Evaluate your generative AI application locally with the Azure AI Evaluation SDK (preview)

> **Note**
>
> This document refers to the Microsoft Foundry (classic) portal.
>
> View the Microsoft Foundry (new) documentation to learn about the new portal.

> **Important**
>
> Items marked (preview) in this article are currently in public preview. This preview is provided without a service-level agreement, and we don't recommend it for production workloads. Certain features might not be supported or might have constrained capabilities. For more information, see Supplemental Terms of Use for Microsoft Azure Previews .

You can thoroughly assess the performance of your generative AI application by applying it to a substantial dataset. Evaluate the application in your development environment with the Azure AI Evaluation SDK.

When you provide either a test dataset or a target, your generative AI application outputs are quantitatively measured with both mathematical-based metrics and AI-assisted quality and safety evaluators. Built-in or custom evaluators can provide you with comprehensive insights into the application's capabilities and limitations.

In this article, you learn how to run evaluators on a single row of data and a larger test dataset on an application target. You use built-in evaluators that use the Azure AI Evaluation SDK locally. Then, you learn to track the results and evaluation logs in a Foundry project.

Get started

First, install the evaluators package from the Azure AI Evaluation SDK:

Python

```
pip install azure-ai-evaluation
```

> **Note**
>
> For more information, see Azure AI Evaluation client library for Python .

Built-in evaluators

Built-in quality and safety metrics accept query and response pairs, along with additional information for specific evaluators.

Expand table

Category	Evaluators
General purpose	`CoherenceEvaluator`, `FluencyEvaluator`, `QAEvaluator`
Textual similarity	`SimilarityEvaluator`, `F1ScoreEvaluator`, `BleuScoreEvaluator`, `GleuScoreEvaluator`, `RougeScoreEvaluator`, `MeteorScoreEvaluator`
Retrieval-augmented generation (RAG)	`RetrievalEvaluator`, `DocumentRetrievalEvaluator`, `GroundednessEvaluator`, `GroundednessProEvaluator`, `RelevanceEvaluator`, `ResponseCompletenessEvaluator`
Risk and safety	`ViolenceEvaluator`, `SexualEvaluator`, `SelfHarmEvaluator`, `HateUnfairnessEvaluator`, `IndirectAttackEvaluator`, `ProtectedMaterialEvaluator`, `UngroundedAttributesEvaluator`, `CodeVulnerabilityEvaluator`, `ContentSafetyEvaluator`
Agentic	`IntentResolutionEvaluator`, `ToolCallAccuracyEvaluator`, `TaskAdherenceEvaluator`
Azure OpenAI	`AzureOpenAILabelGrader`, `AzureOpenAIStringCheckGrader`, `AzureOpenAITextSimilarityGrader`, `AzureOpenAIGrader`

Data requirements for built-in evaluators

Built-in evaluators can accept query and response pairs, a list of conversations in JSON Lines (JSONL) format, or both.

Expand table

Evaluator	Conversation & single-turn support for text	Conversation & single-turn support for text and image	Single-turn support for text only	Requires `ground_truth`	Supports agent inputs
Quality Evaluators					
`IntentResolutionEvaluator`					✓
`ToolCallAccuracyEvaluator`					✓
`TaskAdherenceEvaluator`					✓
`GroundednessEvaluator`	✓				✓
`GroundednessProEvaluator`	✓				
`RetrievalEvaluator`	✓				
`DocumentRetrievalEvaluator`	✓			✓	
`RelevanceEvaluator`	✓				✓
`CoherenceEvaluator`	✓				
`FluencyEvaluator`	✓				
`ResponseCompletenessEvaluator`			✓	✓	
`QAEvaluator`			✓	✓	
Natural Language Processing (NLP) Evaluators					
`SimilarityEvaluator`			✓	✓	
`F1ScoreEvaluator`			✓	✓	
`RougeScoreEvaluator`			✓	✓	
`GleuScoreEvaluator`			✓	✓	
`BleuScoreEvaluator`			✓	✓	
`MeteorScoreEvaluator`			✓	✓	
Safety Evaluators					

Evaluator	Conversation & single-turn support for text	Conversation & single-turn support for text and image	Single-turn support for text only	Requires `ground_truth`	Supports agent inputs
`ViolenceEvaluator`		✓			
`SexualEvaluator`		✓			
`SelfHarmEvaluator`		✓			
`HateUnfairnessEvaluator`		✓			
`ProtectedMaterialEvaluator`		✓			
`ContentSafetyEvaluator`		✓			
`UngroundedAttributesEvaluator`			✓		
`CodeVulnerabilityEvaluator`			✓		
`IndirectAttackEvaluator`	✓				
Azure OpenAI Graders					
`AzureOpenAILabelGrader`	✓				
`AzureOpenAIStringCheckGrader`	✓				
`AzureOpenAITextSimilarityGrader`	✓			✓	
`AzureOpenAIGrader`	✓				

> **Note**
>
> AI-assisted quality evaluators, except for `SimilarityEvaluator`, include a reason field. They use techniques like chain-of-thought reasoning to generate an explanation for the score.
>
> They consume more token usage in generation as a result of improved evaluation quality. Specifically, `max_token` for evaluator generation is set to 800 for most AI-assisted evaluators. It has the value 1600 for `RetrievalEvaluator` and 3000 for `ToolCallAccuracyEvaluator` to accommodate longer inputs.

Azure OpenAI graders require a template that describes how their input columns are turned into the *real* input that the grader uses. For example, if you have two inputs called *query* and *response*, and a template formatted as `{{item.query}}`, only the query is used. Similarly, you could have something like `{{item.conversation}}` to accept a conversation input, but the ability of the system to handle that depends on how you configure the rest of the grader to expect that input.

For more information on data requirements for agentic evaluators, see Evaluate your AI agents.

Single-turn support for text

All built-in evaluators take single-turn inputs as query-and-response pairs in strings. For example:

Python

```python
from azure.ai.evaluation import RelevanceEvaluator

query = "What is the capital of life?"
response = "Paris."

# Initialize an evaluator:
relevance_eval = RelevanceEvaluator(model_config)
relevance_eval(query=query, response=response)
```

To run batch evaluations by using local evaluation or upload your dataset to run a cloud evaluation, represent the dataset in JSONL format. The previous single-turn data, which is a query-and-response pair, is equivalent to a line of a dataset like the following example, which shows three lines:

JSON

```json
{"query":"What is the capital of France?","response":"Paris."}
{"query":"What atoms compose water?","response":"Hydrogen and oxygen."}
{"query":"What color is my shirt?","response":"Blue."}
```

The evaluation test dataset can contain the following elements, depending on the requirements of each built-in evaluator:

- **Query**: The query sent to the generative AI application.
- **Response**: The response to the query generated by the generative AI application.
- **Context**: The source the generated response is based on. That is, the grounding documents.
- **Ground truth**: The response generated by a user or human as the true answer.

To see what each evaluator requires, see Evaluators.

Conversation support for text

For evaluators that support conversations for text, you can provide `conversation` as input. This input includes a Python dictionary with a list of `messages`, which includes `content`, `role`, and optionally `context`.

See the following two-turn conversation in Python:

Python

```
conversation = {
        "messages": [
        {
            "content": "Which tent is the most waterproof?",
            "role": "user"
        },
        {
            "content": "The Alpine Explorer Tent is the most waterproof",
            "role": "assistant",
            "context": "From the our product list the alpine explorer tent is the most
waterproof. The Adventure Dining Table has higher weight."
        },
        {
            "content": "How much does it cost?",
            "role": "user"
        },
        {
            "content": "The Alpine Explorer Tent is $120.",
            "role": "assistant",
            "context": None
        }
        ]
}
```

To run batch evaluations by using local evaluation or upload your dataset to run cloud evaluation, you need to represent the dataset in JSONL format. The previous conversation is equivalent to a line of dataset in a JSONL file like the following example:

JSON

```
{"conversation":
    {
        "messages": [
        {
```

```
            "content": "Which tent is the most waterproof?",
            "role": "user"
        },
        {
            "content": "The Alpine Explorer Tent is the most waterproof",
            "role": "assistant",
            "context": "From the our product list the alpine explorer tent is the most
waterproof. The Adventure Dining Table has higher weight."
        },
        {
            "content": "How much does it cost?",
            "role": "user"
        },
        {
            "content": "The Alpine Explorer Tent is $120.",
            "role": "assistant",
            "context": null
        }
        ]
    }
}
```

Our evaluators understand that the first turn of the conversation provides valid `query` from `user`, `context` from `assistant`, and `response` from `assistant` in the query-response format. Conversations are then evaluated per turn and results are aggregated over all turns for a conversation score.

> **Note**
>
> In the second turn, even if `context` is `null` or a missing key, the evaluator interprets the turn as an empty string instead of failing with an error, which might lead to misleading results.
>
> We strongly recommend that you validate your evaluation data to comply with the data requirements.

For conversation mode, here's an example for `GroundednessEvaluator`:

Python

```python
# Conversation mode:
import json
import os
from azure.ai.evaluation import GroundednessEvaluator, AzureOpenAIModelConfiguration

model_config = AzureOpenAIModelConfiguration(
```

```
    azure_endpoint=os.environ.get("AZURE_ENDPOINT"),
    api_key=os.environ.get("AZURE_API_KEY"),
    azure_deployment=os.environ.get("AZURE_DEPLOYMENT_NAME"),
    api_version=os.environ.get("AZURE_API_VERSION"),
)

# Initialize the Groundedness evaluator:
groundedness_eval = GroundednessEvaluator(model_config)

conversation = {
    "messages": [
        { "content": "Which tent is the most waterproof?", "role": "user" },
        { "content": "The Alpine Explorer Tent is the most waterproof", "role": "as-
sistant", "context": "From the our product list the alpine explorer tent is the most
waterproof. The Adventure Dining Table has higher weight." },
        { "content": "How much does it cost?", "role": "user" },
        { "content": "$120.", "role": "assistant", "context": "The Alpine Explorer Tent
is $120."}
    ]
}

# Alternatively, you can load the same content from a JSONL file.
groundedness_conv_score = groundedness_eval(conversation=conversation)
print(json.dumps(groundedness_conv_score, indent=4))
```

For conversation outputs, per-turn results are stored in a list and the overall conversation score `'groundedness': 4.0` is averaged over the turns:

Python

```
{
    "groundedness": 5.0,
    "gpt_groundedness": 5.0,
    "groundedness_threshold": 3.0,
    "evaluation_per_turn": {
        "groundedness": [
            5.0,
            5.0
        ],
        "gpt_groundedness": [
            5.0,
            5.0
        ],
        "groundedness_reason": [
            "The response accurately and completely answers the query by stating that
the Alpine Explorer Tent is the most waterproof, which is directly supported by the
context. There are no irrelevant details or incorrect information present.",
            "The RESPONSE directly answers the QUERY with the exact information pro-
vided in the CONTEXT, making it fully correct and complete."
        ],
```

```
        "groundedness_result": [
            "pass",
            "pass"
        ],
        "groundedness_threshold": [
            3,
            3
        ]
    }
}
```

> **Note**
>
> To support more evaluator models, use the key without prefixes. For example, use `groundedness.groundedness`.

Conversation support for images and multimodal text and image

For evaluators that support conversations for image and multimodal image and text, you can pass in image URLs or Base64-encoded images in `conversation`.

Supported scenarios include:

- Multiple images with text input to image or text generation.
- Text-only input to image generations.
- Image-only input to text generation.

Python

```python
from pathlib import Path
from azure.ai.evaluation import ContentSafetyEvaluator
import base64

# Create an instance of an evaluator with image and multi-modal support.
safety_evaluator = ContentSafetyEvaluator(credential=azure_cred,
azure_ai_project=project_scope)

# Example of a conversation with an image URL:
conversation_image_url = {
    "messages": [
        {
            "role": "system",
            "content": [
                {"type": "text", "text": "You are an AI assistant that understands im-
ages."}
```

```
            ],
        },
        {
            "role": "user",
            "content": [
                {"type": "text", "text": "Can you describe this image?"},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": "https://cdn.britannica.com/68/178268-050-5B4E7FB6/Tom-
Cruise-2013.jpg"
                    },
                },
            ],
        },
        {
            "role": "assistant",
            "content": [
                {
                    "type": "text",
                    "text": "The image shows a man with short brown hair smiling, wear-
ing a dark-colored shirt.",
                }
            ],
        },
    ]
}

# Example of a conversation with base64 encoded images:
base64_image = ""

with Path.open("Image1.jpg", "rb") as image_file:
    base64_image = base64.b64encode(image_file.read()).decode("utf-8")

conversation_base64 = {
    "messages": [
        {"content": "create an image of a branded apple", "role": "user"},
        {
            "content": [{"type": "image_url", "image_url": {"url":
f"{base64_image}"}}],
            "role": "assistant",
        },
    ]
}

# Run the evaluation on the conversation to output the result.
safety_score = safety_evaluator(conversation=conversation_image_url)
```

Currently, the image and multimodal evaluators support:

- Single turn only: a conversation can have only one user message and one assistant message.
- Conversations that have only one system message.
- Conversation payloads that are smaller than 10 MB, including images.
- Absolute URLs and Base64-encoded images.
- Multiple images in a single turn.
- JPG/JPEG, PNG, and GIF file formats.

Set up

For AI-assisted quality evaluators, except for `GroundednessProEvaluator` preview, you must specify a GPT model (`gpt-35-turbo`, `gpt-4`, `gpt-4-turbo`, `gpt-4o`, or `gpt-4o-mini`) in your `model_config`. The GPT model acts as a judge to score the evaluation data. We support both Azure OpenAI or OpenAI model configuration schemas. For the best performance and parseable responses with our evaluators, we recommend using GPT models that aren't in preview.

> **Note**
>
> Replace `gpt-3.5-turbo` with `gpt-4o-mini` for your evaluator model. According to **OpenAI**, `gpt-4o-mini` is cheaper, more capable, and as fast.
>
> To make inference calls with the API key, make sure you have at least the `Cognitive Services OpenAI User` role for the Azure OpenAI resource. For more information about permissions, see **Permissions for an Azure OpenAI resource**.

For all risk and safety evaluators and `GroundednessProEvaluator` (preview), instead of a GPT deployment in `model_config`, you must provide your `azure_ai_project` information. This accesses the back end evaluation service by using your Foundry project.

Prompts for AI-assisted built-in evaluators

For transparency, we open-source the prompts of our quality evaluators in our Evaluator Library and the Azure AI Evaluation Python SDK repository, except for the Safety Evaluators and `GroundednessProEvaluator`, powered by Azure AI Content Safety. These prompts serve as instructions for a language model to perform their evaluation task, which requires a human-friendly definition of the metric and its associated scoring rubrics. We highly recommend that you customize the definitions and grading rubrics to your scenario specifics. For more information, see Custom evaluators.

Composite evaluators

Composite evaluators are built-in evaluators that combine individual quality or safety metrics. They provide a wide range of metrics right out of the box for both query response pairs or chat messages.

Expand table

Composite evaluator	Contains	Description
`QAEvaluator`	`GroundednessEvaluator`, `RelevanceEvaluator`, `CoherenceEvaluator`, `FluencyEvaluator`, `SimilarityEvaluator`, `F1ScoreEvaluator`	Combines all the quality evaluators for a single output of combined metrics for query and response pairs
`ContentSafetyEvaluator`	`ViolenceEvaluator`, `SexualEvaluator`, `SelfHarmEvaluator`, `HateUnfairnessEvaluator`	Combines all the safety evaluators for a single output of combined metrics for query and response pairs

Local evaluation on test datasets using `evaluate()`

After you spot-check your built-in or custom evaluators on a single row of data, you can combine multiple evaluators with the `evaluate()` API on an entire test dataset.

Prerequisite set up steps for Microsoft Foundry projects

If this session is your first time running evaluations and logging it to your Foundry project, you might need to do the following setup steps:

1. Create and connect your storage account to your Foundry project at the resource level. This bicep template provisions and connects a storage account to your Foundry project with key authentication.
2. Make sure the connected storage account has access to all projects.
3. If you connected your storage account with Microsoft Entra ID, make sure to give Microsoft Identity permissions for **Storage Blob Data Owner** to both your account and Foundry project resource in Azure portal.

Evaluate on a dataset and log results to Foundry

To ensure the `evaluate()` API can correctly parse the data, you must specify column mapping to map the column from the dataset to key words that the evaluators accept. This example specifies the data mapping for `query`, `response`, and `context`.

Python

```
from azure.ai.evaluation import evaluate

result = evaluate(
    data="data.jsonl", # Provide your data here:
    evaluators={
        "groundedness": groundedness_eval,
        "answer_length": answer_length
    },
    # Column mapping:
    evaluator_config={
        "groundedness": {
            "column_mapping": {
                "query": "${data.queries}",
                "context": "${data.context}",
                "response": "${data.response}"
            }
        }
    },
    # Optionally, provide your Foundry project information to track your evaluation re-
sults in your project portal.
    azure_ai_project = azure_ai_project,
    # Optionally, provide an output path to dump a JSON file of metric summary, row-
level data, and the metric and Foundry project URL.
    output_path="./myevalresults.json"
)
```

Tip

Get the contents of the `result.studio_url` property for a link to view your logged evaluation results in your Foundry project.

The evaluator outputs results in a dictionary, which contains aggregate `metrics` and row-level data and metrics. See the following example output:

Python

```
{'metrics': {'answer_length.value': 49.333333333333336,
             'groundedness.gpt_groundeness': 5.0, 'groundedness.groundeness': 5.0},
 'rows': [{'inputs.response': 'Paris is the capital of France.',
```

```
           'inputs.context': 'Paris has been the capital of France since '
                             'the 10th century and is known for its '
                             'cultural and historical landmarks.',
           'inputs.query': 'What is the capital of France?',
           'outputs.answer_length.value': 31,
           'outputs.groundeness.groundeness': 5,
           'outputs.groundeness.gpt_groundeness': 5,
           'outputs.groundeness.groundeness_reason': 'The response to the query is sup-
ported by the context.'},
          {'inputs.response': 'Albert Einstein developed the theory of '
                              'relativity.',
           'inputs.context': 'Albert Einstein developed the theory of '
                             'relativity, with his special relativity '
                             'published in 1905 and general relativity in '
                             '1915.',
           'inputs.query': 'Who developed the theory of relativity?',
           'outputs.answer_length.value': 51,
           'outputs.groundeness.groundeness': 5,
           'outputs.groundeness.gpt_groundeness': 5,
           'outputs.groundeness.groundeness_reason': 'The response to the query is sup-
ported by the context.'},
          {'inputs.response': 'The speed of light is approximately 299,792,458 '
                              'meters per second.',
           'inputs.context': 'The exact speed of light in a vacuum is '
                             '299,792,458 meters per second, a constant '
                             "used in physics to represent 'c'.",
           'inputs.query': 'What is the speed of light?',
           'outputs.answer_length.value': 66,
           'outputs.groundeness.groundeness': 5,
           'outputs.groundeness.gpt_groundeness': 5,
           'outputs.groundeness.groundeness_reason': 'The response to the query is sup-
ported by the context.'}],
 'traces': {}}
```

Requirements for `evaluate()`

The `evaluate()` API requires a specific data format and evaluator parameter key names to display the evaluation results charts in your Foundry project correctly.

Data format

The `evaluate()` API accepts only data in JSONL format. For all built-in evaluators, `evaluate()` requires data in the following format with the required input fields. See the previous section on the required data input for built-in evaluators. The following code snippet is a sample of what one line can look like:

JSON

```
{
  "query":"What is the capital of France?",
  "context":"France is in Europe",
  "response":"Paris is the capital of France.",
  "ground_truth": "Paris"
}
```

Evaluator parameter format

When you pass in your built-in evaluators, specify the right keyword mapping in the `evaluators` parameter list. The following table shows the keyword mapping required for the results from your built-in evaluators to show up in the UI when logged to your Foundry project.

Expand table

Evaluator	Keyword parameter
`GroundednessEvaluator`	`"groundedness"`
`GroundednessProEvaluator`	`"groundedness_pro"`
`RetrievalEvaluator`	`"retrieval"`
`RelevanceEvaluator`	`"relevance"`
`CoherenceEvaluator`	`"coherence"`
`FluencyEvaluator`	`"fluency"`
`SimilarityEvaluator`	`"similarity"`
`F1ScoreEvaluator`	`"f1_score"`
`RougeScoreEvaluator`	`"rouge"`
`GleuScoreEvaluator`	`"gleu"`
`BleuScoreEvaluator`	`"bleu"`
`MeteorScoreEvaluator`	`"meteor"`
`ViolenceEvaluator`	`"violence"`
`SexualEvaluator`	`"sexual"`

Evaluator	Keyword parameter
`SelfHarmEvaluator`	`"self_harm"`
`HateUnfairnessEvaluator`	`"hate_unfairness"`
`IndirectAttackEvaluator`	`"indirect_attack"`
`ProtectedMaterialEvaluator`	`"protected_material"`
`CodeVulnerabilityEvaluator`	`"code_vulnerability"`
`UngroundedAttributesEvaluator`	`"ungrounded_attributes"`
`QAEvaluator`	`"qa"`
`ContentSafetyEvaluator`	`"content_safety"`

Here's an example of how to set the `evaluators` parameters:

Python

```
result = evaluate(
    data="data.jsonl",
    evaluators={
        "sexual":sexual_evaluator,
        "self_harm":self_harm_evaluator,
        "hate_unfairness":hate_unfairness_evaluator,
        "violence":violence_evaluator
    }
)
```

Local evaluation on a target

If you have a list of queries that you want to run and then evaluate, the `evaluate()` API also supports a `target` parameter. This parameter sends queries to an application to collect answers, and then runs your evaluators on the resulting query and response.

A target can be any callable class in your directory. In this example, there's a Python script `askwiki.py` with a callable class `askwiki()` that is set as the target. If you have a dataset of queries that you can send into the simple `askwiki` app, you can evaluate the groundedness of the outputs. Make sure that you specify the proper column mapping for your data in `"column_mapping"`. You can use `"default"` to specify column mapping for all evaluators.

Here's the content in `"data.jsonl"`:

JSON

```
{"query":"When was United States found ?", "response":"1776"}
{"query":"What is the capital of France?", "response":"Paris"}
{"query":"Who is the best tennis player of all time ?", "response":"Roger Federer"}
```

Python

```
from askwiki import askwiki

result = evaluate(
    data="data.jsonl",
    target=askwiki,
    evaluators={
        "groundedness": groundedness_eval
    },
    evaluator_config={
        "default": {
            "column_mapping": {
                "query": "${data.queries}",
                "context": "${outputs.context}",
                "response": "${outputs.response}"
            }
        }
    }
)
```

Related content

- Azure AI Evaluation client library for Python
- Troubleshoot AI Evaluation SDK Issues
- Observability in generative AI
- Run evaluations in the cloud by using the Microsoft Foundry SDK
- Generate synthetic and simulated data for evaluation
- See evaluation results in the Foundry portal
- Get started with Foundry
- Get started with evaluation samples

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Last updated on 11/25/2025