Why Evaluate Agents - Agent Development Kit

Why Evaluate Agents¶

Currently supported in Python

In traditional software development, unit tests and integration tests provide confidence that code functions as expected and remains stable through changes. These tests provide a clear "pass/fail" signal, guiding further development. However, LLM agents introduce a level of variability that makes traditional testing approaches insufficient.

Due to the probabilistic nature of models, deterministic "pass/fail" assertions are often unsuitable for evaluating agent performance. Instead, we need qualitative evaluations of both the final output and the agent's trajectory - the sequence of steps taken to reach the solution. This involves assessing the quality of the agent's decisions, its reasoning process, and the final result.

This may seem like a lot of extra work to set up, but the investment of automating evaluations pays off quickly. If you intend to progress beyond prototype, this is a highly recommended best practice.

intro_components.png

Preparing for Agent Evaluations¶

Before automating agent evaluations, define clear objectives and success criteria:

- **Define Success:** What constitutes a successful outcome for your agent?
- **Identify Critical Tasks:** What are the essential tasks your agent must accomplish?
- **Choose Relevant Metrics:** What metrics will you track to measure performance?

These considerations will guide the creation of evaluation scenarios and enable effective monitoring of agent behavior in real-world deployments.

What to Evaluate?¶

To bridge the gap between a proof-of-concept and a production-ready AI agent, a robust and automated evaluation framework is essential. Unlike evaluating generative models, where the focus is primarily on the final output, agent evaluation requires a deeper understanding of the decision-making process. Agent evaluation can be broken down into two components:

1. **Evaluating Trajectory and Tool Use:** Analyzing the steps an agent takes to reach a solution, including its choice of tools, strategies, and the efficiency of its approach.
2. **Evaluating the Final Response:** Assessing the quality, relevance, and correctness of the agent's final output.

The trajectory is just a list of steps the agent took before it returned to the user. We can compare that against the list of steps we expect the agent to have taken.

Evaluating trajectory and tool use¶

Before responding to a user, an agent typically performs a series of actions, which we refer to as a 'trajectory.' It might compare the user input with session history to disambiguate a term, or lookup a policy document, search a knowledge base or invoke an API to save a ticket. We call this a 'trajectory' of actions. Evaluating an agent's performance requires comparing its actual trajectory to an expected, or ideal, one. This comparison can reveal errors and inefficiencies in the agent's process. The expected trajectory represents the ground truth -- the list of steps we anticipate the agent should take.

For example:

```
 // Trajectory evaluation will compare
 expected_steps = ["determine_intent", "use_tool", "review_results", "r
 actual_steps = ["determine_intent", "use_tool", "review_results", "rep
```

Several ground-truth-based trajectory evaluations exist:

1. **Exact match:** Requires a perfect match to the ideal trajectory.
2. **In-order match:** Requires the correct actions in the correct order, allows for extra actions.
3. **Any-order match:** Requires the correct actions in any order, allows for extra actions.
4. **Precision:** Measures the relevance/correctness of predicted actions.
5. **Recall:** Measures how many essential actions are captured in the prediction.
6. **Single-tool use:** Checks for the inclusion of a specific action.

Choosing the right evaluation metric depends on the specific requirements and goals of your agent. For instance, in high-stakes scenarios, an exact match might be crucial, while in more flexible situations, an in-order or any-order match might suffice.

How Evaluation works with the ADK¶

The ADK offers two methods for evaluating agent performance against predefined datasets and evaluation criteria. While conceptually similar, they differ in the amount of data they can process, which typically dictates the appropriate use case for each.

First approach: Using a test file¶

This approach involves creating individual test files, each representing a single, simple agent-model interaction (a session). It's most effective during active agent development, serving as a form of unit testing. These tests are designed for rapid execution and should focus on simple session complexity. Each test file contains a single session, which may consist of multiple turns. A turn represents a single interaction between the user and the agent. Each turn includes

- `User Content`: The user issued query.
- `Expected Intermediate Tool Use Trajectory`: The tool calls we expect the agent to make in order to respond correctly to the user query.

- `Expected Intermediate Agent Responses` : These are the natural language responses that the agent (or sub-agents) generates as it moves towards generating a final answer. These natural language responses are usually an artifact of an multi-agent system, where your root agent depends on sub-agents to achieve a goal. These intermediate responses, may or may not be of interest to the end user, but for a developer/owner of the system, are of critical importance, as they give you the confidence that the agent went through the right path to generate final response.
- `Final Response` : The expected final response from the agent.

You can give the file any name for example `evaluation.test.json` .The framework only checks for the `.test.json` suffix, and the preceding part of the filename is not constrained. Here is a test file with a few examples:

NOTE: The test files are now backed by a formal Pydantic data model. The two key schema files are [Eval Set](#) and [Eval Case](#)

```
# Do note that some fields are removed for sake of making this doc rea
{
  "eval_set_id": "home_automation_agent_light_on_off_set",
  "name": "",
  "description": "This is an eval set that is used for unit testing `x
  "eval_cases": [
    {
      "eval_id": "eval_case_id",
      "conversation": [
        {
          "invocation_id": "b7982664-0ab6-47cc-ab13-326656afdf75", # U
          "user_content": { # Content provided by the user in this inv
            "parts": [
              {
                "text": "Turn off device_2 in the Bedroom."
              }
            ],
            "role": "user"
          },
```

```
        "final_response": { # Final response from the agent that act
          "parts": [
            {
              "text": "I have set the device_2 status to off."
            }
          ],
          "role": "model"
        },
        "intermediate_data": {
          "tool_uses": [ # Tool use trajectory in chronological orde
            {
              "args": {
                "location": "Bedroom",
                "device_id": "device_2",
                "status": "OFF"
              },
              "name": "set_device_info"
            }
          ],
          "intermediate_responses": [] # Any intermediate sub-agent
        },
      }
    ],
    "session_input": { # Initial session input.
      "app_name": "home_automation_agent",
      "user_id": "test_user",
      "state": {}
    },
  }
],
}
```

Test files can be organized into folders. Optionally, a folder can also include a
`test_config.json` file that specifies the evaluation criteria.

How to migrate test files not backed by the Pydantic schema?¶

NOTE: If your test files don't adhere to [EvalSet](#) schema file, then this section is relevant to you.

Please use `AgentEvaluator.migrate_eval_data_to_new_schema` to migrate your existing `*.test.json` files to the Pydanctic backed schema.

The utility takes your current test data file and an optional initial session file, and generates a single output json file with data serialized in the new format. Given that the new schema is more cohesive, both the old test data file and initial session file can be ignored (or removed.)

Second approach: Using An Evalset File¶

The evalset approach utilizes a dedicated dataset called an "evalset" for evaluating agent-model interactions. Similar to a test file, the evalset contains example interactions. However, an evalset can contain multiple, potentially lengthy sessions, making it ideal for simulating complex, multi-turn conversations. Due to its ability to represent complex sessions, the evalset is well-suited for integration tests. These tests are typically run less frequently than unit tests due to their more extensive nature.

An evalset file contains multiple "evals," each representing a distinct session. Each eval consists of one or more "turns," which include the user query, expected tool use, expected intermediate agent responses, and a reference response. These fields have the same meaning as they do in the test file approach. Each eval is identified by a unique name. Furthermore, each eval includes an associated initial session state.

Creating evalsets manually can be complex, therefore UI tools are provided to help capture relevant sessions and easily convert them into evals within your evalset. Learn more about using the web UI for evaluation below. Here is an example evalset containing two sessions.

NOTE: The eval set files are now backed by a formal Pydantic data model. The two key schema files are [Eval Set](#) and [Eval Case](#)

```
 # Do note that some fields are removed for sake of making this doc re
 {
```

```json
    "eval_set_id": "eval_set_example_with_multiple_sessions",
    "name": "Eval set with multiple sessions",
    "description": "This eval set is an example that shows that an eval
    "eval_cases": [
      {
        "eval_id": "session_01",
        "conversation": [
          {
            "invocation_id": "e-0067f6c4-ac27-4f24-81d7-3ab994c28768",
            "user_content": {
              "parts": [
                {
                  "text": "What can you do?"
                }
              ],
              "role": "user"
            },
            "final_response": {
              "parts": [
                {

                  "text": "I can roll dice of different sizes and check
                }
              ],
              "role": null
            },
            "intermediate_data": {
              "tool_uses": [],
              "intermediate_responses": []
            },
          },
        ],
        "session_input": {
          "app_name": "hello_world",
          "user_id": "user",
          "state": {}
```

```json
        },
      },
      {
        "eval_id": "session_02",
        "conversation": [
          {
            "invocation_id": "e-92d34c6d-0a1b-452a-ba90-33af2838647a",
            "user_content": {
              "parts": [
                {
                  "text": "Roll a 19 sided dice"
                }
              ],
              "role": "user"
            },
            "final_response": {
              "parts": [
                {
                  "text": "I rolled a 17."
                }
              ],
              "role": null
            },
            "intermediate_data": {
              "tool_uses": [],
              "intermediate_responses": []
            },
          },
          {
            "invocation_id": "e-bf8549a1-2a61-4ecc-a4ee-4efbbf25a8ea",
            "user_content": {
              "parts": [
                {
                  "text": "Roll a 10 sided dice twice and then check if
                }
              ],
```

```
            "role": "user"
          },
          "final_response": {
            "parts": [
              {
                "text": I got 4 and 7 form the dice roll, and 9 is not
              }
            ],
            "role": null
          },
          "intermediate_data": {
            "tool_uses": [
              {
                "id": "adk-1a3f5a01-1782-4530-949f-07cf53fc6f05",
                "args": {
                  "sides": 10
                },
                "name": "roll_die"
              },
              {
                "id": "adk-52fc3269-caaf-41c3-833d-511e454c7058",
                "args": {
                  "sides": 10
                },
                "name": "roll_die"
              },
              {
                "id": "adk-5274768e-9ec5-4915-b6cf-f5d7f0387056",
                "args": {
                  "nums": [
                    9
                  ]
                },
                "name": "check_prime"
              }
            ],
```

```
              "intermediate_responses": [
                [
                  "data_processing_agent",
                  [
                    {
                      "text": "I have rolled a 10 sided die twice. The f
                    }
                  ]
                ]
              ]
            },
          }
        ],
        "session_input": {
          "app_name": "hello_world",
          "user_id": "user",
          "state": {}
        },
      }
    ],
  }
```

How to migrate eval set files not backed by the Pydantic schema?¶

NOTE: If your eval set files don't adhere to [EvalSet](#) schema file, then this section is relevant to you.

Based on who is maintaining the eval set data, there are two routes:

1. **Eval set data maintained by ADK UI** If you use ADK UI to maintain your Eval set data then *no action is needed* from you.
2. **Eval set data is developed and maintained manually and used in ADK eval Cli** A migration tool is in the works, until then the ADK eval cli command will continue to support data in the old format.

Evaluation Criteria¶

The evaluation criteria define how the agent's performance is measured against the evalset. The following metrics are supported:

- `tool_trajectory_avg_score`: This metric compares the agent's actual tool usage during the evaluation against the expected tool usage defined in the `expected_tool_use` field. Each matching tool usage step receives a score of 1, while a mismatch receives a score of 0. The final score is the average of these matches, representing the accuracy of the tool usage trajectory.
- `response_match_score`: This metric compares the agent's final natural language response to the expected final response, stored in the `reference` field. We use the ROUGE metric to calculate the similarity between the two responses.

If no evaluation criteria are provided, the following default configuration is used:

- `tool_trajectory_avg_score`: Defaults to 1.0, requiring a 100% match in the tool usage trajectory.
- `response_match_score`: Defaults to 0.8, allowing for a small margin of error in the agent's natural language responses.

Here is an example of a `test_config.json` file specifying custom evaluation criteria:

```
{
  "criteria": {
    "tool_trajectory_avg_score": 1.0,
    "response_match_score": 0.8
  }
}
```

How to run Evaluation with the ADK¶

As a developer, you can evaluate your agents using the ADK in the following ways:

1. **Web-based UI (** `adk web` **):** Evaluate agents interactively through a web-based interface.
2. **Programmatically (** `pytest` **):** Integrate evaluation into your testing pipeline using `pytest` and test files.
3. **Command Line Interface (** `adk eval` **):** Run evaluations on an existing evaluation set file directly from the command line.

1. `adk web` - Run Evaluations via the Web UI¶

The web UI provides an interactive way to evaluate agents and generate evaluation datasets.

Steps to run evaluation via the web ui:

1. Start the web server by running: `bash adk web samples_for_testing`
2. In the web interface:
3. Select an agent (e.g., `hello_world`).
4. Interact with the agent to create a session that you want to save as a test case.
5. Click the **"Eval tab"** on the right side of the interface.
6. If you already have an existing eval set, select that or create a new one by clicking on **"Create new eval set"** button. Give your eval set a contextual name. Select the newly created evaluation set.
7. Click **"Add current session"** to save the current session as an eval in the eval set file. You will be asked to provide a name for this eval, again give it a contextual name.
8. Once created, the newly created eval will show up in the list of available evals in the eval set file. You can run all or select specific ones to run the eval.
9. The status of each eval will be shown in the UI.

2. `pytest` - Run Tests Programmatically¶

You can also use `pytest` to run test files as part of your integration tests.

Example Command¶

```
pytest tests/integration/
```

Example Test Code¶

Here is an example of a `pytest` test case that runs a single test file:

```
from google.adk.evaluation.agent_evaluator import AgentEvaluator

def test_with_single_test_file():
    """Test the agent's basic ability via a session file."""
    AgentEvaluator.evaluate(
        agent_module="home_automation_agent",
        eval_dataset_file_path_or_dir="tests/integration/fixture/home_
    )
```

This approach allows you to integrate agent evaluations into your CI/CD pipelines or larger test suites. If you want to specify the initial session state for your tests, you can do that by storing the session details in a file and passing that to `AgentEvaluator.evaluate` method.

3. `adk eval` - Run Evaluations via the cli¶

You can also run evaluation of an eval set file through the command line interface (CLI). This runs the same evaluation that runs on the UI, but it helps with automation, i.e. you can add this command as a part of your regular build generation and verification process.

Here is the command:

```
adk eval \
    <AGENT_MODULE_FILE_PATH> \
    <EVAL_SET_FILE_PATH> \
    [--config_file_path=<PATH_TO_TEST_JSON_CONFIG_FILE>] \
    [--print_detailed_results]
```

For example:

```
adk eval \
    samples_for_testing/hello_world \
    samples_for_testing/hello_world/hello_world_eval_set_001.evalset.j
```

Here are the details for each command line argument:

- `AGENT_MODULE_FILE_PATH`: The path to the `__init__.py` file that contains a module by the name "agent". "agent" module contains a `root_agent`.
- `EVAL_SET_FILE_PATH`: The path to evaluations file(s). You can specify one or more eval set file paths. For each file, all evals will be run by default. If you want to run only specific evals from a eval set, first create a comma separated list of eval names and then add that as a suffix to the eval set file name, demarcated by a colon `:`.
- For example:
  `sample_eval_set_file.json:eval_1,eval_2,eval_3`
  `This will only run eval_1, eval_2 and eval_3 from`
  `sample_eval_set_file.json`
- `CONFIG_FILE_PATH`: The path to the config file.
- `PRINT_DETAILED_RESULTS`: Prints detailed results on the console.