

LLMkit Documentation

A library for making it easy for Salesforce's OmniStudio to call GPT

Important Note:

The full code and documentation repository can be found at <https://github.com/cmccguinness/LLMkit>.

There are three sources of information on how this library works. In order from most authoritative to least, they are:

1. The source to LLMkit.cls, which is the actual implementation
2. The source to LLMkitTest.cls, which is the test class for LLMkit.cls, and has working examples of exercising the class
3. [Demonstrations](#)
4. This README

That is the inverse of the order of readability, however.

Philosophy

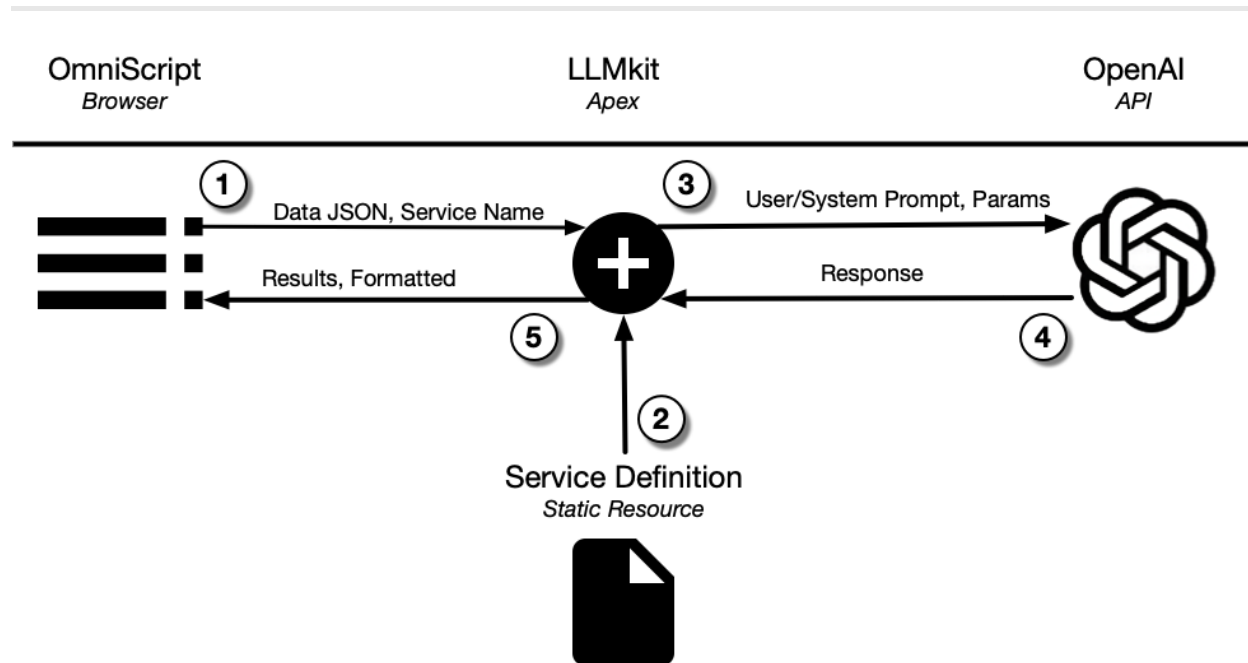
Rather than building a series of specific, one-off interfaces to OpenAI, the class provides a completely generic, parameter driven interface that does a lot of the heavy lifting for you. It supports the notion of bundling the parameters into a "service" (for example, a life insurance recommender) that can be re-used and shared. The users of the service need not know how the service works (although it's not hidden) to get the benefit from it.

The definition of a service consists of:

- Template(s) for building prompts; the templates are used to merge live data into the prompts at runtime.
- Settings for OpenAI models like model, temperature, max_tokens
- Instructions on how to handle the response

The motivation for this is that it takes a fair amount of effort to write, test, and refine prompts for LLMs, and prompt engineering is likely to require a different skill set than writing OmniScripts, so it makes sense to separate that out from building UI flows.

In general, the deployment-time flow will look like this:



1. The OmniScript call the class, passing the current, in-flight data (the Data JSON as it's called) for the input and the name of the service as an option.
2. The LLMkit class reads in the specification of the service from static resources and, optionally, retrieves the specified templates as well. It then merges the input data into the templates to product the final prompts.
3. The prompts and other parameters are sent as part of the call to OpenAI's API
4. OpenAI's API responds
5. Using the post processing instructions given in the service definition, LLMkit returns the results to the OmniScript

Installation

There are two .cls files to install. It's easiest to upload them as well in the Setup section of the web UI. Start by going to Setup, search for "classes", and click on Apex Classes.

From there you'll have a list of Apex classes. Hiding in the top of the list is a button labeled "New". Press it, and you are taken to an Apex Editor. Now, copy the contents of LLMkit.cls (all ~800 lines of it), and paste it into the editor. Press Save.

Repeat the same thing for the over 300 lines of LLMkitTest.cls.

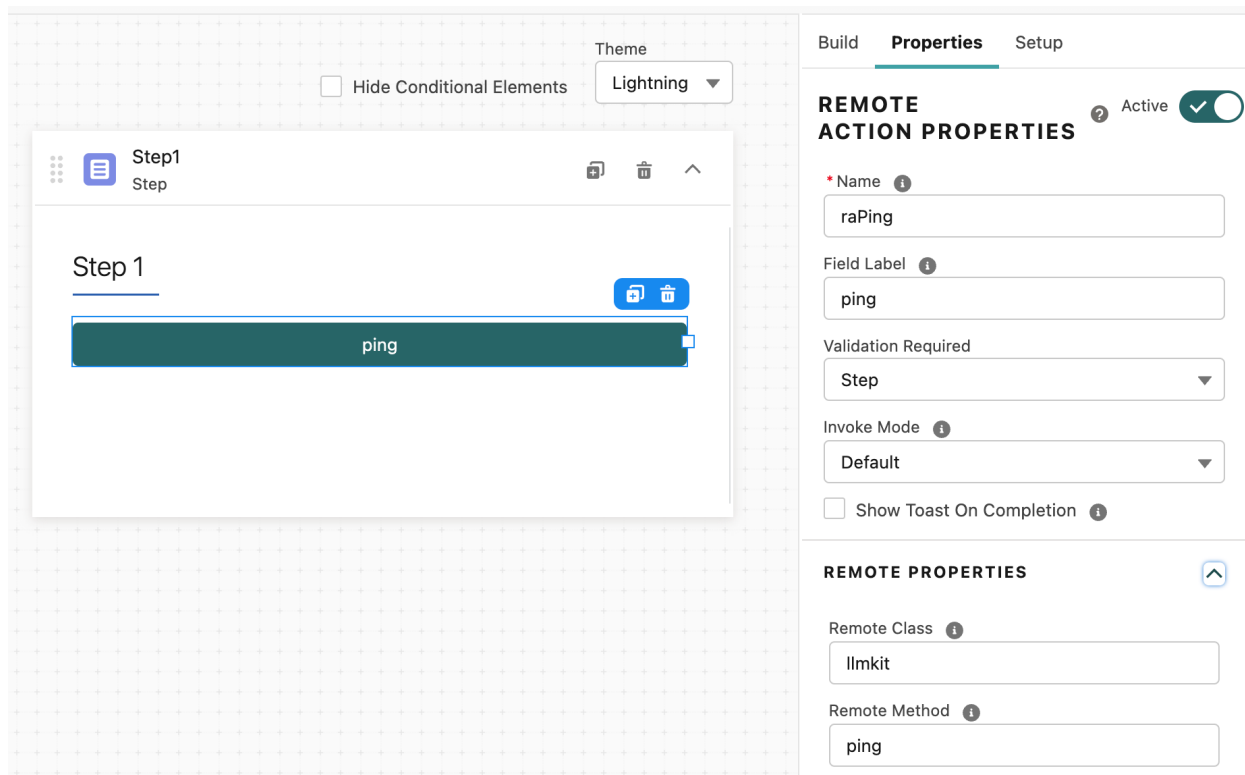
Once you've loaded it, you can go to Class Summary and run the test. You now have access to the LLM kit from OmniScript (and Apex and ...) and have a test class with high-80s% coverage if you wish to deploy it to production.

Calling Pattern

The class is named "LLMkit", and it follows the OmniStudio (aka Vlocity) Open Interface standard. If you're not used to it, it represents a very different approach from standard Apex classes. In general technical terms, it implements a "late-binding" approach: the method we want to call is passed in as a string. The class finds the method for you at runtime (late) rather than the normal Apex way, at compile time (early).

This makes it very easy to call from OmniScript, but a bit harder to call from "normal" Apex. The copious examples in the test class, however, should suffice to demonstrate how to call the methods from Apex directly (it's just a bit of additional overhead).

Here's an example of calling the "ping" interface from OmniScript:



You can see down at the bottom right that we just tell it the remote class is "llmkit" and the method is "ping". Automatically, there are three "inputs" passed along with the name of the method. The three are called "input", "output", and "options". (We'll see more about them in the other methods.) When calling from OmniScript or Integration Procedures, this is implicit in the UI that allows you to specify your call. From Apex, it's a bit messier. Here's a snippet of Apex from the test class that demonstrates how calls work:

```
Map<String,Map<String,Object>> args = new Map<String,Map<String,Object>>();
args.put('input',new Map<String,Object>());
args.put('output',new Map<String,Object>());
args.put('options',new Map<String,Object>());

LLMkit llmkit = new LLMkit();
llmkit.call('ping', args);

String response = (String) (args.get('output').get('response'));
```

Generally speaking, the test class contains many useful (if somewhat artificial) examples of calling the OpenAlkit class from APEX and should be sufficient to get you going.

Methods Supported

There are four callable methods:

Method	Description
ping	Just returns "pong", but slightly useful to check to see if you can call the class during installation. See the test class for an example of how it works.
generate_template	You pass it in your data, a template (or the name of a static resource file that holds the template), and it inserts the data into the template and returns the results to you.
call_openai	This takes your data, populates the prompt(s), sends it off to OpenAI, patiently (very patiently) awaits a reply and then sends it to you.
get_sr_as_json	Reads in a static resource which contains a JSON structure, parses it, and returns it to the caller. Useful for holding ad-hoc collections of data. (Also callable as getSRasJSON for backwards compatibility)

Method: ping

Only useful as a quick test to see if the class is available. Has no inputs, returns in output a "response" property with the value "pong".

To call from Apex see the method `test_ping()` in the test class.

Method Name	ping
options	<i>not used</i>
input	<i>not used</i>
output	response == "pong"

Method: generate_template

This will take a template specified in *options* (see below for format) and populate it with data out of *input*. The resulting template is returned in *response* as a string.

To call generate_template from Apex, see the method `test_generate_template_good()` in the test class.

Method Name	generate_template
options	<i>template</i> is either the template, or starts with '@' and is the name of a static resource that holds a template.
input	The source of data used to populate the template. With OmniScript, this is typically just the "Data JSON" which is sent automatically and contains all the in-flight data of the OmniScript.
output	<i>response</i> will be a string containing the populated template if successful.. <i>error</i> will hold an error message of something goes wrong. It does not exist on success.

Note that the *template* can either be a template literal (that is, just the text of the template) or start with an '@' character followed immediately (no space) with the name of a static resource (without its extension). In the second case, the template generator will read in the static resource and use its contents. This allows you to build up a library of prompt templates, if you wish, and re-use them for multiple purposes.

Template Format

The template format is a very simplified version of the Jinja2 templating system used by Python. This is so that you can develop templates using a Python tool such as [this Google Colab notebook](#) and then use them in Salesforce. In general, the text of the template is copied verbatim except when one of these three is encountered:

- {{ which is the start of an insertion of data, followed by }}

- {% which indicates some sort of control logic, followed by %}
- {# which indicates a comment that is not included in the output, followed by #}

The insertion of data uses a standard "." notation (and so NOT the OmniScript ":" notation) to traverse a data structure. For arrays, you can use array.0, array.1, etc. to refer to members, although it is expected that you will use a {% for %} loop almost always instead.

The control logic supports the following:

- {% if *variable* %}
- {% else %}
- {% endif %}
- {% for *loop_variable* in *array_variable* %}
- {% end for %}

For the {% if *variable* %} statement, the class looks to see if:

- *variable* exists
- When turned into a string, it does not equal "false" (case insensitive comparison)
- When turned into a string, it does not equal "0"

If all those are true, the variable is declared to be *true*.

{% else %} and {% endif %} work as expected.

Important limitations:

- ***You cannot have nested ifs.*** That's a future feature.
- ***You cannot have expressions in your if's test.*** That's a distant future feature

For the {% for %} statement, it is designed to simply iterate over an array.

Important limitations:

- ***At this point in time, you cannot have nested fors.*** That's a future feature

However, you can have a *for* inside an *if*, or an *if* inside a *for*, but you cannot do anything more complicated than that yet.

Template Example

This is an example used in the unit test *test_generate_template_logic* from the test class:

Input (Data JSON)	Template
<pre>{ "a": 0, "b": 1, "bdata": [10, 11, 12] }</pre>	<pre>{% if a %} a is non-zero {% endif %} {% if b %} bdata is {% for i in bdata %}{{ i }}, {% endfor %}{% endif %}</pre>

This generates:

```
bdata is 10, 11, 12,
```

Method: call_openai

This is the most important method, and therefore the most complicated.

The starting point is that it will send your inputs to OpenAI's "completion" interfaces and return the result to you. There's tremendous variability in how you do that and how the response gets handled. Much of the variability is there to make development a bit easier or testing work.

Let's start with the inputs to `call_openai`. When you are calling from OmniScript, the inputs are split into two bit buckets:

1. input, which holds either the entire in-flight data from your OmniScript at the point you make the call to LLMkit (this is the default), or some edited subset of that data that you've chosen (not default). In this documentation, we assume you've just sent the whole kit and caboodle of data, but it makes no real difference to the method.
2. options, which instruct the class on how to pre-process the data, call OpenAI, and post-process the results.

There's not much to say about the input (and what there is to say is mostly covered in the documentation for `generate_template`).

Options

Here are the options that affect the call. They will be further discussed after the table.



Option	Required?	Description
openai_service	N	Specifies a second source of options. Follows the '@' syntax (see below)
model	Y	Which OpenAI model are you using (e.g., gpt-4, text-divinci-003)
named_credential	Y*	The named credential used to provide the API key to OpenAI. Either this or api_key must be specified.
api_key	Y*	The actual, sk-..., OpenAI key you'll use instead of a named credential
system_template, user_template, prompt_template	Y	If you are calling a chat model (gpt-4, gpt-3.5-turbo), you need a system_template and a user_template. Otherwise you need just a prompt_template. All templates follow the '@' syntax.
temperature	N	The temperature setting for OpenAI.
max_tokens	N	The max_tokens setting for OpenAI
timeout	N	How long Salesforce will await a response. Defaults to 120000 milliseconds, which is the maximum allowed.
		If provided, this is used as the result and the actual call to OpenAI is not made. Useful for testing and

mock_response	N	demos.
raw_response	N	Normally, the text response from OpenAI is fished out of the JSON structure which the REST calls return. If this is provided and is true, the entire return from OpenAI will be returned from the method.
json_response	N	If you expect that the response from OpenAI is going to be a JSON structure rather than just text (that is, your prompts tell OpenAI to return JSON), including this option and setting it to true will cause the method to convert the string to JSON so you get a data structure back instead of a string.
history	N	A set of previous messages (see "History" below) that comprise earlier chat history
max_history	N	The maximum number of previous interactions to return as a means of pruning history
text_history	N	Indicates whether a pre-formatted string of the history should be returned
additional_data	N	Hard coded data the method should return (see "Additional Data" below)
additional_name	N	The top level name the additional data should have

@ Syntax

The options `openai_service`, `system_template`, `user_template`, `prompt_template`, and `mock_response` allows you to either include a literal string or the name of a static resource to fetch the value from. If the first character is an '@', then the rest of the option is taken as the name of a static resource that will be read and used. Here's an example that shows one of each style:

Remote Options

Key	Value	
openai_service	@chatcpq	
mock_response	Your order is valid.	

In this case, the service definition is coming from a static resource called chatcpq, but the literal text "Your order is valid." will be used as the mock response.

openai_service

While you can provide all the options directly to make a call to OpenAI, the intent (see Philosophy) is that you will put them together into a service specification. A service's specification is a JSON structure, which holds all of the parameters and options needed to call a service (but not the input data, of course). While you can pass in a complete service specification as an option to the call, it is normal to keep the service specification in a static resource. There are samples provided in the [Demos](#) subdirectory of this repository.

The logic of the class is that it will first look for the value of an option in the directly passed in values and, if not found, then look in the service definition. This allows you to use a service definition but override values in it (or add things, like the mock_response in the image above).

model

Specifies the name of the model, using the same naming convention that OpenAI uses. Note that if OpenAI adds new models, the source to the class will need to be updated. The name of the model is used to determine what the calling convention is (chat or not). Any model that starts with *ft:* is assumed be a fine-tuned version of *gpt-3.5-turbo*.

named_credential, api_key

All calls to OpenAI require a API key. The best way to do that, in Salesforce, is to used a Named Credential. If you are not familiar with Named Credentials, you can see my [newsletter article](#) on the topic. As an alternative, and although I discourage it, is to supply the api_key you'll use directly. One of the two must be present.

system_template, user_template, prompt_template

These are the templates that will be sent to OpenAI as your prompts. `system_template`, `user_template` are both required if you're calling gpt-4 or gpt-3.5-turbo, and just `prompt_template` is required for older models. These follow the `@` syntax, so they can be specified directly or as static resources. Note, however, in OmniScript, it is nearly impossible to pass literal templates if you are using any logic in your templates, as OmniScript will delete anything between matching `%`s. That problem does not happen if the template is embedded in the service definition, fortunately.

temperature, max_tokens

These is passed directly to OpenAI. Refer to [their documentation](#) for an explanation of the values.

timeout

Salesforce has a default timeout of 10 seconds for REST calls, which is way too short. The library replaces that with a default timeout of 120 seconds, the maximum. If you have a reason for something different, specify it here, in milliseconds.

mock_response

If you specify this parameter, the call will *not* make an actual call to OpenAI; instead, it will pretend this was the content of the response from a fake call. This is useful if you are working on other aspects of your integration and don't need to make endless calls, you want your demos to work quickly, or are building test classes. This follows the `@` syntax, so you can either give a string or the name of a static resource that holds your mock response.

raw_response

OpenAI returns a JSON structure containing all sorts of information, including the text of the response. Normally, we just fish the text out of the JSON structure and return "response" set to that in the output. Additionally, we return an output of "raw_response" that contains the full response from OpenAI. Passing in this option with a value that's true will cause response to hold the full response too.

json_response

With clever prompt engineering, you can get OpenAI to return not merely a text response, but a structured JSON object that has a more complicated response. OmniScript is not great at handling strings with JSON data in them, however, and normally you have to either use a DataRaptor or custom code to turn the string into proper data. If you include this option in your call, the response from OpenAI will be deserialized back to a real data structure, avoiding additional work for the OmniScript.

history, max_history, text_history

If you wish to build a conversational ("Chat") UI, with subsequent user inputs being understood in the context of earlier interactions, it is incumbent on the client application to maintain the history of interactions and supply that on each call to the OpenAI API. These three parameters are used to accomplish that.

Upon return from a call to gpt-4 or gpt-3.5-turbo, call_openai will return a "history" output that contains the conversation up to and including the last interaction. The structure of `history` is the following:

```
[ {"role": "user", "content": "user's input"}, {"role": "assistant", "content": "model's output"}, ... ]
```

I.e., an array of dictionaries, even numbered elements containing the user's input and odd numbered elements containing the model's output (starting numbering at 0, as one does).

The simplest method for carrying the history forward is to feed this back into the `history` option (both the option and output are named the same, which is less confusing than it seems.). That, however, presents a problem in that a long winded chat can grow quite a lengthy history and the models have to parse the entire history anew on each call.

The solution to that problem is the `max_history` option, which limits the number of previous conversations. If you specify `5` as the `max_history`, then only up to the previous 5 interactions will be returned in the `history` output (which means the size of the array will be 10, because each interaction takes two elements). A higher value for `max_history` may require you to have a larger value for `max_tokens` to handle the sizeable input or even require switching to the 16k or 32k models at some point (with risks of significant increased costs).

Because the returned `history` structure can be a bit difficult to consume directly in OmniScript (you might consider assembling a Flexcard to display it), there's an additional option called `text_history` that can be set to `true`. If you do that, then another output, `history_text` is created where the history is returned as a single string with newlines and labels to format the

conversation.

additional_data, additional_name

This is a bit of a hack to make it a bit easier to build chat-driven OmniScripts. If both are specified, then the output of the call includes a top level entry with the name specified in `additional_name`, containing the data specified in `additional_data`. I used this to clear the input field upon return from the call.

Return from call_openai

The method returns, in output:

Key	Value
elapsed_seconds	How long, in seconds, it took to call OpenAI and get a response
raw_response	A string holding the body of the response from OpenAI. This would need to be deserialized most likely for further use, but it's intended mostly as a debugging.
response	The response from OpenAI. Absent the options <code>raw_response</code> and <code>json_response</code> , it's just the text of the response. With <code>json_response</code> , it's deserialized into a data structure (error if not deserializable). With <code>raw_response</code> , it's the same data as in <code>raw_response</code> except deserialized – that is, the entire response from OpenAI in a parsed structure.
error	Rut-roh, something bad happened.
history	An array of user inputs and system responses, as documented above in the section on <code>history</code>
history_text	A human readable form of <code>history</code> , optionally returned
<i>additional_name</i>	Any additional data that was passed into the options, as documented above.

Method: get_sr_as_json

In OmniScript, you often need to build bundles of data as JSON (or, really, JavaScript) structures. The typical method is to use a Set Values which you edit as JSON. The editor, however, is not good at complicated structures. The `get_sr_as_json` method offers an alternative, at a cost of a Remote Action call; it reads in a static resource and parses it as JSON. In this manner, you can maintain (and potentially reuse) more complicated JSON structures outside of OmniScript.

Unusual for many calls, all the arguments to the call are options. They are:

Key	Value
resource	Required. The name of the static resource
path	Required. The top level name to put the retrieved JSON under.
mock_response	Optional. Not typically used, this is a JSON string that will be parsed and returned as if it were coming from a static resource. The intended user is the test class (which does not rely on any given static resource being present), but it does provide a way to deserialize a JSON string, I supposed.

It returns the retrieved JSON data under the name provided in `path`.

Source to LLMkit.cls

Here is the source to the Apex class:

```
/**
 * LLM KIT
 *
 * https://github.com/cm McGuinness/LLMkit
 *
 * Version 2.1
 *
 * An Apex class for building complicated prompts for LLMs
 * and submitting them to ChatGPT - all for use with OmniStudio
```

```

*
* There are two big pieces to this class. The first is a templating system
* that allow us to store complicated prompt templates as static resources.
* A template might look like:
*
*     Give me 10 ideas for a company name in the {{ Step1.industry }}
industry.
*
* The design of the templating language follows the conventions established
* in the Jinja2 system, so that there is a portability of simple templates
* between this and Jinja2 itself.
*
* The second part drives creation of the system and user inputs to ChatGPT
* using the templating system and then submits the requests to OpenAI via
* their API.
*
* Note 1:
*
*     You might say "gosh, this is way too big and should be
*     broken up into smaller classes." And you'd be right.
*     However, for ease of distribution, I'm going to keep
*     this in one class for now.
*
* Note 2:
*
*     It is early days for this code, and there will be bugs, lack
*     of error checking, and missing features. Subscribe to my
*     substack, https://mcguinnessai.substack.com, to stay in
*     the loop.
*/

/**
* Copyright (c) 2024 Charles McGuinness<charles@mcguinness.us>
*
* Permission is hereby granted, free of charge, to any person obtaining a copy
* of this software and associated documentation files (the "Software"), to deal
* in the Software without restriction, including without limitation the rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included in
all

```



```

* copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
* AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
* LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
* OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*
***/

```

```

global with sharing class LLMkit implements Callable
{
    /**
     * TABLE OF CONTENTS
     */
    /**
     *
     * There are five main sections to this class:
     *
     * 1. The interface section, which provides the appropriate methods to
     *    allow the code to be called from OmniStudio
     *
     * 2. Internal Utilities
     *
     * 3. External Utilities
     *
     * 4. The template processing section, which implements our Jinja2-
inspired
     *    templating language for inserting data from Salesforce into a
prompt
     *
     * 5. The OpenAI interface section, which manages the calls to OpenAI
     */
}

```

```

/*****
/*  Section 1: The interfaces to the outside world      */
/*****

/**
 * call
 *
 * The main entry point using the OmniStudio style interface.
 *
 * This basically pulls apart the inputs to feed it to an old-school
 * invokeMethod (so if you need to go really old-school, the code
 * is basically all here for you.)
 */
public Object call(String action, Map<String, Object> args) {

    Map<String, Object> input = (Map<String, Object>)args.get('input');
    Map<String, Object> output = (Map<String, Object>)args.get('output');
    Map<String, Object> options = (Map<String, Object>)args.get('options');

    return invokeMethod(action, input, output, options);
}

/**
 * invokeMethod
 *
 * This is the dispatcher between the external call and our internal
methods.
 */
public Object invokeMethod(String methodName, Map<String, Object> input,
Map<String, Object> output, Map<String, Object> options) {

    if (methodName == 'ping') {
        return ping(output);
    }

    if (methodName == 'generate_template') {
        return generate_template(input, output, options);
    }
}

```

```

        if (methodName == 'call_openai') {
            return call_openai(input, output, options);
        }

        if (methodName == 'get_sr_as_json') {
            return get_sr_as_json(input,output,options);
        }

        // For backwards compatibility; what was I thinking when I came up with
this name?
        if (methodName == 'getSRasJSON') {
            return get_sr_as_json(input,output,options);
        }

        output.put('error', 'No Such Method '+methodName);
        return false;
    }

}

/*****
/*  Section 2: Internal Utilities                                     */
*****/

/**
 * u_get_object_type
 *
 * Debugging function, taken from:
https://ideas.salesforce.com/s/feed/0D58W000069LiQ8SAK
 *
 * Commented out when not used to avoid affecting coverage counts...
 */

// public static String u_get_object_type(Object obj) {
//     String result = 'DateTime';
//     try { DateTime typeCheck = (DateTime)obj; }
//     catch(System.TypeException te) {
//         String message = te.getMessage().substringAfter('Invalid
conversion from runtime type ');

```

```

//      result = message.substringBefore(' to Datetime');
//    }
//    return result;
// }

/**
 * u_get_option (as String)
 * u_get_option_object (as JSON)
 *
 * We have two sources of options: the ones passed in and (potentially) a
service specification
 * Look first in the service specification and then override with the
passed in options
 */
Object u_get_option_object(String name, Map<String,Object> passed,
Map<String,Object> service) {
    Object result = null;
    if (service != null) {
        if (service.containsKey(name)) {
            result = service.get(name);
        }
    }

    if (passed.containsKey(name)) {
        result = passed.get(name);
    }

    return result;
}

String u_get_option(String name, Map<String,Object> passed,
Map<String,Object> service) {

    Object result = u_get_option_object(name, passed, service);

    if (result != null) {
        return String.valueOf(result);
    }
    return null;
}

```

```

/**
 * u_read_static_resource:
 *
 * resource_name:    The name of the static resource to retrieve (without
its extension)
 *
 * returns:          A string with the contents of the static resource
 *                  null if not found
 *
 * (Obviously this is for text resources only...)
 */
private String u_read_static_resource(String resource_name) {
    StaticResource[] svc = [SELECT Id, Body FROM StaticResource WHERE Name
= :resource_name LIMIT 1];
    if (svc.size() == 0) {
        return null;
    }

    return svc[0].Body.toString();
}

/**
 * u_literal_or_static
 *
 * If the first character of the string is "@", the rest is the name of a
 * static resource. In that case, fetch it and return the contents.
 *
 * If the first character is not, just return the input string
 */
private String u_literal_or_static(String los) {
    if (los == null) {
        return null;
    }
    if (!los.startsWith('@')) {
        return los;
    } else {
        String sr = u_read_static_resource(los.substring(1));
        return sr;
    }
}

```

```

/**
 * u_add_error
 *
 * Sets or adds an error message to the output
 *
 */
private static void u_add_error(map<String,Object> output, String message)
{
    String err = '';

    if (output.containsKey('error')) {
        err = ((String) output.get('error')) + '\n';
    }
    err = err + message;
    output.put('error', err);
}

/**
 * u_get_history
 *
 * Gets the chat history, if any, from options, to support ongoing chats
 */
private List<Map<String,String>> u_get_history(Map<String,Object> options)
{
    List<Map<String,String>> history = new List<Map<String,String>>();

    // If history is being passed in but there's none yet, we'll get a type
    // exception. Apex is deficient in ways to test for types, so catching
    // the exception is the least worst solution
    try {
        Object oHistory = options.get('history');

        if (oHistory != null) {
            List<Object> lhistory = (List<Object>) oHistory;

            for (Integer i = 0; i < lhistory.size();i++) {
                Map<String,String> out_row = new Map<String,String>();
                Map<String,Object> in_row = (Map<String,Object>)
lhistory.get(i);
                out_row.put('role', String.valueOf(in_row.get('role')));
                out_row.put('content',
String.valueOf(in_row.get('content')));

```

```

        history.add(out_row);
    }
}
} catch(System.TypeException te) {
    return new List<Map<String,String>>();
}

return history;
}

/**
 * u_history_to_text
 *
 * Converts history to a string format suitable for a preformatted HTML
element
 */
private String u_history_to_text(List<Map<String,String>> history) {
    String ret = '';

    for (Integer i=0;i<history.size();i++) {
        Map<String,String> ele = history.get(i);
        ret += ele.get('role').toUpperCase() + ':\n';
        ret += ele.get('content') + '\n\n';
    }

    return ret;
}

```

```

/*****
/*  Section 3: External Utilities                                     */
/*****

```

```

/**
 * ping
 *

```

```

    * Stupidest of test methods, yet when you need it, you really need it...
    * It's in the interface section because it's just here to let people
    * do easy testing of their ability to call into this class
    */
private Object ping(Map<String, Object> output) {
    output.put('response', 'pong');
    return true;
}

/**
 * get_sr_as_json
 *
 * Read a static resource in that is a JSON file, deserialize the JSON
 * and return it to the caller
 */
private Object get_sr_as_json(Map<String, Object> input, Map<String, Object>
output, Map<String, Object> options) {
    String resource_name = (String) (options.get('resource'));
    String path_name = (String) (options.get('path'));

    String sr;

    if (options.containsKey('mock_response')) {
        sr = (String) (options.get('mock_response'));
    } else {
        sr = u_read_static_resource(resource_name);
    }
    Map<String, Object> response = new Map<String, Object>();
    output.put(path_name, JSON.deserializeUntyped(sr));
    return true;
}

/*****
/* Section 3: Template Processing */
*****/

```



```

/**
 * class Token
 *
 * We're going to tokenize the template (at a very high level),
 * and this class holds the tokens. It's kind of vague what it
 * does, because over time it will do more...
 */
private class Token {
    public String type;    // Usually text, logic, comment, etc.
    public String value;

    public Token(String type, String value) {
        this.type = type;
        this.value = value;
    }
}

/**
 * getPath
 *
 * Given a a.b.c kind of notation for accessing JSON structures, this tries
to find
 * it in the input structure (which is usually the Data JSON from
OmniScript).
 * For the moment arrays can be indexed in an a.0.b kind of way, will add
proper
 * subscripts later when I feel the pain too much not to.
 */
private Object getPath(String path, map<String,Object> input, Boolean
return_actual) {
    List<String> path_parts = path.split('\\.');
    Object current = input;
    for (String path_part : path_parts) {
        if (current instanceof Map<String,Object> && ((Map<String,Object>)
current).containsKey(path_part)) {
            current = ((Map<String,Object>)current).get(path_part);
        } else if (current instanceof List<Object>) {
            current =
((List<Object>)current).get(Integer.valueOf(path_part));
        } else {
            return null;

```

```

    }
}
if (return_actual) {
    return current;
}
return String.valueOf(current);
}

/**
 * Tokenizer
 *
 * Read through the template and break it up into sections.  Sections are
 * one of:
 *
 *     Just plain text
 *     {{ some value from input to insert }}
 *     {% if | else | endif | for | endfor %}
 *     {# comment #}
 *
 * We don't output the comments, though.
 */

Token[] tokenize(String template) {
    Integer loc_insertion;
    Integer loc_logic;
    Integer loc_comment;

    Token[] tokens = new Token[0];

    while (template.length() > 0 ) {
        // Find the start of the next insertion, logic block, or comment
        loc_insertion = template.indexOf('{{');
        loc_logic = template.indexOf('{%');
        loc_comment = template.indexOf('{#');

        // If none is found, then we are done
        if (loc_insertion == -1 && loc_logic == -1 && loc_comment == -1) {
            tokens.add(new Token('text', template));
            return tokens;
        }
        if (loc_insertion == -1) {
            loc_insertion = 999999;

```

```

    }

    if (loc_logic == -1) {
        loc_logic = 999999;
    }

    if (loc_comment == -1) {
        loc_comment = 999999;
    }

    // Is the next thing an insertion, a logic block, or a comment?

    // Insertion
    if (loc_insertion < loc_logic && loc_insertion < loc_comment) {
        tokens.add(new Token('text', template.substring(0,
loc_insertion)));
        template = template.substring(loc_insertion);
        loc_insertion = template.indexOf('}}');
        tokens.add(new Token('insertion', template.substring(2,
loc_insertion).trim()));
        template = template.substring(loc_insertion+2);
        continue;
    }

    // Logic Block
    if (loc_logic < loc_comment) {
        tokens.add(new Token('text', template.substring(0,
loc_logic)));
        template = template.substring(loc_logic);
        loc_logic = template.indexOf('%}');
        tokens.add(new Token('logic', template.substring(2,
loc_logic).trim()));
        template = template.substring(loc_logic+2);
        continue;
    }

    // Comment, which is simply removed
    tokens.add(new Token('text', template.substring(0, loc_comment)));
    template = template.substring(loc_comment);
    loc_comment = template.indexOf('#}');
    template = template.substring(loc_comment+2);

```

```

    }    /* END WHILE */

    return tokens;

}

/**
 * iterate_tokens
 *
 * Loop over the tokens, copying them from input to putput
 *
 * Text tokens get copied as is
 * Expansion tokens ({{ path_in_json}}) get replaced with data
 * Logic tokens {% if | else | endif | for | endfor %}) affect what's
getting copied conditionally
 */
private String iterate_tokens(Token[] tokens, Map<String,Object>input) {
    String output_template = '';

    // Loop over the tokens
    Boolean outputting = true;

    Token t;

    for (Integer token_iter=0;token_iter<tokens.size();token_iter++) {
        t = tokens[token_iter];

        if (t.type == 'text') {
            if (outputting) {
                output_template += t.value;
            }
            continue;
        }

        if (t.type == 'insertion') {
            if (outputting) {
                String path = t.value;
                Object value = getPath(path, input, false);
                if (value == null) {
                    return '*** PATH ' + String.valueOf(path) + ' NOT FOUND
***';

```

```

        }
        output_template += String.valueOf(value);
    }
    continue;
}

if (t.type == 'logic') {
    String[] parts = t.value.split(' ');
    if (parts[0] == 'endif') {
        continue;
    }

    if (parts[0] == 'if') {
        String path = parts[1];
        Object value = getPath(path, input, false);

        if (value == null) {
            outputting = false;
            continue;
        }

        String svalue = String.valueOf(value);
        if (svalue.toUpperCase() == 'FALSE' || svalue == '0') {
            outputting = false;
        } else {
            outputting = true;
        }
        continue;
    }

    if (parts[0] == 'else') {
        outputting = !outputting;
        continue;
    }

    if (parts[0] == 'endif') {
        outputting = true;
        continue;
    }

    if (!outputting) {
        continue;
    }
}

```

```

    }

    if (parts[0] == 'for') {
        String iteration_variable = parts[1];
        String path = parts[3];
        Object value = getPath(path, input, true);
        // System.debug('Value class: ' + u_get_object_type(value));
        // If uncommented, need to uncomment method too
        if (value == null) {
            return '*** PATH ' + path + ' NOT FOUND ***';
        }
        if (!(value instanceof List<Object>)) {
            return '*** PATH ' + path + ' NOT ARRAY ***';
        }

        List<Object> list_of_values = (List<Object>)value;

        Token[] inner_tokens = new Token[0];

        for (Integer
endfor=token_iter+1;endfor<tokens.size();endfor++) {
            if (tokens[endfor].type == 'logic' &&
tokens[endfor].value == 'endfor') {
                break;
            }
            inner_tokens.add(tokens[endfor]);
            token_iter = endfor+1;
        }

        for (Object o_iter: list_of_values) {
            Map<String,Object> inner_input = new
Map<String,Object>();

            inner_input.putAll(input);
            inner_input.put(iteration_variable, o_iter);
            output_template += iterate_tokens(inner_tokens,
inner_input);
        }
        continue;
    }
}
}
}

```

```

        return output_template;
    }

    /**
     * populate_template
     *
     * template is one of:
     *
     *     - A string that starts with @ and is followed by the name of a
     *       static resource, in which case we read in the static resource
     *       and use it as the source of our template, or if not...
     *
     *     - A string that does not start with @ and is the text of our
     *       template.
     *
     * Why the two choices? You might want to start with your templates being
in
     * individual static resources to make it easier to edit them, but once
they've
     * become final move them into your openai_service file so everything is in
one
     * place for easy DevOps and sharing.
     */
    private String populate_template(String template, Map<String, Object> input)
    {

        // Make sure we actually got a string (if the options didn't include
        // a template, this can happen
        if (template == null) {
            return null;
        }

        String input_template = u_literal_or_static(template);    // function
handles @ or not
        if (input_template == null) {
            return null;
        }

        String output_template = '';

        // Tokenize the template

```

```

        Token[] tokens = tokenize(input_template);

        output_template = iterate_tokens(tokens, input);

        return output_template;
    }

    /**
     * generate_template
     *
     * An OmniScript callable version of populate_template, if you wanted to
     generate a template but not use it yet.
     * Could be useful if you were building an interface to other OpenAI
     endpoints (e.g., /completions) that this
     * class doesn't support yet.
     */
    private Object generate_template(Map<String,Object> input,
    map<String,Object> output, map<String,Object> options) {
        String template_name = (String)(options.get('template'));

        String output_template = populate_template(template_name, input);
        if (output_template == null) {
            output.put('error', 'The template ' + template_name + ' was not
found');
            return false;
        }
        output.put('response', output_template);
        return true;
    }

```

```

/*****
/*  Section 4: Calling OpenAI                                     */
/*****

```

```

/**
 * openai_model

```



```

    *
    * For this class, there are two types of models we use. One is an older
    * set of models (e.g., text-davinci-003), which are NOT chat models. They
    * accept one prompt as an input. The other are the chat models (e.g.,
gpt-4)
    * and they accept a series of prompts.
    */
private class openai_model {
    String model_name;
    String model_endpoint;
    Boolean isChat;          // basically, old-style or new-style

    public openai_model(String mn, String me, Boolean ic) {
        this.model_name = mn;
        this.model_endpoint = me;
        this.isChat = ic;
    }
}

/**
 * get_models
 *
 * Assembles a list of models, mostly so we know what kind of input
 * to pass to them.
 *
 * For an up to date list of models, see:
https://platform.openai.com/docs/models
 */
private Map<String,openai_model> get_models() {
    Map<String,openai_model> map_of_models = new Map<String,openai_model>();

    // This are the models that use the "older" completions endpoint (just
one input prompt)
    map_of_models.put('text-davinci-002', new openai_model('text-davinci-
002','/v1/completions', false));
    map_of_models.put('text-babbage-002', new openai_model('text-babbage-
001','/v1/completions', false));
    map_of_models.put('gpt-3.5-turbo-instruct', new openai_model('gpt-3.5-
turbo-instruct','/v1/completions', false));

```

```

        // These are the models that uses the "newer" chat/completions endpoint
(system and user inputs)
        map_of_models.put('gpt-4', new openai_model('gpt-
4','/v1/chat/completions',true));
        map_of_models.put('gpt-4-32k', new openai_model('gpt-4-
32k','/v1/chat/completions',true));
        map_of_models.put('gpt-3.5-turbo', new openai_model('gpt-3.5-
turbo','/v1/chat/completions',true));
        map_of_models.put('gpt-3.5-turbo-16k', new openai_model('gpt-3.5-turbo-
16k','/v1/chat/completions',true));

        // These are the time stamped models that will constantly change...
        map_of_models.put('gpt-3.5-turbo-0301', new openai_model('gpt-3.5-turbo-
0301','/v1/chat/completions',true));

        map_of_models.put('gpt-4-0613', new openai_model('gpt-4-
0613','/v1/chat/completions',true));
        map_of_models.put('gpt-4-32k-0613', new openai_model('gpt-4-32k-
0613','/v1/chat/completions',true));
        map_of_models.put('gpt-3.5-turbo-0613', new openai_model('gpt-3.5-turbo-
0613','/v1/chat/completions',true));
        map_of_models.put('gpt-3.5-turbo-16k-0613', new openai_model('gpt-3.5-
turbo-16k-0613','/v1/chat/completions',true));

        map_of_models.put('gpt-4-1106-preview', new openai_model('gpt-4-1106-
preview','/v1/chat/completions',true));
        map_of_models.put('gpt-3.5-turbo-1106', new openai_model('gpt-3.5-turbo-
1106','/v1/chat/completions',true));

        // Retired Models
        //      map_of_models.put('text-davinci-003', new openai_model('text-
davinci-003','/v1/completions', false));
        //      map_of_models.put('gpt-4-0314', new openai_model('gpt-4-
0314','/v1/chat/completions',true));
        //      map_of_models.put('gpt-4-32k-0314', new openai_model('gpt-4-32k-
0314','/v1/chat/completions',true));
        //      map_of_models.put('text-davinci-001', new openai_model('text-
davinci-001','/v1/completions', false));
        //      map_of_models.put('text-babbage-001', new openai_model('text-
babbage-001','/v1/completions', false));

```

```

        //      map_of_models.put('text-ada-001', new openai_model('text-ada-
001','/v1/completions', false));

    return map_of_models;
}

/**
 * call_openai
 *
 * The whole enchilada. You give it a system and user template name, a
model, a named credential
 * (mostly for the OpenAI API Key), and some optional parameters, and it
will populate your
 * templates, send everything to ChatGPT, and then fish out the response
and return it to you.
 * If everything works, it works. Needs a lot of error checking to be
reliable.
 *
 * Options:
 *      * means mandatory, - is optional
 *
 *      Either:
 *          * openai_service (which contains the other parameters in JSON
format { 'named_credential': '...', ...}
 *      or:
 *          * One and only one of:
 *              named_credential
 *              api_key
 *          If the model uses a /v1/chat/completion endpoint
 *              * system_template
 *              * user_template
 *          If the model uses a /v1/completion endpoint
 *              * template
 *          * model
 *          - history (previous dialog)
 *          - max_history (max number of interactions to save in returned
history, where one interaction is a pair of user and assistant responses)
 *          - text_history (We always return history as JSON, this does a
trivial formatting to a string)
 *          - temperature

```

```

    *           - max_tokens
    *           - timeout (defaults to 120000 here)
    *           - mock_response (if you want to test without calling the REST
endpoint)
    *           - raw_response (if you want to see the raw response from the
REST endpoint)
    *           - json_response (we take the text response and deserialize it
for you)
    *
    *           Additional Data: Data passed in that should just be passed out.
Either both or neither of:
    *           - additional_data (the data to return)
    *           - additional_name (the top level name to return)
    *
    *           For openai_service, system_template, user_template, and template
parameters, they can
    *           either be literal strings or they can start with the '@' character
followed by the
    *           name of a static resource. In the second case, the contents of the
named static resource
    *           is used.
    */
    private Boolean call_openai(Map<String,Object> input, map<String,Object>
output, map<String,Object> options) {

        // We use a Long time, but we hope our response times are short ...
(Sorry, Dad Joke)
        Long start_time = Datetime.now().getTime();

        // -----
        // First step: Collect all our options
        // -----

        Map <String,Object> service_options = null;

        // Look to see if we have a openai_service file for one and done
sourcing of parameters
        if (options.containsKey('openai_service')) {
            String openai_service_name = (String)
(options.get('openai_service'));
            String openai_service = u_literal_or_static(openai_service_name);
            if (openai_service == null) {

```

```

        output.put('error', 'The openai_service static resource ' +
openai_service_name + ' was not found');
        return false;
    }

    service_options = (Map<String,Object>)
JSON.deserializeUntyped(openai_service);
}

// First, get the model name as that drives some other option choices:
String model_name = u_get_option('model', options, service_options);
if (model_name == null) {
    output.put('error', 'model_name is required option');
    return false;
}

openai_model model = get_models().get(model_name);

// Fine tuning?
if (model == null && model_name.startsWith('ft:')) {
    model = new openai_model(model_name, '/v1/chat/completions', true);
}

if (model == null) {
    output.put('error', 'I do not know of a model named '+model_name);
    return false;
}

// Retrieve the options that control what we do

// We need either a Named Credential or an API key
String named_credential_name = u_get_option('named_credential',
options, service_options);
String api_key = null;

if (named_credential_name == null) {
    api_key = u_get_option('api_key', options, service_options);
    if (api_key == null) {
        output.put('error', 'Require either named_credential or
api_key');
        return false;
    }
}

```

```

    }

    String system_template_name;
    String user_template_name;
    String prompt_template_name;

    // Get the template(s) for the prompt (depending on what type of call
this is)
    if (model.isChat) {
        system_template_name = u_get_option('system_template', options,
service_options);
        user_template_name = u_get_option('user_template', options,
service_options);
    } else {
        prompt_template_name = u_get_option('prompt_template', options,
service_options);
    }

    String temperature = u_get_option('temperature', options,
service_options);

    String max_tokens = u_get_option('max_tokens', options,
service_options);

    // Note that the default timeout in Salesforce is too low to work
    // with OpenAI reliably, we our default is the maximum, 2 minutes.
    Integer timeout_limit = 120000;
    String s_timeout = u_get_option('timeout', options, service_options);
    if (s_timeout != null) {
        timeout_limit = Integer.valueOf(s_timeout);
    }

    // A mock response has our code skip the actual calling of OpenAI and
    // instead return whatever is passed in to us here. This allows for
    // avoiding unnecessary charges. This is also used by ChatKitTest,
    // as you are NOT allowed to make outbound REST calls when called from
    // a unit test, and so this will prevent that from happening too.
    String mock_response =
u_literal_or_static(u_get_option('mock_response', options, service_options));

    // If you want to see the raw response from ChatGPT, set this to true
    Boolean raw_response = false;

```

```

    String s_raw_response = u_get_option('raw_response', options,
service_options);
    if (s_raw_response != null) {
        raw_response = Boolean.valueOf(s_raw_response);
    }

    // We can take the text message generated by OpenAI and convert it
    // into JSON for you. Hopefully, your prompt works :-))
    Boolean json_response = false;
    String s_json_response = u_get_option('json_response', options,
service_options);
    if (s_json_response != null) {
        json_response = Boolean.valueOf(s_json_response);
    }

    // Note that non-chats don't have history, but this doesn't hurt

    Boolean text_history = false;
    String s_text_history = u_get_option('text_history', options,
service_options);
    if (s_text_history != null) {
        text_history = Boolean.valueOf(s_text_history);
    }

    String max_history = u_get_option('max_history', options,
service_options);
    // System.debug('max_history='+String.valueOf(max_history));
    List<Map<String,String>> history = u_get_history(options);

    // System.debug('Finished fetching options');

    // -----
    // Second step: Populate the data into the template(s)
    // -----

    String user_template;
    String system_template;
    String prompt_template;

    if (model.isChat) {
        user_template = populate_template(user_template_name, input);

```

```

        if (user_template == null) {
            output.put('error', 'The user template ' +
String.valueOf(user_template_name) + ' was not found');
            return false;
        }
        // System.debug('User template generated');

        system_template = populate_template(system_template_name, input);
        if (system_template == null) {
            output.put('error', 'The system template ' +
String.valueOf(system_template_name) + ' was not found');
            return false;
        }
        // System.debug('System template generated');
    } else {
        prompt_template = populate_template(prompt_template_name, input);
        if (prompt_template == null) {
            output.put('error', 'The prompt template ' +
String.valueOf(prompt_template_name) + ' was not found');
            return false;
        }
        // System.debug('Prompt template generated');
    }

    // -----
    // Third step: Create the input JSON for Open AI
    // -----

    Map<String,Object> input_request = new Map<String,Object>();

    input_request.put('model', model_name);
    if (temperature != null) {
        input_request.put('temperature', Double.valueOf(temperature));
    }
    if (max_tokens != null) {
        input_request.put('max_tokens', Integer.valueOf(max_tokens));
    }

    // Need this later for history
    Map<String,String>role_user = null;

```



```

if (model.isChat) {
    List<Map<String,String>> messages = new List<Map<String,String>>();

    Map<String,String>role_system = new Map<String,String>();
    role_system.put('role','system');
    role_system.put('content', system_template);
    messages.add(role_system);

    // Add in previous chat history
    for (Integer i=0;i<history.size();i++) {
        messages.add(history.get(i));
    }

    role_user = new Map<String,String>();
    role_user.put('role','user');
    role_user.put('content', user_template);
    messages.add(role_user);

    input_request.put('messages', messages);
} else {
    input_request.put('prompt', prompt_template);
}

String request_body = JSON.serialize(input_request);

// System.debug('Input JSON generated');

// -----
// Fourth step: Make the REST call
// -----

HttpRequest req = new HttpRequest();

// Our Endpoint varies based upon whether we have a named credential
or not

if (named_credential_name != null) {
    req.setEndpoint('callout:' + named_credential_name +
model.model_endpoint);
} else {
    req.setHeader('Authorization', 'Bearer ' + api_key);
}

```

```

        req.setEndpoint('https://api.openai.com' + model.model_endpoint);
    }

    req.setMethod('POST');
    req.setBody(request_body);
    req.setHeader('Content-Type', 'application/json');
    req.setHeader('Accept', 'application/json');
    req.setTimeout(timeout_limit);

    Map<String, Object> results;

    // Why a Try here? Because even though you expect JSON, it may not be
    so and throw an error
    try {

        // If we're using a MOCK response, then create a data structure
        that looks like
        // what we'd get back from OpenAI. Note that in unit tests, we
        are not allowed
        // to make actual REST calls, so we use the MOCK response in that
        case.

        if (mock_response != null) {
            if (model.isChat) {
                results = (Map<String, Object>)
JSON.deserializeUntyped('{"choices": [{"message":{"role": "assistant",
"content":"' + mock_response + '"}]}');
            } else {
                results = (Map<String, Object>)
JSON.deserializeUntyped('{"choices": [{"text":"' + mock_response + '"}]}');
            }
        } else {
            Http http = new Http();
            HTTPResponse res = http.send(req);
            System.debug('Response Received: ' + res.getBody());
            output.put('raw_response', res.getBody());
            results = (Map<String, Object>)
JSON.deserializeUntyped(res.getBody());
        }

        // Our "long" wait for results is over!
        Long end_time = Datetime.now().getTime();

```

```

        output.put('elapsed_seconds', String.valueOf((end_time-
start_time)/1000));

        // Normally, we just return the text generated, but if you ask
nicely, we'll
        // give you everything
        if (raw_response) {
            output.put('response', results);
            return true;
        }
        List<object> choices = (List<object>) (results.get('choices'));
        if (choices == null) {
            Map<String,Object> error_obj = (Map<String,Object>)
(results.get('error'));
            if (error_obj == null) {
                u_add_error(output, 'Did not receive a response from
OpenAI');
            } else {
                u_add_error(output, 'Error from OpenAI: ' +
String.valueOf(error_obj.get('message')));
            }
            return false;
        }
        Map<String,Object> choice = ( Map<String,Object> ) (choices[0]);

        String response;

        // Depending upon which call we made, the response is at different
places
        if (model.isChat) {
            Map<String,Object> message = ( Map<String,Object> )
(choice.get('message'));
            response = String.valueOf(message.get('content'));
            // Update the chat history
            history.add(role_user);
            Map<String,String> new_history = new Map<String,String>();
            new_history.put('role', String.valueOf(message.get('role')));
            new_history.put('content',
String.valueOf(message.get('content')));
            history.add(new_history);

            // Trim away too old of history

```

```

        if (max_history != null) {
            // System.debug('Checking History length');
            Integer imax = 2 * Integer.valueOf(max_history);
            Integer overage = history.size()-imax;
            if (overage > 0) {
                for (Integer i=0;i<overage;i++) {
                    // System.debug('Remove');
                    history.remove(0);
                }
            }
        }

        output.put('history', history);

        if (text_history) {
            output.put('history_text', u_history_to_text(history));
        }

    } else {
        response = String.valueOf(choice.get('text'));
    }

    // In cases where we've instructed OpenAI to return JSON, we can go
ahead
    // and parse the JSON here (since that's hard to do directly other
places)

    if (json_response) {
        output.put('response', JSON.deserializeUntyped(response));
    } else {
        output.put('response', response);
    }

    // See if there's anything to reset
    Object additional = u_get_option_object('additional_data', options,
service_options);
    String additional_name = u_get_option('additional_name', options,
service_options);
    if (additional != null) {
        output.put(additional_name, additional);
    }

} catch (Exception e) {

```

```
        output.put('error', e.getMessage() + ' at line ' +
String.valueOf(e.getLineNumber()));
        return false;
    }

```

```
        return true;
    }

```

```
}
```

```
/*
```

"All good things must come to an end."

This quote, FWIW, is usually attributed to Geoffrey Chaucer from "Troilus and Criseyde", 1380s

Since he wrote in middle english, he actually wrote:

> But at the laste, as every thing hath ende, She took hir leve, and
[nedes wolde] wende.

Which I translated (via google and a bunch of sites) as:

> But at the last, as everything has ended, She took her faith, and
[must] go.

Seems a bit different, but middle english was before my time...

(You may delete this if you're worried about space)

```
*/
```