LLM Reference Guide

Overview

This guide provides instructions for configuring Large Language Models (LLMs), for example configuring access to LLMs hosted in external systems such as AWS Bedrock, or enabling LLMs to use Tools.

AWS

Vantiq integrates with AWS SageMaker and Bedrock, allowing access to their hosted models. A list of available models can be found in the IDE LLM creation pane, in the Model Name drop-down list.

When you choose a SageMaker or Bedrock model from the Model Name list, the user interface automatically displays the necessary fields for entering your AWS credentials:

- Region
- Access Key Id
- Secret Access Key
- Role Arn *(optional)*

The AWS credentials that you specify are used to access the model inference endpoint. For example, if you have enabled access to a Bedrock model hosted in the `us-east-1` region, you must specify the same region in the Region field.

Note that all fields, except Region, are treated as confidential. These credentials are managed as secret references and must be defined before you can use them in the LLM configuration.

SageMaker

To access a SageMaker model, you need to provide:

- **Model Name** – either `sagemaker` for a generative model, or `sagemaker-embedding` for an embedding model.
- **AWS Credentials** – enter these as detailed in the previous section.
- **Endpoint Name** – the name of the SageMaker inference endpoint.
- **Message Format** – the message format name or message format configuration defining the message syntax used to access the model.

For advanced settings, the Configuration field allows you to input a JSON document with specific parameters. For example, a `Llama3-Chat` model configuration may look like this:

```
{
   "endpoint_kwargs": {
      "CustomAttributes": "accept_eula=true"
   },
   "model_kwargs": {
      "max_new_tokens": 350,
      "temperature": 0.1,
      "return_full_text": false
   }
}
```

Please refer to the model documentation to understand the available configuration options and proper format.

SageMaker Message Format

The Message Format specifies the expected syntax for all messages sent to the inference endpoint. This format encompasses two primary aspects:

- *Marshalling*: it sets the rules for converting a request text message (e.g., prompt) into a format that the inference endpoint can process.
- *Unmarshalling*: it sets the rules for converting the response received from the inference endpoint.

For models available through SageMaker JumpStart, message format documentation can often be found in their associated Notebook as python examples.

Many models, generative or embedding, use the following generic message syntax:

```
    request:    {"<request_key>": <input_value>, "<request_parameters_key>": model_kwargs}
    response:   response_json<response_path>
```

For example,

```
     request:   {"inputs": <input_value>, "parameters": model_kwargs}
     response:  response_json[0]["generated_text"]
```

or,

```
    request:   {"inputs": <input_value>, **model_kwargs}
    response:  response_json["vectors"]
```

You can configure the message format of any model following this generic syntax by specifying the configuration property `message_format_config`, and providing values for:

- `request_key`
- `request_nesting_level`
- `request_parameters_key`
- `response_path`

Both `request_nesting_level` and `request_parameters_key` are optional.

`request_nesting_level` is the number of nesting levels for `<input_value>`.

For example,

```
    # request_nesting_level = 0  (default value)
    request: {"inputs": <input_value>, "parameters": model_kwargs}

    # request_nesting_level = 1
    request: {"inputs": [<input_value>], "parameters": model_kwargs}

    # request_nesting_level = 2
    request: {"inputs": [[<input_value>]], "parameters": model_kwargs}
```

Message Format Examples

Below are message format configuration examples from some of the models available through SageMaker JumpStart.

```
# Falcon
#
# request:  {"inputs": <input_value>, "parameters": model_kwargs}
# response: response_json[0]["generated_text"]

{
    "message_format_config": {
        "request_key": "inputs",
        "request_parameters_key": "parameters",
        "response_path": "0,generated_text"
    }
}
```

```
# GPTJ
#
# request:  {"text_inputs": <input_value>, **model_kwargs}
# response: response_json[0][0]["generated_text"]

{
    "message_format_config": {
        "request_key": "text_inputs",
        "response_path": "0,0,generated_text"
    }
}
```

```
# Bloom
#
# request:  {"text_inputs": <input_value>, **model_kwargs}
# response: response_json["generated_texts"][0]

{
    "message_format_config": {
        "request_key": "text_inputs",
        "response_path": "generated_texts,0"
    }
}
```

```
# Embedding
#
# request:  {"text_inputs": <input_value>, **model_kwargs}
# response: response_json["embedding"]

{
    "message_format_config": {
        "request_key": "text_inputs",
        "response_path": "embedding"
    }
}
```

Message Format Names

To simplify the configuration of common message formats, Vantiq provides the following pre-defined names that can be specified in the `Message Format Name` field:

- `llama2-chat`
- `llama3-chat`
- `mistral-instruct`
- `llama2`
- `llama3`
- `falcon`
- `mistral-instruct`
- `mistral`
- `bloom`
- `gptj`

All names except `llama2-chat`, `llama3-chat` and `mistral-instruct` are simple built-in aliases to generic syntax message format configurations, as described above. `llama2-chat`, `llama3-chat` and `mistral-instruct` require dedicated code specific to the LLM model, so they do not rely on the generic syntax.

If you do not find a configuration that matches your model, please contact Vantiq Support.

Bedrock

To access a Bedrock model, you need to provide:

- **Model Name** – select a name from the Model Name Bedrock drop-down list. A name corresponds to the Bedrock `modelId` prefixed with `bedrock/`, for example, `bedrock/meta.llama3-8b-instruct-v1:0`. If your desired model is not listed, you can manually enter it using the syntax `bedrock/<model-name>`, where `<model-name>` represents the Bedrock `modelId`.
- **AWS Credentials** – enter these as detailed in the previous AWS section.

For advanced settings, the Configuration field allows you to input a JSON document with specific parameters. For example, providing advanced settings for the Titan Text model would look like this:

```
{
   "model_kwargs": {
      "maxTokenCount": 1024,
      "topP": 1,
      "temperature": 0.1
   }
}
```

Please refer to each model documentation to understand the available configuration options and their proper format.

Custom Bedrock Configuration

Alternatively, instead of manually entering the Bedrock model using the syntax `bedrock/<model-name>`, you can specify a custom configuration.

The custom configuration must be a JSON document with the following properties:

- **class_name** – `vantiq.llms.ChatAWSBedrockConverse` for a generative model, or `vantiq.embeddings.BedrockEmbeddings` for an embedding model.
- **model_id** – the Bedrock *modelId* value, for example `meta.llama3-8b-instruct-v1:0`.
- **aws_region** – the AWS region where the model is hosted.
- **aws_access_key_id** - the AWS access key id.
- **aws_secret_access_key** - the AWS secret access key.
- **aws_role_arn** - the AWS role arn *(optional)*.
- **model_kwargs** - a JSON document with the model-specific configuration parameters *(optional)*.

For example,

```
{
   "class_name": "vantiq.llms.ChatAWSBedrockConverse",
   "model_id": "meta.llama3-8b-instruct-v1:0",
   "aws_region": "us-east-1",
   "aws_access_key_id": "@secrets(AWSAccessKeyId)",
   "aws_secret_access_key": "@secrets(AWSSecretAccessKey)",
   "max_tokens": 512,
   "temperature": 0.2
}
```

The UI field *API Key Secret* should be left empty and the *Model Name* field should contain your LLM name, for example `MyBedrockLLM`. Make sure to rely on the secrets notation to protect the AWS credentials, as shown in the above example.

Note that manually specifying a model name and its configuration does not guarantee functionality. It is likely to work if the model you are trying to configure is a variant of a supported model family. For example, if the Model Name Bedrock drop-down list includes `bedrock/meta.llama3-2-11b-instruct-v1:0` and you manually specify `bedrock/meta.llama3-2-90b-instruct-v1:0`, a custom configuration is likely to function correctly.

If the model you are trying to configure is not a variant of a supported model family, please contact Vantiq Support.

Google

To specify a Google model, prefix the model name with `google-genai/`, for example `google-genai/gemini-2.0-flash`.

Safety Settings

Gemini models have built-in safety filters to prevent generating harmful or unsafe content.

To change the default safety filter settings, specify the property `safety_settings` in the LLM configuration, providing a JSON document with a set of key-value pairs where the key is the filter category and the value is the block level value.

For example,

```
{
    "safety_settings": {
      "HARM_CATEGORY_DANGEROUS_CONTENT": "BLOCK_LOW_AND_ABOVE",
      "HARM_CATEGORY_HATE_SPEECH": "BLOCK_MEDIUM_AND_ABOVE"
   }
}
```

Gemini models support the following categories:

```
    HARM_CATEGORY_HARASSMENT
    HARM_CATEGORY_HATE_SPEECH
    HARM_CATEGORY_SEXUALLY_EXPLICIT
    HARM_CATEGORY_DANGEROUS_CONTENT
```

The block levels available are:

```
    BLOCK_LOW_AND_ABOVE
    BLOCK_MEDIUM_AND_ABOVE
    BLOCK_ONLY_HIGH
    BLOCK_NONE
```

Azure

Vantiq integrates with Azure OpenAI, allowing access to OpenAI models deployed in Azure OpenAI Studio.

OpenAI

To define an Azure OpenAI model using the IDE, you need to provide:

- **Model Name** – `azure-openai` for a generative model, or `azure-openai-embedding` for an embedding model.
- **Azure Resource Name** – the name of the Azure OpenAI resource.
- **Deployment Name** – the name of the Azure OpenAI model deployment.
- **API Version** – the API version of the model. You may refer to the Azure documentation.
- **Credentials** – the model access key (e.g., from *Keys and Endpoint* in the OpenAI Azure resource Portal).

For a JSON definition format, you need to specify the model endpoint name (`azure_endpoint`), which is constructed from the Azure resource name: `https://<resource-name>.openai.azure.com/`

For example,

```
{
  "name": "AzureOpenAI",
  "modelName": "azure-openai",
  "type": "generative",
  "config": {
    "azure_deployment": "gpt-4o-dep",
    "openai_api_version": "2024-10-21",
    "azure_endpoint": "https://vq-open-ai.openai.azure.com/",
    "temperature": 0.1,
    "apiKeySecret": "AzureOpenAISecret"
  }
}
```

Where `vq-open-ai` is the Azure resource name, `gpt-4o-dep` is the deployment name, and `2024-10-21` is the API version. A secret named `AzureOpenAISecret` holds the model access key.

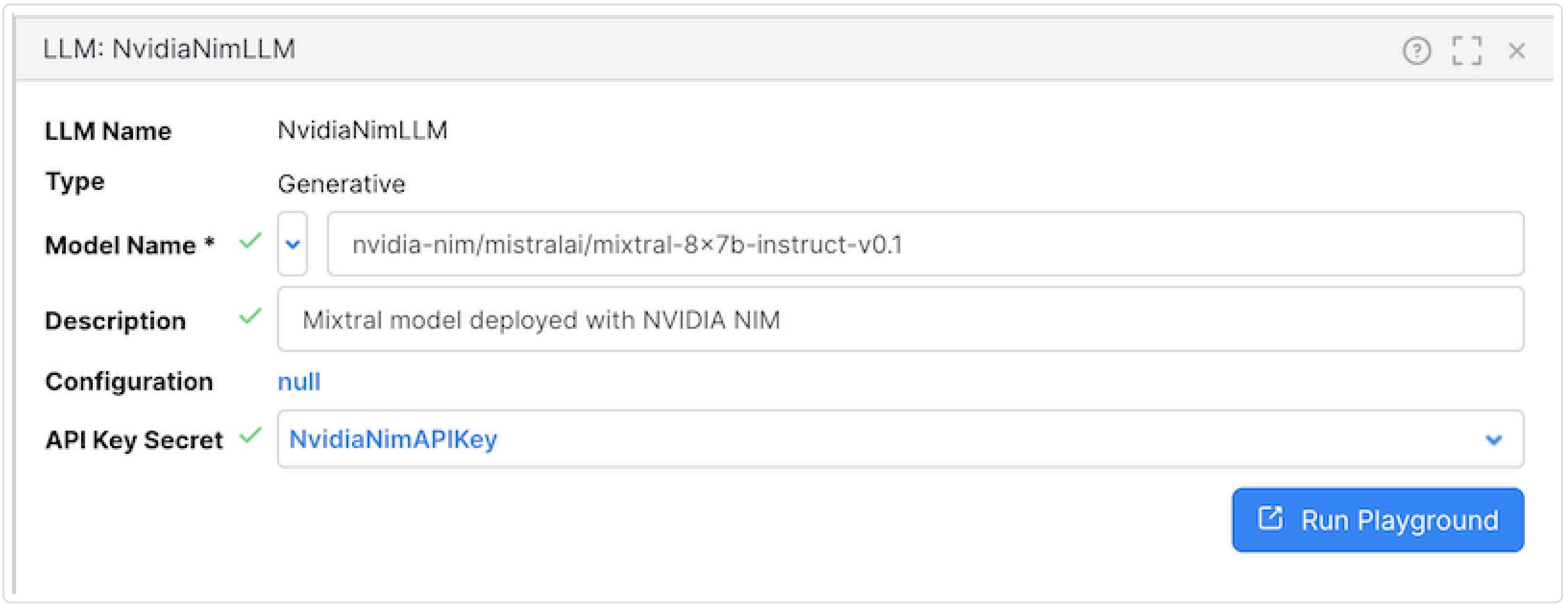
NVIDIA NIM

Vantiq integrates with NVIDIA NIM, allowing access to models deployed with NVIDIA NIM.

To define an NVIDIA NIM generative or embedding model using the IDE, you need to provide:

- **Model Name** – `nvidia-nim/<model>` where `<model>` is the name of the model.
- **Credentials** – the NVIDIA NIM API Key.

For example,


LLM: NvidiaNimLLM

LLM Name	NvidiaNimLLM
Type	Generative
Model Name *	nvidia-nim/mistralai/mixtral-8x7b-instruct-v0.1
Description	Mixtral model deployed with NVIDIA NIM
Configuration	null
API Key Secret	NvidiaNimAPIKey

Where the model name is `mistralai/mixtral-8x7b-instruct-v0.1` and `NvidiaNimAPIKey` is the secret name for the NVIDIA NIM API Key.

The JSON definition for the above example is:

```
{
  "name": "NvidiaNimLLM",
  "modelName": "nvidia-nim/mistralai/mixtral-8x7b-instruct-v0.1",
  "type": "generative",
  "description": "Mixtral model deployed with NVIDIA NIM",
  "config": {
    "apiKeySecret": "NvidiaNimAPIKey"
  }
}
```

OpenAI

To specify an OpenAI model, prefix the model name with `openai/`, such as `openai/gpt-4o` or `openai/o3-mini`.

If you want to interface with a model compatible with OpenAI, you can specify the model using the syntax `openai/<model-name>` or create a custom model definition with the property `class_name` set to the OpenAI supporting class `vantiq.llms.ChatOpenAI`.

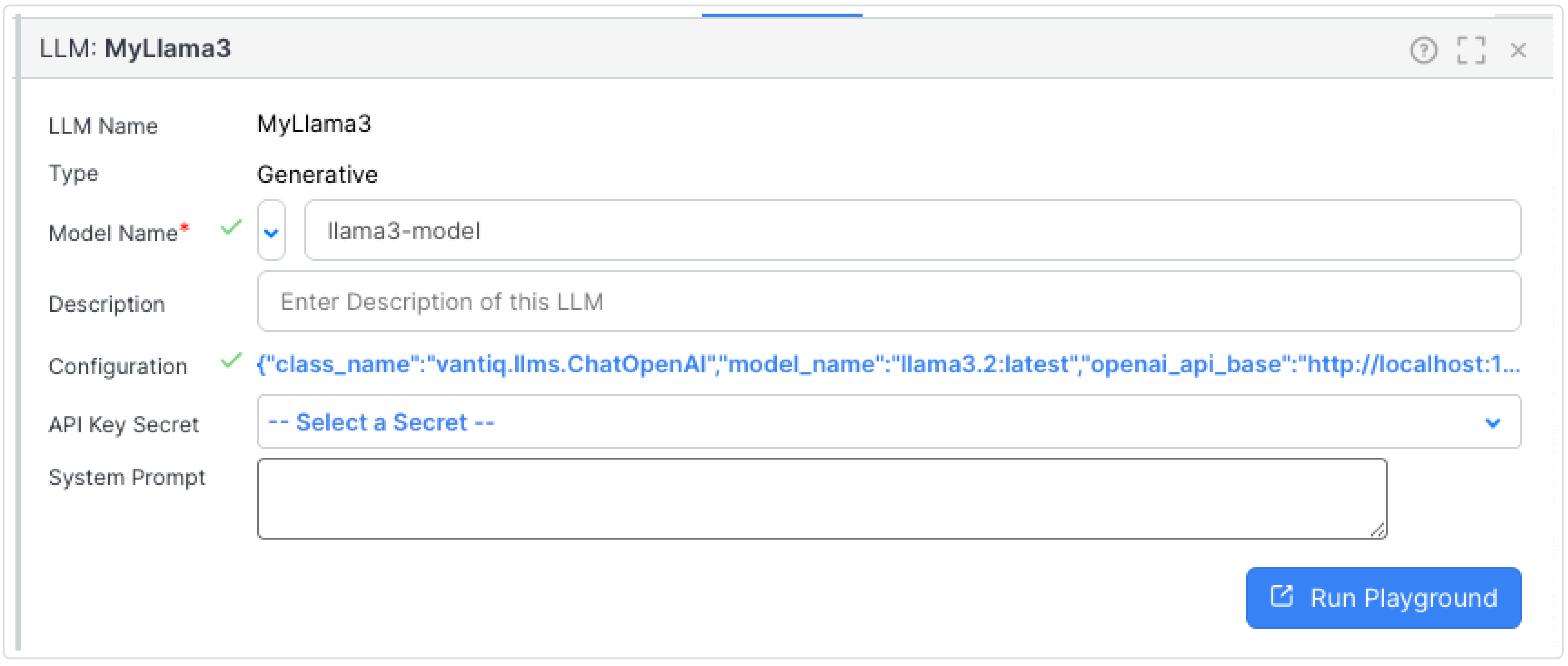
For example, let's assume that we are running a local `llama3` model using *Ollama*, which is compatible with the OpenAI Chat Completions API. Based on the *Ollama* documentation, the custom configuration would look like,

```
{
   "class_name": "vantiq.llms.ChatOpenAI",
   "model_name": "llama3.2:latest",
   "openai_api_base": "http://localhost:11434/v1",
   "openai_api_key": "unused"
}
```

Where `model_name` is the local model to use, `openai_api_base` (or `base_url`) is the *Ollama* endpoint and the property `openai_api_key` (or `api_key`) is set to any value (it is ignored).

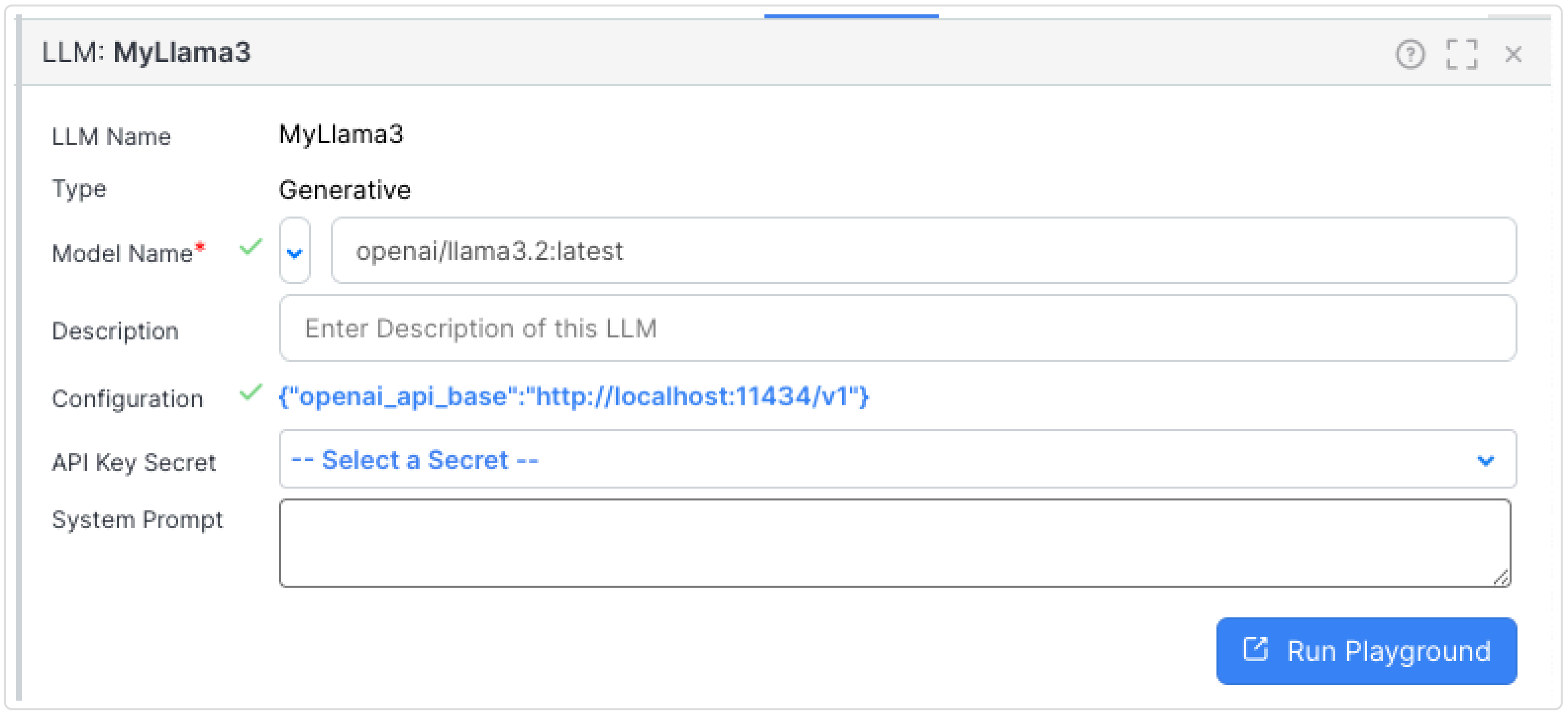
Because we create a custom LLM definition, the Model Name specified must be a descriptive name (e.g., llama3-model) and not a name from the suggested models list.

For example,


LLM: **MyLlama3**

LLM Name	MyLlama3
Type	Generative
Model Name*	llama3-model
Description	Enter Description of this LLM
Configuration	{"class_name":"vantiq.llms.ChatOpenAI","model_name":"llama3.2:latest","openai_api_base":"http://localhost:1...
API Key Secret	-- Select a Secret --
System Prompt	

The above example can also be specified using a model name prefixed with `openai/`. In this case, the Model Name value would be `openai/llama3.2:latest` and since the `openai/` prefix is specified, the custom configuration would omit the `class_name` property.


LLM: **MyLlama3**

LLM Name	MyLlama3
Type	Generative
Model Name*	openai/llama3.2:latest
Description	Enter Description of this LLM
Configuration	{"openai_api_base":"http://localhost:11434/v1"}
API Key Secret	-- Select a Secret --
System Prompt	

The examples above are scoped as *Ollama* examples. Depending on the use case, any other ChatOpenAI configuration property might be used.

Usage

Configuration

LLMs can be customized using configuration parameters such as `temperature`, `top_p`, or `max_tokens`. Although these parameters generally have the same function across different LLMs, not all models expose the same parameters in the same manner. A parameter might be unavailable or its syntax could differ between models.

While Vantiq tries to normalize the configuration parameter names across the LLMs it supports,
the LLM documentation should always be consulted as the definitive source for configuration syntax and availability.

For example, OpenAI expects a configuration property `max_tokens` while Bedrock Llama expects `model_kwargs.max_tokens` and Bedrock Titan `model_kwargs.maxTokenCount`.

Both LLM configuration property and runtimeConfig property (e.g., submitPrompt) expect the same configuration syntax.

For example, let's assume that we have the following LLM configuration (e.g., Titan model)

```
{
    "model_kwargs": {
        "maxTokenCount": 1024
    }
}
```

If we want to override this default LLM configuration with a lower value for the time of a `submitPrompt` call, we would provide a `runtimeConfig` value that would have the same syntax:

```
{
    "model_kwargs": {
        "maxTokenCount": 256
    }
}
```

Note that Vantiq tries to normalize the configuration parameter names across the LLMs it supports. For example, if two LLMs support the same maxToken parameter, both LLMs will expose it as `max_tokens`, allowing a similar configuration between LLMs.

The normalized configuration parameter names are: `temperature`, `max_tokens`, `top_p`, `top_k`, `stop`, `presence_penalty`, `frequency_penalty`. Not all models support all of these parameters. If a normalized parameter is specified but not supported by an LLM, it gets ignored.

A normalized configuration parameter can be pictured as an alias to the actual parameter name used by the LLM. For example, `max_tokens` is an alias to `model_kwargs.maxTokenCount` for the Bedrock Titan model.

Usage of normalized configuration parameters is optional. As mentioned previously, an LLM documentation is the definitive source for configuration syntax and availability.

Tools

In the context of a Large Language Model, a tool is an external function or system that the model can use to perform tasks beyond its inherent abilities. Tool calling, also known as function calling, refers to the LLM's capability to invoke a tool to carry out a specific task. Tool calling is only available on LLM models that support it.

An LLM can be configured with one or more tools by specifying, for each tool, a schema that defines how the tool should be invoked. This schema includes the tool's name, parameters, return type, and description. It's essential for the schema to have clear parameter names and detailed descriptions to help the LLM understand the tool's purpose and usage. When the LLM decides to use one of the tools it has been configured with, it returns a document matching the provided tool schema. This document, referred to as a *tool call* document, describes the tool invocation. It includes the selected tool's name, along with the parameter names and their corresponding values needed for invocation. However, the LLM does not invoke the tool itself; invocation is handled by the Vantiq server based on the tool usage configuration.

To configure an LLM with tools, you can use VAIL procedures where each procedure acts as a tool. The Vantiq server automatically generates a schema for each procedure and configures the LLM using the generated schemas. If the LLM chooses to use a tool, it returns a message containing a *tool call* description that details how to invoke the procedure. The Vantiq server then executes the procedure on behalf of the LLM.

For example, let's use the following `getCurrentTemperature` VAIL procedure as an LLM tool,

```
// Return the temperature of the specified location in Celsius
package tool.example
procedure ToolService.getCurrentTemperature(location String Required Description "The location to get the temperature for"): Intege
    // VAIL code to get the temperature from the specified location
    var temp = <logic to retrieve the temperature>
    return temp
```

Notice that the procedure `getCurrentTemperature` contains a description for the tool itself and for the `location` parameter.

With the `submitPrompt` LLM built-in service, tools are provided as the `tools` parameter,

```
var ref = "/procedures/tool.example.ToolService.getTemperature"
io.vantiq.ai.LLM.submitPrompt(llmName: "ChatLLM", prompt: "What is the temperature in New York?", tools: [ref])
```

If a service contains multiple procedures, and each public procedure is a tool, you can specify a service reference. This will include all public procedures as tools,

```
var ref = "/services/tool.example.ToolService"
io.vantiq.ai.LLM.submitPrompt(llmName: "ChatLLM", prompt: "What is the temperature in New York?", tools: [ref])
```

When the LLM is invoked with the provided prompt, it selects the `getCurrentTemperature` tool and returns a response containing a *tool call* describing how to invoke the tool. The Vantiq server then invokes the procedure `getCurrentTemperature` on behalf of the LLM and adds the returned value to the LLM's prompt context. The LLM is then automatically re-invoked with this enriched context, enabling it to generate a response that includes the temperature.

If several tools are specified, the LLM prompt context is enriched with the return values from all the tools it decides to use.

Tool calling is supported by the `submitPrompt` built-in service, `SubmitPrompt` activity pattern and `Tool` GenAI component. Each of these supports defining tools with services and procedures and also with user-provided schemas.

Execution of a Vail procedure tool can be controlled using the options explained in the Tool Authorizer section.

MCP Tools

The `mcpTools` property of the GenAI Tools component allows you to specify MCP servers, providing access to the MCP tools defined in those servers.

The value of the `mcpTools` property is a JSON object in which each key is an MCP server alias, and each corresponding value is an MCP server configuration object. Each MCP server configuration must include the `url` and `transport` properties. If the server requires headers (e.g., for authentication), they can be provided using the optional `headers` property.

For example, the following configuration specifies two MCP servers, `mcpServer1` and `mcpServer2`, with their respective URLs, transports, and authentication headers:

```
{
  "mcpTools": {
    "mcpServer1": {
      "url": "https://mcp-server1.example.com/mcp",
      "transport": "streamable_http",
      "headers": {
        "Authorization": "Bearer @secrets(MCPKey_Server1)"
      }
    },
    "mcpServer2": {
      "url": "https://mcp-server2.example.com/mcp",
      "transport": "streamable_http",
      "headers": {
        "Authorization": "Bearer @secrets(MCPKey_Server2)"
      }
    }
  }
}
```

Note: The `transport` property supports the values `streamable_http` and `sse`.

We recommend using the `@secrets` notation to protect sensitive information such as API keys or authentication tokens.

Defining MCP servers in the `mcpTools` property makes their tools available to the LLM. If the `tools` property also defines tools, the LLM will have access to both sets of tools.

You can use the `toolAuthorizer` property to control the execution of MCP tools, as described in the Tool Authorizer section below.

Tool Custom Schema

Using VAIL procedures is the most convenient way to configure and work with tools. VAIL procedures offer a simple framework for tool execution and a control mechanism with authorizers. However, VAIL procedures are not the only way to define tools; you can also define tools by specifying custom schemas.

With a tool defined by a custom schema, the LLM returns a *tool call* description that the Vantiq server does not attempt to invoke. Instead, the Vantiq server returns the LLM message containing the *tool call* description to your application, giving you full control over how the *tool call* information is processed. For example, you could implement a custom invocation tailored to a tool, then provide the tool's response back to the LLM.

Another scenario would be to simply leverage a tool's schema mapping capability. Based on an input document (prompt) the LLM can extract and map the necessary information conforming to the tool's schema, creating a document instance of this schema based on the prompt data.

For example, imagine a music record classifier tool that defines a schema with fields like title, year, artist, etc. The LLM prompt would include data from a music record, and the LLM would return a message with a *tool call* description containing the music record in the schema format. In this scenario, no tool invocation is necessary, and no further LLM re-invocation is required. We are simply taking advantage of the LLM capability to generate a document that conforms to a schema.

Let's take a first example with a tool defined using a GenAI component, relying on a Pydantic schema definition. This example shows the schema mapping capability. Another example that illustrate a custom tool invocation is outlined in the Tool Custom Invocation section below.

We keep the above `getCurrentTemperature` procedure example, but this time defined as a Pydantic model class definition in the Tool GenAI component,

```
class getTemperature(BaseModel):
  """Return the temperature of the specified location in Celsius"""
  location: str = Field(..., description="The location to get the temperature for")
```

When we invoke the Tool component with the prompt `What is the temperature in Paris?`, the response is an `ai` message containing the following `tool_calls` property,

```
[
   {
      "name": "getTemperature",
      "args": {
         "location": "Paris"
      },
      "id": "toolu_bdrk_012yxuj8nqjSnT9SEsTL9ios",
      "type": "tool_call"
   }
]
```

This *tool call* description is a JSON document that conforms to the `getTemperature` schema. From this description, we could decide to invoke an external tool that would provide the temperature, then feed the response back into the LLM (see example in the section below), or the response might be sufficient for our processing logic, without the need to call the LLM again for this prompt.

Note that the `tool_calls` property is a list that may contain multiple tools. Each *tool call* description in this list includes the necessary invocation information. The `id` property uniquely identifies a *tool call*, and if a tool result message needs to be built, the `id` value must be provided as the `tool_call_id` parameter. See section Tool Custom Invocation for an example.

Tool Custom Invocation

Another use case for defining a tool with a custom schema is to explicitly manage its invocation, providing complete control over the process. Based on the *tool call* description returned by the LLM, you can access the necessary invocation information, invoke the tool, and then provide the tool's response back to the LLM. This allows the LLM to generate an appropriate response using the context provided by the tool.

To do this, you can use the built-in service procedure `ChatMessage.buildToolResultMessage`. This procedure wraps the tool invocation response as a tool message, which can be added back to the LLM prompt context. Note that when adding a tool result message to the prompt context, you must also add the initial LLM result with the *tool call* descriptions. This ensures that the LLM has the necessary context.

The following VAIL code illustrates how to define a tool with a custom schema and how to explicitly manage its invocation. We continue using `getCurrentTemperature` as a tool, but now we use the `submitPrompt` built-in service procedure along with a function descriptor schema,

```
package tool.example

import service io.vantiq.ai.LLM
import service io.vantiq.ai.ChatMessage

// LLM resource must have tool calling capability, e.g., OpenAI, Gemini
// Prompt should ask for the temperature of a location (e.g., "What is the temperature in Paris?")
PROCEDURE ToolSamples.customInvoke(llmName String, prompt String)

// getCurrentWeather schema (io.vantiq.ai.FunctionDescriptor)
var schema = {
    name: "getCurrentWeather",
    description: "Return the temperature of the specified location in Celsius",
    parameters: {
        type: "object",
        properties: {
            location: {
                type: "string",
                description: "The location to get the temperature for"
            }
        },
        required: ["location"]
    }
}

// initial prompt followed by LLM invocation
var prompts = [ChatMessage.buildHumanMessage(prompt)]

// LLM is configured with the getCurrentWeather tool
var result = LLM.submitPrompt(llmName: llmName, prompt: prompts, tools: [schema])

if (typeOf(result) == "Object") {
    // add message with tool description to prompt context
    prompts.add(result)

    // tool call describes the tool (name, args) - we assume that it gets invoked here and returns a value
    var toolCall = result.tool_calls[0]
    // var tempFromTool = <tool invocation>
    var tempFromTool = 18   // assume the tool returns 18 as the temperature

    // wrap tool returned value as a tool result message and add to prompt context
    var toolResult = ChatMessage.buildToolResultMessage(toolCall.name, tempFromTool, toolCall.id)
    prompts.add(toolResult)

    // with complete context, invoke LLM again - we should now get the expected response
    result = LLM.submitPrompt(llmName: llmName, prompt: prompts, tools: [schema])
} else {
    exception("org.test.error", "expected a message response with tool_calls", [])
}

// e.g., ret = "The temperature in Paris is 18 degrees Celsius."
return result
```

Once the tool is invoked using the information from the `tool_calls` property description (`<tool invocation>` placeholder above), a tool result message is created and added to the prompt context. The tool response message contains the tool name, tool returned value and the tool `id` value from the *tool call* description.

This example uses a single tool for simplicity. If multiple schemas are provided and the LLM chooses to use more than one tool (i.e., the *tool call* description list contains multiple tools), you must handle each tool invocation separately and provide a tool result message for each response. The LLM prompt context should then include the initial LLM result with the *tool call* descriptions, followed by a tool result message for each invoked tool.

Tool Parameter Override

When an LLM selects a tool, it returns a *tool call* document that specifies the tool name and input parameters. The Vantiq server uses this information to perform the tool invocation. In some cases, you may want to override certain parameters - for example, to provide a value for an optional parameter the LLM did not set, or to replace a value it selected. To do this, you can use the GenAI Flow runtime configuration property `toolParameterOverride`.

Its general syntax is:

```
{
  "toolParameterOverride": {
    tool: <tool_name>
    name: <parameter_name>
    value: <searchable_value>
    override: <override_value>
  }
}
```

Properties of the `toolParameterOverride` object are:

- **tool** – The name of the tool that the parameter belongs to.
- **name** – The name of the parameter to override.
- **value** – A substring to match against the parameter value selected by the LLM. If a match is found, the override is applied.
- **override** – The value to use instead of the value returned by the LLM.

To illustrate how `toolParameterOverride` works, consider the following VAIL procedure `getMedicalRecord`, defined as a tool:

```
// Get the medical record for the specified patient, providing their name and optionally their social security number
PROCEDURE getMedicalRecord(name String Required Description "Name of the person",
                           ssn  String Description "Person social security number")
```

Override parameter by name

Suppose the LLM selects this tool and returns a *tool call* with the name parameter set to "Alice" and the ssn parameter set to "123-45-6789". To override the ssn value regardless of what the LLM selected, specify the following `toolParameterOverride` in your runtime configuration:

```
{
  "toolParameterOverride": {
    "tool": "getMedicalRecord",
    "name": "ssn",
    "override": "987-65-4321"
  }
}
```

In this case, the Vantiq server will invoke the `getMedicalRecord` procedure with the name parameter set to "Alice" and ssn set to "987-65-4321", replacing the value selected by the LLM.

> For a Vantiq procedure tool, the `tool` property must specify the fully qualified procedure name. If the `getMedicalRecord` procedure were part of the `MedicalService` service in the `health.records` package, the tool property value would be `health.records.MedicalService.getMedicalRecord`.

Override parameter by value

Now suppose your prompt is: `Get the medical record for Alice. For the SSN value, use the string: SSN_VALUE.`

The LLM returns a tool call with name set to "Alice" and ssn set to "SSN_VALUE". To override this value based on a match, use the following configuration:

```
{
  "toolParameterOverride": {
    "tool": "getMedicalRecord",
    "value": "SSN_VALUE",
    "override": "987-65-4321"
  }
}
```

The Vantiq server searches all *tool call* parameters for values containing the substring "SSN_VALUE". For each match, the override is applied, and the entire LLM-selected value is replaced with "987-65-4321".

> Note: The match is based on substring search. This means that even if the LLM-selected value was "SSN_VALUE_Alice", the override would still apply, replacing "SSN_VALUE_Alice" with "987-65-4321".

Override parameter by name and value

You must specify at least the name or value property - but you can provide both. If both are set, the override applies only when the parameter name matches and the value contains the specified substring.

Using *by name* makes the override deterministic: it always applies to that parameter, no matter what value the LLM picks. Using *by value* is more flexible: it lets you match based on the selected content, which is useful if you do not know the parameter name.

Several parameters override

If you have several parameters to override, you can specify a list as the value of the `toolParameterOverride` property.

For example,

```
{
  "toolParameterOverride": [
    {
      "tool": "getMedicalRecord",
      "name": "ssn",
      "override": "987-65-4321"
    },
    {
      "tool": "getPersonalInfo",
      "name": "redacted",
      "override": "true"
    }
  ]
}
```

> Note that `toolParameterOverride` is a runtime configuration, so it applies only to the call where it is specified.

Tool parameter overrides can be used with both VAIL procedure and service tools and MCP tools.

Tool Authorizer

For LLMs capable of tool calling, tools can be specified as VAIL resources (either procedures or services) in the `submitPrompt` invocation, `SubmitPrompt` activity pattern or `Tool` GenAI component. Tools can also be specified as MCP server resources using the GenAI `Tool` component `mcpTools` property.

When the LLM selects one of these specified tools, it can automatically execute the corresponding VAIL procedure or MCP server tool, thus providing the LLM with the additional context it requires.

To control tool execution, you can include a tool authorizer. For example as a parameter to the `submitPrompt` LLM built-in service, as the `functionAuthorizer` configuration property of the `SubmitPrompt` activity pattern or as the `authorizer` configuration property of the `Tool` GenAI component.

The authorizer is a VAIL procedure that is evaluated before executing any tool. It must return `true` to authorize the tool's execution or `false` to deny it. If denied, tool execution fails. For example, in the case of a `submitPrompt` invocation, it would fail with the error `io.vantiq.llm.function.exec.denied`.

A tool authorizer requires *name* and *arguments* parameters. The *name* parameter specifies the tool to be executed, and the *arguments* parameter holds the values of the tool's parameters.

For example,

```
procedure confirmExecution(name String, arguments Object): Boolean
```

To control if a tool can be automatically executed or not, you can specify the `aiExecute` property in its procedure definition. The `aiExecute` property can have the following values:

- **allow** – invocation is allowed - the procedure is automatically executed, unless an authorizer invocation denies it.
- **authorized** – invocation must be authorized - the procedure cannot be executed unless an authorizer invocation authorizes it.

For example, the following procedure `submitOrder` can only be executed as an LLM tool if a tool authorizer allows it:

```
// Submit an order named 'item' for a total price specified by the integer property 'price'
procedure submitOrder(item String Required Description "The name of the item to order",
                      price Integer Required Description "The total price of the order"): String
                      with properties = {aiExecute = authorized}
```

Let's assume that we have an `AIService` with an authorizer as defined above. Let's also assume that we have a reference to the `submitOrder` procedure. A call to `submitPrompt` would look like this:

```
var ref = "/procedures/submitOrder"
var refAuthorizer = "/procedures/org.test.AIService.confirmExecution"
io.vantiq.ai.LLM.submitPrompt(llmName: "OpenAI", prompt: "Order one large pizza for $20.",
                              functions: [ref], functionAuthorizer: refAuthorizer)
```

As the LLM chooses to invoke `submitOrder`, the authorizer `confirmExecution` is invoked with the name parameter set to `submitOrder` and the arguments parameter set to `{"item": "large pizza", "price": 20}`. The authorizer decides to either grant execution (return `true`) or deny it (return `false`).

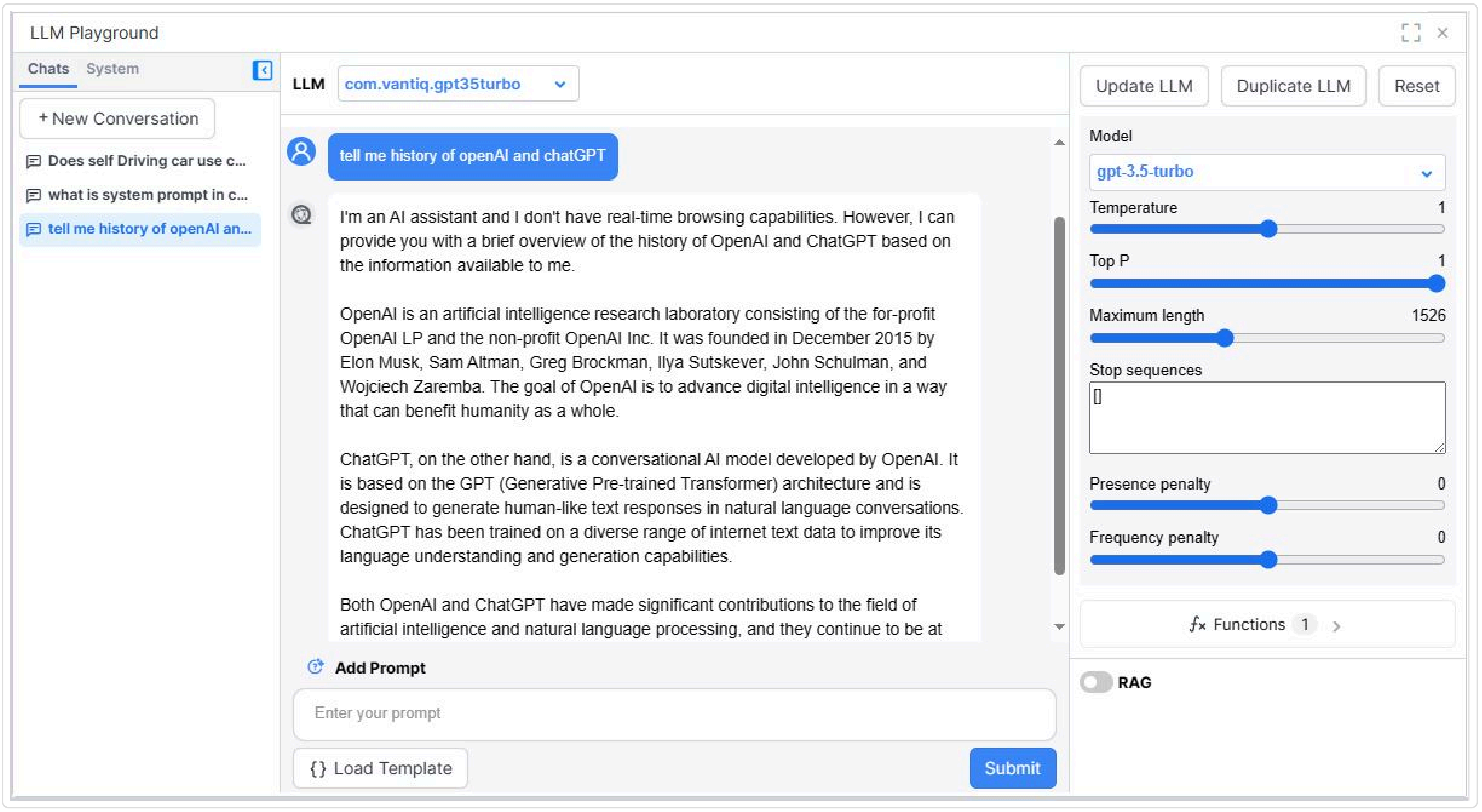
MCP tools have an implied execution mode of `allow`. Therefore, if the GenAI Tools component's `mode` property is set to `Execute`, the MCP tools are executed automatically unless explicitly denied by an authorizer.

LLM Playground

The LLM playground offers a tool for verifying LLM connections and experimenting with different large language models and their settings. It is also a useful tool for creating and refining prompts (a task often referred to as *Prompt Engineering*).

There are two ways to access the LLM playground in the IDE. Use the *Test - LLM Playground* menu from the top navigation bar menu. Or use the "Run Playground" button found in an LLM's detail pane.

The LLM playground has three main UI components: navigation panel, main chatting area and settings panel.



tell me history of openAI and chatGPT

I'm an AI assistant and I don't have real-time browsing capabilities. However, I can provide you with a brief overview of the history of OpenAI and ChatGPT based on the information available to me.

OpenAI is an artificial intelligence research laboratory consisting of the for-profit OpenAI LP and the non-profit OpenAI Inc. It was founded in December 2015 by Elon Musk, Sam Altman, Greg Brockman, Ilya Sutskever, John Schulman, and Wojciech Zaremba. The goal of OpenAI is to advance digital intelligence in a way that can benefit humanity as a whole.

ChatGPT, on the other hand, is a conversational AI model developed by OpenAI. It is based on the GPT (Generative Pre-trained Transformer) architecture and is designed to generate human-like text responses in natural language conversations. ChatGPT has been trained on a diverse range of internet text data to improve its language understanding and generation capabilities.

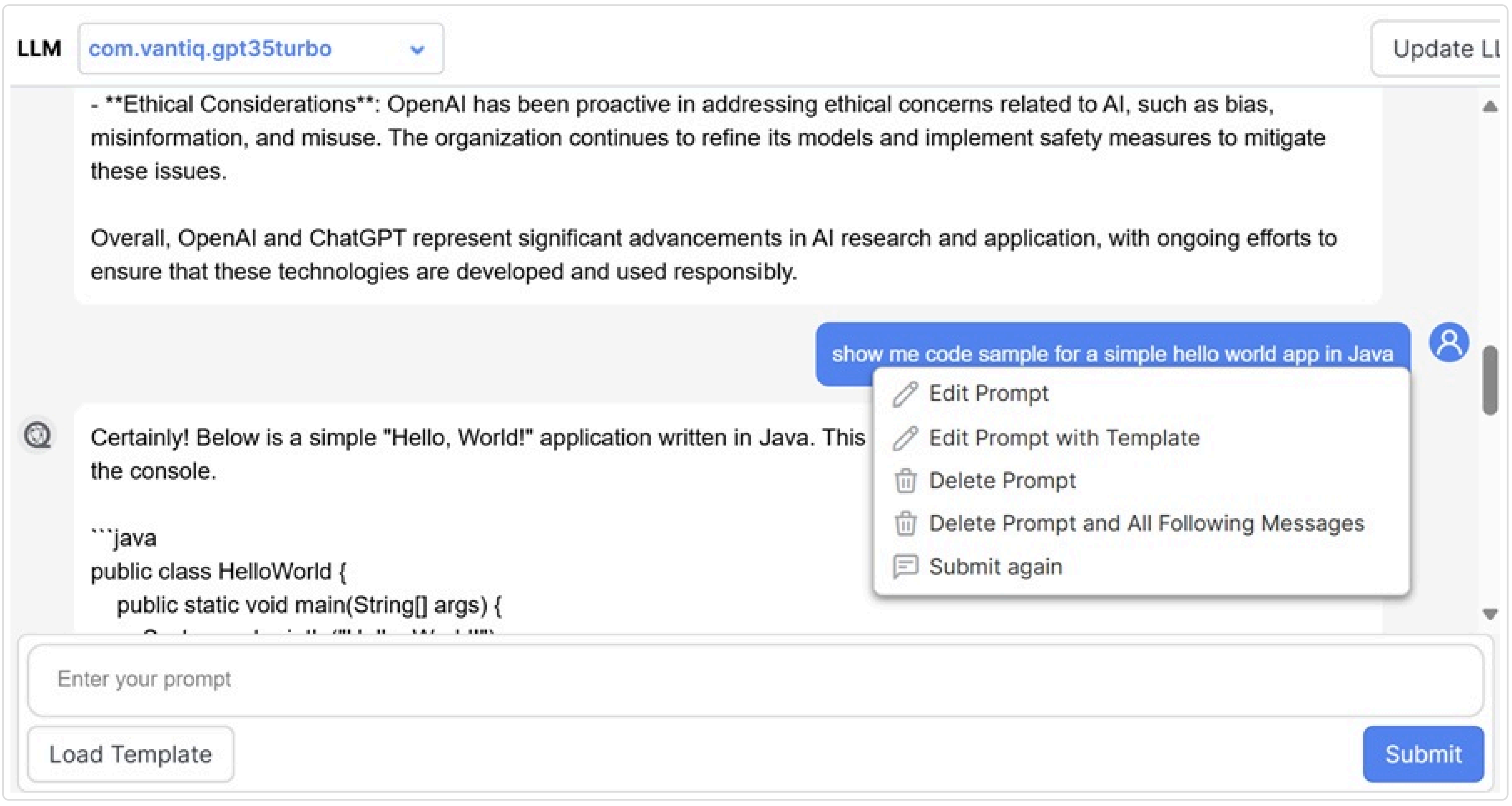
Both OpenAI and ChatGPT have made significant contributions to the field of artificial intelligence and natural language processing, and they continue to be at

Model: gpt-3.5-turbo
Temperature: 1
Top P: 1
Maximum length: 1526
Stop sequences: []
Presence penalty: 0
Frequency penalty: 0

Main Chatting Area

The main chatting area is where you can choose the LLM to work on and type in a prompt to see the response from the selected LLM.

Once a prompt is submitted, it is added to the end of the current conversation, along with its response. There is a context menu associated with each user prompt.



- **Ethical Considerations**: OpenAI has been proactive in addressing ethical concerns related to AI, such as bias, misinformation, and misuse. The organization continues to refine its models and implement safety measures to mitigate these issues.

Overall, OpenAI and ChatGPT represent significant advancements in AI research and application, with ongoing efforts to ensure that these technologies are developed and used responsibly.

show me code sample for a simple hello world app in Java

Certainly! Below is a simple "Hello, World!" application written in Java. This
the console.

```java
public class HelloWorld {
    public static void main(String[] args) {
```

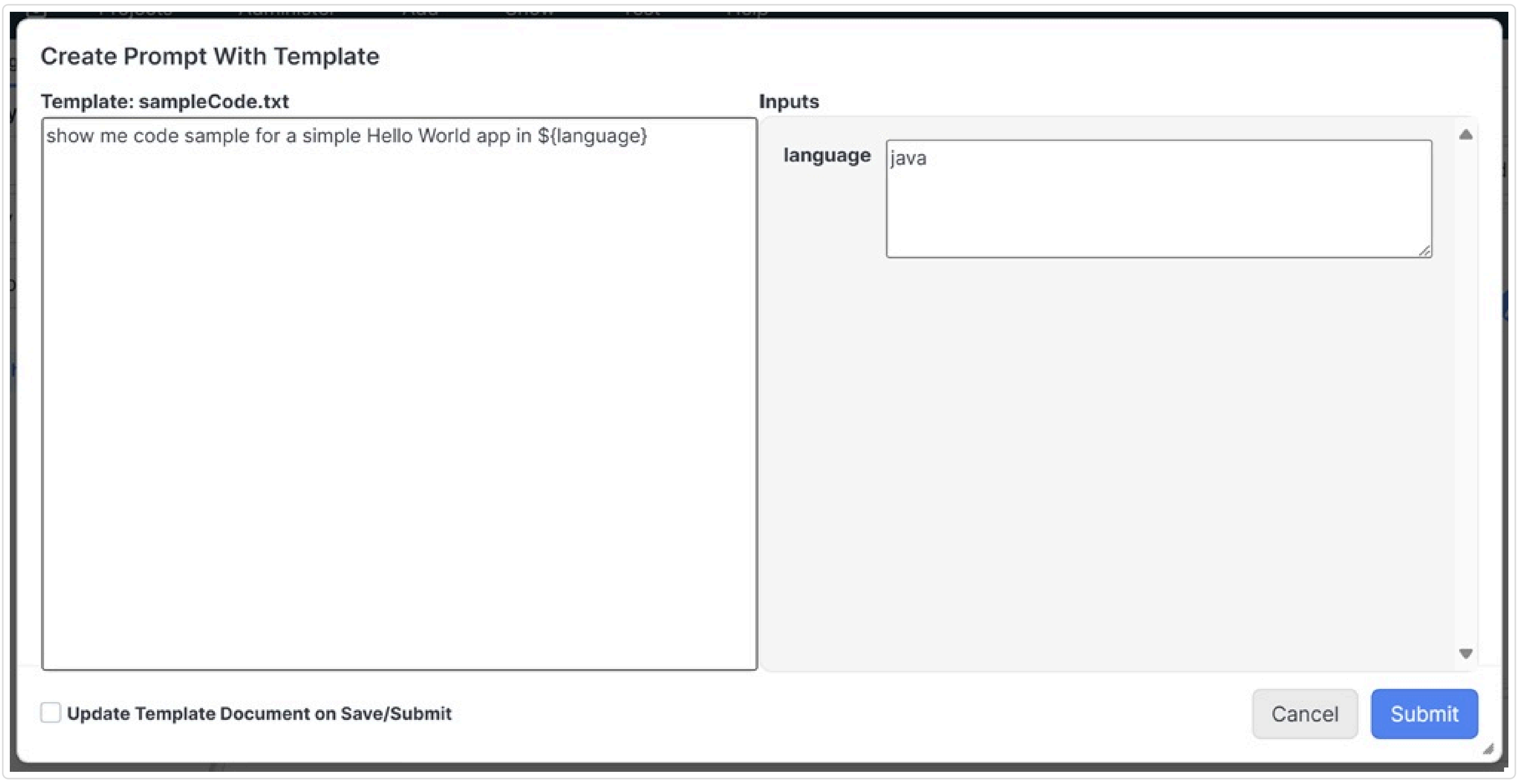
- ***Edit Prompt*** – Edit the prompt in place. You will have the option to save the modified prompt only or submit the modified prompt. If you choose to submit, the modified prompt will be submitted and the original prompt and everything after it will be deleted. It is a quicker alternative for deleting the prompt first, then submitting a new prompt.

- ***Edit Prompt with Template*** – Works just like *Edit Prompt*, should only be used if the prompt was created from a template. See section Working with Template below.
- ***Delete Prompt*** – Delete one prompt only.
- ***Delete Prompt and All Following Messages*** – Delete the prompt and all responses and prompts after that prompt. Any new prompt submitted will be continuing the conversation from the response before the deleted prompt.
- ***Submit again*** – Do not change existing conversation, but the same prompt will be submitted again. The new submission and its response will be appended to the end of the conversation. This is a quick way to test different settings without retyping the same prompt.

Working with Template

Using a template is an efficient way to build your prompt. A Template is a Vantiq text document which contains placeholders that are dynamically replaced with actual data when the template is used.

Use the ***Load Template*** button to bring up the template editor with your chosen Document. Then provide values for placeholder variables before submitting the prompt.


Create Prompt With Template

Template: sampleCode.txt

show me code sample for a simple Hello World app in ${language}

Inputs

language: java

Update Template Document on Save/Submit

You can also edit an existing prompt that was created using template. The same template editor will be shown with the template document and user input values loaded.

The editor has the option to "Update Template Document on Save/Submit" which allows you to build your template without leaving the LLM playground.

Settings Panel

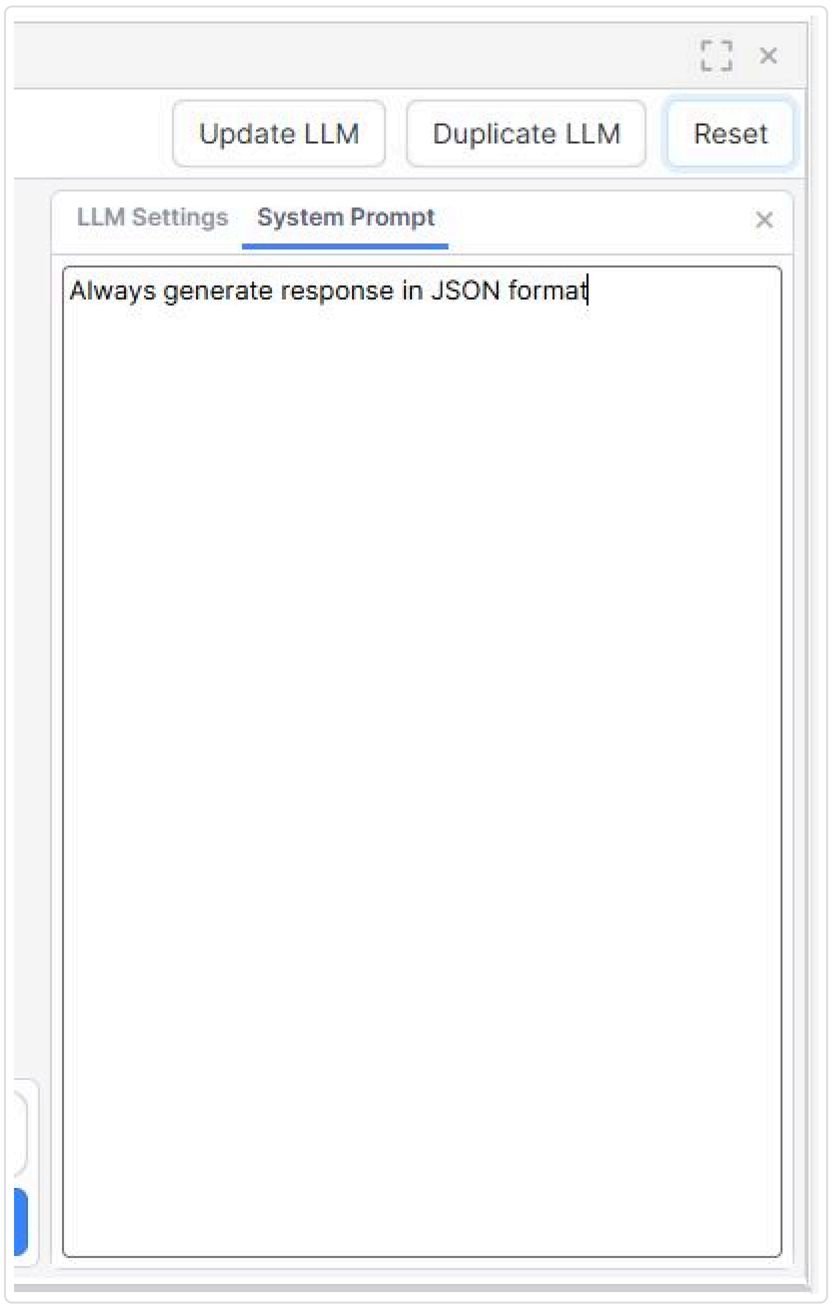
The settings panel is where you can modify LLM to experiment with and understand the capabilities of the LLM. Edit the settings via the various sliders, or by clicking the setting's value to open a text field.

Initially, settings are loaded from the saved LLM config. Modified settings will be used on the next prompt submission within the playground. It does not modify the LLM until you click the "***Update LLM***" button on top of the settings panel. You can also create a new LLM with the new settings by clicking the "***Duplicate LLM***" button.

The "***Reset***" button resets all settings back to the current configuration state of the selected LLM.

Note that each LLM has its own settings, and some settings may not apply to all LLMs. For example, the "Functions" setting is currently only available for GPT and Google Gemini models.

There is another sub-tab titled "System". It lets you specify a "System" prompt which is submitted with each user prompt. The system prompt is a predefined message or question provided by the system to guide the conversation or elicit a specific type of response from the LLM. For example, a system prompt of "Always generate response in JSON format" instructs the LLM to respond using JSON instead of plain text.



Always generate response in JSON format

The selected LLM may have already defined a default system prompt in its configuration. In this case, the default system prompt will be loaded automatically and displayed as read-only text.

There is also a collapse icon on the panel to minimize the panel to the left side of the screen. This gives you more space to work on the main chat area.

Navigation Panel

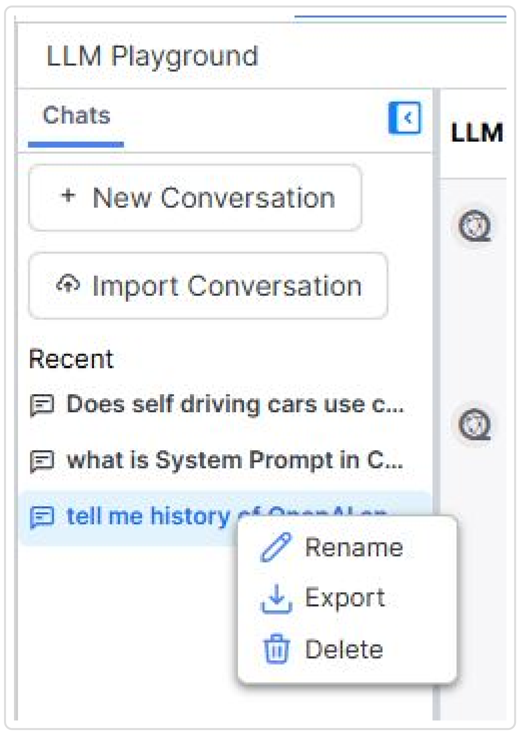
The navigation panel shows the history of conversations.

You can use the “**New Conversation**” button to start a new conversation. The conversation does not have a title until you submit the first prompt. The first prompt is also used as the initial name of the conversation. You can use the context menu “***Rename***” to rename the conversation.

You can click on a conversation name to jump back to an existing conversation. The main chatting area will load all previous prompts and responses saved for the selected conversation.

To export a conversation, use “**Export**” from its context menu. The exported conversation is saved as a JSON file. You can import the conversation back to the playground using the “**Import Conversation**” button on the navigation panel.

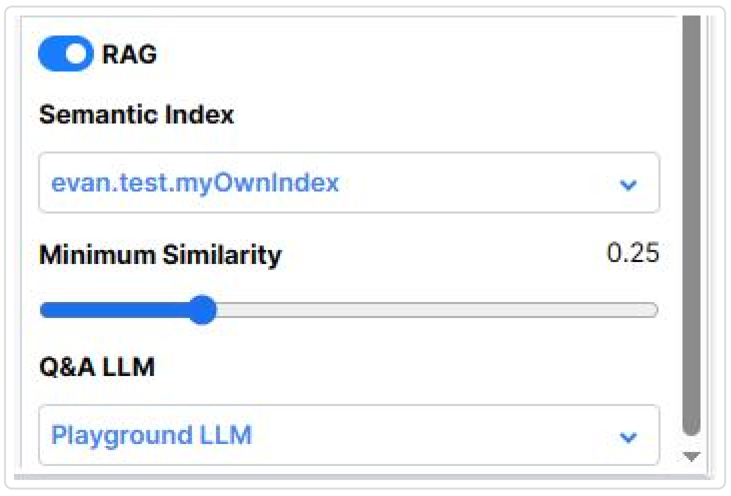
To delete a conversation, use “**Delete**” from its context menu.


LLM Playground
Chats
LLM
+ New Conversation
Import Conversation
Recent
Does self driving cars use c...
what is System Prompt in C...
tell me history of OpenAI an
Rename
Export
Delete

Testing Semantic Index

Besides testing an LLM, you can also test a Semantic Index you have created in Vantiq.

At the bottom of the Settings panel, there is a radio button “RAG” (this stands for Retrieval Augmented Generation). Enabling it allows you to choose a Semantic Index to test.



RAG

Semantic Index: evan.test.myOwnIndex

Minimum Similarity: 0.25

Q&A LLM: Playground LLM

Once you have selected a Semantic Index, any new prompt submitted will have the response generated using the Semantic Index (as part of Vantiq’s RAG algorithm) instead of from an LLM directly.

The LLM used to generate the response depends on the “Q&A LLM” setting. This lets you choose to either use the current LLM settings in the playground or use the default LLM specified in the Semantic Index. If Semantic Index’s default LLM is used, the LLM and settings seen in the playground will not be used.