Prompt flow

Prompt flow is a suite of development tools designed to streamline the end-to-end development cycle of LLM-based AI applications, from ideation, prototyping, testing, evaluation to production deployment and monitoring. It makes prompt engineering much easier and enables you to build LLM apps with production quality.

With prompt flow, you will be able to:

- **Create flows** that link LLMs, prompts, Python code and other tools together in a executable workflow.
- **Debug and iterate your flows**, especially the interaction with LLMs with ease.
- **Evaluate your flows**, calculate quality and performance metrics with larger datasets.
- **Integrate the testing and evaluation into your CI/CD system** to ensure quality of your flow.
- **Deploy your flows** to the serving platform you choose or integrate into your app's code base easily.
- (Optional but highly recommended) **Collaborate with your team** by leveraging the cloud version of Prompt flow in Azure AI.

> Welcome to join us to make prompt flow better by participating discussions, opening issues, submitting PRs.

This documentation site contains guides for prompt flow sdk, cli and vscode extension users.

Develop chat flow

This is an experimental feature, and may change at any time. Learn more.

From this document, you can learn how to develop a chat flow by writing a flow yaml from scratch. You can find additional information about flow yaml schema in Flow YAML Schema.

Flow input data

The most important elements that differentiate a chat flow from a standard flow are **chat input** and **chat history**. A chat flow can have multiple inputs, but **chat history** and **chat input** are required inputs in chat flow.

- **Chat Input**: Chat input refers to the messages or queries submitted by users to the chatbot. Effectively handling chat input is crucial for a successful conversation, as it involves understanding user intentions, extracting relevant information, and triggering appropriate responses.

- **Chat History**: Chat history is the record of all interactions between the user and the chatbot, including both user inputs and AI-generated outputs. Maintaining chat history is essential for keeping track of the conversation context and ensuring the AI can generate contextually relevant responses. Chat history is a special type of chat flow input, that stores chat messages in a structured format.

  An example of chat history:

```
[
  {"inputs": {"question": "What types of container software there
are?"}, "outputs": {"answer": "There are several types of container
software available, including: Docker, Kubernetes"}},
  {"inputs": {"question": "What's the different between them?"},
"outputs": {"answer": "The main difference between the various
container software systems is their functionality and purpose. Here
are some key differences between them..."}},
]
```

You can set **is_chat_input**/**is_chat_history** to **true** to add chat_input/chat_history to the chat flow.

```
inputs:
  chat_history:
    type: list
    is_chat_history: true
    default: []
  question:
    type: string
    is_chat_input: true
    default: What is ChatGPT?
```

For more information see develop the flow using different tools.

Develop the flow using different tools

In one flow, you can consume different kinds of tools. We now support built-in tool like LLM, Python and Prompt and third-party tool like Serp API, Vector Search, etc.

For more information see develop the flow using different tools.

Chain your flow - link nodes together

Before linking nodes together, you need to define and expose an interface.

For more information see chain your flow.

Set flow output

Chat output is required output in the chat flow. It refers to the AI-generated messages that are sent to the user in response to their inputs. Generating contextually appropriate and engaging chat outputs is vital for a positive user experience.

You can set **is_chat_output** to **true** to add chat_output to the chat flow.

```
outputs:
  answer:
    type: string
    reference: ${chat.output}
    is_chat_output: true
```

Develop standard flow

This is an experimental feature, and may change at any time. Learn more.

From this document, you can learn how to develop a standard flow by writing a flow yaml from scratch. You can find additional information about flow yaml schema in Flow YAML Schema.

Flow input data

The flow input data is the data that you want to process in your flow.

:sync: CLI You can add a flow input in inputs section of flow yaml.

```
inputs:
  url:
    type: string
    default: https://www.microsoft.com/en-us/d/xbox-wireless-controller-
stellar-shift-special-edition/94fbjc7h0h6h
```

:sync: VS Code Extension When unfolding Inputs section in the authoring page, you can set and view your flow inputs, including input schema (name and type), and the input value.

flow_input

For Web Classification sample as shown the screenshot above, the flow input is an url of string type. For more input types in a python tool, please refer to Input types.

Develop the flow using different tools

In one flow, you can consume different kinds of tools. We now support built-in tool like LLM, Python and Prompt and third-party tool like Serp API, Vector Search, etc.

Add tool as your need

:sync: CLI You can add a tool node in nodes section of flow yaml. For example, yaml below shows how to add a Python tool node in the flow.

```
nodes:
- name: fetch_text_content_from_url
  type: python
  source:
    type: code
    path: fetch_text_content_from_url.py
  inputs:
    url: ${inputs.url}
```

:sync: VS Code Extension By selecting the tool card on the very top, you'll add a new tool node to flow.

add_tool

Edit tool

:sync: CLI You can edit the tool by simply opening the source file and making edits. For example, we provide a simple Python tool code below.

```python
from promptflow import tool

# The inputs section will change based on the arguments of the tool function, after you save the code
# Adding type to arguments and return value will help the system show the types properly
# Please update the function name/signature per need
@tool
def my_python_tool(input1: str) -> str:
  return 'hello ' + input1
```

We also provide an LLM tool prompt below.

```
Please summarize the following text in one paragraph. 100 words.
Do not add any information that is not in the text.
Text: {{text}}
Summary:
```

:sync: VS Code Extension When a new tool node is added to flow, it will be appended at the bottom of flatten view with a random name by default. At the top of each tool node card, there's a toolbar for adjusting the tool node. You can move it up or down, you can delete or rename it too. For a python tool node, you can edit the tool code by clicking the code file. For a LLM tool node, you can edit the tool prompt by clicking the prompt file and adjust input parameters like connection, api and etc. edit_tool

Create connection

Please refer to the Create necessary connections for details.

Chain your flow - link nodes together

Before linking nodes together, you need to define and expose an interface.

Define LLM node interface

LLM node has only one output, the completion given by LLM provider.

As for inputs, we offer a templating strategy that can help you create parametric prompts that accept different input values. Instead of fixed text, enclose your input name in `{{}}`, so it can be replaced on the fly. We use Jinja as our templating language. For example:

```
Your task is to classify a given url into one of the following types:
Movie, App, Academic, Channel, Profile, PDF or None based on the text
content information.
The classification will be based on the url, the webpage text content
summary, or both.

Here are a few examples:
{% for ex in examples %}
URL: {{ex.url}}
Text content: {{ex.text_content}}
OUTPUT:
{"category": "{{ex.category}}", "evidence": "{{ex.evidence}}"}

{% endfor %}

For a given URL : {{url}}, and text content: {{text_content}}.
Classify above url to complete the category and indicate evidence.
OUTPUT:
```

Define Python node interface

Python node might have multiple inputs and outputs. Define inputs and outputs as shown below. If you have multiple outputs, remember to make it a dictionary so that the downstream node can call each key separately. For example:

```python
import json
from promptflow import tool

@tool
def convert_to_dict(input_str: str, input_str2: str) -> dict:
    try:
        print(input_str2)
        return json.loads(input_str)
    except Exception as e:
        print("input is not valid, error: {}".format(e))
        return {"category": "None", "evidence": "None"}
```

Link nodes together

After the interface is defined, you can use:

- ${inputs.key} to link with flow input.
- ${upstream_node_name.output} to link with single-output upstream node.
- ${upstream_node_name.output.key} to link with multi-output upstream node.

Below are common scenarios for linking nodes together.

Scenario 1 - Link LLM node with flow input and single-output upstream node

After you add a new LLM node and edit the prompt file like Define LLM node interface, three inputs called `url`, `examples` and `text_content` are created in inputs section.

:sync: CLI You can link the LLM node input with flow input by `${inputs.url}`. And you can link `examples` to the upstream `prepare_examples` node and `text_content` to the `summarize_text_content` node by `${prepare_examples.output}` and `${summarize_text_content.output}`.

```
- name: classify_with_llm
  type: llm
  source:
    type: code
    path: classify_with_llm.jinja2
  inputs:
    deployment_name: text-davinci-003
    suffix: ""
    max_tokens: 128
    temperature: 0.2
    top_p: 1
    echo: false
    presence_penalty: 0
    frequency_penalty: 0
    best_of: 1
    url: ${inputs.url}    # Link with flow input
    examples: ${prepare_examples.output} # Link LLM node with single-output
upstream node
    text_content: ${summarize_text_content.output} # Link LLM node with
single-output upstream node
```

When running the flow, the `url` input of the node will be replaced by flow input on the fly, and the `examples` and `text_content` input of the node will be replaced by `prepare_examples` and `summarize_text_content` node output on the fly.

Scenario 2 - Link LLM node with multi-output upstream node

Suppose we want to link the newly created LLM node with `covert_to_dict` Python node whose output is a dictionary with two keys: `category` and `evidence`.

:sync: CLI You can link `examples` to the `evidence` output of upstream `covert_to_dict` node by `${convert_to_dict.output.evidence}` like below:

```
- name: classify_with_llm
  type: llm
  source:
    type: code
    path: classify_with_llm.jinja2
  inputs:
    deployment_name: text-davinci-003
    suffix: ""
    max_tokens: 128
    temperature: 0.2
    top_p: 1
    echo: false
    presence_penalty: 0
    frequency_penalty: 0
    best_of: 1
    text_content: ${convert_to_dict.output.evidence} # Link LLM node with
multi-output upstream node
```

When running the flow, the `text_content` input of the node will be replaced by `evidence` value from `convert_to_dict node` output dictionary on the fly.

Scenario 3 - Link Python node with upstream node/flow input

After you add a new Python node and edit the code file like Define Python node interface], two inputs called `input_str` and `input_str2` are created in inputs section. The linkage is the same as LLM node, using `${flow.input_name}` to link with flow input or `${upstream_node_name.output}` to link with upstream node.

:sync: CLI

```
- name: prepare_examples
  type: python
  source:
    type: code
    path: prepare_examples.py
  inputs:
    input_str: ${inputs.url}  # Link Python node with flow input
    input_str2: ${fetch_text_content_from_url.output} # Link Python node
with single-output upstream node
```

When running the flow, the `input_str` input of the node will be replaced by flow input on the fly and the `input_str2` input of the node will be replaced by `fetch_text_content_from_url` node output dictionary on the fly.

Set flow output

When the flow is complicated, instead of checking outputs on each node, you can set flow output and check outputs of multiple nodes in one place. Moreover, flow output helps:

- Check bulk test results in one single table.
- Define evaluation interface mapping.
- Set deployment response schema.

:sync: CLI You can add flow outputs in outputs section of flow yaml . The linkage is the same as LLM node, using `${convert_to_dict.output.category}` to link `category` flow output with with `category` value of upstream node `convert_to_dict`.

```
outputs:
  category:
    type: string
    reference: ${convert_to_dict.output.category}
  evidence:
    type: string
    reference: ${convert_to_dict.output.evidence}
```

Develop evaluation flow

This is an experimental feature, and may change at any time. Learn more.

The evaluation flow is a flow to test/evaluate the quality of your LLM application (standard/chat flow). It usually runs on the outputs of standard/chat flow, and compute key metrics that can be used to determine whether the standard/chat flow performs well. See Flows for more information.

Before proceeding with this document, it is important to have a good understanding of the standard flow. Please make sure you have read Develop standard flow, since they share many common features and these features won't be repeated in this doc, such as:

- `Inputs/Outputs definition`
- `Nodes`
- `Chain nodes in a flow`

While the evaluation flow shares similarities with the standard flow, there are some important differences that set it apart. The main distinctions are as follows:

- `Inputs from an existing run`: The evaluation flow contains inputs that are derived from the outputs of the standard/chat flow. These inputs are used for evaluation purposes.
- `Aggregation node`: The evaluation flow contains one or more aggregation nodes, where the actual evaluation takes place. These nodes are responsible for computing metrics and determining the performance of the standard/chat flow.

Evaluation flow example

In this guide, we use eval-classification-accuracy flow as an example of the evaluation flow. This is a flow illustrating how to evaluate the performance of a classification flow. It involves comparing each prediction to the groundtruth and assigns a `Correct` or `Incorrect` grade, and aggregating the results to produce metrics such as `accuracy`, which reflects how good the system is at classifying the data.

Flow inputs

The flow `eval-classification-accuracy` contains two inputs:

```yaml
inputs:
  groundtruth:
    type: string
    description: Groundtruth of the original question, it's the correct
label that you hope your standard flow could predict.
    default: APP
  prediction:
    type: string
    description: The actual predicted outputs that your flow produces.
    default: APP
```

As evident from the inputs description, the evaluation flow requires two specific inputs:

- `groundtruth`: This input represents the actual or expected values against which the performance of the standard/chat flow will be evaluated.
- `prediction`: The prediction input is derived from the outputs of another standard/chat flow. It contains the predicted values generated by the standard/chat flow, which will be compared to the groundtruth values during the evaluation process.

From the definition perspective, there is no difference compared with adding an input/output in a `standard/chat flow`. However when running an evaluation flow, you may need to specify the data source from both data file and flow run outputs. For more details please refer to Run and evaluate a flow.

Aggregation node

Before introducing the aggregation node, let's see what a regular node looks like, we use node `grade` in the example flow for instance:

```yaml
- name: grade
  type: python
  source:
    type: code
    path: grade.py
  inputs:
    groundtruth: ${inputs.groundtruth}
    prediction: ${inputs.prediction}
```

It takes both `groundtruth` and `prediction` from the flow inputs, compare them in the source code to see if they match:

```python
from promptflow import tool

@tool
def grade(groundtruth: str, prediction: str):
    return "Correct" if groundtruth.lower() == prediction.lower() else
"Incorrect"
```

When it comes to an `aggregation node`, there are two key distinctions that set it apart from a regular node:

1. It has an attribute `aggregation` set to be `true`.

```yaml
- name: calculate_accuracy
  type: python
  source:
    type: code
    path: calculate_accuracy.py
  inputs:
    grades: ${grade.output}
  aggregation: true  # Add this attribute to make it an aggregation node
```

2. Its source code accepts a `List` type parameter which is a collection of the previous regular node's outputs.

```python
from typing import List
from promptflow import log_metric, tool

@tool
def calculate_accuracy(grades: List[str]):
    result = []
    for index in range(len(grades)):
        grade = grades[index]
        result.append(grade)

    # calculate accuracy for each variant
    accuracy = round((result.count("Correct") / len(result)), 2)
    log_metric("accuracy", accuracy)

    return result
```

The parameter `grades` in above function, contains all results that are produced by the regular node `grade`. Assuming the referred standard flow run has 3 outputs:

```
{"prediction": "App"}
{"prediction": "Channel"}
{"prediction": "Academic"}
```

And we provides a data file like this:

```
{"groundtruth": "App"}
{"groundtruth": "Channel"}
{"groundtruth": "Wiki"}
```

Then the `grades` value would be `["Correct", "Correct", "Incorrect"]`, and the final accuracy is `0.67`.

This example provides a straightforward demonstration of how to evaluate the classification flow. Once you have a solid understanding of the evaluation mechanism, you can customize and design your own evaluation method to suit your specific needs.

More about the list parameter

What if the number of referred standard flow run outputs does not match the provided data file? We know that a standard flow can be executed against multiple line data and some of them could fail while others succeed. Consider the same standard flow run mentioned in above example but the `2nd` line run has failed, thus we have below run outputs:

```
{"prediction": "App"}
{"prediction": "Academic"}
```

The promptflow flow executor has the capability to recognize the index of the referred run's outputs and extract the corresponding data from the provided data file. This means that during the execution process, even if the same data file is provided(3 lines), only the specific data mentioned below will be processed:

```
{"groundtruth": "App"}
{"groundtruth": "Wiki"}
```

In this case, the `grades` value would be `["Correct", "Incorrect"]` and the accuracy is `0.5`.

How to set aggregation node in VS Code Extention

img

How to log metrics

You can only log metrics in an `aggregation node`, otherwise the metric will be ignored.

Promptflow supports logging and tracking experiments using `log_metric` function. A metric is a key-value pair that records a single float measure. In a python node, you can log a metric with below code:

```python
from typing import List
from promptflow import log_metric, tool

@tool
def example_log_metrics(grades: List[str]):
    # this node is an aggregation node so it accepts a list of grades
    metric_key = "accuracy"
    metric_value = round((grades.count("Correct") / len(result)), 2)
    log_metric(metric_key, metric_value)
```

After the run is completed, you can run `pf run show-metrics -n <run_name>` to see the metrics.

img