Google Agent Development Kit (ADK): Evaluation, Guides, and API

Evaluation of Agents

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, qualitative evaluations of both the final output and the agent's trajectory (the sequence of steps taken to reach the solution) are needed. This involves assessing the quality of the agent's decisions, its reasoning process, and the final result.

Preparing for Agent Evaluations

Before automating agent evaluations, it's important to define clear objectives and success criteria:

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

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

What to Evaluate?

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 the 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 is referred to as a 'trajectory.' It might compare the user input with session history to disambiguate a term, look up a policy document, search a knowledge base, or invoke an API. Evaluating an agent's performance requires comparing its actual trajectory to an expected, or ideal, one.

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:

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:

- `query` : The user query.
- `expected_tool_use` : The tool call(s) that we expect the agent to make to respond correctly to the user query.
- `expected_intermediate_agent_responses` : Natural language responses produced by the agent as it progresses towards a final answer.

- `reference`: The expected final response from the model.

Example test file format:

```
[
 {
  "query": "hi",
  "expected_tool_use": [],
  "expected_intermediate_agent_responses": [],
  "reference": "Hello! What can I do for you?\n"
 },
 {
  "query": "roll a die for me",
  "expected_tool_use": [
    {
      "tool_name": "roll_die",
      "tool_input": {
        "sides": 6
      }
    }
  ],
  "expected_intermediate_agent_responses": []
 }
]
```

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

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.

Evaluation Criteria

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

1. **`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.

2. **`response_match_score`** : This metric compares the agent's final natural language response to the expected final response, stored in the `reference` field. The ROUGE metric is used 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.

Example of a custom evaluation criteria configuration:

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

How to Run Evaluation with the ADK

The ADK provides three methods for running evaluations:

1. adk web - Run Evaluations via the Web UI

The ADK web interface provides a user-friendly way to run and visualize evaluations. This approach is particularly useful for developers who prefer a graphical interface for monitoring and analyzing agent performance.

2. pytest - Run Tests Programmatically

For developers who prefer a programmatic approach, the ADK supports running evaluations using pytest. This method integrates well with existing CI/CD pipelines and allows for automated testing.

Example command:

```
python -m pytest path/to/test_file.py -v
```

Example test code:

```
 from google.adk.evaluation import run_test_file

def test_agent():
    result = run_test_file(
        agent_factory=lambda: my_agent,
        test_file_path="path/to/test_file.json"
    )
    assert result.passed
```

3. adk eval - Run Evaluations via the CLI

The ADK command-line interface provides a convenient way to run evaluations from the terminal. This approach is particularly useful for quick evaluations during development.

Guides

Contributing Guide

The ADK project welcomes contributions to both the core Python framework and its documentation. The contributing guide provides information on how to get involved.

Repositories

1. **google/adk-python**: Contains the core Python library source code.
2. **google/adk-docs**: Contains the source for the documentation site.

Before You Begin

Before contributing, you need to:

1. **Sign the Contributor License Agreement (CLA)**: Contributions must be accompanied by a CLA. This gives Google permission to use and redistribute your contributions as part of the project.

2. **Review Community Guidelines**: Familiarize yourself with the project's community guidelines to ensure your contributions align with the project's goals and standards.

How to Contribute

There are several ways to contribute to the ADK project:

1. **Reporting Issues (Bugs & Errors)**: Help improve the project by reporting bugs and errors you encounter.

2. **Suggesting Enhancements**: Propose new features or improvements to existing functionality.

3. **Improving Documentation**: Help make the documentation more comprehensive, clear, and accurate.

4. **Writing Code**: Contribute code improvements, bug fixes, or new features.

Code Reviews

All submissions, including submissions by project members, require review. The project uses GitHub pull requests for this purpose.

API Reference

The ADK provides a comprehensive API for building, deploying, and evaluating agents. The API is organized into several modules:

Key Modules

1. **google.adk.agents**: Contains the core agent classes, including `BaseAgent` and `LlmAgent`.

2. **google.adk.artifacts**: Provides functionality for managing binary data or large text content.

3. **google.adk.code_executors**: Contains classes for executing code within agents.

4. **google.adk.evaluation**: Provides tools for evaluating agent performance.

5. **google.adk.events**: Contains classes for representing events in the agent execution lifecycle.

6. **google.adk.memory**: Provides functionality for managing long-term knowledge.

7. **google.adk.models**: Contains classes for interacting with language models.

8. **google.adk.planners**: Provides functionality for planning agent actions.

9. **google.adk.runners**: Contains the `Runner` class for orchestrating agent execution.

10. **google.adk.sessions**: Provides functionality for managing conversation sessions.

11. **google.adk.tools**: Contains various tool implementations for agents.

Key Classes

BaseAgent

The `BaseAgent` class is the foundation for all agents in the ADK. It provides the basic structure and functionality that all agents share.

Key properties and methods: - `name` : The name of the agent. - `description` : A description of the agent's purpose and capabilities. - `parent_agent` : The parent agent, if this agent is a sub-agent. - `sub_agents` : A list of sub-agents. - `run_async()` : Runs the agent asynchronously. - `run_live()` : Runs the agent with live updates. - `find_agent()` : Finds an agent by name. - `find_sub_agent()` : Finds a sub-agent by name.

LlmAgent

The `LlmAgent` class extends `BaseAgent` to provide functionality specific to language model-based agents.

Key properties and methods: - `model` : The language model to use. - `tools` : The tools available to the agent. - `instruction` : The instruction to give to the model. - `global_instruction` : A global instruction that applies to all interactions. - `examples` : Example interactions to guide the model. - `planner` : The planner to use for action planning. - `code_executor` : The code executor to use for running code. - `input_schema` : The schema for input data. - `output_schema` : The schema for output data. - `generate_content_config` : Configuration for content generation.

GitHub Repository Structure

The ADK is organized into two main repositories:

1. **google/adk-python**: Contains the core Python library, including:
2. Source code for all ADK modules
3. Tests for verifying functionality
4. Examples demonstrating usage patterns

5. Documentation comments within the code

6. **google/adk-docs**: Contains the documentation site, including:

7. Conceptual guides explaining ADK concepts
8. Tutorials for getting started
9. API reference documentation
10. Examples and best practices

Conclusion

The ADK provides a comprehensive set of tools for evaluating agents, contributing to the project, and leveraging the API to build sophisticated agent applications. By understanding these components, developers can create robust, well-tested agents that meet their specific requirements and contribute to the growing ADK ecosystem.

Evaluation is a critical aspect of agent development, allowing developers to measure and improve agent performance. The ADK's flexible evaluation framework supports both simple unit testing and complex integration testing, enabling developers to ensure their agents meet the desired quality standards.

The contributing guide provides a clear path for getting involved with the project, whether through reporting issues, suggesting enhancements, improving documentation, or writing code. By following these guidelines, contributors can help shape the future of the ADK.

The API reference provides detailed information about the ADK's modules and classes, enabling developers to leverage the full power of the framework. By understanding the API, developers can create sophisticated agent applications that meet their specific requirements.