GenAI Builder Guide

Introduction

The Vantiq platform enables developers to leverage Generative AI (GenAI) technologies such as Large Language Models (LLMs) and Vector Databases as part of their Real-Time Business Applications. GenAI provides powerful reasoning and analytic capabilities and is a key element of the applications that power a modern enterprise. The features provided by the core Vantiq platform are designed to be general purpose and to enable developers to quickly add GenAI to their applications (using the SubmitPrompt and AnswerQuestion activity patterns). As a result, their behavior is relatively fixed and intended to address the broadest number of use cases. There are times however, when an application requires more specialized GenAI features or needs to leverage the newest GenAI algorithms. Addressing these requirements is the purpose of the *GenAI Builder*.

Using the GenAI Builder, a developer can describe a detailed GenAI Flow (we sometimes shorten this to just “Flow” in this document) using the Vantiq GenAI resources that they are already familiar with:

- Prompt Templates – expressed as text strings or stored in Vantiq Documents. Prompt Templates are used to format the inputs for both LLMs and Semantic Indexes.
- LLMs – the connection to a Large Language Model, along with the default configuration of that model.
- Semantic Indexes – a knowledge base which has been processed and stored in a vector database to facilitate similarity searches (which are a key element of RAG).

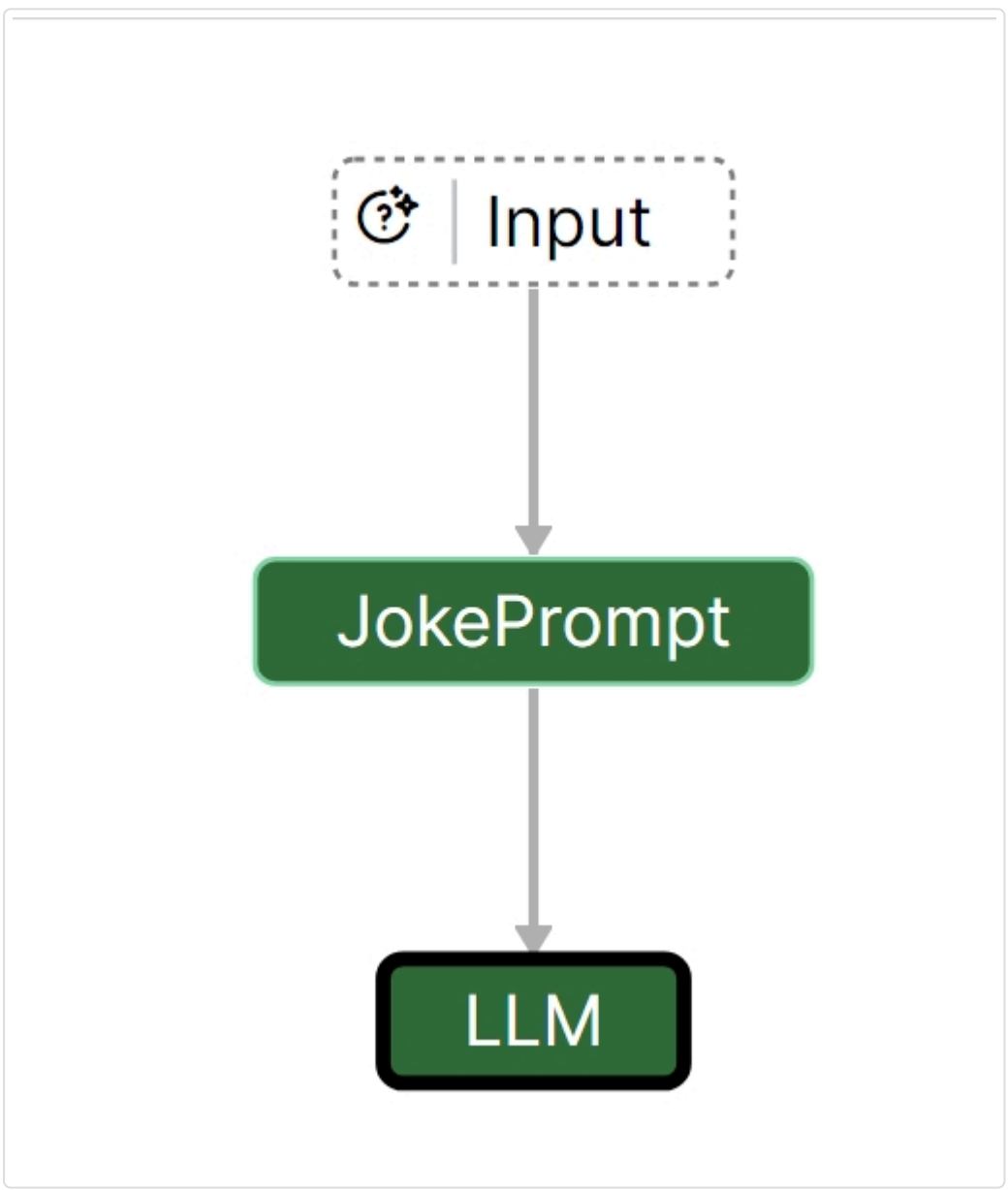
These flows can range from a simple two step process of formatting a prompt for an LLM, to complex, multi-step flows implementing advanced Retrieval Augmented Generation (RAG). Flows can also be used to implement custom content ingestion pipelines used to load data into a semantic index.

Getting Started

Before we dive into a detailed description of the GenAI Builder, let’s first see how it can be used to build a very basic flow. The most basic and common use case is chaining a prompt template and an LLM together. To see how this works, let’s create a flow that takes a topic and generates a joke.

> Detailed steps for this process can be found in the GenAI Builder tutorial. Here we are going to provide a more informal overview of the results.

The resulting flow looks like this:



Flow diagrams are directed, acyclic graphs depicting the “flow” of data as it is processed. They are read from the top to the bottom. The nodes in the graph are called “tasks” and each one represents some work that is being done. Tasks are created from GenAI Components (often referred to as simply “components” in this document) which determine the specifics of that work. The first task is always called “Input” and represents the initial data sent to the flow for processing. From there each task receives its input from its parent task(s), processes that input, and produces its output. This output is then sent to any children or becomes part of the flow’s final result.

Applying this to the flow above, we can see that the initial input is provided to the *JokePrompt* task which creates the prompt. This prompt is then sent to the *LLM* task and the response from the LLM is returned as the result of the flow. Note that a flow's terminal task(s) (here *LLM*) are denoted by a thick, black border.

GenAI Builder Overview

Launching the GenAI Builder

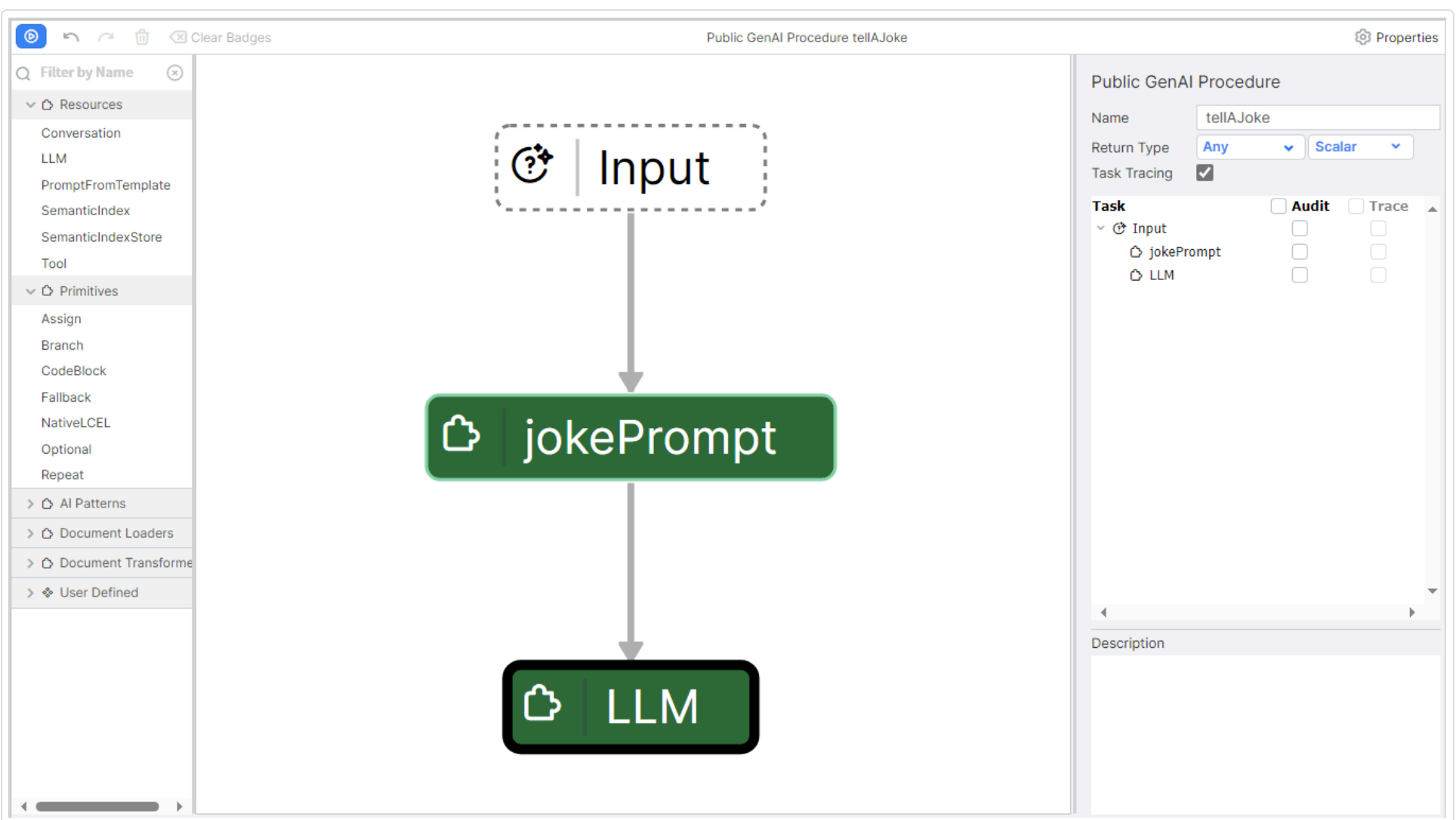
The GenAI Builder can be launched in one of three ways, depending on how the resulting GenAI Flow is intended to be used:

- From the App Builder – launched by creating/editing a task based on the GenAI Flow Activity Pattern. This results in an *embedded* flow which is part of an event handler.
- From the Service Builder – launched by creating/editing a GenAI Procedure. This results in a flow which is executed by invoking the procedure (which may be either public or private).
- From the Add…GenAI Component menu item – launched with creating/editing a standalone GenAI Component.

Once launched, the behavior of the GenAI Builder is generally the same in all three cases (with some notable exceptions in the GenAI Component case which we will discuss below).

General Layout

The GenAI Builder uses a layout which is very similar to the App Builder.



There is a *palette* on the left hand side which contains the components used to construct flows, a *canvas* in the center where the tasks that make up a flow are depicted, and a *configuration panel* which slides out on the right hand side upon request. Tasks are created by dragging components from the palette and dropping them on the canvas such that they connect to an existing task.

The palette is subdivided into *sections* which serve to group the available components according to their function/purpose. The categories (and their contents) are:

Resources

The resource components perform the basic functions in a flow and are configured from an associated Vantiq resource.

- Conversation – maintains a conversation for the specified sub-flow. The conversation is managed via the Vantiq ConversationMemory service.
- LLM – used to send requests to a Large Language Model (LLM) and parse/format any response. Configured via an LLM resource.
- Procedure – execute the specified procedure, emitting its result.
- PromptFromMessages – formats a prompt using a specified list of chat messages.
- PromptFromTemplate – formats a prompt using a specified template.
- SemanticIndex – used to retrieve information from a SemanticIndex resource based on semantic similarity.
- SemanticIndexStore – used to store information in a SemanticIndex resource.
- SemanticIndexWithCompression – Queries the specified semantic index and returns an Array of similar documents after processing by the specified document compressors.
- Tool – used to send requests to a Large Language Model (LLM) configured for tool calling and parse/format any response. Configured via an LLM resource.

Primitives

The primitive components (aka primitives) provide various utility/control functions such as decision making and invoking custom code.

- Assign – adds additional properties to an initial `Object` input value.
- Branch – chooses a sub-flow to execute based on an associated condition.
- CodeBlock – executes the given block of code.
- Fallback – Supports execution of a 'fallback' sub-flow, if the primary flow fails.
- Loop – executes a sub-flow until the defined condition returns `false` or the specified loop limit is reached.
- Map – concurrently executes the contained sub-flow once for each item in the input array. Returns an array containing each of the resulting values.
- NativeLCEL – executes a Python code block which must return an instance of a LangChain component.
- Transform – transforms the event passing through the component based on the provided code.

Guardrails

- GuardrailsAI – An implementation of the Guardrails-AI toolkit with a limited set of Validators. Used to protect against invalid or improper prompts and responses.
- NeMoGuardrails – An implementation of Nvidia's NeMo Guardrails toolkit. Used to protect against invalid or improper prompts and responses, and to tailor LLM prompts or responses for certain topics. Intended for more complex or comprehensive use-cases, compared to the Guardrails-AI toolkits.

Document Loaders

The document loaders support reading content from some source and producing an Array of LangChain Documents (these are not instances of the Vantiq Documents resource). The loaders differ in the source of the content and/or the techniques used to extract their content.

- UnstructuredURL – reads from a URL (or an array of URLs) and uses the unstructured library to extract content from whatever the URL references.
- ConversationMemory – reads the current state of a specified conversation.

Document Transformers

Document transformers accept content as input (typically as an Array of LangChain Documents) and perform the specified transformation, producing the result as an Array of LangChain Documents. Typically these are used to prepare previously ingested content for storage in a Semantic Index.

- ParagraphSplitter – analyzes text (ignoring any formatting) and divides it into paragraphs based on the placement of newlines. If any given paragraph is "too large" (based on configuration) then it may be further subdivided, first into sentences, then into words, and then arbitrarily.
- MarkdownSplitter – analyzes a Markdown document and divides it first into sections (based on the presence of headers). For any sections that are "too large" it further subdivides them into code blocks, horizontal lines, paragraphs, sentences, and words.
- HTMLSplitter – analyzes an HTML document and sub-divides it based on the formatting tags.
- CodeSplitter – analyzes a document containing a specified programming language and sub-divides it based on that language's keywords.

Document Compressors

Document compressors are used to perform contextual compression of content to improve the performance of retrieval algorithms. They can be used as standalone components or as part of a SemanticIndexWithCompression task to create a Contextual Compression Retriever. To do this work the compressor is provided with a list of LangChain Documents and the query being processed.

- CohereRerank – Uses the specified Cohere re-ranking model to order the documents based on their relevance to the query.
- ExtractRelevantContent – Extracts any content from the documents based on its relevance to the query.
- FilterForRelevance – Filters the documents to remove any that lack content relevant to the query.
- ListwiseRerank – Uses an LLM to perform a "listwise" re-ranking of the documents based on their relevance to the query.

AI Patterns

The AI Patterns are implementations of several common GenAI algorithms (aka patterns) which are provided by Vantiq. These can be used as-is or as the basis for further customization in the creation of your own GenAI Flows.

- Categorize – analyzes the given input and places it in one of the categories that are part of its configuration.
- Consensus – uses multiple LLMs to arrive at a consensus response to the supplied input prompt.
- RAG – an implementation of Retrieval Augmented Generation (RAG). This implementation matches the one provided by the AnswerQuestion activity pattern.
- ReduceDocuments – uses a given prompt and LLM to reduce an array of documents to a single result.

Configuration Processors

The configuration processors are designed to assist in the creation of user defined GenAI Components. They define behaviors that are based on the initial configuration values provided when a user defined component is used to create a task.

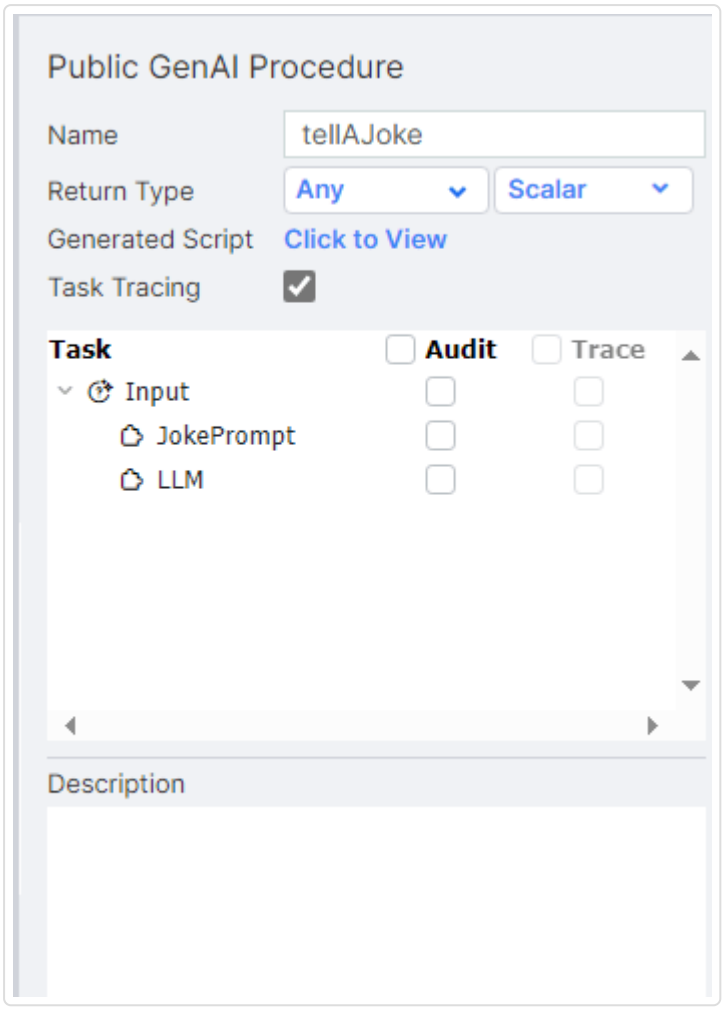
- Optional – defines a sub-flow which will be conditionally executed based on the supplied initial configuration.
- Repeat – defines a sub-flow that will be executed multiple times, based on how many initial configuration values are specified.

Components

This is where any standalone GenAI Components created by the user are shown.

GenAI Flow Properties

The properties of the current GenAI Flow are accessed using the *Properties* button in the upper right hand corner of the GenAI Builder. This opens the configuration pane slide-out like this:



The values shown here will vary depending the type of the GenAI Flow.

GenAI Procedure and Embedded GenAI Flow Properties

The following properties can be edited:

- **name** – the name of the GenAI Flow. When editing an *embedded* flow, this property is read-only.
- **return type** – the expected type of the data returned by the flow. This can be either a scalar value or a sequence. Using a sequence enables streaming of the flow's results. Setting this will trigger standard VAIL type checking.
- **description** – a user supplied description of the GenAI Flow.

Clicking on the *Generated Script* button will bring up a read-only view of the Python code generated for the GenAI Flow. This code uses the LangChain Expression Language (LCEL) along with some Vantiq specific extensions.

The *Task Tracing* checkbox provides flow-wide control over the GenAI Flow Tracing feature (which is enabled by default). The per-task checkboxes below that allow for enabling either auditing or tracing on a per-task basis. Per-task control of tracing is only available if tracing has been disabled for the flow as a whole.

GenAI Component Properties

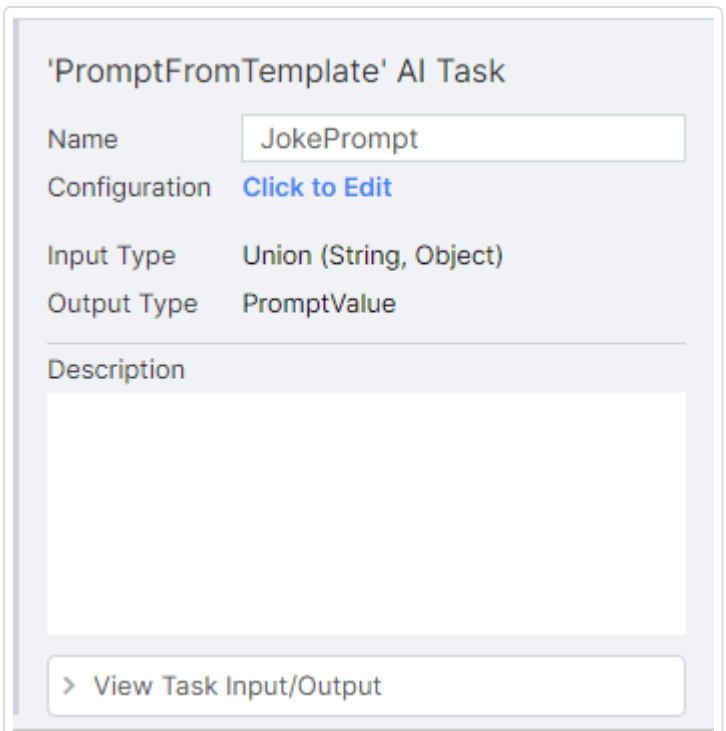
- **name** – the name of the GenAI Component.
- **package** – the GenAI Component's package.

Clicking on the *Generated Script* button will bring up a read-only view of the Python code generated for the GenAI Flow. This code uses the LangChain Expression Language (LCEL) along with some Vantiq specific extensions.

Clicking on the *Configuration* button will bring up a property sheet showing the GenAI Component's Configuration Properties.

GenAI Flow Task Properties

The properties for any given task in the GenAI Flow can be seen by clicking on the task. This brings up the properties pane slide out like this:



- **name** – the name of the GenAI Flow task (defaults to the name of the component).
- **configuration** – clicking on “Click to Edit” brings up the property sheet used to configure the task’s initial configuration.
- **input type** – displays the type of the task’s input. For most tasks this will be read-only.
- **output type** – displays the type of the task’s output. For most tasks this will be read-only.
- **description** – a user supplied description of the task.
- **view task input/output** – expands a field in which the task’s input and output values will be displayed during execution, as well as their classes. Only active if tracing is enabled.

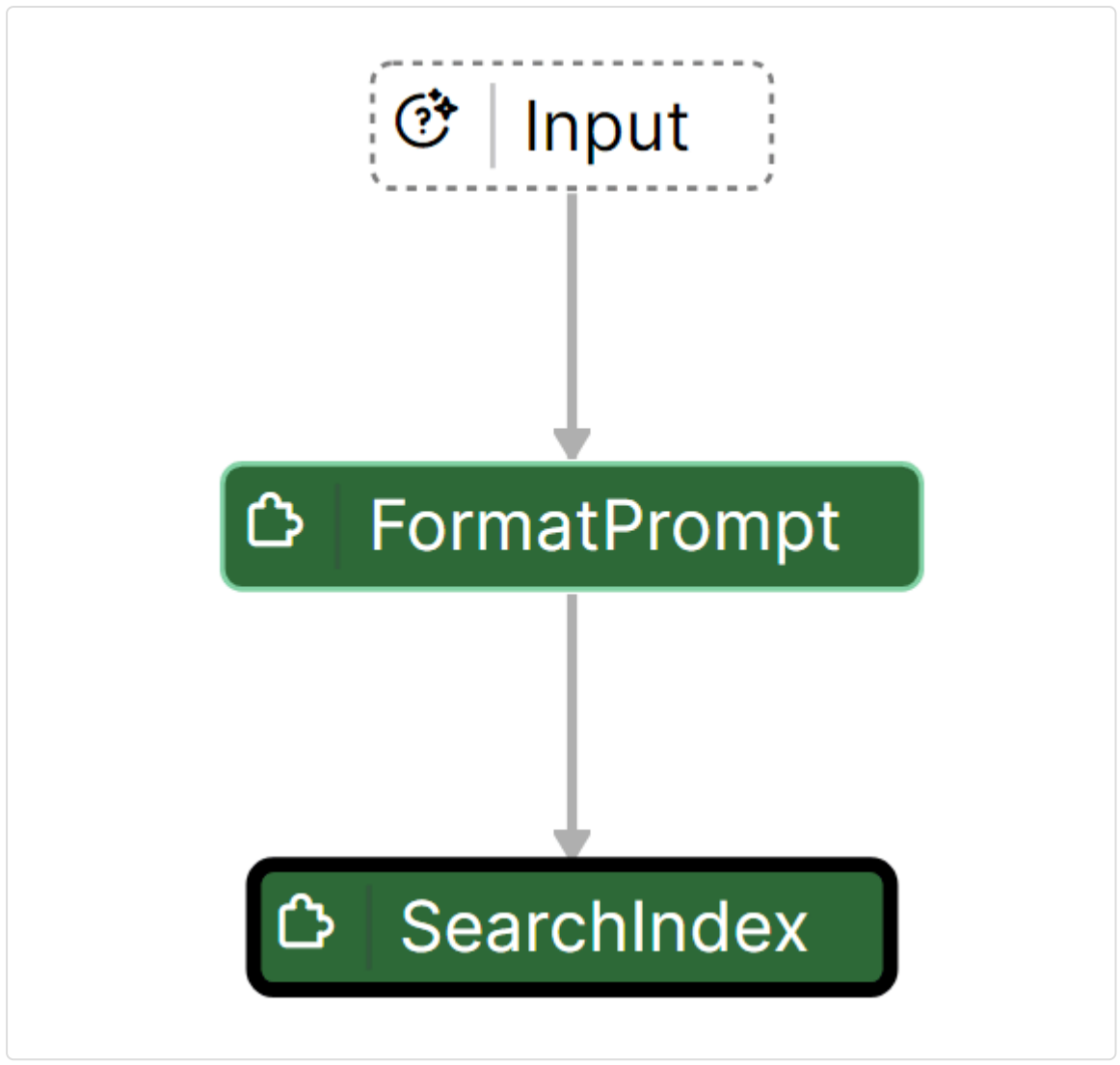
Defining GenAI Flows

Now that we are familiar with the layout and structure of the GenAI Builder, let’s explore how we use it to define GenAI Flows. GenAI Flows consist of a directed sequence of “tasks”, each of which process the data they receive. In a simple diagram like the one shown earlier, data flows linearly from the top of the diagram to the bottom and the functionality is wholly contained in the flow. However, you will encounter situations where more complex flows are required to implement the desired GenAI functionality. In the following sections we cover how to accomplish this.

Task Typing

The input and output of every task in a GenAI Flow has an associated *type*. In most cases these types will be determined automatically, based on the definition of the task’s component and/or the configuration of the task itself (see the GenAI Component section for details). The types for any given task can be viewed on its properties pane. When connecting one task to another, the GenAI Builder will check to confirm that it is legal to assign values of the output type to the specified input type. This check is done when the GenAI Flow is saved.

For example, let’s say we try to create a flow to first format a prompt and then use that value as the input to a semantic search:



When we attempt to save the flow, we will see the following error:

⚠ Compilation Errors (1)

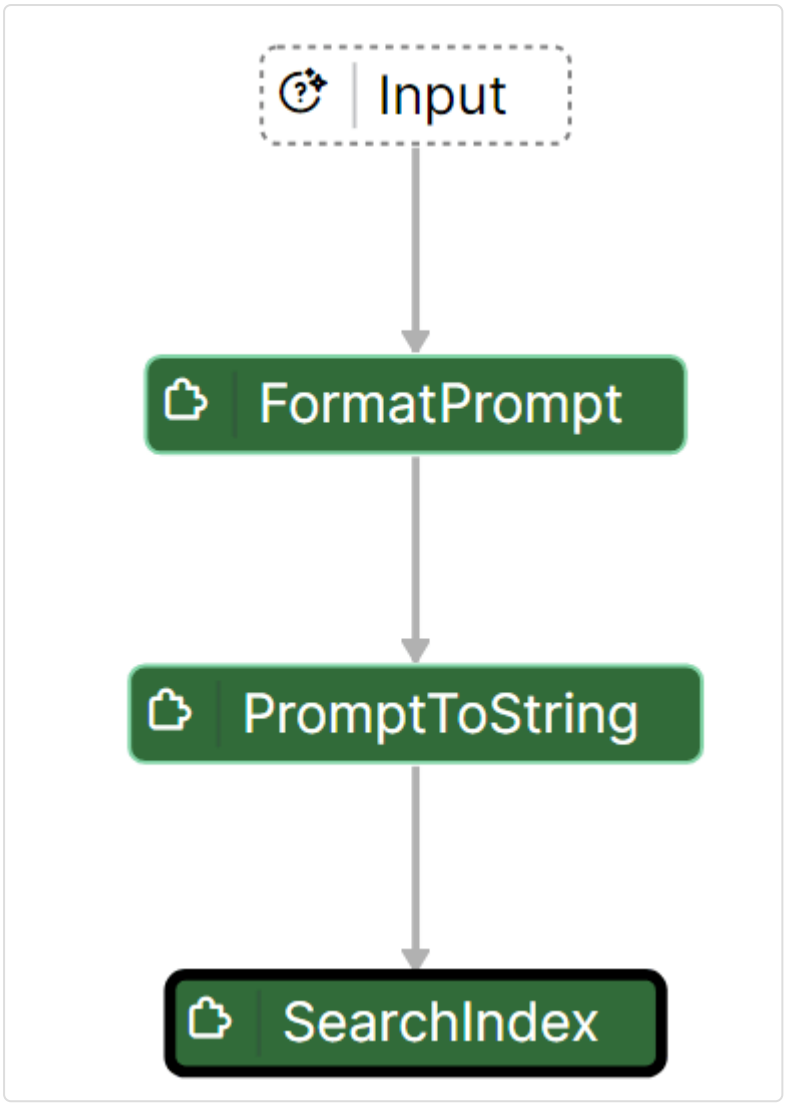
One or more tasks contained an invalid or missing property or unmatched input/output type. The relevant tasks have been outlined in red in the graph.

Task Name: **SearchIndex**
Code: **io.vantiq.aimanager.component.assembly.type.mismatch**
Message: **The task 'SearchIndex' expected an input value of type 'String', but was provided with a value of type 'langchain_core.PromptValue'.**

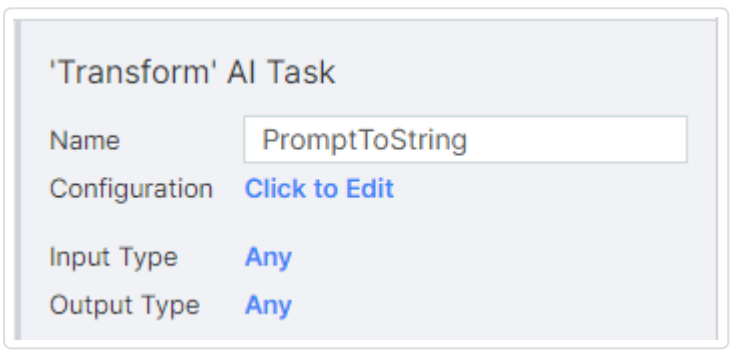
This is because the PromptFromTemplate component has an output type of `PromptValue` and the SemanticIndex component expects its input to be of type `String`. Fixing this will require that we transform the prompt output to the required input type. So let's add a Transform task configured as follows:

- *name* – PromptToString
- *transformation type* – **Projection**
- *transformation* – `input.toString()`

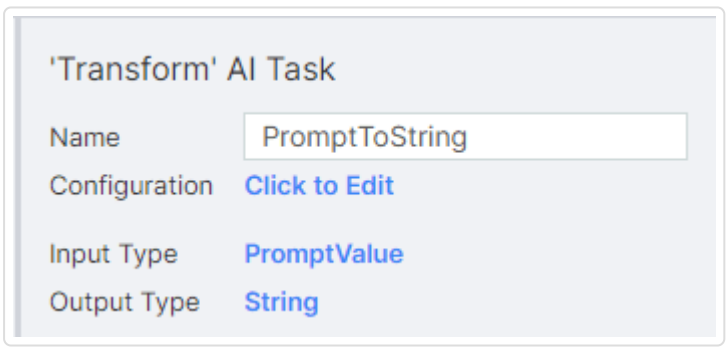
Now when we save the flow, there should be no errors:



However, if we take a closer look at the newly added task's properties, we'll see that we've actually lost some information:



Notice that the new task's input and output types are both `Any`. This is because the transform component can (in the limit) accept any type of input and produce any type of output, so the system cannot know (a priori) what the task's types are. When this happens, the task types will be editable (as they are here) so that the user can provide the missing type information. Clicking on the type gives us a drop list from which we can select the appropriate type. For this task we want the input type to be `PromptValue` and the output type to be `String`:



Supported Types

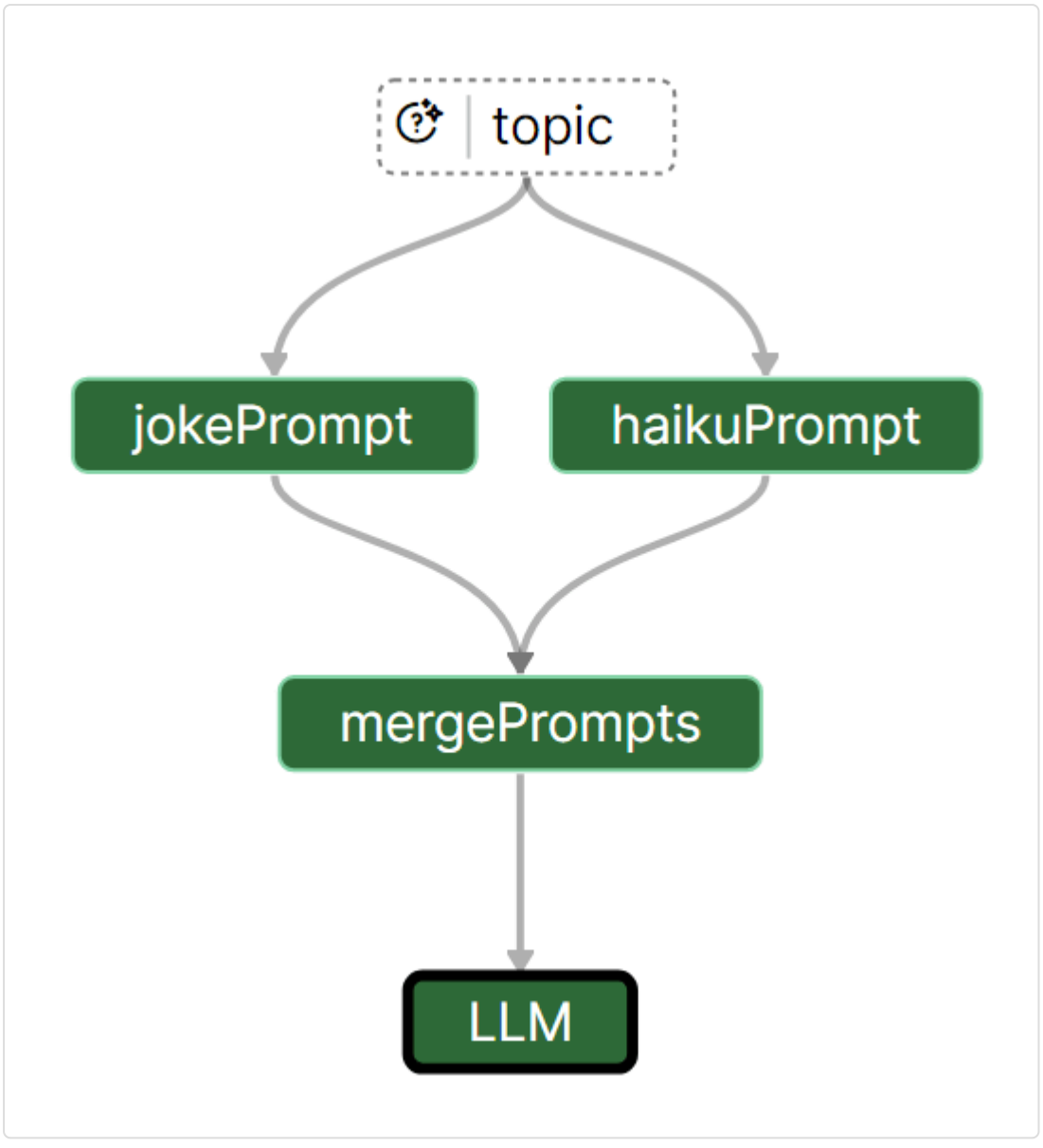
The GenAI Builder supports the following types as the input/output of a task:

- VAIL Standard Types – the types `Object`, `String`, `Integer`, `Real`, `Boolean`, and `DateTime` (other standard types are not supported).
- LangChain Core Types – `PromptValue`, `ChatMessage`, and `Document`.
- Union Types – union types allow a list of possible types (known as type choices). Effectively this means that the Union can be treated as if it were any of the supplied choices when performing type checking.

> The LangChain core type `Document` is *not* the same as the Vantiq `system.documents` resource.

Merging Task Outputs

One fairly common requirement is to have a flow where the output of one task is sent to more than one child task and then the results of those children are merged into a single result. To illustrate this, let's expand on our previous "joke" generation flow by assuming that we want both a joke and a haiku for our chosen topic. To do this we might define the following flow:

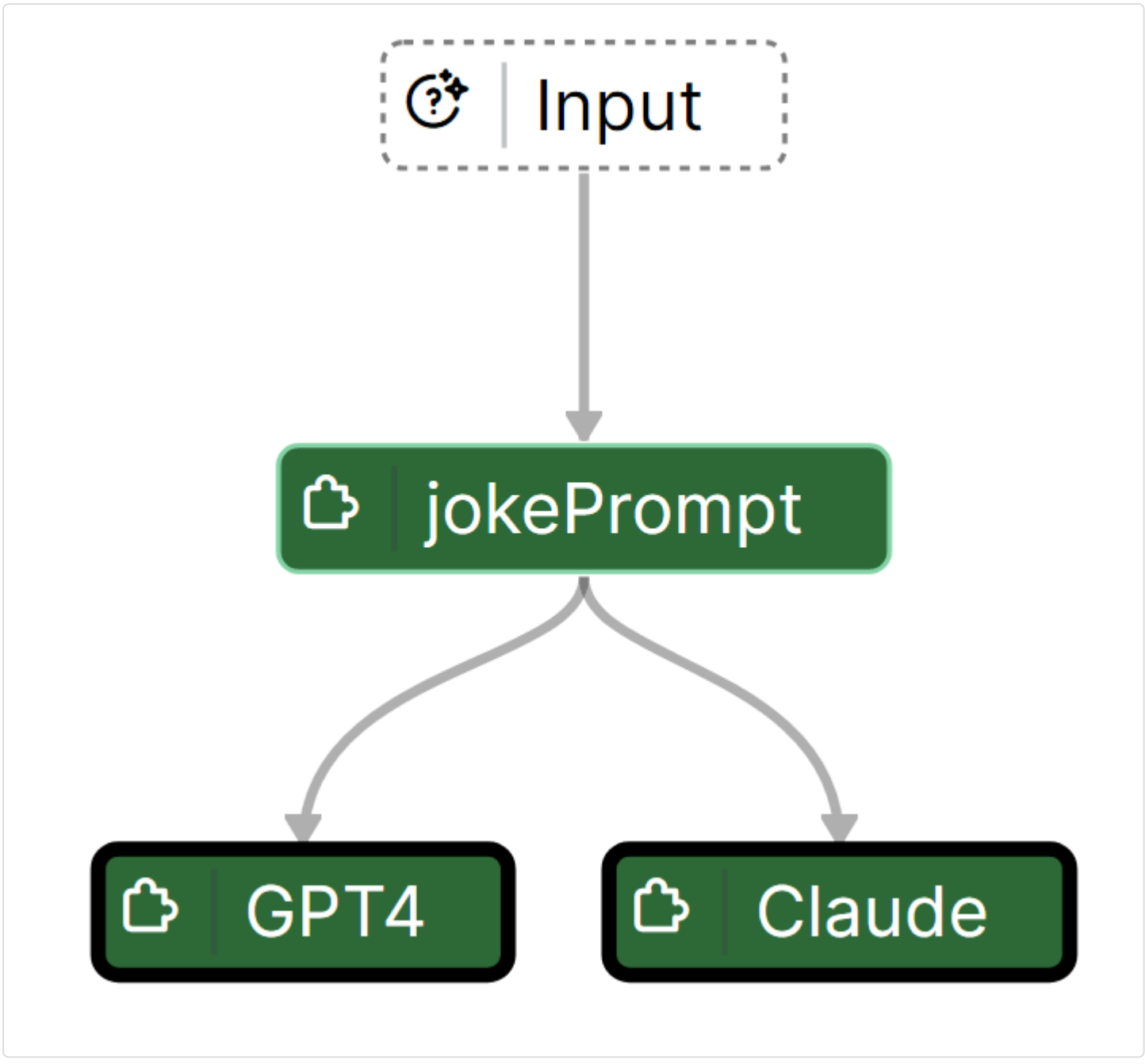


In this flow we are sending the initial input to both the *jokePrompt* and *haikuPrompt* tasks. Once they complete, their results will be merged and sent to the *mergePrompts* task. Merging task outputs results in an `Object` with one property for each contributing task. So here we will get:

```
{
    "jokePrompt": <result of jokePrompt task>,
    "haikuPrompt": <result of haikuPrompt task>
}
```

The prompt used by *mergePrompts* is `Please respond to both of the following requests:\n${jokePrompt}\n\n${haikuPrompt}`. This results in a single prompt with both requests being sent to the LLM. When tasks are merged in this way, they will be executed in parallel. This can be handy when you need responses from more than one LLM to perform a task (assuming that the LLM requests are independent).

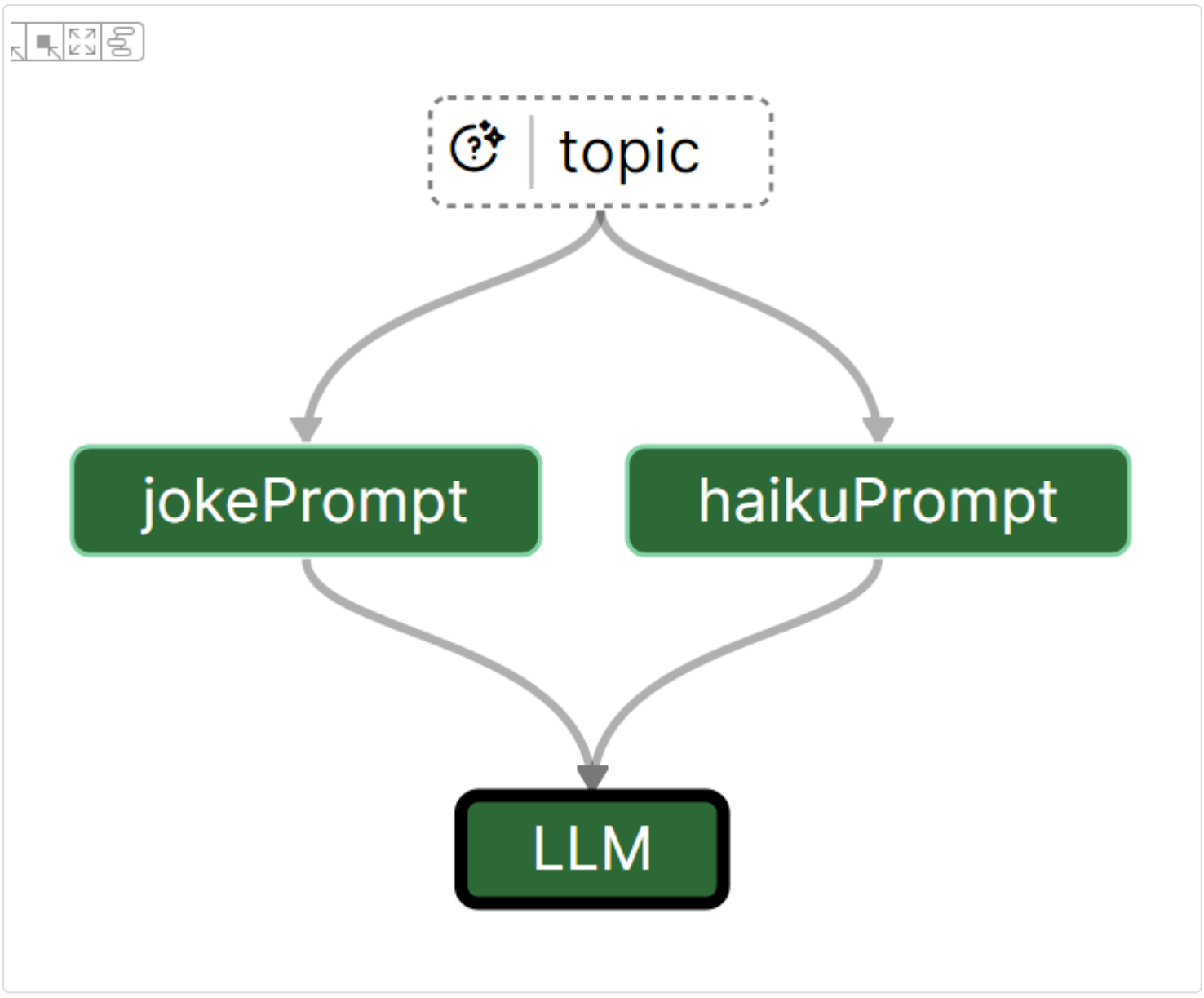
Flows with multiple terminal tasks perform an implicit merge of these tasks upon completion. This will result in the flow returning an `Object` with one property for each of the terminal tasks. For example given the following flow:



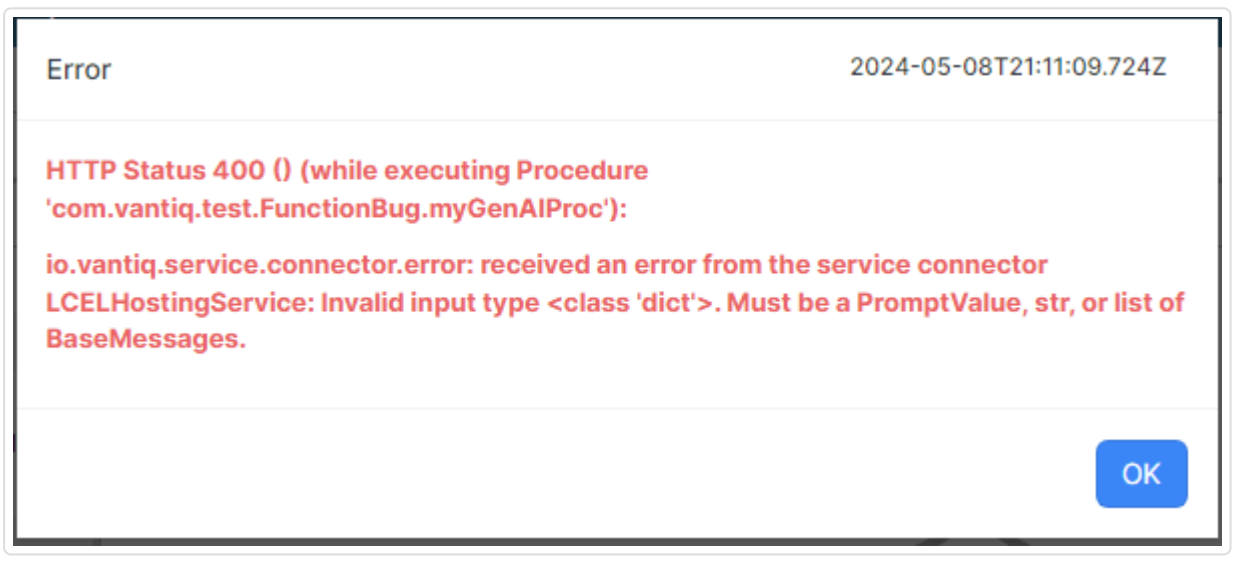
The result is an `Object` with the properties *GPT4* and *Claude*. You can tell which tasks in a flow will contribute to the final output by the fact that they are outlined in a thick, black border (these are know as *terminal* tasks).

Difference from the App Builder

For readers who are familiar with the App Builder, you might have assumed that the flow would look like this:



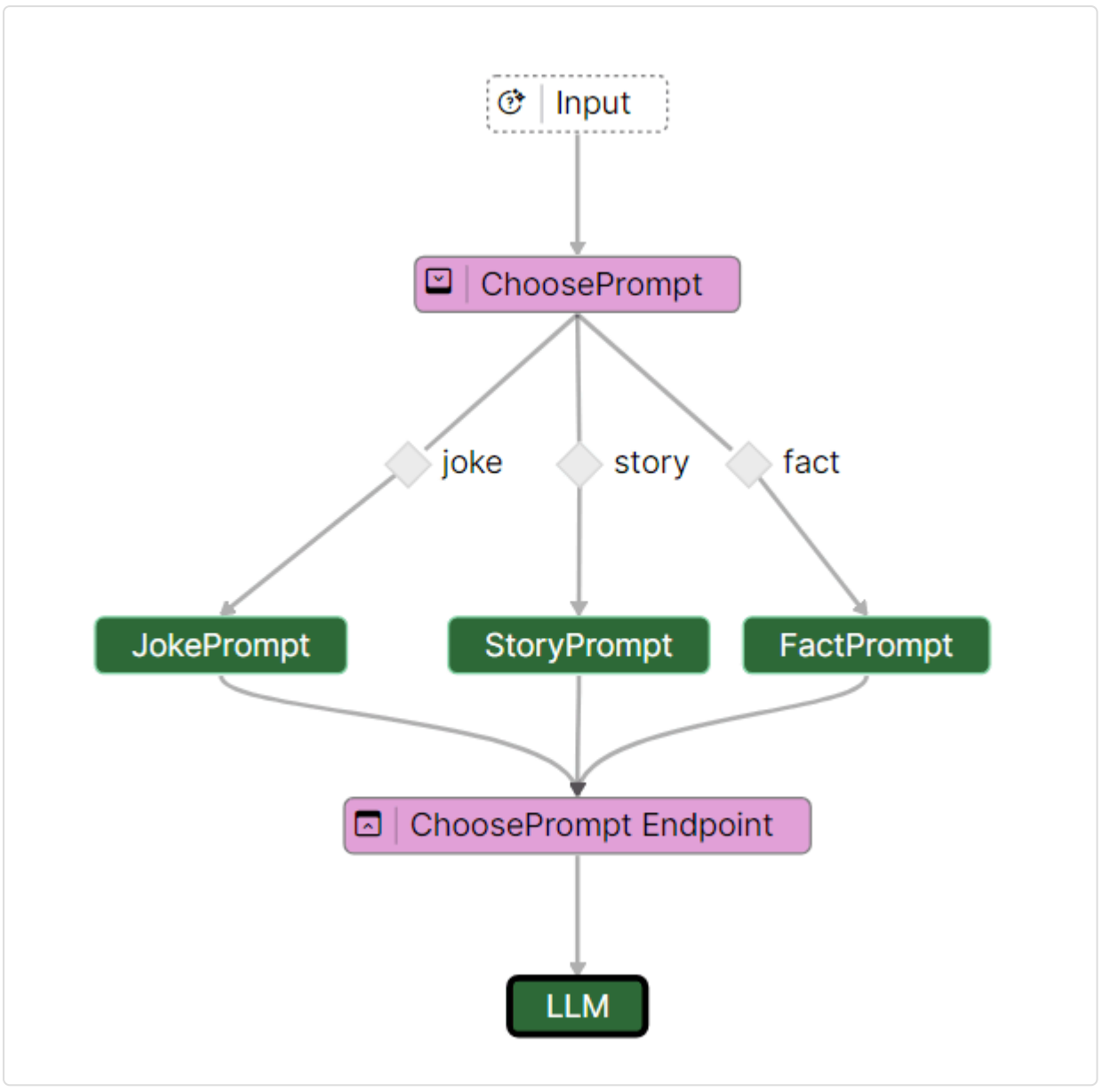
However, running this flow leads to the following error:



This is because unlike in the App Builder, a GenAI Flow represents a *synchronous* data flow and not an asynchronous sequence of events. As a result, the output of the parent tasks is merged before it is sent to the child, which is then executed *once*. This makes the semantics equivalent to the Join Activity Pattern. There simply is no need for the explicit join task since only one option for merging results exists in this case.

Sub-Flows

Many of the GenAI primitives serve to apply additional functionality to a portion of a GenAI Flow or allow for the selection between multiple flows. In such cases, the visualization of the component will denote a “start” and “end”. The task(s) that are placed between these are referred to as a “sub-flow”. Take the following flow for example:



Here we have a Branch primitive (*ChoosePrompt*) configured with 3 possible outcomes – a joke, a story, or a fact (the default). Each of these "branches" has a sub-flow associated with it, consisting of a single *PromptFromTemplate* task. The semantics of "Branch" are that for each input, one of the sub-flows will be executed based on the associated conditions. Once that's done, the output is fed to the *LLM* task. Of course the exact semantics will depend on the task containing the sub-flows, but in each case, those semantics will be applied to the sub-flows only.

Memory and Conversations

A very common use case involves an application managing a sequence of interactions between the application user and an LLM (or possibly even multiple LLMs). Rather than treating each interaction as a discrete request (which is how the LLM views them), the application often wants to use information from previous interactions as part of subsequent ones. This capability is referred to by many different names, including "memory" or "message history". In the Vantiq platform, we refer to it as a "conversation". Conversations are managed in memory by the ConversationMemory service. They can also be managed over much longer time periods via service defined Conversations.

It is important that GenAI Flows be able to participate in these conversations, both in order to leverage the state that they contain and in order to contribute to that state. This is the reason for the Conversation resource component and for the inclusion of the **useConversation** configuration property in AI patterns such as RAG. More details on how to incorporate conversations into your GenAI Flows can be found in the description of the Conversation resource component below.

A special memory state called `flowstate` is also available. This memory space, represented as a VAIL Object or a Python dictionary, is exclusive to the current flow and is not shared with other flows. It is useful for storing intermediate results or other flow-specific data. Typically, it can be directly accessed by any `Code Block` within the flow.

Runtime Configuration

Sometimes it is useful to provide a way to configure some aspects of a component's behavior at runtime. This can allow for experimentation with different values and the ability to more easily adapt a component dynamically. This is especially true of LLMs, most of which offer a variety of options to control their operation. For this reason, GenAI Flows have a notion of "runtime" configuration properties. As the name suggests, these are properties whose values can be set each time a GenAI Flow is executed. The values are made available to every task and will be applied based on whether or not the task supports the property in question.

For "embedded" GenAI Flows, the values are set using the **runtimeConfiguration** property of the GenAI Flow Activity Pattern. For GenAI Procedures they are passed using the **config** parameter. In both cases the value is expected to be of type `Object` where the names of the properties correspond to the runtime property names.

Streaming Results

Due to the fact that LLM processing can be lengthy, most LLMs support incremental generation of results which is also known as "streaming". This is useful if you want to display the response to the user as it's being generated, or if you want to process the response as it's being generated. GenAI Flows support streamed execution when they are declared to return a sequence and not a scalar value.

Taking full advantage of streaming requires cooperation from the caller of the GenAI Flow. The caller must be prepared to receive the results incrementally and must be able to do some reasonable processing of the results as they appear (instead of waiting for the completion of the flow). The options for this style of invocation include:

- Calling a GenAI Procedure from VAIL using an EXECUTE statement with an attached processing block.
- Execution of a GenAI Procedure from a Vantiq Client using client.executeStreamed(). This option is also available in the Conversation Widget.
- Execution of a GenAI Procedure by a REST client with the *stream* URL parameter set to `true`.

The Streaming Results section of the GenAI Builder Tutorial shows this in action.

One context where this feature cannot be used is in a GenAI Flow task. This is because the task needs to have the complete result from the GenAI Flow before it can generate its downstream event. Given this you never need to declare that an *embedded* GenAI Flow returns a sequence.

Supported VAIL Language Features

The GenAI runtime system is implemented in Python. Most of the time this doesn't matter as the GenAI Builder relies primarily on the configuration of tasks to express semantics. Some components do require the specification of either an "expression" or a full code block. In these cases, we support a subset of VAIL so that the use of Python is not required (though it too is supported). The following VAIL features are supported, if something is not listed here, then it is **not** supported.

VAIL Standard Types

The supported VAIL standard types are: `String`, `Integer`, `Real`, `Boolean`, `DateTime`, and `Object`.

VAIL Statements

The following VAIL statements are supported. Anything not listed here (or explictly excluded), is not supported.

- Variable Declaration
- Variable Assignment
- Type Literals
  - Only for the supported standard types.
  - Interval literals are supported.
- Expressions
  - No support for the Lambda Operator.
  - No Decimal and Currency Support.
- Flow Control
- Iteration
- RETURN

VAIL Built-in Services and Procedures

These are the supported built-in procedures. Anything not listed here is not supported.

Type Specific Procedures

String Procedures

- **toInteger(str)**
- **toReal(str)**
- **toDate(str)**
- **toBoolean(str)**
- **concat(str, str1, … strN)**
- **endsWith(str, substr)**
- **includes(str, substr)**
- **indexOf(str, substr)**
- **lastIndexOf(str, substr)**
- **length(str)**
- **match(str, pattern)**
- **repeat(str, count)**
- **replace(str, pattern, newSubStr)**
- **search(str, pattern)**
- **slice(str, start, end)**
- **split(str, pattern, limit)**
- **startsWith(str, substr)**
- **substr(str, begin, length)**
- **substring(str, begin, end)**
- **toLowerCase(str)**
- **toUpperCase(str)**
- **trim(str)**
- **isAlpha(str)**
- **isAlphaNumeric(str)**
- **isHexadecimal(str)**
- **isName(str)**

- **isNumeric(str)**
- **isReal(str)**
- **trimLeft(str)**
- **trimRight(str)**
- **whitespaceTo(str, char)**

Integer Procedures

- **toReal(intValue)**
- **toString(intValue)**
- **toDate(str)**
- **toBoolean(intValue)**
- **range(from, to, increment)**

Real Procedures

- **toInteger(realValue)**
- **toString(realValue)**
- **toBoolean(realValue)**

DateTime Procedures

- **toInteger(date)**
- **toString(date)**
- **getMilliseconds(date)**
- **getSeconds(date)**
- **getMinutes(date)**
- **getHours(date)**
- **getDate(date)**
- **getMonth(date)**
- **getFullYear(date)**
- **getDay(date)**
- **getTime(date)**
- **date(value, sourceRepresentation, destinationRepresentation)**

  The **sourceRepresentation** and **destinationRepresentation** parameters may be one of the following string values:

  - date
  - ISO
  - epochMilliseconds
- **now()**

- **parseDate(str, dateFormat)**
- **truncateTo(date, chronoUnit)**
- **addMonth(date, monthOffset, preserveLastDay)**
- **addYear(date, yearOffset, preserveLastDay)**
- **week(date)**
- **DoW(date)**
- **HoW(date)**
- **MoW(date)**
- **DoY(date)**
- **MoD(date)**
- **SoD(date)**
- **SoH(date)**

Interval Procedures

- **stringToInterval(str)**
- **durationInMillis(str)**

Object Procedures

- **toObject(obj)**
- **clear(obj)**
- **deleteKey(obj, key)**
- **has(obj, key)**

Array Procedures

- **concat(arr, joinedArray)**
- **fill(arr, value, begin, end)**
- **includes(arr, element)**
- **indexOf(arr, element)**
- **join(arr, separator)**
- **lastIndexOf(arr, element)**

- **length(arr)**
- **pop(arr)**
- **push(arr, element)**
- **reverse(arr)**
- **shift(arr)**
- **slice(arr, begin, end)**
- **sort(arr, property)**
- **splice(arr, begin, deleteCount, list)**
- **unshift(arr, element)**
- **size(arr)**
- **removeAt(arr, index)**
- **remove(arr, element)**
- **removeElement(arr, element)**
- **addAll(arr, elementArr)**
- **flatten(arr)**
- **clear(arr)**

General Use Procedures

URI Procedures

- **decodeUri(encodedUri)**
- **decodeUriComponent(encodedUriComponent)**
- **encodeUri(str)**
- **encodeUriComponent(str)**
- **escape(str)**

Content Parsing Procedures

- **parseInt(str)**
- **parseFloat(str)**

Miscellaneous Procedures

- **typeOf(target)**
- **exception(code, message, params)**
- **rethrow(exception)**

Content Ingestion Flows

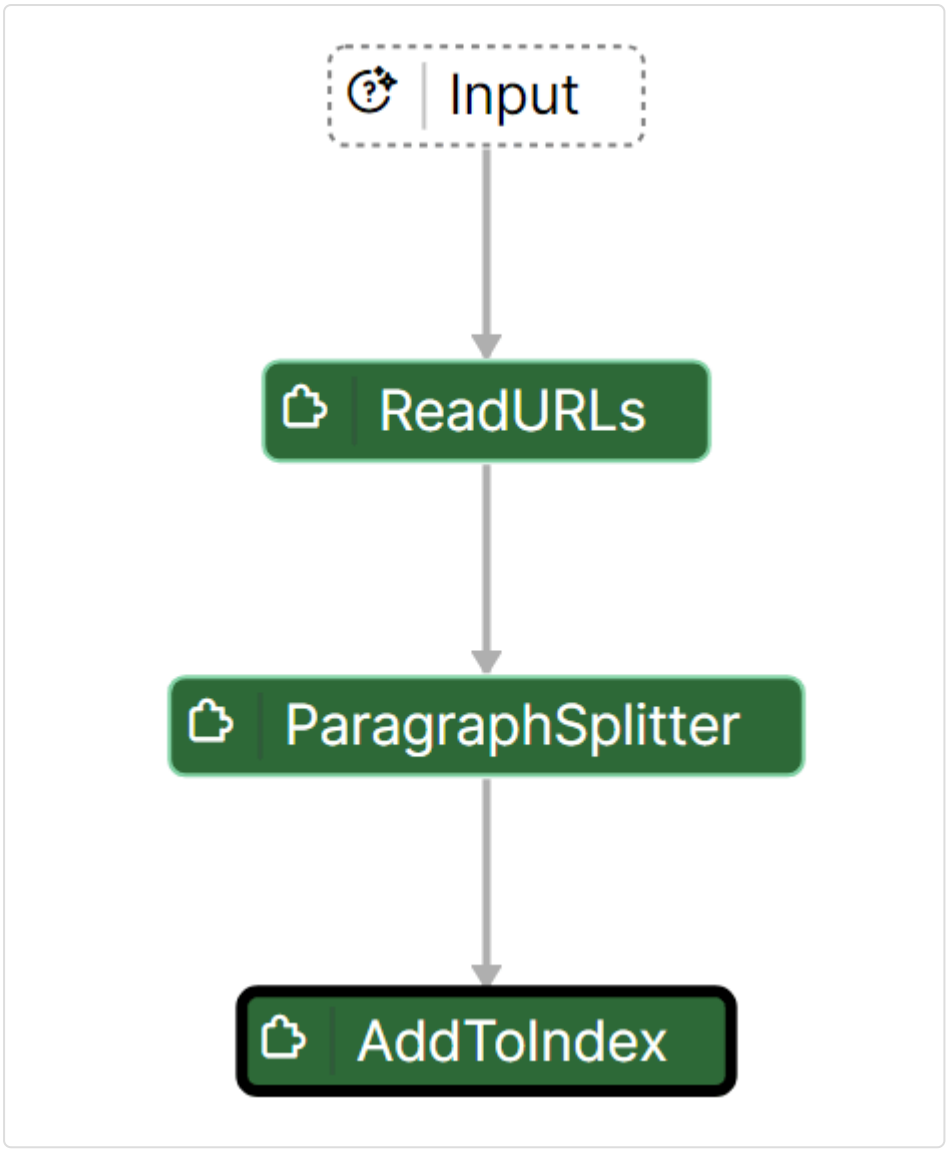
In addition to using the GenAI Flow builder to define flows which leverage the existing content of a Semantic Index, it can also be used to define flows which add to that content. This process is known as *content ingestion*.

> Just as the Vantiq platform provides an out-of-the-box RAG implementation (in the form of the AnswerQuestion activity pattern and service procedure), it also provides a default content ingestion implementation as part of the Semantic Index resource. Therefore using the GenAI Flow builder to construct a custom content ingestion flow is only necessary in cases where the default ingestion approach doesn't work well for the target application (something you will likely only determine by trying it first).

A typical content ingestion flow consists of the following three steps:

1. The raw form of the content is loaded from some external source (typically via a URL). Depending on the content type this may be as simple as reading the content directly or it may involve more complex processing such as OCR.
2. Once the content has been loaded, it may then be reformatted to make it more usable for performing a semantic search. This typically means *splitting* the content in some manner to create smaller *chunks* for storage in a semantic index.
3. Lastly, the final form of the content is stored in the target Semantic Index. This involves creating an *embedding* for each chunk and storing that in the underlying vector database.

For example, the following flow represents the default content ingestion implementation:



A more complex flow might involve requesting a summary of the content from an LLM and then processing the summaries for storage in one semantic index while the full content is stored in another.

Using Flows as the Default for Semantic Index Content Ingestion

Content ingestion flows may be leveraged as the default ingest mechanism for a semantic index. This is done by setting the **Ingest GenAI Procedure** field when creating or updating a Semantic Index in the UI. The setting must correspond to a GenAI Flow procedure. Attempts to use VAIL or other procedures will result in an error. This field becomes the **contentIngestionFlow** property of the Semantic Index resource. Subsequently, as new documents are added to the index, the procedure will be executed by the Semantic Index Service to process them. The service breaks the document content into chunks and passes each chunk (possibly multiple chunks concurrently) to the flow for ingest.

One difference between using a flow as the default content ingestion mechanism and invoking it directly is that when executing a `Semantic Index Store` task, it will not add a new entry to the index, instead returning the results of adding the current chunk of content to the vector data store. This allows for the Semantic Index Service to manage the status of incremental ingestion. Another difference is that the flow will be executed in an asynchronous manner with respect to the caller. The Semantic Index Service waits for a call back from the GenAI Flow service connector to proceed to subsequent ingestion steps and update the status of the index. This means that the ingestion process is not subject to the usual time constraints of a GenAI Flow procedure execution. This can be helpful when ingesting complex content where each chunk may take a significant amount of time to process.

GenAI Flow Tracing and Auditing

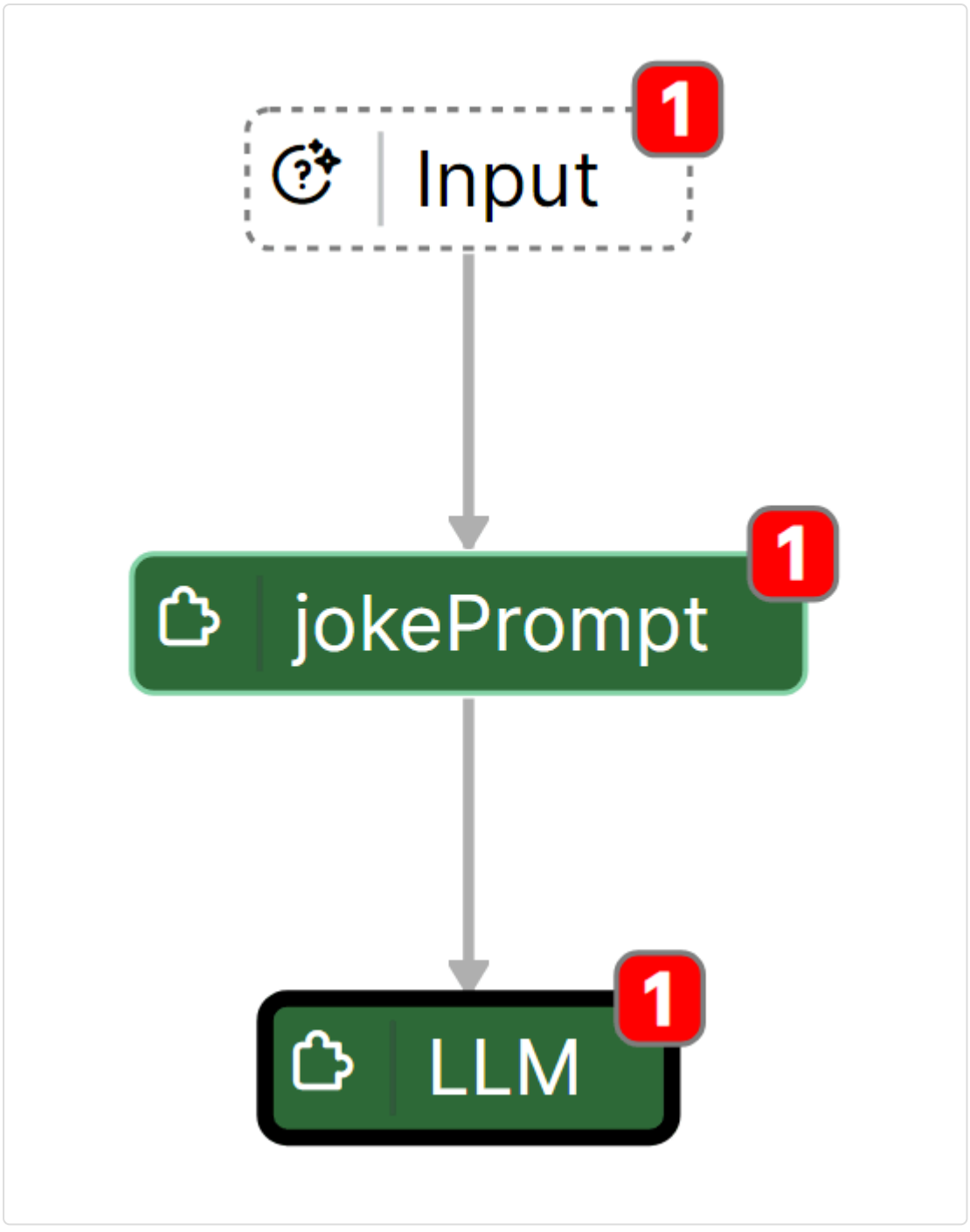
The GenAI Flow Builder provides two ways to record the operation of a GenAI Flow:

- Tracing – visualizes the execution of a flow through the use of task “badges” and allows for examination of the task outputs as they are produced (including intermediate outputs).
- Auditing – permanently records an audit record whenever selected tasks execute which show when the execution occurred and the inputs and outputs for the task.

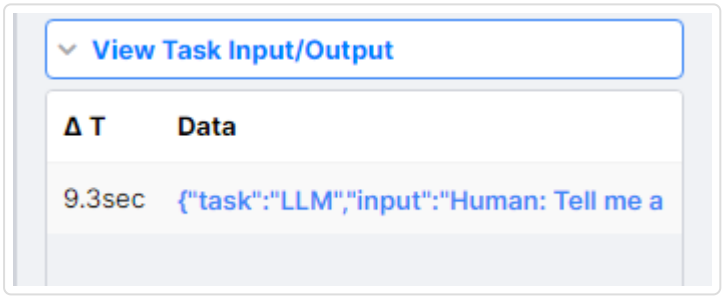
Let’s examine each in more detail.

GenAI Flow Tracing

GenAI Flow tracing is designed to be used during the development of a GenAI Flow. As such, it is enabled by default when a GenAI Flow is created and we recommend that it be disabled when deploying a flow into production. When enabled, task execution is visualized via the display of a “task badge” which increments whenever a task is executed:



It also makes the input and output of each task available via the “View Task Input/Output” pane in the slide-out panel on the right:



Clicking on a row will show the full details:

```
{
   "task": "LLM",
   "input": "Human: Tell me a Dad joke about goats.",
   "inputType": "langchain_core.prompt_values.ChatPromptValue",
   "output": "Why don't goats ever have secrets?\n\nBecause they always goat to tell someone.",
   "outputType": "builtins.str"
}
```

The information recorded in the UI is transient and will never be persisted. However, it does impose a processing overhead to collect, which is why we recommend that it be disabled in production.

GenAI Flow Task Auditing

GenAI Flow Task Auditing is designed to create a permanent record of the execution of specific tasks in order to ensure that the important decisions made by a GenAI Flow can be tracked and analyzed. Since auditing involves persisting data it is disabled by default and must be enabled on a task by task basis. Care should be taken to ensure that only critical tasks are recorded and developers should resist the temptation to turn on auditing for all tasks.

Once enabled for a task, the system will record a Vantiq Audit record any time the task executes. It will include the inputs to the task and the outputs that the task produced (along with standard information such as a timestamp and information about the security context being used). The audit records can be viewed under the Administer menu in Modelo.

Vantiq Provided GenAI Components

The tasks that comprise a GenAI Flow are configurations of predefined GenAI Components. Each GenAI Component represents a discrete bit of GenAI functionality that can be customized via its declared configuration properties. When a task is created, the user determines its underlying component and then provides the specific configuration values which govern its actual behavior. This makes every task created from a given GenAI Component distinct from all other such tasks.

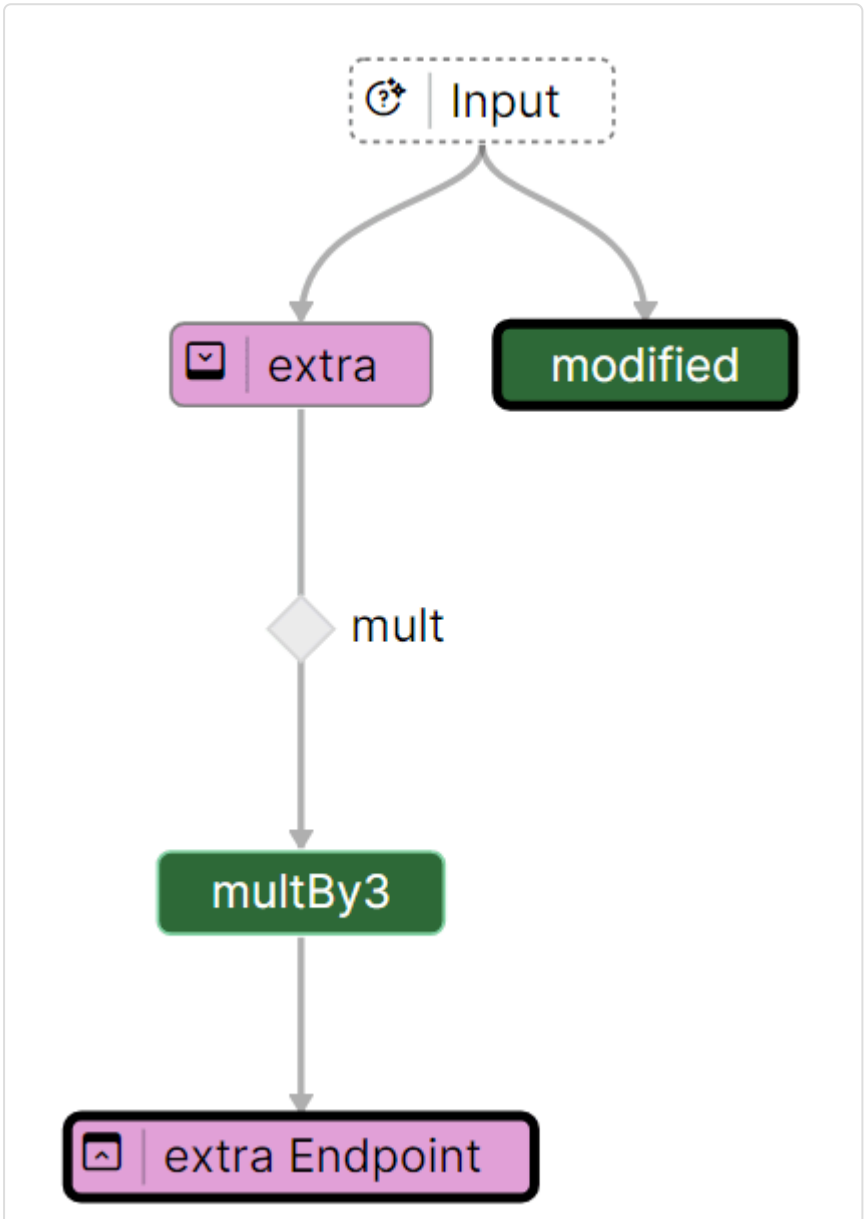
The core functionality of the GenAI Builder is represented by the “built-in” GenAI Components provided as part of the Vantiq platform. The following sections describe these components and their configuration properties. For each component we provide:

- A description of the component’s behavior/functionality. These often include a simple example of the component’s use (more complete examples of how components are used/combined can be found in the GenAI Builder Tutorial).
- A definition of the **Input** and **Output** types. These are typically expressed using the standard VAIL types or common Vantiq schema types.
- The configuration properties defined by the component. These are used to customize the behavior of each task created from a given component.
- The runtime configuration properties defined by the component. These can be used to further adjust behavior for each individual request.

Assign

Adds the specified properties to the input value (which must be of type `Object`). The value of each property is the result of its associated sub-chain. This is useful when additively creating an `Object` to use as input to a later step.

For example, the following flow expects an `Object` with the property `num`. The task *modified* returns `num + 1` and the task *extra* assigns `num * 3` to the property `mult`:



Given the input:

```
{
   "num": 3
}
```

The output is:

```
{
   "extra": {
      "num": 3,
      "mult": 9
   },
   "modified": 4
}
```

Input Type – `Object`

Output Type – `Object`

Configuration Properties

- **properties** (`String[]`) – A *required* list of the properties to be added to the input value. Each property has an associated sub-flow which will be executed to produce the assigned value.

Runtime Configuration – None

Branch

Selects which sub-flow to run based on a condition. The conditions are evaluated using the supplied input value, in the order that they are specified. The sub-flow associated with the first condition that evaluates to `true` will be executed (even if other conditions might also evaluate to `true`). If none of the conditions evaluate to `true`, then the default sub-flow (the one with no condition) will be executed.

Input Type – computed based on the inputs expected by each sub-flow.

Output Type – computed based on the outputs produced by each sub-flow.

Configuration Properties

- *branches* (`LabeledExpression[]`) – A *required* list of entries, each of which consists of a label and an expression (except for the last entry, which has only a label). The expressions can be written in either VAIL or Python. The supplied input value is available in a predefined variable named **input**.

Runtime Configuration – None

The `flowstate` memory is accessible to the branch expressions.

Categorize

Instructs an LLM to classify the given prompt value into the configured categories. The input value will be used as the prompt to be categorized.

Input Type – `Union[String | Object]`

If the supplied input value is an `Object` then the property named `prompt` will be used; otherwise the value will be converted to a `String` and used as-is.

Output Type

Produces an `Object` with the best matching category and confidence score for each category. For example, given the categories of “Land”, “Sea”, and “Sky” an input value of “A whale” would produce the following output:

```
{
   "category": [
      "Sea"
   ],
   "categoryScores": {
      "Land": 0,
      "Sea": 100,
      "Sky": 0
   }
}
```

Configuration Properties

- **categorizerLLM** (`LLM`) – A *required* reference to the LLM used to perform the categorization.
- **categories** (`String[]`) – A *required* list of the categories to use.

Runtime Configuration – See LLM.

CodeBlock

Runs a specified block of code to process the input. The code can be written in either VAIL or Python. The input value is supplied via the predefined `input` variable. The runtime configuration value (if any) is supplied via the predefined `config` variable. The supplied block **must** contain an explicit `return` statement for any value it wishes to return (there is no implied return value).

For example, the following `VAIL` code would take the value of the *num* property from the input and return that value plus one:

```
return input.num + 1
```

The equivalent Python code is:

```
return input["num"] + 1
```

The memory `flowstate` is available to any code block component, and its content is shared among all code blocks within the same flow execution.

For example, the following Python code increments the value of a `count` property in the `flowstate` memory:

```
flowstate['count'] = flowstate.get('count', 0) + 1
return input
```

Input Type – User specified, defaults to `Any`

Output Type – User specified, defaults to `Any`

Configuration Properties

- **codeBlock** (`Code Block`) – The *required* block of code to be executed, along with the language being used. The language must be either `VAIL` or `Python` .

Runtime Configuration – None

CodeSplitter

Splits the provided content based on the keywords used by the specified programming language.

Input Type – `Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`

Any input provided will converted to an array of langchain_core.documents.Document instances prior to processing. If the input is a `String` (or `String[]`) then the document(s) created will have their *page_content* attribute set to that string and will have no *metadata*. If the input is an `Object` (or `Object[]`) then the document(s) created will have the *page_content* and *metadata* attributes set from the corresponding object properties.

Output Type – langchain_core.documents.Document[]

Configuration Properties

- **language** (`Enum`) – The *required* language used to interpret the provided content. Legal values are: `js` , `ts` , `java` , `python` , `cpp` , `go` , `kotlin` , `php` , `proto` , `rst` , `ruby` , `rust` , `scala` , `swift` , `latex` , `sol` , `csharp` , `cobol` , `c` , `lua` , `perl` , and `haskell` .

Runtime Configuration – None

CohereRerank

Uses a Cohere re-ranking model to order a collection of documents based on their relevance to a query.

Input Type – `Object`

The input must be a single `Object` with the properties:

- **query** (`String`) – the retrieval query being tested.
- **documents** (`Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`) – the document(s) to process.

Output Type – langchain_core.documents.Document[]

Configuration Properties

- **model** (`String`) – The name of the re-ranking model to use (can be any model supported by the Cohere API).
- **cohere_api_key** (`String`) – The API key required to access to model provider (if any). Supports the use of @secrets to reference a Vantiq secret value (recommended).
- **top_n** (`Integer`) – The maximum number of documents to include in the reordered list (default is 3).

Runtime Configuration – None

Consensus

Accepts as input an `Object` whose properties are assumed to refer to a collection of statements from which a consensus should be derived. The statements are submitted to the specified LLM with instructions to determine if a consensus exists and if so to produce a response which embodies that consensus. The response will draw from the supplied statements, but will not match any of them. If no consensus can be determined it will return `No consensus found.` . Effective at reducing “hallucinations” or otherwise inconsistent results.

Input Type – `Object`

Output Type – `String`

Configuration Properties

- **consensusLLM** (`LLM`) – The *required* LLM used to form the consensus response.

Runtime Configuration

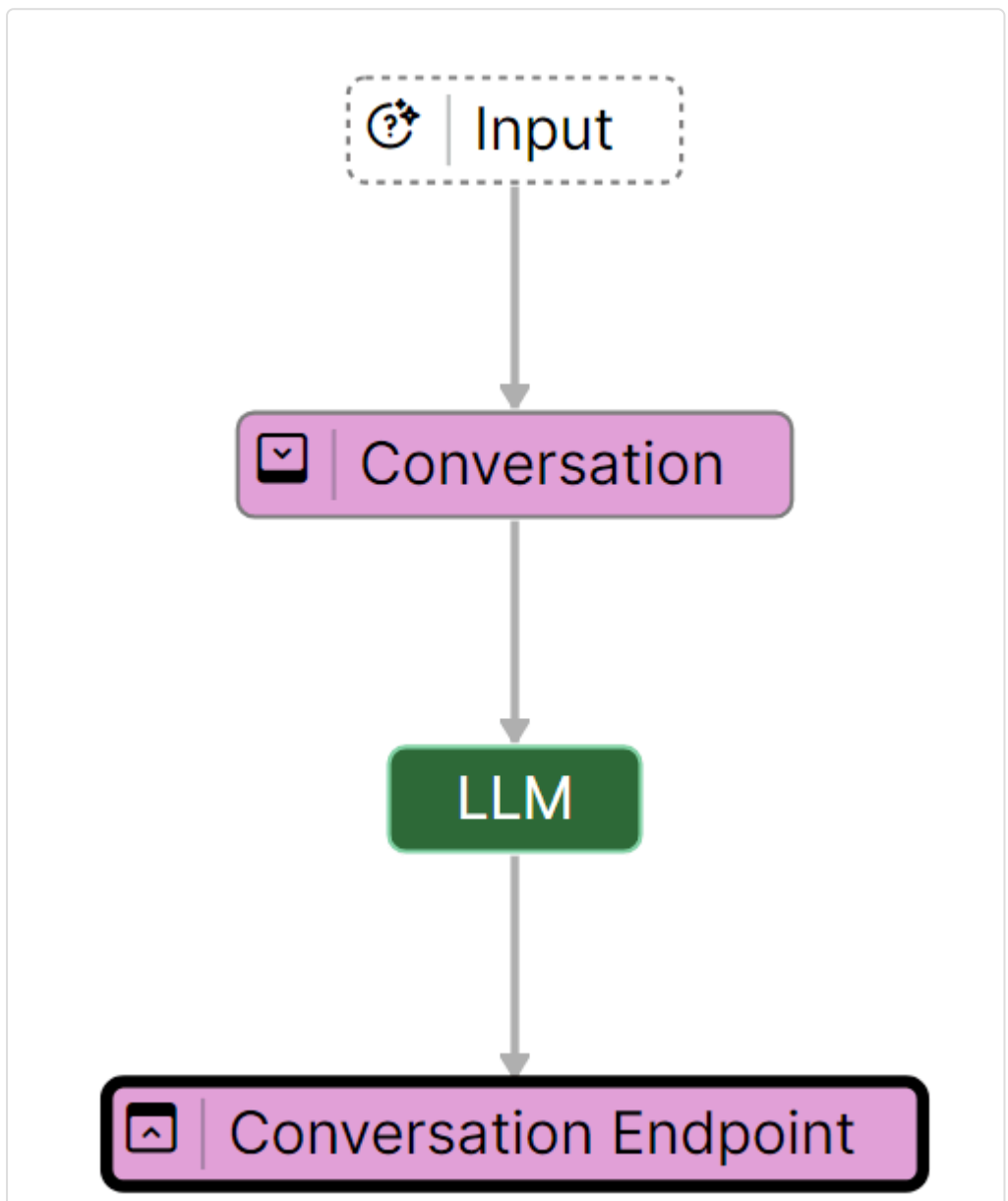
Supports all properties defined by the LLM resource component.

Conversation

Manages conversational history for the enclosed sub-flow (there can be only one). The conversation is identified via the runtime configuration property **conversationId**. Each time the sub-flow is executed, the current conversation state will be injected into its request and both the initial input and result of the sub-flow will be added to the conversation state at the end. The details of how this occurs depend on the input and output types and the configuration of the conversation task.

By default, using a conversation id that does not refer to an active conversation will generate a runtime error. This means that it is the responsibility of the caller of the GenAI Flow to establish any conversation prior to invoking a flow that requires one. It is possible to configure the conversation such that it is "optional". In this case if there is no conversation id present, the conversation task is effectively ignored.

The simplest way to use this component (requiring no configuration) is with a single LLM task as the sub-flow, like this:



Doing this will record all of the LLM interactions as part of the supplied conversation. For example, if this flow was invoked twice, once with the prompt "Why is the sky blue?" and then with "Why not green?", the resulting conversation might look like this:

```
[
   {
      "type": "human",
      "content": "Why is the sky blue?"
   },
   {
      "type": "ai",
      "content": "The sky appears blue because of the way Earth's atmosphere scatters sunlight. The molecules in the air scatter sh
   },
   {
      "type": "human",
      "content": "Why not green?"
   },
   {
      "type": "ai",
      "content": "Although the sky appears blue to our eyes, it can also appear green under certain conditions, such as during a th
   }
]
```

It is important to note that while this example is simple, it is also somewhat of an outlier. In most cases the enclosed sub-flow will contains multiple tasks and the developer will need to properly configure the *Conversation* task to control its use. The GenAI Builder tutorial has additional examples of how to leverage conversations in your flows.

Input Type – `Union[Object | io.vantiq.ai.ChatMessage[]]`

The initial input to the conversation sub-flow is either an `Object` or an array of `io.vantiq.ai.ChatMessage` instances. If the input is an `Object`, then the conversation component will expect to find the input messages in the property specified by **inputMessagesKey**. The value can be either a `String`, a `io.vantiq.ai.ChatMessage` or a `io.vantiq.ai.ChatMessage[]`.

Output Type – `Union[String | Object | io.vantiq.ai.ChatMessage | io.vantiq.ai.ChatMessage[]]`

The output of the sub-flow must be either:

- A `String` which will be treated as the content of a `io.vantiq.ai.ChatMessage` of type *AI*.
- A `io.vantiq.ai.ChatMessage` or `io.vantiq.ai.ChatMessage[]`.
- An `Object` with a key (determined by **outputMessagesKey**) whose value is a `io.vantiq.ai.ChatMessage` or `io.vantiq.ai.ChatMessage[]`.

Configuration Properties

- **optional** (`Boolean`) – Indicates whether or not a conversation is required. When `false` (the default) it is an error to execute the sub-flow without a valid **conversationId**.
- **inputMessagesKey** (`String`) – The name of the key in which the input message can be found. Must be specified if the sub-flow accepts an `Object` as input. The default value is “input”.
- **outputMessagesKey** (`String`) – The name of the key in which the output message can be found. Must be specified if the sub-flow returns an `Object` as output. The default value is “output”.
- **historyMessagesKey** (`String`) – The name of the key used to add the message history to the input. Must be specified if the sub-flow accepts an `Object` as input and expects a separate key for historical messages.

Runtime Configuration

- **conversationId** (`String`) – An opaque value which identifies a Vantiq conversation.

ConversationMemory

Constructs a list of Document instances from the current state of a conversation.

Input Type – `Union[String | Object]`

If the input type is a `String` then it must be a legal conversation id. If it is an `Object`, then it must have the following properties:

- *conversation_id* (`String`) – the id of the conversation to load.
- *metadata* (`Object`) – the metadata to add to each created document.

Output Type – langchain_core.documents.Document[]

There is one document instance for each chat message found in the conversation.

Configuration Properties – None

Runtime Configuration – None

ExtractRelevantContent

Extracts content from the supplied documents based on its relevance to the query.

Input Type – `Object`

The input must be a single `Object` with the properties:

- **query** (`String`) – the retrieval query being tested.
- **documents** (`Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`) – the document(s) to process.

Output Type – langchain_core.documents.Document[]

Configuration Properties

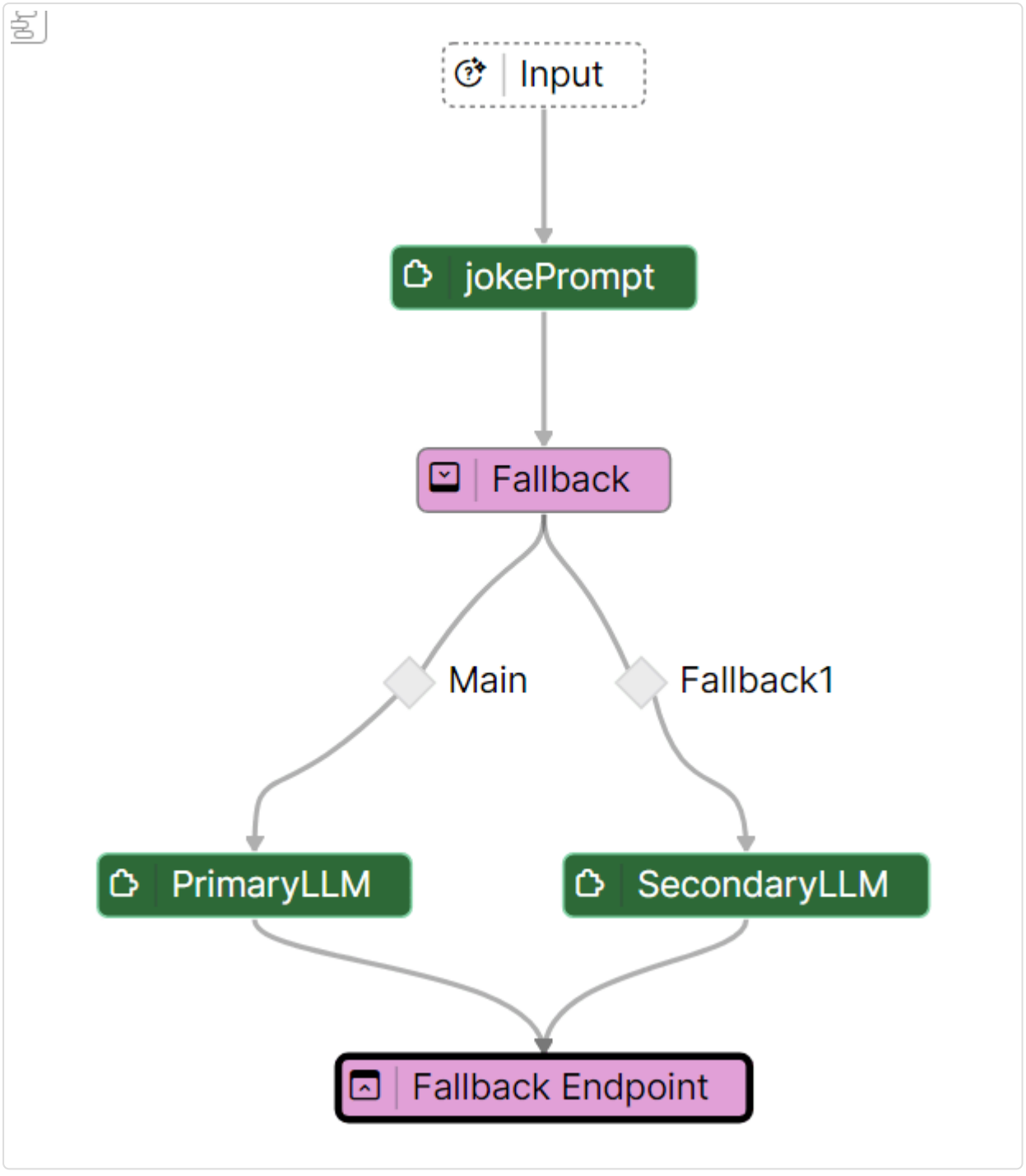
- **llm** (`LLM`) – A *required* reference to an LLM resource.

Runtime Configuration

Supports all values specified by the LLM component.

Fallback

Supports execution of a ‘fallback’ sub-flow, if the primary flow fails. Multiple such fallbacks can be provided and they will be tried in order. For example, the following flow will first try sending the formatted prompt to the primary LLM and if that fails, it will try a secondary one:



Input Type – computed based on the inputs expected by each sub-flow.

Output Type – computed based on the outputs produced by each sub-flow.

Configuration Properties

- **fallbackCount** (`Integer`) – The number of fallback sub-flows to configure. Must be an integer that is greater than or equal to one.

Runtime Configuration – None

FilterForRelevance

Filters the documents to remove any that lack content relevant to the query.

Input Type – `Object`

The input must be a single `Object` with the properties:

- **query** (`String`) – the retrieval query being tested.
- **documents** (`Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`) – the document(s) to process.

Output Type – langchain_core.documents.Document[]

Configuration Properties

- **llm** (`LLM`) – A *required* reference to an LLM resource.

Runtime Configuration

Supports all values specified by the LLM component.

GuardrailsAI

A local implementation of the Guardrails-AI toolkit with a limited set of Validators. It's able to restrict LLM inputs and outputs, and fix them up to a limited degree.

Input Type – `String`

The input must be a single `String`, and is expected to be either an LLM prompt or an LLM response.

Output Type – `String`

The output will be the same as the input or, if a Validator has "fix" as its `on_fail` action, a fixed-up version of the input String.

Configuration Properties

- **validators** (`Object[]`) – A *required* array of Validator configurations. The expected format for the configurations is `{name: <Validator name>, parameters: {<Validator parameters>}}`. E.g. `{'name': 'RestrictToTopic', parameters: {invalid_topics: ["crime"], on_fail: "exception"}}`.

The available Validators are:

- **BanList** – Catches words explicitly included in a ban list. See here for documentation.
- **DetectPII** – Catches identifying information such as names, email addresses, and SSNs. See here for documentation. The list of possible information that can be identified and redacted can be found here
- **LowerCase** – Catches upper-case letters. Mostly intended for testing. See here for documentation.
- **ProfanityFree** – Catches profanity. See here for documentation.
- **RestrictToTopic** – Catches sentences that either cover banned topics or don't cover allowed topics. See here for documentation. Note that we disable the LLM fallback option, so the topic detection will rely entirely on the classifier. This is the classifier used by this Validator.

Runtime Configuration – None

HTMLSplitter

Splits the provided HTML content using HTML tags.

Input Type – `Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`

Any input provided will converted to an array of langchain_core.documents.Document instances prior to processing. If the input is a `String` (or `String[]`) then the document(s) created will have their *page_content* attribute set to that string and will have no *metadata*. If the input is an `Object` (or `Object[]`) then the document(s) created will have the *page_content* and *metadata* attributes set from the corresponding object properties.

Output Type – langchain_core.documents.Document[]

Configuration Properties

- **chunk_size** (`Integer`) – Maximum size of chunks to return. Applies when a given element is longer than this value.
- **chunk_overlap** (`Integer`) – Maximum overlap in characters between chunks. Chunks will only overlap when split because the maximum chunk size is exceeded.

Runtime Configuration – None

LLM

Submits a prompt to an LLM and returns the result (possibly after parsing/formatting).

Input Type – `Union[String | langchain_core.prompt_values.PromptValue | io.vantiq.ai.ChatMessage[]]`

Output Type – Determined by the **outputType** property, default is `String`.

Configuration Properties

- **llm** (`LLM`) – A *required* reference to an LLM resource.
- **outputType** (`Enum`) – Indicates how to parse the response from the LLM and transform it based on the given type. The legal output types are:
  - *String* (default) – return the response from the LLM as a `String`.
  - *Json* – parse the LLM response as JSON and return the resulting value.
  - *Boolean* – parse the LLM response as a boolean value. The text *TRUE* or *YES* is interpreted as `true`, while *FALSE* or *NO* is interpreted as `false` (all checks are case insensitive).
  - *DateTime* – parse the LLM response as an ISO-8601 date/time value.
  - *CSV* – parse the LLM response as a comma separated list.
  - *Numbered List* – parse the LLM response as a numbered list.
  - *Markdown List* – parse the LLM response using the Markdown list format.
  - *Token Count* – return the size of the input value in terms of LLM tokens (does not submit the request to the LLM).
  - *None* – returns the raw response from the LLM as a LangChain *BaseMessage*.

- **outputTypeSchema** (`Type`) – Uses structured output to format the response object into the provided schema. Properties not in the schema will be ignored by the parser. Only used when the outputType is *Json*.
    - Most models do not need you to specify the schema in the prompt. It is strongly advised that you include comprehensive descriptions for each property in the Type used, so that the LLM will know what information belongs there.

Runtime Configuration

- **temperature** (`Real`) – The temperature value to use when calling the LLM.
- **max_tokens** (`Integer`) – Maximum number of tokens that will be generated.
- **top_p** (`Real`) – Maximum cumulative probability for words sampled during generation.
- **stop** (`String[]`) – The 'stop words' to provide to the LLM. Token generation terminates once any of these are generated.
- **presence_penalty** (`Real`) – Penalty applied for words that have already appeared in the generated text. Used to encourage the model to include a diverse range of tokens in the generated text.
- **frequency_penalty** (`Real`) – Penalty applied based on the frequency of the appearance of a token in the generated text. Used to discourage the model from repeating the same words or phrases too frequently within the generated text.

ListwiseRerank

Uses an LLM to perform a "listwise" re-ranking of the documents based on their relevance to the query.

Input Type – `Object`

The input must be a single `Object` with the properties:

- **query** (`String`) – the retrieval query being tested.
- **documents** (`Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`) – the document(s) to process.

Output Type – langchain_core.documents.Document[]

Configuration Properties

- **llm** (`LLM`) – A *required* reference to an LLM resource.
- **top_n** (`Integer`) – The maximum number of documents to include in the reordered list (default is 3).

Runtime Configuration

Supports all values specified by the LLM component.

Loop

Execute the enclosed sub-flow while the specified condition evaluates to `true`. The condition is checked at the start of each iteration, and the loop terminates when the condition evaluates to `false`.

Input Type – computed based on the input expected by the sub-flow.

Output Type – computed based on the output produced by the sub-flow.

Configuration Properties

- **condition** (`Code Block`) – The *required* condition to evaluate at the beginning of each iteration. The condition can be written in either VAIL or Python and must return a *Boolean* value.
- **loopLimit** (`Integer`) – The maximum number of iterations allowed. If the loop exceeds this limit, it terminates with an error.

The `flowstate` memory is accessible to the condition code block.

Map

Concurrently executes the contained sub-flow once for each item in the input array. Returns an array containing each of the resulting values. These values are *not* guaranteed to be in the same order as the inputs that produced them.

Input Type – computed based on the input expected by the sub-flow.

Output Type – computed based on the output produced by the sub-flow.

Configuration Properties

- **withIndex** (`Boolean`) – Whether or not to return the index of the input value used to produce each output value. When set to `true`, the return will be an Object with the **index** and **value** properties. Otherwise, only the value produced by the sub-flow is returned.
- **maxConcurrency** (`Integer`) – Maximum number of concurrent executions. Must be an Integer greater than or equal to `1`. In general this should not be set unless sequential execution is required.

MarkdownSplitter

Splits the provided Markdown content using Markdown formatting tags.

Input Type – `Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`

Any input provided will converted to an array of langchain_core.documents.Document instances prior to processing. If the input is a `String` (or `String[]`) then the document(s) created will have their *page_content* attribute set to that string and will have no *metadata*. If the input is an `Object` (or `Object[]`) then the document(s) created will have the *page_content* and *metadata* attributes set from the corresponding object properties.

Output Type – langchain_core.documents.Document[]

Configuration Properties

- **chunk_size** (`Integer`) – Maximum size of chunks to return. Applies when a given element is longer than this value.
- **chunk_overlap** (`Integer`) – Maximum overlap in characters between chunks. Chunks will only overlap when split because the maximum chunk size is exceeded.

Runtime Configuration – None

NativeLCEL

Executes a Python code block which must return an instance of a LangChain component. The behavior of the task will be determined by the LangChain component instance created.

An example of this in use is shown in the NativeLCEL section of the GenAI Builder tutorial.

Input Type – specified by the user.

Output Type – specified by the user.

Configuration Properties

- **codeBlock** (`Code Block`) – the Python code that will be executed in order to create the LangChain component.

Runtime Configuration – depends on the LangChain component created

NeMoGuardrails

An implementation of the Nvidia's NeMo Guardrails toolkit. Used to protect against invalid or improper prompts and responses, and to tailor LLM prompts or responses for certain topics. Intended for more complex or comprehensive use-cases, compared to the Guardrails-AI toolkit.

Input Type – `langchain_core.prompt_values.PromptValue`

Output Type – `io.vantiq.ai.ChatMessage`

Configuration Properties

- **generativeLlm** (`LLM`) – The *required* name of the LLM used by NeMo for internal LLM calls and response generation. Must be a "generative" type LLM.
- **embeddingsLlm** (`LLM`) – The *required* name of the LLM that NeMo will use when it needs to create embeddings and generate intents from messages. Must be an "embedding" type LLM.
- **colang** (`ResourceReference[documents]`) – A CoLang document, which specifies the flows available to the LLM when handling the prompt. See here for Nvidia's CoLang documentation. Note that only CoLang v1 is currently supported.
- **yamlConfig** (`ResourceReference[documents]`) – Specifies the configuration of the NeMo instance. Largely used for prompts and for general instructions to the LLM. Anything that the NeMo documentation says should go into "config.yml" should go in this document. Expected to be in YAML format. Specify "vantiq_llm" as the model where models should be specified. See here for information on prompts. Make sure to see our tutorial for warnings if you intend to set properties in "rails", "models", or "actions_server_url".
- **inputRails** (`String[]`) – The CoLang flows to apply to inputs to the LLM. These may be any flows specified in the "colang" property, as well as NeMo's pre-defined flows such as "self check input". See here for documentation on input rails.
- **outputRails** (`String[]`) – The CoLang flows to apply to outputs of the LLM. These may be any flows specified in the "colang" property, as well as NeMo's pre-defined flows such as "self check output". See here for more documentation on output rails.
- **customModels** (`Object[]`) – NeMo allows models to be set for specific flows. To add a Vantiq LLM, use "vantiq_llm" as the engine property for generative LLMs or "vantiq_embedding_llm" for embedding LLMs, and include the LLM's name as "llm_name" in the parameters. E.g. `{type: <flow name>, engine: "vantiq_llm"|"vantiq_embedding_llm", parameters: {llm_name: <vantiq LLM name>}}`. Note that `embeddingsLlm` and `generativeLlm` are automatically used for the "embeddings" and "main" types, respectively. See here for documentation on what flows are available.
- **actionsServer** (`String`) – The URL for a server that handles any custom actions required by the flows. See here for documentation on the creation and use of an actions server.

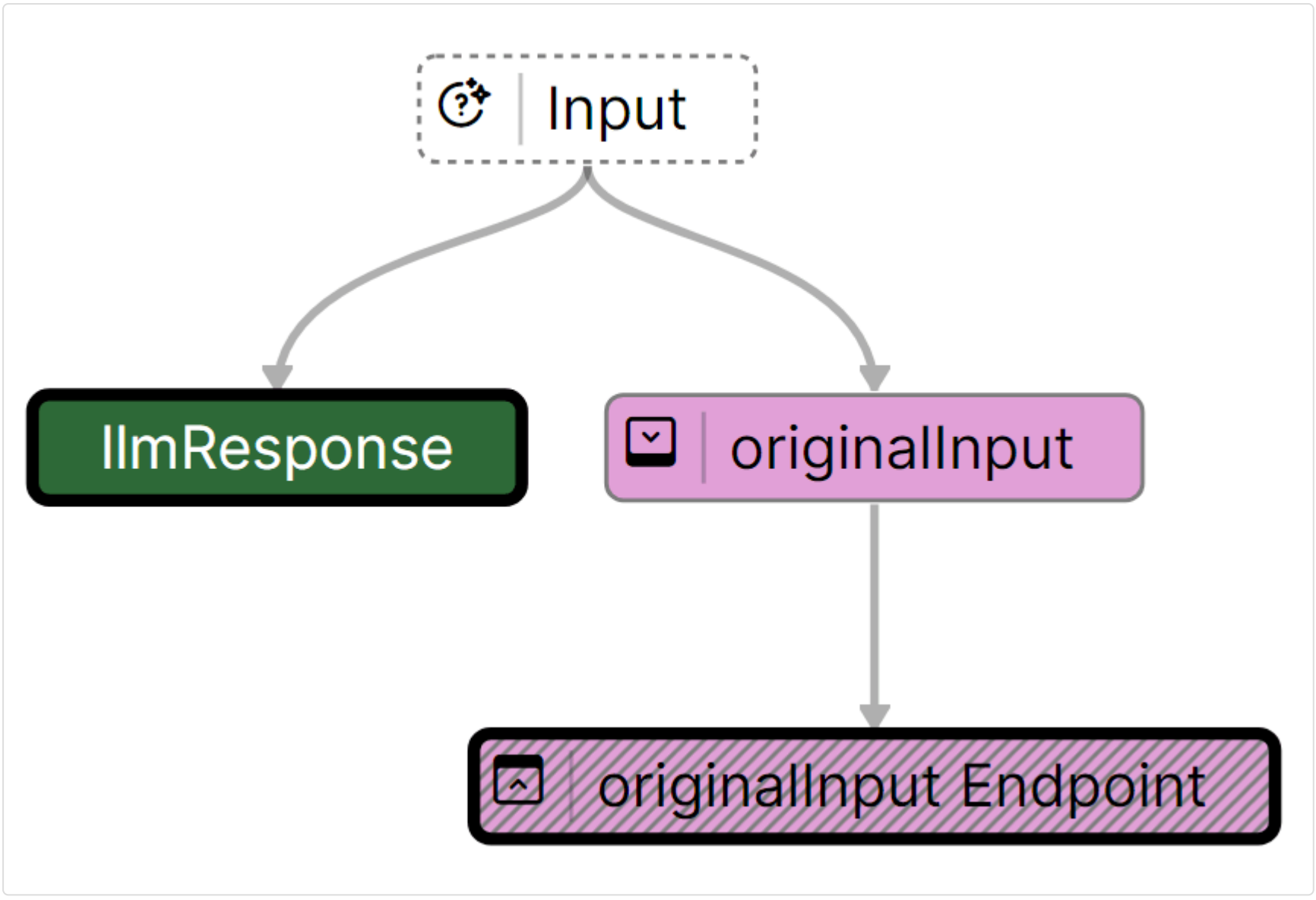
Runtime Configuration – None

Optional

Allows the contained sub-flow to be disabled via configuration (**not** runtime configuration).

> Since the choice of whether to execute the optional flow is made at configuration time, this component is only useful when creating a GenAI Component which will expose the configuration property to its users.

For example, in the following GenAI Flow, the task *originalInput* is used to optionally return the input to the flow in addition to the LLM response:



When the task is enabled, the result is:

```
{
   "llmResponse": "The sky appears blue because of the way Earth's atmosphere scatters sunlight. The molecules in the air scatter sl
   "originalInput": "Why is the sky blue?"
}
```

When it is disabled the output is:

```
{
   "llmResponse": "The sky appears blue because of the way Earth's atmosphere scatters sunlight. The molecules in the air scatter sl
}
```

Input Type – computed based on the input expected by the sub-flow.

Output Type – computed based on the output produced by the sub-flow.

Configuration Properties

- **enabled** (`Boolean`) – Whether or not the sub-flow should be executed. If 'true' (the default), then the sub-flow will be executed.

Runtime Configuration – None

ParagraphSplitter

Accepts textual content and splits it by paragraph. Prioritizes keeping all paragraphs (and then sentences, and then words) together as long as possible, while maximizing the chunk size subject to the configured limit.

Input Type – `Union[String | String[] | Object | Object[] | langchain_core.documents.Document | langchain_core.documents.Document[]]`

Any input provided will converted to an array of langchain_core.documents.Document instances prior to processing. If the input is a `String` (or `String[]`) then the document(s) created will have their *page_content* attribute set to that string and will have no *metadata*. If the input is an `Object` (or `Object[]`) then the document(s) created will have the *page_content* and *metadata* attributes set from the corresponding object properties.

Output Type – langchain_core.documents.Document[]

Configuration Properties

- **chunk_size** (`Integer`) – Maximum size of chunks to return. Applies when a given element is longer than this value.
- **chunk_overlap** (`Integer`) – Maximum overlap in characters between chunks. Chunks will only overlap when split because the maximum chunk size is exceeded.

Runtime Configuration – None

Procedure

Executes a specified VAIL procedure. The procedure must be a Vantiq Procedure, and the provided parameters serve as its input.

> The procedure will be invoked using the permissions granted by the GenAI Flow Service connector's token and not those of the user who invoked the GenAI Flow. This may result in the invocation taking place at a lower privilege level than expected. Given this you may need to use the procedure's WITH Clause to elevate to the correct privilege level.

Input Type – specified by the user.

Output Type – if configured to use a procedure reference then this is based on the return type of that procedure. Otherwise, it must be specified by the user.

Configuration Properties

- **procedure** (`Union[ResourceReference[procedures] | Procedure Name Expression]`) – The *required* Vantiq procedure to execute. Can be given as either a direct reference to a procedure or as an expression used to produce the procedure name at runtime.
- **parameters** (`Code Block`) – Code that returns an Object (VAIL) or a dictionary (Python) representing the procedure parameters. Keys correspond to parameter names, and values correspond to their respective values.

The `flowstate` memory is accessible to the procedure name definition expression and the parameters code block.

PromptFromMessages

Creates an LLM prompt from the supplied list of ChatMessages instances. These may reference a template (which behave just as they do for PromptFromTemplate]) or an already existing list of message instances.

Input Type – `Union[String | Object | langchain_core.documents.Document]`

Normally the input to a *PromptFromMessages* task is expected to be an `Object` the properties of which are used to replace any matching template variables or to resolve "placeholder" variable references (see below for details). However, if the template has **exactly** one variable (no more, no less), then it is legal to supply any value as the input. If the input is *not* an `Object`, then the task will convert it to a String and use it as the value for the template variable. It is also legal to supply a LangChain document instance as input. In this case the document will be converted to an `Object` value with the `page_content` property set to the document's content and all other properties taken from the document's `metdata` value.

Output Type – `langchain_core.prompt_values.PromptValue`

Configuration Properties

- **messageTemplates** (`langchain_core.prompt.MessageLikeRepresentation`) – A *required* list of messages to create. There must be at least one entry in the list. The entries must be of the following types:
  - `system` – Creates an LLM "system" message from the provided template.
  - `ai` or `assistant` – Creates a message representing a response from the LLM from the provided template.
  - `human` or `user` – Creates a message representing input from the user from the provided template.
  - `placeholder` – Creates a message representing a spot where an additional list of messages will be inserted. The value provided is the name of the input key holding the message instances. This is typically from a conversation using the **historyMessagesKey**.
- **defaultValues** (`Labeled Expression[]`) – A mapping of template substitution variables to their default values (computed via an expression). These default values are used to populate the template so that you don't need to pass them in every time you call the prompt. The current input is available via the predefined `input` variable.

Runtime Configuration – None

PromptFromTemplate

Creates an LLM prompt from the supplied template which is either a `String`, a Vantiq Document, or a URL. The input value is used to substitute values into the template using either Vantiq Template syntax or Python f-string syntax.

> The fact that f-string syntax is supported means that any literal brace characters must be escaped using `{{` or `}}`.

Input Type – `Union[String | Object | langchain_core.documents.Document]`

Normally the input to a *PromptFromTemplate* task is expected to be an `Object` the properties of which are used to replace any matching template variables. However, if the template has **exactly** one variable (no more, no less), then it is legal to supply any value as the input. If the input is *not* an `Object`, then the task will convert it to a String and use it as the value for the template variable. It is also legal to supply a LangChain document instance as input. In this case the document will be converted to an `Object` value with the `page_content` property set to the document's content and all other properties taken from the document's `metdata` value.

Output Type – `langchain_core.prompt_values.PromptValue`

Configuration Properties

- **promptTemplate** (`Union[String | ResourceReference[documents]]`) – A *required* template from which the prompt will be created.
- **defaultValues** (`Labeled Expression[]`) – A mapping of template substitution variables to their default values (computed via an expression). These default values are used to populate the template so that you don't need to pass them in every time you call the prompt. The current input is available via the predefined `input` variable.

Runtime Configuration – None

RAG

Submits a question to a flow that utilizes a Semantic Index to generate a response from the configured LLM.

Input Type – `Union[String | Object]`

If the input value is an `Object`, it must have the property `question`.

Output Type – `Object`

Produces an `Object` with the properties:

- **question** (`String`) – the submitted question.
- **context** (`Object[]`) – the entries from the semantic index used to formulate the answer.
- **answer** (`String`) – the answer generated by the Q&A LLM.

For example, given a semantic index containing information about various physical phenomenon and the question "Why is the sky blue?", the response might look like:

```
{
  "question" : "Why is the sky blue?",
  "context" : [ {
    "page_content" : "Sep 8, 2017,10:00am EDT This article is more than 6 years old. The combination of a blue sky, dark overhead,
    "metadata" : {
      "fileName" : "BlueSky.txt",
      "filetype" : "text/plain",
      "languages" : [ "eng" ],
      "id" : "a7238450-0d67-11ef-8253-482ae351284f",
      "filename" : "BlueSky.txt",
      "_id" : "96a0dc07-dd80-4113-a354-f2dfac36d191",
      "_collection_name" : "TestIndex__UVNS1"
    },
    "type" : "Document"
  }, {
    "page_content" : "of three simple factors put together: that sunlight is made out of light of many different wavelengths, that
    "metadata" : {
      "fileName" : "BlueSky.txt",
      "filetype" : "text/plain",
      "languages" : [ "eng" ],
      "filename" : "BlueSky.txt",
      "id" : "a7238450-0d67-11ef-8253-482ae351284f",
      "_id" : "ecb50681-dac8-4243-babe-6c0e1dbbb41a",
      "_collection_name" : "TestIndex__UVNS1"
    },
    "type" : "Document"
  }, {
    "page_content" : "the fact that sunlight is made up of light of many different wavelengths, that atmospheric particles are very
    "metadata" : {
      "filetype" : "text/plain",
      "fileName" : "BlueSky.txt",
      "languages" : [ "eng" ],
      "id" : "a7238450-0d67-11ef-8253-482ae351284f",
      "filename" : "BlueSky.txt",
      "_id" : "8554363f-ae0f-4d99-bacb-0a682219f7bf",
      "_collection_name" : "TestIndex__UVNS1"
    },
    "type" : "Document"
  }, {
    "page_content" : "Why The Sky Is Blue, According To Science Ethan Siegel Senior Contributor Starts With A Bang Contributor Grou
    "metadata" : {
      "filetype" : "text/plain",
      "fileName" : "BlueSky.txt",
      "languages" : [ "eng" ],
      "id" : "a7238450-0d67-11ef-8253-482ae351284f",
      "filename" : "BlueSky.txt",
      "_id" : "91d5e489-2ef7-4062-9af9-5cc969ff63c6",
      "_collection_name" : "TestIndex__UVNS1"
    },
    "type" : "Document"
  } ],
  "answer" : "The sky is blue because of the scattering of different-wavelength light by Earth's atmosphere and the sensitivity of
}
```


Configuration Properties

- **semanticIndex** (`SemanticIndex`) – A *required* reference to the Semantic Index to use for the similarity search.
- **qaLLM** (`LLM`) – A reference to the LLM used to generate the response based on the question and context.
- **useConversation** (`Boolean`) – Specifies whether or not to participate in a conversation when formulating the prompt. The id of the conversation is provided via the `conversationId` configuration property.

Runtime Configuration – Supports the properties used by the LLM and SemanticIndex components.

ReduceDocuments

Uses the given prompt and LLM to reduce an array of documents to a single result. This is used as the final step of a Map/Reduce algorithm. For example, when producing a summary of a large number of documents, which is typically implemented by summarizing each document individually and then producing a summary of those summaries.

> If you know in advance that the total size of the prompt and the documents to be processed will not exceed the LLM's token buffer, then this step can be accomplished by simply "stuffing" all the document content into the prompt and submitting it to the LLM as a single request. This component is only useful when it is not possible to guarantee that the token buffer will not be exceeded by such a request.

Input Type – `Union[String[] | Object[] | langchain_core.documents.Document[]]`

Output Type – `langchain_core.documents.Document`

Configuration Properties

- **reducePrompt** (`Union[String | ResourceReference[documents]]`) – A *required* template from which the prompt will be created.
- **reduceLLM** (`LLM`) – A reference to the LLM used to process the reduce request(s).
- **maxTokensPerBatch** (`Integer`) – The maximum number of tokens permitted in a single reduce request. It is an error if any of the supplied documents exceeded this threshold. The default value is `8196` .
- **maxPartialReductions** (`Integer`) – The maximum number of "partial" reductions allowed before the final result is achieved. If no value is provided then the reduction process will continue until done.

Repeat

Repeats a sub-flow multiple times, with different configurations for each instance of the flow. The contained sub-flow can only have a single task. If multiple tasks are required, then a GenAI Component must be defined to contain them. This component is useful when defining a GenAI Component or to avoid having to create multiple tasks based on the identical component (can make the flow easier to read).

Input Type – computed based on the input expected by the sub-flow.

Output Type – computed based on the output produced by the sub-flow.

Configuration Properties

- **repeatProperties** (`String[]`) – A *required* list of properties in the sub-flow that will be different in each instance of the sub-flow. The expected format is `<sub-flow task name>.<property name>` .
- **repeatPropertyValues** (`Object`) – The values for properties in the sub-flow that will be different in each instance of the sub-flow. Each property in the supplied `Object` should match a value in **repeatProperties**, and the values should all be arrays of equal size. When used in the definition of a GenAI Component, this value must be null or unset.

Runtime Configuration – None

SemanticIndex

Queries the specified semantic index and returns an `Array` of similar documents.

Input Type – `String`

Output Type – determined by the **contentOnly** configuration property. When it is unchecked (the default), the output is langchain_core.documents.Document[]. When **contentOnly** is "checked", the output is a `String` .

Configuration Properties

- **semanticIndex** (`Union[SemanticIndex | Expression]`) – A *required* reference to the Semantic Index to use for the similarity search. This can be provided either as a direct reference to a semantic index or as an expression that dynamically resolves to a semantic index name at runtime.
- **contentOnly** (`Boolean`) – When `true` the result will be a `String` containing the combined content of the selected documents (instead of the langchain_core.documents.Document[]). The default value is `false` .
- **minSimilarity** – The minimum similarity value to use when working with the semantic index. This value will override the default value provided by the referenced Semantic Index.
- **metadataFilter** – A filter that will be applied against the documents' metadata before the similarity search.

Runtime Configuration

- **minSimilarity** (`Real`) – The minimum similarity value to use when working with the semantic index. This value will override any previously supplied value, but only for the current execution.
- **limit** (`Integer`) – The maximum number of documents to return from the index (default is 4).

The semantic index name expression definition has access to both the `flowstate` memory and the `config` property.

SemanticIndexStore

Stores the supplied documents in the specified semantic index based on their embeddings. Documents are grouped into semantic index entry as follows:

- If the *metadata* for a given document instance has a value for the *id* property, then this will be used as the index entry id.
- Otherwise, the system will assign the document a "default" index entry id.

The "default" index entry id can be specified as part of the runtime configuration. If it is not specified, then it will be an automatically generated UUID. All documents with the same value of the *id* property will be stored in the same index entry. If the id refers to an existing entry, then its contents will be replaced with the provided documents. Otherwise, a new entry will be created.

Input Type – langchain_core.documents.Document[]

Output Type – Semantic Index instance

Configuration Properties

- **semanticIndex** (`Union[SemanticIndex | Expression]`) – A *required* reference to the Semantic Index used to store the documents. This can be provided either as a direct reference to a semantic index or as an expression that dynamically resolves to a semantic index name at runtime.

Runtime Configuration

- **default_entry_id** (`String`) – The index entry id to use for any document which does not have one specified.

The semantic index name expression definition has access to both the `flowstate` memory and the `config` property.

SemanticIndexWithCompression

Queries the specified semantic index and returns an Array of similar documents after processing by the specified document compressors.

Input Type – `String`

Output Type – determined by the **contentOnly** configuration property. When it is unchecked (the default), the output is langchain_core.documents.Document[]. When **contentOnly** is "checked", the output is a `String` .

Configuration Properties

- **semanticIndex** (`Union[SemanticIndex | Expression]`) – A *required* reference to the Semantic Index to use for the similarity search. This can be provided either as a direct reference to a semantic index or as an expression that dynamically resolves to a semantic index name at runtime.
- **documentCompressorCount** (`Integer`) – The *required* number of document compressors to configure. Must be an integer that is greater than or equal to one.
- **contentOnly** (`Boolean`) – When `true` the result will be a `String` containing the combined content of the selected documents (instead of the langchain_core.documents.Document[]). The default value is `false` .
- **minSimilarity** – The minimum similarity value to use when working with the semantic index. This value will override the default value provided by the referenced Semantic Index.
- **metadataFilter** – A filter that will be applied against the documents' metadata before the similarity search.

Runtime Configuration

- **minSimilarity** (`Real`) – The minimum similarity value to use when working with the semantic index. This value will override any previously supplied value, but only for the current execution.
- **limit** (`Integer`) – The maximum number of documents to return from the index (default is 4).

The semantic index name expression definition has access to both the `flowstate` memory and the `config` property.

Tool

Submits a prompt to an LLM configured for tool calling and returns the result (possibly after parsing/formatting).

Input Type – `Union[String | langchain_core.prompt_values.PromptValue | io.vantiq.ai.ChatMessage[]]`

Output Type – Determined by the **outputType** property, default is `String` .

Configuration Properties

- **llm** (`LLM`) – A *required* reference to an LLM resource.
- **tools** (`Union[ServiceReference | ProcedureReference]`) – A list of VAIL procedures and/or services available to the LLM when responding to the prompt. Each entry is given as a resource reference (e.g. `/services/com.vantiq.MyService`). For services, all public procedures are accessible to the LLM. Each VAIL procedure is treated as a tool that the LLM can request to invoke.
- **mode** (`Enum`) – Specifies the execution mode when the LLM selects a tool to invoke. When a tool is selected, the LLM generates a schema document that describes the tool invocation. If the mode is set to `Execute` (the default), the tool is automatically invoked, and the tool's response is added to the LLM's context. The LLM is then invoked again with this enriched context. If the mode is set to `Return Schema Mapping` , a LangChain *BaseMessage* containing a description of the tools to be invoked (*tool_calls*) is returned, and no further tool execution or LLM processing occurs. Since the output is a *BaseMessage*, the `outputType` should be set to `None` . If set to `Ignore Tools` , no tool is made available to the LLM, functioning similarly to the LLM GenAI Component.
- **authorizer** (`Procedure`) – The name of a procedure used to authorize execution of a tool. 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.
- **customTools** (`Code Block`) – A code block used to define custom tools as Pydantic model classes. When the LLM selects a custom tool, the execution mode always defaults to `Return Schema Mapping` .
- **mcpTools** (`Object`) – MCP Server Tools available to the LLM. The configuration is an `Object` mapping server aliases to MCP server definitions. Each definition is itself an `Object` specifying a `url` , `transport` , and optional `headers` .
- **outputType** (`Enum`) – Indicates how to parse the response from the LLM and transform it based on the given type. The legal output types are: *String* (default), *Json*, *Boolean*, *DateTime*, *CSV*, *Numbered List*, *Markdown List*, and *None*.

Runtime Configuration

- **tools (`Union[ServiceReference | ProcedureReference]`)** – A list of VAIL procedures and/or services which will override the statically configured `tools` value. The semantics are otherwise identical to the `tools` configuration property.
- **temperature** (`Real`) – The temperature value to use when calling the LLM.
- **max_tokens** (`Integer`) – Maximum number of tokens that will be generated.
- **top_p** (`Real`) – Maximum cumulative probability for words sampled during generation.
- **stop** (`String[]`) – The 'stop words' to provide to the LLM. Token generation terminates once any of these are generated.
- **presence_penalty** (`Real`) – Penalty applied for words that have already appeared in the generated text. Used to encourage the model to include a diverse range of tokens in the generated text.

- **frequency_penalty** (`Real`) – Penalty applied based on the frequency of the appearance of a token in the generated text. Used to discourage the model from repeating the same words or phrases too frequently within the generated text.
- **tools** (`ResourceReference`) – List of additional tool references that should be used for this execution.
- **mcpTools** (`Object`) – MCP Server Tools available to the LLM for this execution.
- **toolParameterOverride** (`Object`) – A configuration to override LLM selected tool parameter values.

Transform

Transforms the input based on the configured transformation.

Input Type – User specified, defaults to `Any`

Output Type – User specified, defaults to `Any`

Configuration Properties

- **transformation** (`Union[]`) – A transformation that will be applied to events passing through the component. There are three different options for the transformation.
  - **Visual Transformation** (`ExpressionObject`) – A visual interface that allows users to map inputs to outputs without having to write their own code. Specifies a mapping between keys and the expressions that will become their values.
  - **Projection** (`Expression`) – The expression or input property (for example: 'input.metadata') that will become the output of this transformation.
  - **Transformation code** (`Code Block`)– Code that will transform the input, if you require a particularly complex transformation. Identical to the CodeBlock component.
  - **transformInPlace** (`Boolean`) – Start with the inbound event and apply the transformation only to the properties specified by the transformation object. The remaining properties will be unchanged. Only applies to Visual Expressions, ignored by other types of transformation. Optional, defaults to `false` .

Runtime Configuration

The properties used from the runtime configuration are determined by the user. Access to runtime configuration is provided via the well-known 'config' variable (which may be 'None'), and it is usable with any transformation option.

The `flowstate` memory is accessible to the transformation code block.

UnstructuredURL

Loads content from the supplied URL(s) using the unstructured library. This can process a number of textual and binary formats including Markdown, HTML, DOCX, and PDF.

Using OCR for content extraction

Depending on the type of document being loaded and the selected processing *strategy*, the system may choose to use Tesseract OCR to extract information. Use of OCR is required when extracting content from a binary image such as a JPEG or PNG file. It can also be used to extract information from images embedded in a PDF document. It will not be used in any other case (for example, OCR is not used to process images embedded in a Word Document).

> Use of OCR can result in very long content processing times. Therefore, care should be taken to restrict it to processing of relatively short documents (especially when processing PDFs). This is the primary reason that the default processing strategy is *fast* (as opposed to *auto*). If the processing takes too long, the GenAI Flow will receive a timeout error (default timeout is 90 seconds). This timeout is 2 hours if the flow is being used for content ingestion.

Input Type – `Union[String | String[] | Object | Object[]]`

The input is the URL(s) from which the document content should be loaded; it can be provided in the following forms:

- `String` or `String[]` – either a single URL or an array of URLs. These will be read by the content processing server. The document's content type will be determined based on the URL's file extension (if present) or by examination of the content itself.
- `Object` or `Object[]` – each `Object` represents the source of the content to be processed. The expected properties are:
  - *url* (`String`) – the *required* URL from which to load the content.
  - *content_type* (`String`) – an *optional* "Content-Type" used when processing the content.
  - *headers* (`Object`) – *optional* headers to use when issuing the `GET` request on the URL. These are provided as key/value pairs.
  - *filename* (`String`) – the *optional* file name to use when processing the content.

Output Type – langchain_core.documents.Document[]

The number of document instances depends both on the task configuration and the structure of the content being processed.

Configuration Properties

- **chunking_strategy** (`Enum`) – The chunking strategy used when constructing Document instances. Determines how the content is divided into Documents. Legal values are:
  - *single* – constructs a single document instance, no matter the content structure or size.
  - *basic* – performs simple chunking based solely on size as determined by the following property values:

        - **max_characters** (`Integer`) – create a new chunk before reaching a length of *n* chars (hard max). Defaults to `500` .
        - **new_after_n_chars** (`Integer`) – create a new chunk after reaching a length of *n* chars (soft max). If set, it should be less than **max_characters**.
        - **overlap** (`Integer`) – specifies the length of a string ("tail") to be drawn from each chunk and prefixed to the next chunk as a context-preserving mechanism. By default, this only applies to split-chunks where an oversized element is divided into multiple chunks by text-splitting. Defaults to `0` .
        - **overlap_all** (`Boolean`) – when `true` , apply overlap between "normal" chunks formed from whole elements and not subject to text-splitting. Use this with caution as it entails a certain level of "pollution" of otherwise clean semantic chunk boundaries. Defaults to `false` .
    - *by_title* – chunks the content based on the appearance of "Title" elements. Each such element will automatically create a new chunk. Chunk sizes are limited as per *basic* chunking. The following additional options are supported:
        - **combine_text_under_n_chars** (`Integer`) – combine elements until a chunk reaches a length of n chars. Mitigates against over-chunking caused by elements misidentified as "Title" elements.
        - **multipage_sections** (`Boolean`) – determines if chunks can span multiple pages. Defaults to `true` .
    - *by_page* (default) – chunks the content based on the appearance of page breaks. Each such element will automatically create a new chunk. Chunk sizes are limited as per *basic* chunking (as a result this is the same as *basic* for any content that lacks the concept of pages). The following additional options are supported:
        - **combine_text_under_n_chars** (`Integer`) – combine elements until a chunk reaches a length of n chars. Mitigates against over-chunking caused by pages with limited content.
    - *none* – constructs one document for each raw "element" found in the processed content (what constitutes an element is content type specific).
- **encoding** (`String`) – The encoding method used to decode the text input. Defaults to `utf-8` .
- **include_page_breaks** (`Boolean`) – If `true` , the output will include page breaks if the filetype supports it.
- **strategy** (`Enum`) – The strategy to use for partitioning PDF/image content (this property is ignored for all other cases). The options are:

    - *auto* – select an appropriate strategy based on the type of content being processed. This strategy is free to use OCR even if it would incur a significant performance cost.
    - *fast* (default) – extract as much information as possible **without** the use of OCR. This strategy cannot be used for purely image content types such as JPEG or PNG.
    - *hi_res* – uses a layout model to determine the document elements combined with OCR for text extraction.
    - *ocr_only* – simply extracts any text from the image content via OCR.
- **languages** (`String[]`) – The languages present in the document, for use in partitioning and/or OCR. More than one language means that any of them may appear in the document. See the Tesseract Documentation for a complete list of the supported languages.

- **skip_infer_table_types** (`String[]`) – The document types (pdf, png, etc…) for which you want to skip table extraction.
- **coordinates** (`Boolean`) – If true, return coordinates for each element (when present).
- **extract_image_block_types** (`String[]`) – The types of elements (e.g. 'table', 'image') to extract. For use in extracting image blocks as Base64 encoded data stored in metadata fields.
- **unique_element_ids** (`Boolean`) – When `True` , assign UUIDs to element IDs, which guarantees their uniqueness (useful when using them as primary keys). Otherwise a SHA-256 of element text is used.
- **chunking_options** (`Object`) – The chunking strategy specific options (as detailed above).

Runtime Configuration – None

UnstructuredContent

Loads content using the unstructured library. This can process a number of textual and binary formats including Markdown, HTML, DOCX, and PDF. It is similar to the UnstructuredURL component, but works with content directly. This is useful when you want to use a GenAI Flow as the default ingest procedure for a Semantic Index. The Semantic Index service will break down content into manageable chunks prior to invoking the GenAI Flow then assemble the results for the entry. Depending on the content type, the encoding may be either base64 or a UTF-8 string. The UnstructuredContent component is designed to process this content and return a set of Document instances. One significant advantage to using this mechanism for ingest is that it is tolerant of longer processing times. The service invokes the GenAI Flow asynchronously and waits for up to 2 hours for it to complete.

Input Type – `Union[String | Object]`

If the input is an `Object` , its properties provide the content and describe how to process it. The expected properties are:

- `page_content` – This is a String representation of the content to be processed. It can be either Base64 or UTF-8 encoded.
- `content_encoding` – This is either `utf-8` or `base64` . If unspecified, the default is `base64` .
- `metadata` – Any metadata that should be associated with the content. This is an `Object` with key/value pairs. The following are expected:
- `fileName` – The name of the file from which the content was extracted. If not specified, the default is `unknown` .
- `content_type` – The type of content being loaded. This is a `String` and can be anything supported by the unstructured library. The following are common types:
- `application/pdf` (default) – A PDF document.
- `text/plain` – A plain text document.
- `text/markdown` – A Markdown document.
- `text/html` – An HTML document.

If the input is a `String` , its value will be used as the **page_content** property. The encoding will be `utf-8` and the content type will be `text/plain` .

Output Type – langchain_core.documents.Document[]

The number of document instances depends both on the task configuration and the structure of the content being processed.

Configuration Properties

The configuration properties are the same as those for the UnstructuredURL component. They control the behavior of Unstructured API library when processing the content. This functionality underpins both.

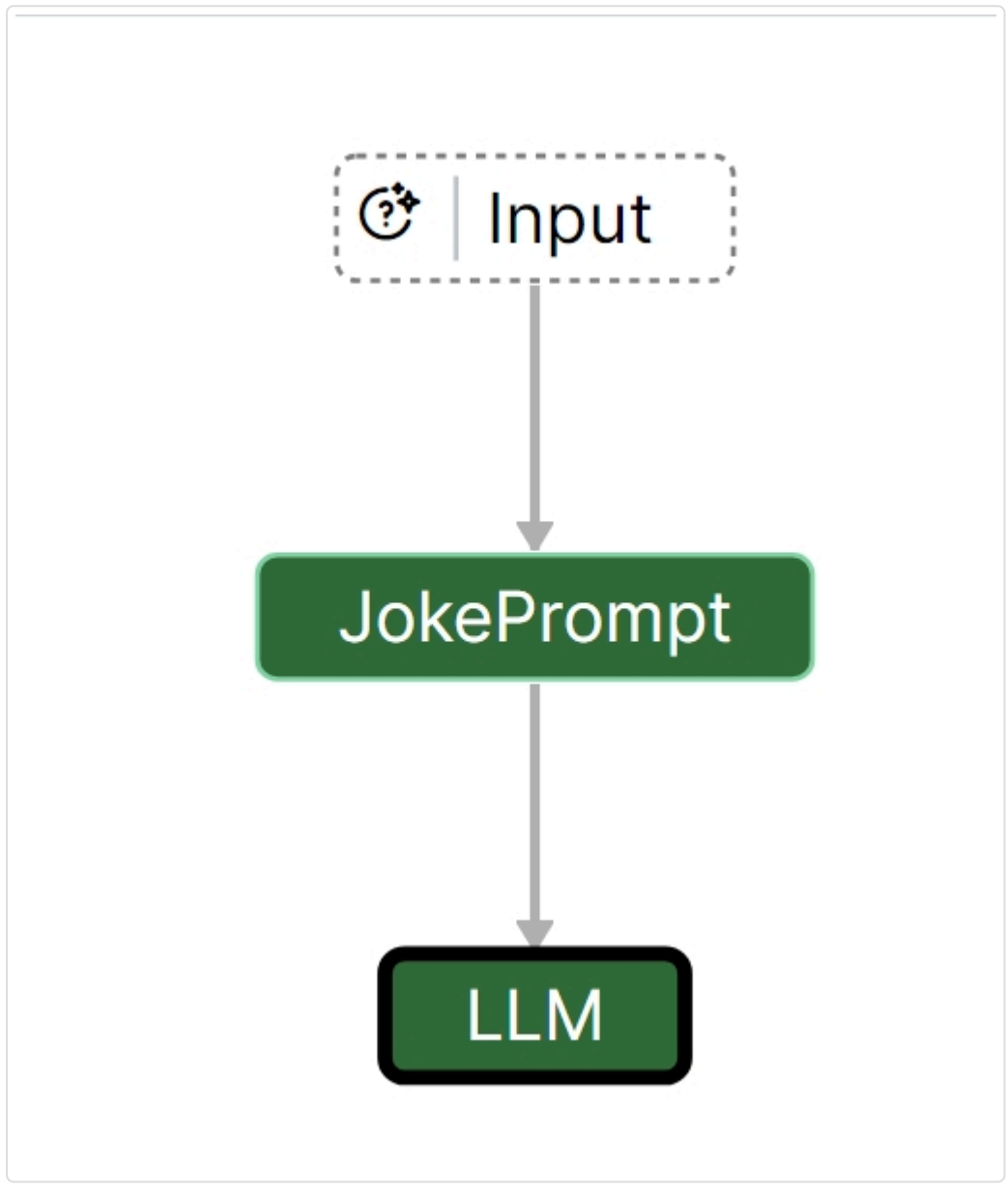
Runtime Configuration – None

User Defined GenAI Components

In addition to the GenAI Components supplied by Vantiq, users may also define their own GenAI Components. User defined GenAI Components allow developers to build reusable GenAI Flows that can be shared with other developers and incorporated into their GenAI Flows. This allows developers to quickly build their own customizable GenAI Components out of the primitive building blocks already at their disposal. These user developed components then become a unit of re-use in the GenAI Builder, appearing in the *palette* alongside the Vantiq supplied components.

Defining GenAI Components

To create a new GenAI Component, you must select the “Add…GenAI Component” menu item. Doing so will launch the GenAI Builder in its component building mode, presenting you with essentially the same UI as we’ve already seen. For the most part, defining a GenAI Flow for a component is no different from defining a GenAI Procedure or an embedded GenAI Flow. As with any GenAI Flow, the component will accept an input, perform the specified processing, and produce an output. All of the behaviors discussed previously apply to GenAI Components. The key difference is that GenAI Components are intended to be used multiple times and therefore may need to adjust their behavior slightly in order to be more generally useful. For example, let’s suppose that we want to take our joke generating flow from earlier and turn it into a component. As a reminder, here is that flow:



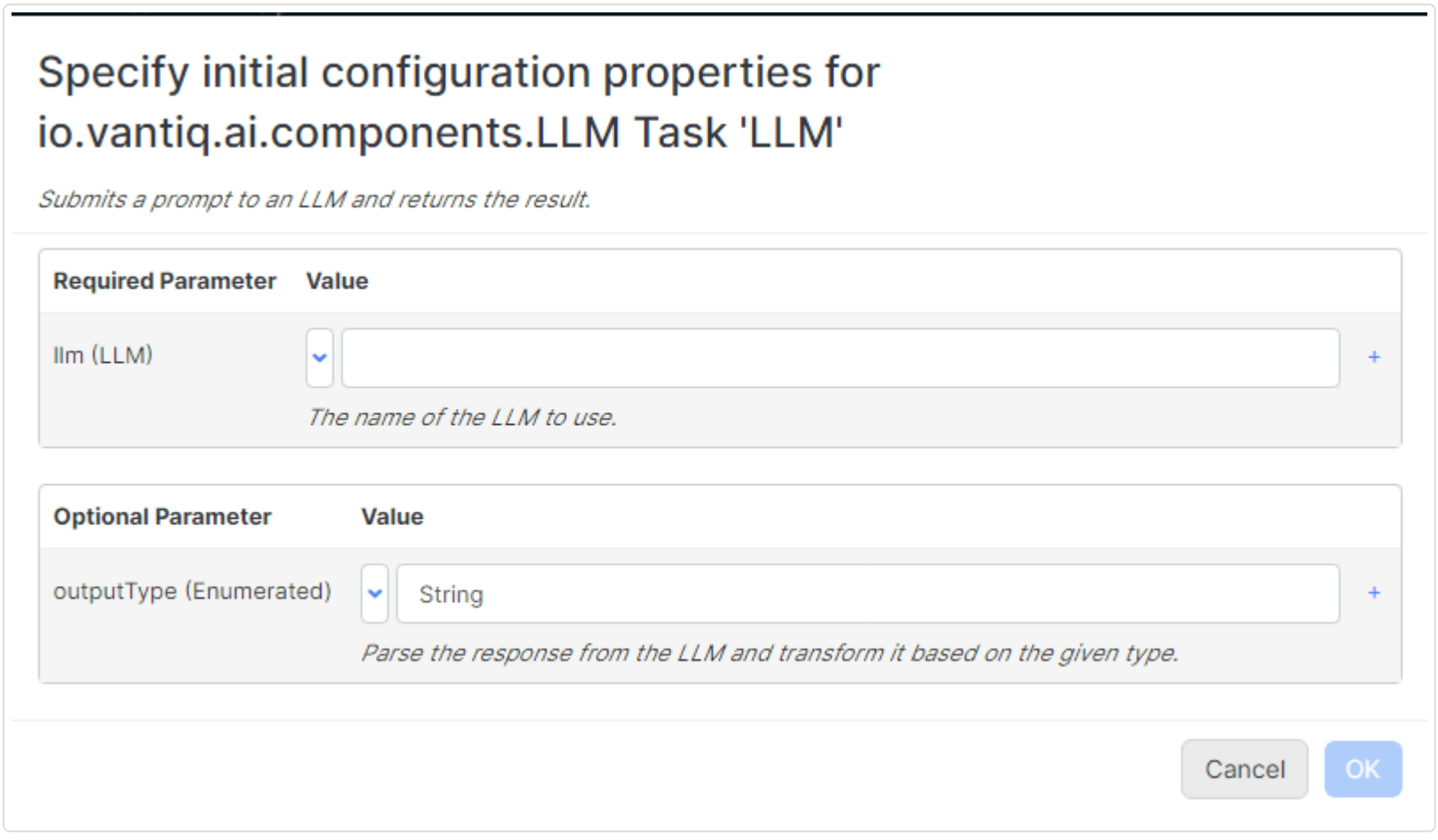
When we defined this flow in the tutorial, we chose a specific LLM instance when configuring the LLM task. However, if we want to make this a component, we probably want the user of that component to determine which LLM to use. This illustrates the key difference between GenAI Components and GenAI Procedures or embedded GenAI Flows. GenAI Components need to be configurable.

Component Configuration Properties

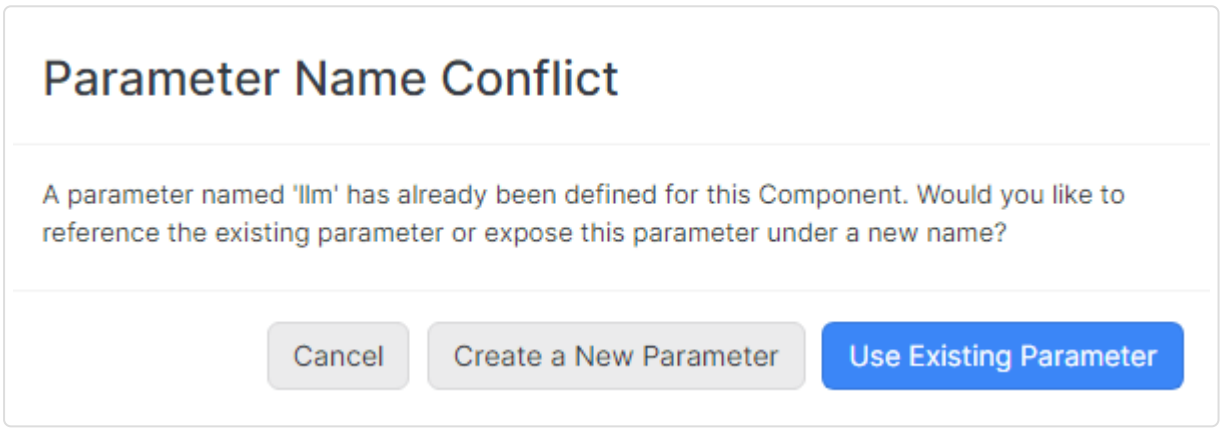
The configuration of a GenAI Component is controlled through the creation and use of Component Configuration Properties. It is up to the developer of the GenAI Component to determine what properties should exist and how they are used by the component they are developing. This is done either by *exposing* one or more task properties or by explicitly creating the property and then making use of it in a task definition.

Exposing Task Properties

Most of the time, a component’s configuration properties are mapped directly to a property of one or more of their tasks. To support this use case, the GenAI Builder allows developers to select a task property and *expose* it as a configuration property. This is done via the task configuration dialog. For example, here is that dialog for our LLM task:

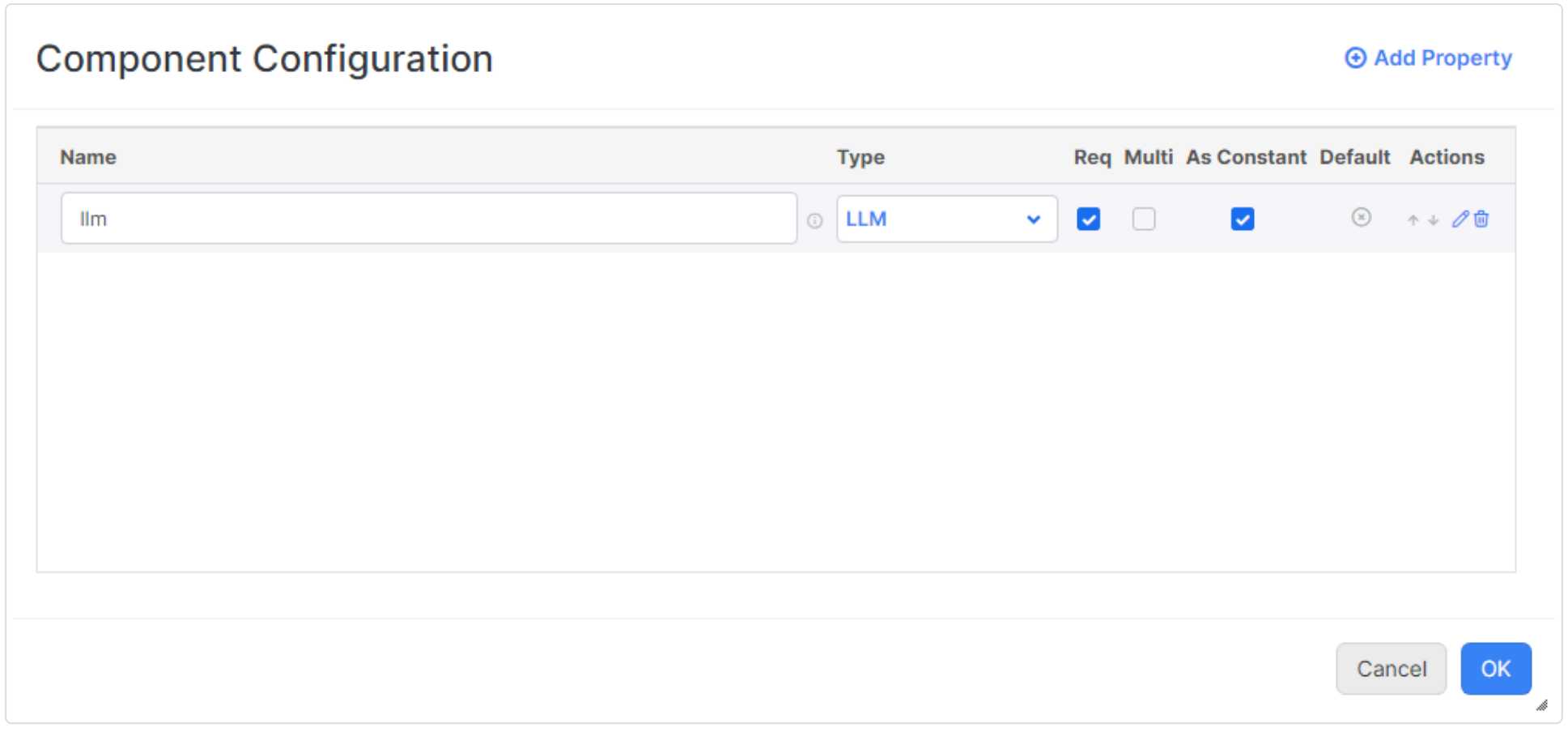


Notice the addition of a *plus* (`+`) on the far right of each task property (this will only appear in component building mode). Clicking on that exposes the task property as a component configuration property. If this is the first time a property of that name has been exposed, it will create the property. If a property of the same name already exists, you will be asked whether you want to use that property or create a new one.



Once the property has been created (or an existing one selected), the task property's value will be specified using a *substitution variable* of the form `${<propertyName>}` . This means that when the component is used, the value provided by the user of the component will be *substituted* wherever that variable appears.

We can see the currently defined component configuration properties by clicking on the *Configuration* button under the GenAI Flow Properties (the button appears when the GenAI Builder is in "component building" mode).



As you can see, each property has a number of attributes which can be defined (we'll discuss these in the next section). However, when exposing a task property you typically do not have to worry about these as they will be inherited from the definition of the component on which the task is based and should not be changed. The one exception is the property's name. This will default to the name of the task property, which may or may not be what you want for the component property name.

Explicitly Creating Properties

While most of the time the component properties can be created by exposing one or more task properties, there are times when it is useful (and even necessary) to create a property explicitly. To do that you go to the *Component Configuration* dialog and click on the *Add Property* button. This will create a new entry in the list of properties which you can fill out as needed.

Component configuration properties have the following attributes:

- **Name** – the name of the property. This will be seen by users of the property when they create tasks from it.
- **Type** – the property's type. Used to validate any value provided when configuring a task based on the component.
- **Req** – whether or not the property is required.
- **Multi** – whether or not the property's value is an Array.
- **As Constant** – should always be checked, will be removed in the next release.
- **Default** – the property's default value.

Once defined, the property can be used as the value of any compatible task property (via its substitution variable). More importantly, it can also be referenced as part of any code that is written when configuring a task. For example, a property can be referenced in the **codeBlock** property of the *CodeBlock* component by using its substitution variable in the code.

Using GenAI Components

User defined components appear in the *Components* section of the GenAI Builder's palette. They are used to create tasks just like any of the Vantiq supplied components.

Importing and Updating GenAI Components

GenAI Components can be added to projects via the "Add…GenAI Component" menu item. Any component that is part of a project, will be exported when that project is exported. Their definitions appear in the `aicomponents` directory. Similarly, when the project is imported, so too will the GenAI Component (unless manually excluded).

When a GenAI Component is updated, either through import or by using the GenAI Builder, it does not immediately have any impact on any other GenAI Flows using the component. Instead, the next time an update is made to a GenAI Flow that references the component, the updated version of the component will be used. This gives the developer a window of time between importing a new GenAI Component definition and updating their GenAI Flows to resolve any incompatibilities that the author may have introduced into the new version. These incompatibilities should be described by the component author in the description, but ultimately that is up to the author.