GenAI Agent Building Guide

Introduction

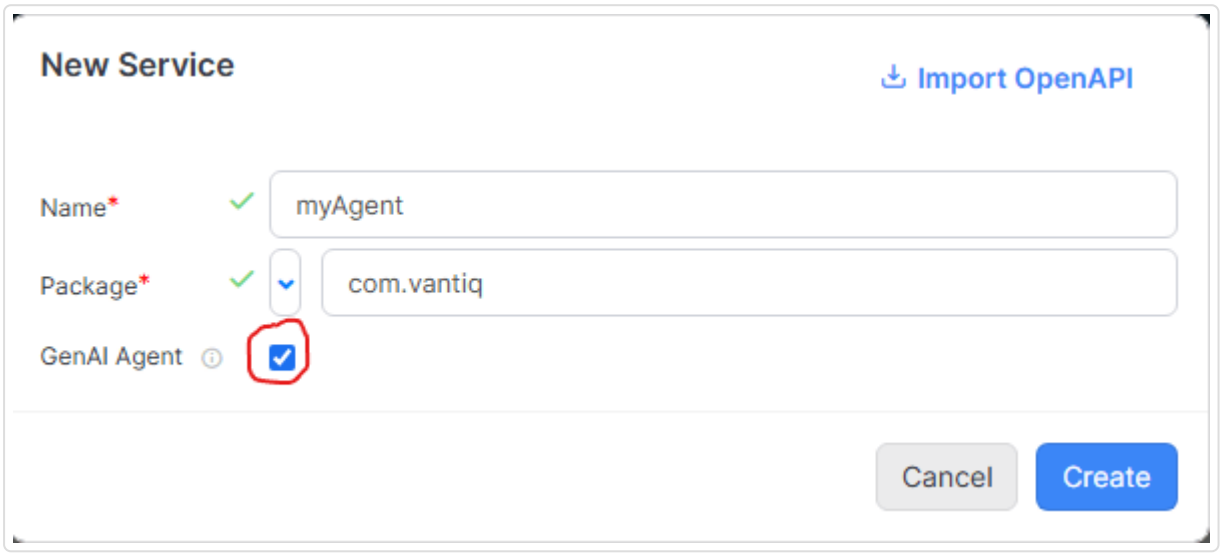
GenAI Agents (hereafter simply Agents) are Services which combine application logic with the use of a large language model (LLM), either directly or via other resources such as Semantic Indexes. When it comes to their basic implementation including issues such as state management, Agents are just standard services. However, the use of LLMs and the nature of the functionality they typically provide, do present some unique issues. That is what we will be focusing on in this document.

Related Documentation

When it comes to leveraging an LLM you may be interested in the SubmitPrompt, AnswerQuestion, and GenAIFlow activity patterns in the Visual Event Handler Guide. For practice adding GenAI functionality to your services see the GenAI Builder Tutorial.

Defining Agents

To create an Agent you start by creating a Service and check the “GenAI Agent” checkbox as shown here:



This triggers generation of a framework which allows Agents to receive and process requests through a standard interface. Currently this is only used during plan execution, but it will see more widespread usage in later releases. The framework also provides a standard way for agents to request user input and obtain context related to the current request. We will see examples of all of these in the sections that follow.

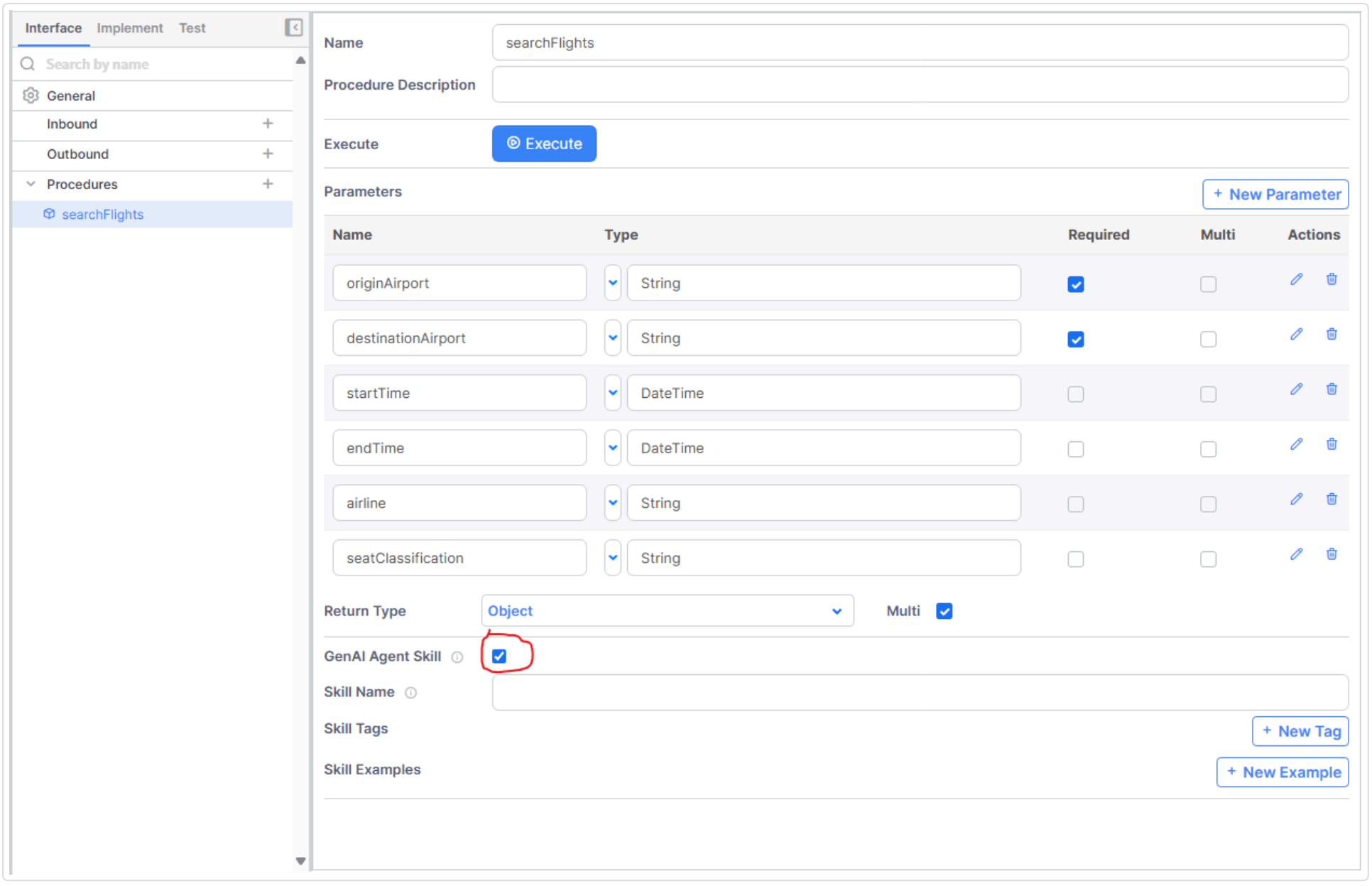
> Explicitly marking services as being “agents” is new in release 1.43. You can convert existing services by checking the “GenAI Agent” checkbox in the “General” section of the “Interface” tab in the service builder.

Agent Prerequisites

Implementation of the Agent’s dispatching framework requires the use of an LLM. The generated implementation uses the LLM *io.vantiq.a2a.agentDispatch* which will be automatically defined if it does not exist in the current namespace. The LLM is configured to use the model `openai/gpt-4.1`. It gets its OpenAI API key from the secret *VANTIQ_A2A_SECRET*. The system will create an empty instance of this secret if it does not already exist and it must be populated with a valid OpenAI API key in order for the agent framework to operate properly.

Agent Skills

As part of its definition, an Agent must declare one or more “skills”. Each skill represents a unique function or capability that the agent provides. Skills are implemented as procedures, so Modelo provides a way to indicate which of an Agent’s public procedures should be treated as skills:



Skills are invoked by the agent framework in response to the requests the Agent receives (for now this will only occur during plan execution). To ensure that skills are used properly, it is very important to provide as complete a description as possible, both of the procedure and its parameters.

Discovering and Describing Agents

Explicitly declaring agents means that it is possible to query the system's metadata and determine what agents are available. The following query selects the names of all agents:

```
var agentNames = SELECT name FROM system.services WHERE isAgent == true
```

Agents also support the notion of "tags" which can be used to form groups of related agents which can be determined dynamically. For example, this query will find all agents tagged as "travel":

```
var qual = {
    isAgent: true,
    "ars_properties.tags": "travel"
}
var travelAgents = SELECT name FROM system.services WHERE qual
```

Once the appropriate agents are selected, you may need to determine exactly what the agent can do. To that end, all agents automatically provide an Agent Card which provides a standard description of their functionality. An agent's "card" can be obtained either via the REST endpoint `/api/v1/resources/services/<agentName>/_agentCard` or by invoking the built-in procedure *io.vantiq.ai.A2A.getAgentCard*.

For example, suppose we want to use a planning algorithm to determine which agents to use when handling a user's request. To do this we need to be able to provide the planning prompt with a description of our target agent(s). If we assume we have the following template stored in a document named "agentSummary.txt":

```
Agent: ${name}
Description: ${description}
Skills:
@repeat(skills)---------
Name: ${name}
Id: ${id}
Description: ${description}
--------@endrepeat
```

Then we can emit a description of an agent like this (assumes that the variable `agentName` holds the name of the target agent:

```
var template = io.vantiq.text.Template.documentReference("agentSummary.txt")
Template.format(template, io.vantiq.ai.A2AA2A.getAgentCard(agentName))
```

Conversations

LLM interactions are stateless. This means that for every request, the LLM only has access to its own “knowledge” and the information in the current request. If you want the LLM to know about previous requests and its responses to them, this information must be presented as part of the current request. Doing this is the role of a *conversation*.

> The word “conversation” gets used in a variety of contexts when talking about GenAI applications. This isn’t surprising given that LLMs appear to “converse” with users and it is a convenient word to describe many application behaviors. However, in the Vantiq platform the term has a very specific meaning and anytime you see it in relation to Vantiq Agents and GenAI Applications, this is what we mean.

A conversation consists of an ordered sequence of *messages* which capture the history of an interaction between an Agent and an LLM. The messages in a conversation have a *type* and associated *content*. A message’s type tells the LLM how to interpret the content and must be one of the following:

- **system** – Instructions to the LLM about how to interpret the conversation or “behave” in general. Typically there is only a single system message which is added automatically by the application. Vantiq supports including a system message as part of an LLM definition to ensure that it is always present.
- **human** – Content provided by the user/client.
- **ai** – Content generated by the LLM in response to a request. Maybe a direct response or instructions for some further action (e.g. the invocation of an LLM “tool”).
- **tool** – Content produced by the execution of an LLM tool.

The structure of a message’s content depends on its type. See ChatMessage for more details.

Transient Conversations

Conversations are managed using the Conversation Memory service which supports creation and manipulation of conversations. The conversations managed by this service can be referenced when sending an LLM request via the built-in submitPrompt and answerQuestion procedures or when invoking a GenAI procedure. Doing so causes the conversation to be automatically updated based on the underlying LLM interactions. On the client side, there is the Conversation Widget to facilitate the user’s participation in a conversation (use the `conversationId` property to refer to a specific conversation)

As its name implies, the resulting conversation state is stored in memory. This means that it has a limited lifetime and can be subject to loss in certain failure cases. We refer to these as “transient” conversations. Transient conversations are suitable for interactions that will last minutes to maybe an hour or so and which do not need to be saved for any reason (such as auditing).

Persistent Conversations

For cases where a conversation must be available over much longer time frames (days or even weeks) or when it needs to be recorded for some reason, conversations can be persisted as part of a collaboration managed by the Agent. This can be accomplished in a variety of ways. Using the SubmitPrompt, AnswerQuestion, or GenAIFlow activity patterns in a visual event handler will automatically bind a conversation to the current collaboration instance (or create one as needed). The Agent may also choose the manage its collaborations more explicitly. If the conversation is associated with some application “entity”, then the entity role procedures are a natural fit. Alternatively, the Agent can directly manage one or more persistent conversations using the collaboration management procedures.

In either case, once bound to a collaboration instance, persistent conversations are automatically saved along with their associated collaboration instance and loaded into memory when the collaboration instance is retrieved. Once loaded, they can be accessed through the Conversation Memory service as described above. This is all done using standard partitioned state and fully supports service replication for stricter reliability guarantees.

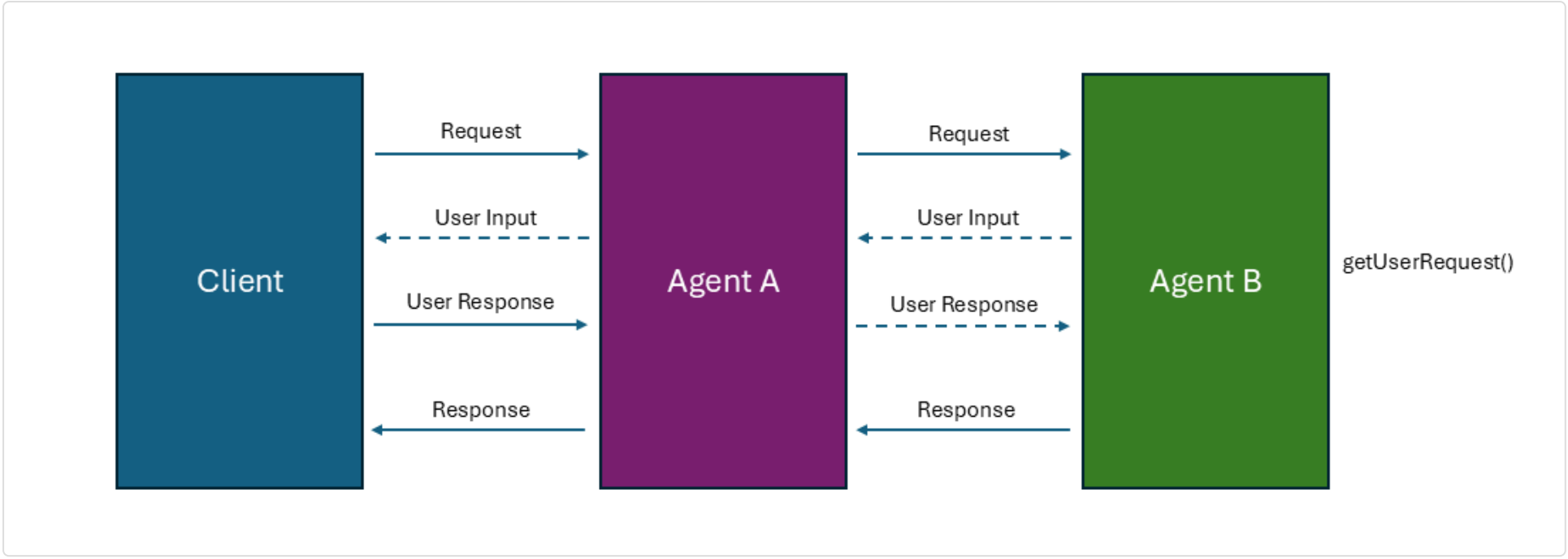
> When using persistent conversations, clients should use `Client.setCollaborationContext` to bind the client to the active collaboration instance. This will trigger binding any conversation widgets to the appropriate conversation (including support for use of named conversations).

Agent to Human Interaction

The use of an LLM can make Agents very powerful, allowing them to make decisions much more dynamically instead of relying solely on predetermined pathways. To help mitigate against the Agent doing something not just unexpected, but inappropriate, it can be necessary to keep the “human in the loop”, requiring that it obtain permission prior to acting. This requires that the Agent be able to initiate an interaction with the user. This mechanism can also be used for other purposes, such as allowing the Agent to gather additional information it might need to complete its task. Requests for user input can occur in two distinct contexts depending on how the agent was invoked.

Agent Framework

The agent framework provides a way to invoke all agents using a common entry point. This is what allows agents to be invoked during plan execution without having to worry about the details of how the requests will be processed. This execution framework also includes a standard way of requesting input from the user using the built-in procedure io.vantiq.ai.Agent.requestUserInput. This procedure can be called at any point and will instruct the framework to contact the caller to obtain the requested information. This works through multiple invocations to ensure that the request ends up with the original requester. For example, suppose a client sends a request to an agent and that agent sends a request to a second agent (one that is unknown to the client). If this second agent wants to obtain user input, we want that request to be forwarded back to the client. Using `io.vantiq.ai.Agent.requestUserInput` triggers exactly this behavior as shown here:



Currently, the agent framework is only used when dispatching requests during plan execution. This means that requests through the Agent's service interface (from any client) are not supported. Future releases will be expanding use of the framework's dispatch mechanism, so to help in the transition agents can implement a "bridge" procedure.

User Request Bridge Procedure

The user request bridge procedure (hereafter simply "the bridge") should be implemented by any agent which will be receiving requests through its service interface and wants to support use of the built-in procedure `io.vantiq.ai.Agent.requestUserInput`. The bridge has the following definition:

```
PRIVATE STATELESS PROCEDURE <AgentService>.a2a_requestUserInput(userRequest Any, requestContext Any): Any
```

The parameters are:

- **userRequest** – the value provided to the invocation of `io.vantiq.ai.Agent.requestUserInput`. It describes what is being requested from the user.
- **requestContext** – the value provided to the invocation of `io.vantiq.ai.Agent.requestUserInput`. This is an opaque value that the agent should use to establish the proper communication context.

The return value of the procedure will be returned to the caller of `io.vantiq.ai.Agent.requestUserInput`. The procedure's implementation should leverage the techniques described in the next section.

Agent Service Interface

When processing requests received through the Agent's service interface (either via a procedure invocation or processing an inbound event), the Agent must implement its own user request handling. Vantiq supports two ways to accomplish this: direct communication and using notifications.

Direct Communication

The direct communication model requires that the agent and the user be actively engaged in a conversation via the Conversation Widget. Whenever the conversation widget loads a conversation (either directly or via an enclosing collaboration instance), it will register a "callback" with the Callback service. The id of the current conversation is used as the *callbackId*. This allows the Agent to contact the user using the `io.vantiq.Callback.invoke` procedure like this:

```
var userPrompt = "I'm about to withdraw money from your account, is that OK?"
var userResponse = io.vantiq.Callback.invoke(conversationId, userPrompt, 5 minutes)
... do something with the response ...
```

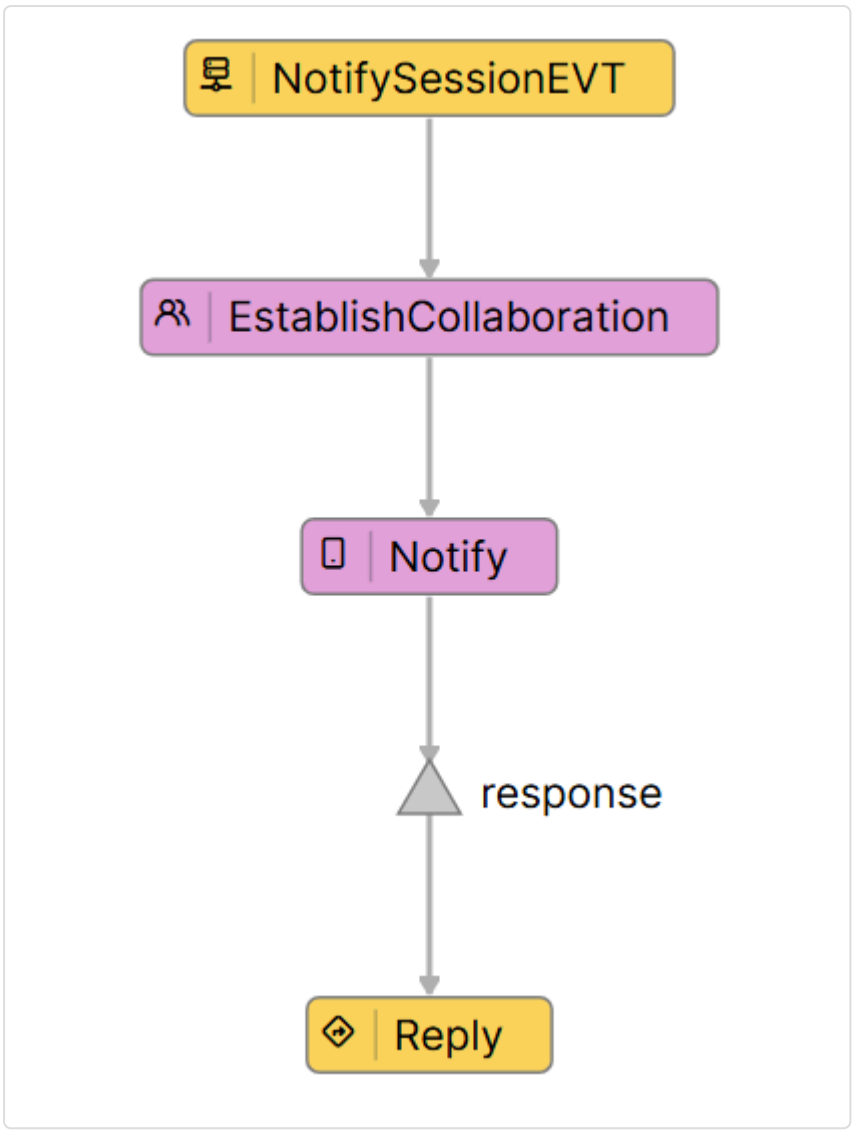
The data sent will be displayed to the user in the conversation widget and then the user's response will be returned as the result of the invocation (unless the user takes longer than *5 minutes* to respond). The advantage of this approach is that the Agent's communication will appear to the user where they are likely already engaged. This avoids the need to pop up additional UI elements or distract them from the current task. The disadvantage is that it won't work if the user isn't actively using a client with a conversation widget.

Notification

If the user cannot be reached directly, then the Agent must instead use a notification based approach to contact them. The Notify activity pattern already provides robust support for sending a notification to one or more users and then managing their responses. The primary limitation is that it works in the context of a visual event handler, which typically operate asynchronously and do not provide a means to produce a result. Rather than invent an alternate notification mechanism, we instead chose to address this limitation.

To do this we added the ability to "invoke" a service event handler. The VAIL `PUBLISH` statement can obviously be used to trigger a handler, but it assumes a fully asynchronous execution model, so it cannot "wait" for a reply. Therefore, we have added the `Event.request` procedure to the built-in event processing service. This procedure triggers the handler for a specified service event. It must be called from a service procedure belonging to the same service as the target event type. The behavior of the handler is unrestricted, but it is assumed that at some point it will provide a response using the Reply activity pattern. The net result is a request/response execution model which uses an event handler as its implementation.

For example, suppose we have the following event handler:



We can “invoke” it from the Agent using code like this:

```
var event = {collaborationId: collaborationId}
var userResponse = Event.request("NotifySessionEVT", event, 5 minutes)
... do something with the response ...
```

When run, the target user will receive the notification and be presented with the associated client. Once the user provided the requested input, the result from the client would be sent back to the Agent as the return from `Event.request` and processing would continue from that point. The advantage of this approach is that it allows the agent to contact users on an “interrupt” basis, not just when they are actively in a conversation. The disadvantage is that the Vantiq Notify pattern is limited to use on a mobile device. So this approach isn’t appropriate for browser only applications.

Planning Agents

One popular agent architecture is known as “plan and execute”. The idea is that you start by asking the LLM to formulate a step-wise plan to address a given request. The LLM is typically given a collection of “tools” which it can use when building its plan (note that these are distinct from LLM Tools). Once the plan is created, it is executed, step by step. This may or may not involve calling the LLM. Once the plan has been executed, the results are analyzed and a result is provided. Some algorithms introduce the notion of “re-planning” which may cause the agent to repeat the execution phase on a new set of tasks (until a final state is reached). Some common algorithms are:

- Plan and Execute – this is the most basic form which constructs a linear sequence of tasks and uses re-planning to refine the results. It performs re-planning after each step, so it can be quite expensive in terms of the number of LLM interactions. This approach does not enumerate the available tools to the planner, relying instead on native LLM capabilities augmented with standard LLM tool calling (which is implicit and therefore unknown to the planner).
- Reasoning Without Observations (ReWOO) – attempts to improve on the basic plan-execute approach in order to reduce the number of LLM calls required. It explicitly enumerates the available tools so the planner can decide when to leverage them and it uses “variables” to pass information from one step to another. The individual tools may or may not use an LLM to do their work and can often use a smaller, cheaper LLM than the one used to construct the plan. The resulting plan can be represented as a directed, acyclic graph. Plan execution occurs once, with no re-planning.
- LLMCompiler – similar to ReWOO in that information can flow from one task to another. Tasks are “streamed” and executed as they are produced (assuming all dependencies are available). Incorporates re-planning after the execution of all current tasks to continue task generation until final result is reached.

Each of these uses different prompts to build the plan and process task results, but what they all have in common is the need to execute a sequence of tasks and report their results. We refer to this as “plan execution” and we have provided the built-in procedure io.vantiq.ai.Agent.executePlan to manage this process.

> Since the “Plan and Execute” algorithm executes its tasks one at a time, it actually doesn’t need to use the plan execution facility. We’ve included it in this section for completeness (since it is a well-known architecture for planning agents).

The primary input to this procedure is a DAG where each node represents a task to perform and the edges represent the flow of information from one task to another. The result is an `Object` with one property for each node which references that node’s result.

Trip Planner Contribution

The *Trip Planner* project found under **Project…Import…Contributions** provides a skeletal example of a multi-agent system. It demonstrates the use of the ReWOO planning algorithm and illustrates how agents can obtain user input and interact with a client.