%. If the `regionResidenceAlgorithm` is `centroid`, this value is ignored.
- **labelFilter** – the label(s) for which region presence is being filtered. If missing, any object located in the regions listed will satisfy this condition.
- **coordinateProperty** – The name of the property from which to glean location. The default value is `location`.
- **labelProperty** – The name of the property from which to determine the object’s label. The default value is `label`.
- **trackedObjectProperty** – The name of the property from which to gather the list of objects to be tracked. The default value is `results`.
- **filteredRegionsProperty** – The name of the property into which to place the region(s) detected. Default is ‘filteredRegions’.”
- **imports** – A list of resources referenced by the `trackingRegionsIncluded` that must be imported.



Required Property	Value
refreshInterval (Interval String)	10 minutes *An interval string such as '1 minute' or '30 seconds' that indicates how frequently the trackingRegionsIncluded set should be updated from the database.*
regionResidenceAlgorithm (Enumerated)	bounding box *Method used to determine if detected objects are within a the region.*

Optional Property	Value
boundingBoxOverlapPercentage (Real)	75 *If the regionResidenceAlgorithm is 'bounding box', then this fraction of the bounding box must be inside the region to qualify. Ignored for 'centroid' algorithm. Default value is 85%.*
coordinateProperty (String)	*The name of the property from which coordinates are taken. Default is 'location'.*
filteredRegionsProperty (String)	*The name of the property into which to place the region(s) detected. Default is 'filteredRegions'.*
imports (Array of Import)	<null> *A list of related resources that must be imported for this activity. This may be required if the trackingRegionsIncluded expression uses other types.*
labelFilter (Array of String)	<null> *The label(s) for which region presence is being filtered. If missing, any object located in the regions listed will satisfy this condition.*
labelProperty (String)	*The name of the property from which an object's label is taken. Default is 'label'.*
trackedObjectProperty (String)	*The name of the property from which the objects to be tracked are gathered. Default is 'results'.*
trackingRegionsIncluded (VAIL Expression)	VAIL Expression *A filter (expression in a Vail SELECT WHERE clause) used to construct the set of tracking regions that will satisfy this activity pattern. If missing, all tracking regions are considered, meaning that the result is to filter out any locations not in a tracking region. If the expression returns an empty set of tracking regions, no location will qualify.*

YOLO From Documents

See YOLO From Images. Used when the input event is a Document.

YOLO From Temp Images

See YOLO From Images. Used when the input event is from a video source.

YOLO From Images

TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

YOLOFromImages, YOLOFromTempImages, and YOLOFromDocuments tasks are used to perform analysis via a YOLO TensorFlow model. YOLOFromImages is used when the input event represents a `system.images` resource, YOLOFromDocuments is used when the input event represents a `document` resource, and YOLOFromTempImages is used when the input event represents the output of a video source. They are otherwise identical. This section will refer to YOLOFromImages.

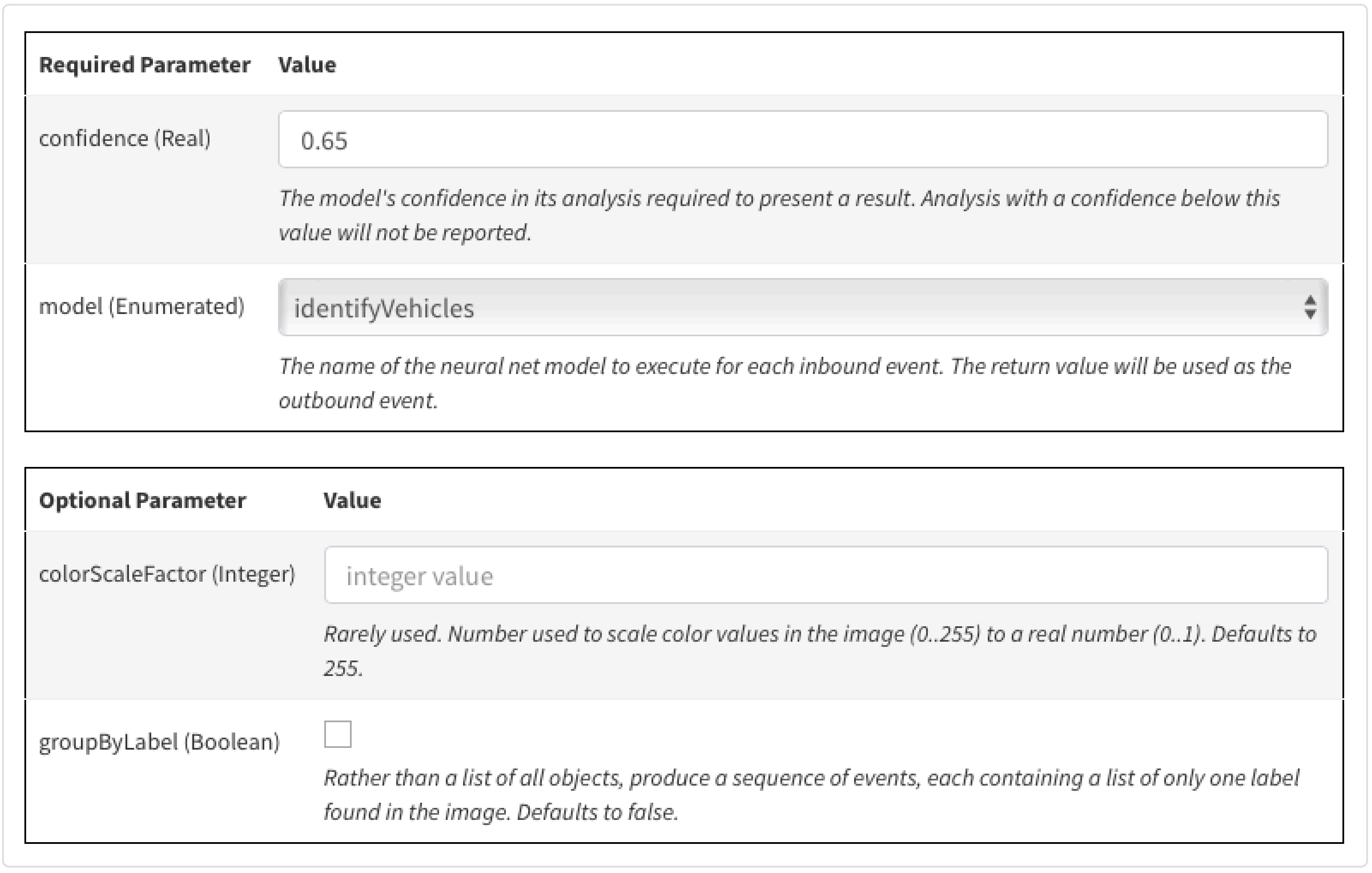
A typical use for YOLOFromImages is to identify objects or other patterns in an image. For example, images may be gathered from a camera overlooking some area. A YOLO TensorFlow model (named, say, *identifyVehicles*) may be used to find certain types of vehicles and other objects of interest in that area.

The YOLOFromImages Activity Pattern contains the following configuration properties:

- **confidence** – a real number between 0 and 1. This indicates the level of confidence in the result that the model must have to return an answer. For example, an entry of 0.65 indicates a 65% confidence in the result.
- **model** – a `tensorflowmodel` element, chosen from the list presented. This is the model used to perform the analysis. This model must be a YOLO model – a model of type `tensorflow/yolo`.

The two properties above are required. In addition, there are optional properties as follows:

- **colorScaleFactor** – an integer used to convert the colors represented in an image to numbers between 0 and 1 (for use by the model analysis). This is very rarely used.
- **groupByLabel** – a boolean (*i.e.* checkbox) indicating that the output should be a set of lists, each containing a single label. This can be used before a Split By Group Activity Pattern grouping by label.



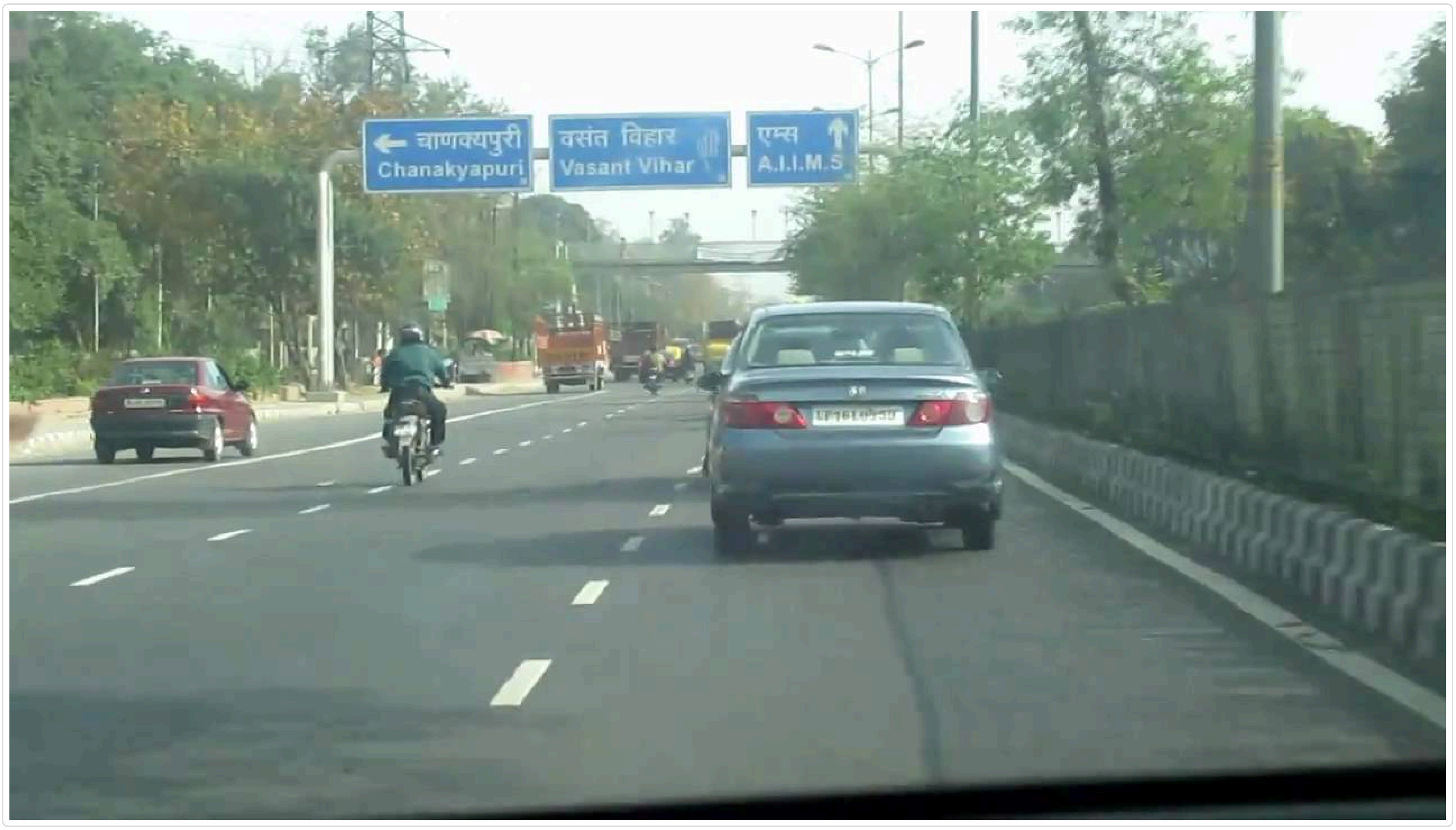
Required Parameter	Value
confidence (Real)	0.65 *The model's confidence in its analysis required to present a result. Analysis with a confidence below this value will not be reported.*
model (Enumerated)	identifyVehicles *The name of the neural net model to execute for each inbound event. The return value will be used as the outbound event.*

Optional Parameter	Value
colorScaleFactor (Integer)	integer value *Rarely used. Number used to scale color values in the image (0..255) to a real number (0..1). Defaults to 255.*
groupByLabel (Boolean)	☐ *Rather than a list of all objects, produce a sequence of events, each containing a list of only one label found in the image. Defaults to false.*

The output event from the YOLOFromImages (or YOLOFromDocument) task contains two properties:

- target – the image that was the input to the task
- results – the results of the analysis. The `results` property contains a list of boxes (possibly empty) identifying the name (label), location within the image, and confidence in the analysis provided.
  - confidence – the confidence in the result
  - location – coordinates of a box surrounding the identified object
  - label – the name of the object identified by the model.

For example, running the JPEG image `freeway.jpg`



through our *identifyVehicles* model might produce the following results.

```
{
    "target":{
        "name": "freeway.jpg",
        "fileType": "image/jpeg",
        "contentSize": 71631,
        ...
    },
    "results":
      [
        {
            "confidence": 0.791946,
            "location": {
                "centerY": 368.95956,
                "top": 259.94803,
                "left": 622.9274,
                "centerX": 760.1899,
                "bottom": 477.97113,
                "right": 897.4523
            },
            "label": "car"
        },
        {
            "confidence": 0.6838598,
            "location": {
                "centerY": 358.36145,
                "top": 294.93753,
                "left": 342.35565,
                "centerX": 373.64096,
                "bottom": 421.78534,
                "right": 404.92627
            },
            "label": "person"
        },
        {
            "confidence": 0.6590094,
            "location": {
                "centerY": 352.7839,
                "top": 306.68915,
                "left": 72.169304,
                "centerX": 147.08789,
                "bottom": 398.8787,
                "right": 222.0065
            },
            "label": "car"
        }
      ]
}
```

Note: we do not recommend photographing while driving.

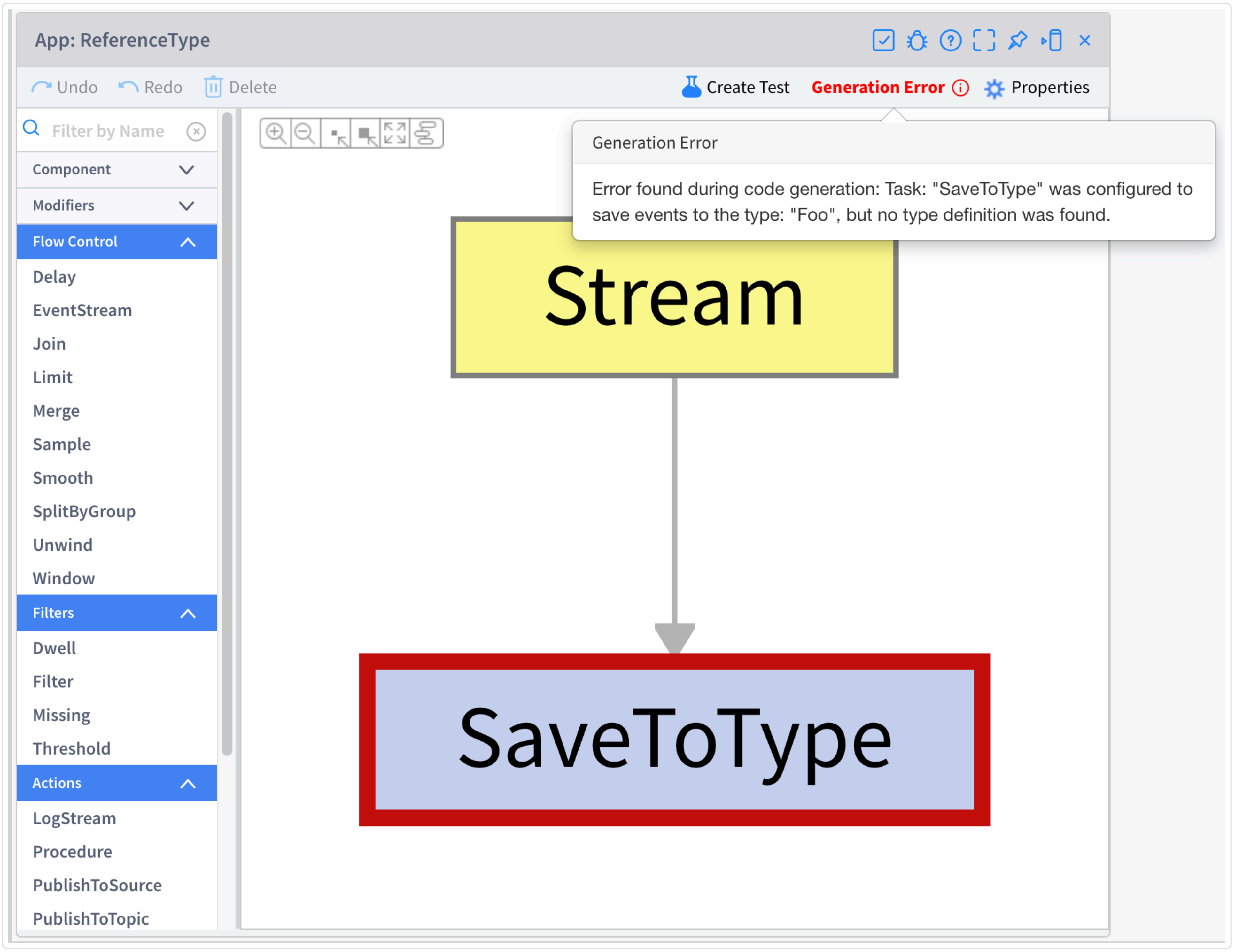
TensorFlow models are not provided as part of the Vantiq product.

Dependencies and Optional Imports

Dependency Management

Many Activity Patterns are configured with references to other Vantiq resources, and those references need to resolve for the VEH to work. All of these references should initially be validated when a service is saved and code is generated, however it's possible to change the related resources independently. Certain changes to referenced resources can trigger the deactivation of a running App if those changes result in invalidating the apps dependencies. For example, if an App references a procedure (correctly), and the procedure is later updated to change the number of parameters, the App will automatically be deactivated.

When a VEH is made inactive because of a dependency error an open App Builder pane will immediately show the new error by highlighting the offending task in red:



Generation Error

Error found during code generation: Task: "SaveToType" was configured to save events to the type: "Foo", but no type definition was found.

When this happens, the error can be seen by hovering over the task highlighted in red, or the red indicator in the menu bar at the top of the pane.

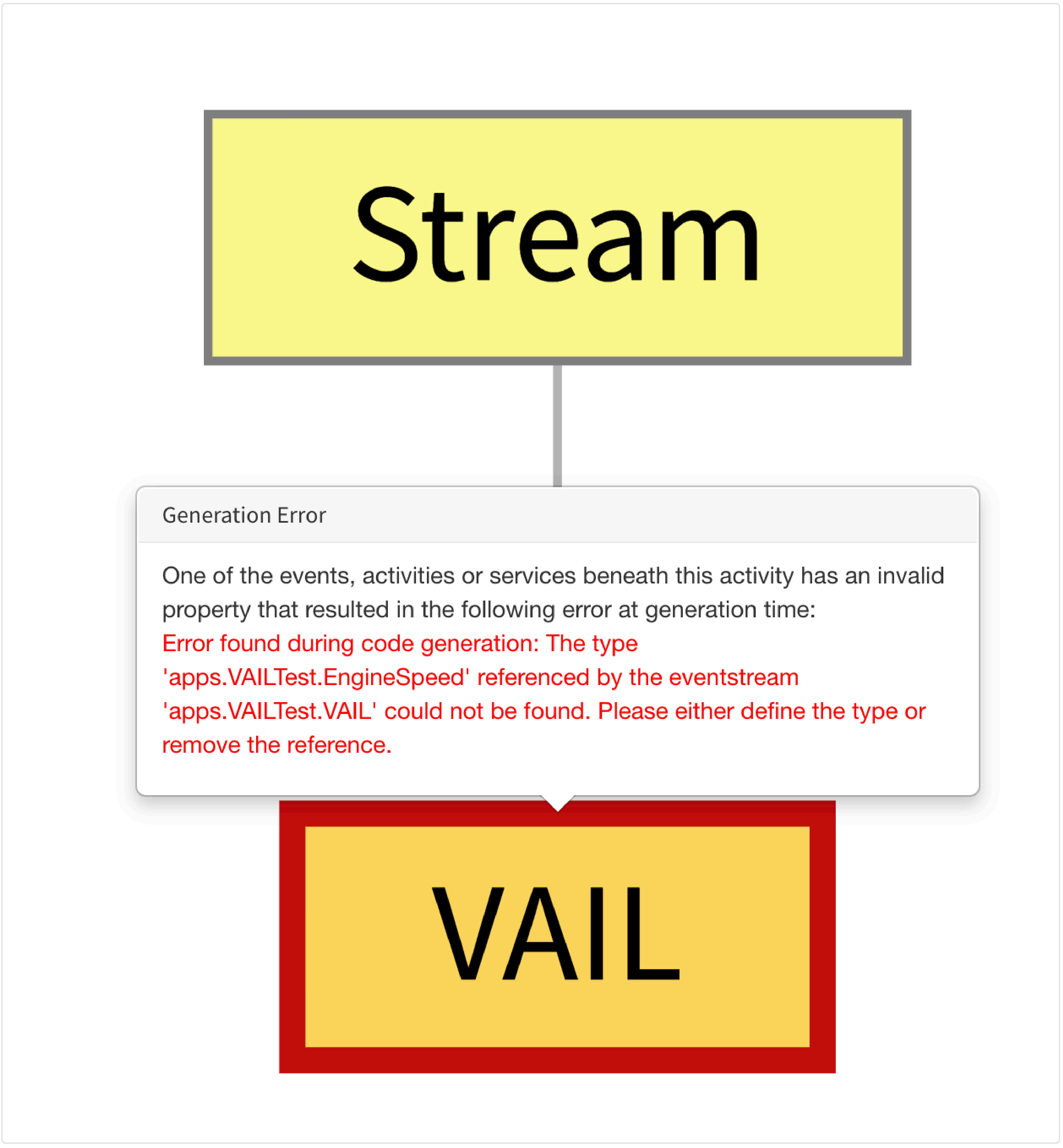
Optional Imports

Some dependencies cannot be inferred automatically by the system. In cases like the *VAIL* Activity Pattern it's possible to enter arbitrary VAIL that could reference any resources in any packages. These references need to be explicitly described in the optional `imports` property. An import can reference any of the following resources:

- services
- procedures (those not in services)
- types
- topics
- sources

> To access a service procedure, you must import the *service*. Importing individual service procedures is not supported.

Every resource that is referenced in a VAIL expression or code block needs to be included in the list of imports. If a resource is not properly imported, you will see an error indicating that the resource you referenced could not be found:



Generation Error

One of the events, activities or services beneath this activity has an invalid property that resulted in the following error at generation time:
Error found during code generation: The type 'apps.VAILTest.EngineSpeed' referenced by the eventstream 'apps.VAILTest.VAIL' could not be found. Please either define the type or remove the reference.