Tracing

The Agents SDK includes built-in tracing, collecting a comprehensive record of events during an agent run: LLM generations, tool calls, handoffs, guardrails, and even custom events that occur. Using the Traces dashboard, you can debug, visualize, and monitor your workflows during development and in production.

Note

Tracing is enabled by default. There are two ways to disable tracing:

1. You can globally disable tracing by setting the env var `OPENAI_AGENTS_DISABLE_TRACING=1`
2. You can disable tracing for a single run by setting `agents.run.RunConfig.tracing_disabled` to `True`

For organizations operating under a Zero Data Retention (ZDR) policy using OpenAI's APIs, tracing is unavailable.

Traces and spans

- **Traces** represent a single end-to-end operation of a "workflow". They're composed of Spans. Traces have the following properties:
  - `workflow_name`: This is the logical workflow or app. For example "Code generation" or "Customer service".
  - `trace_id`: A unique ID for the trace. Automatically generated if you don't pass one. Must have the format `trace_<32_alphanumeric>`.
  - `group_id`: Optional group ID, to link multiple traces from the same conversation. For example, you might use a chat thread ID.
  - `disabled`: If True, the trace will not be recorded.
  - `metadata`: Optional metadata for the trace.
- **Spans** represent operations that have a start and end time. Spans have:
  - `started_at` and `ended_at` timestamps.
  - `trace_id`, to represent the trace they belong to
  - `parent_id`, which points to the parent Span of this Span (if any)

- `span_data`, which is information about the Span. For example, `AgentSpanData` contains information about the Agent, `GenerationSpanData` contains information about the LLM generation, etc.

Default tracing

By default, the SDK traces the following:

- The entire `Runner.{run, run_sync, run_streamed}()` is wrapped in a `trace()`.
- Each time an agent runs, it is wrapped in `agent_span()`
- LLM generations are wrapped in `generation_span()`
- Function tool calls are each wrapped in `function_span()`
- Guardrails are wrapped in `guardrail_span()`
- Handoffs are wrapped in `handoff_span()`
- Audio inputs (speech-to-text) are wrapped in a `transcription_span()`
- Audio outputs (text-to-speech) are wrapped in a `speech_span()`
- Related audio spans may be parented under a `speech_group_span()`

By default, the trace is named "Agent trace". You can set this name if you use `trace`, or you can can configure the name and other properties with the `RunConfig`.

In addition, you can set up custom trace processors to push traces to other destinations (as a replacement, or secondary destination).

Higher level traces

Sometimes, you might want multiple calls to `run()` to be part of a single trace. You can do this by wrapping the entire code in a `trace()`.

```
from agents import Agent, Runner, trace

async def main():
    agent = Agent(name="Joke generator", instructions="Tell funny jokes.")

    with trace("Joke workflow"): 1
        first_result = await Runner.run(agent, "Tell me a joke")
        second_result = await Runner.run(agent, f"Rate this joke:
{first_result.final_output}")
        print(f"Joke: {first_result.final_output}")
        print(f"Rating: {second_result.final_output}")
```

1. Because the two calls to `Runner.run` are wrapped in a `with trace()`, the individual runs will be part of the overall trace rather than creating two traces.

Creating traces

You can use the `trace()` function to create a trace. Traces need to be started and finished. You have two options to do so:

1. **Recommended**: use the trace as a context manager, i.e. `with trace(...) as my_trace`. This will automatically start and end the trace at the right time.
2. You can also manually call `trace.start()` and `trace.finish()`.

The current trace is tracked via a Python `contextvar`. This means that it works with concurrency automatically. If you manually start/end a trace, you'll need to pass `mark_as_current` and `reset_current` to `start()` / `finish()` to update the current trace.

Creating spans

You can use the various `*_span()` methods to create a span. In general, you don't need to manually create spans. A `custom_span()` function is available for tracking custom span information.

Spans are automatically part of the current trace, and are nested under the nearest current span, which is tracked via a Python `contextvar`.

Sensitive data

Certain spans may capture potentially sensitive data.

The `generation_span()` stores the inputs/outputs of the LLM generation, and `function_span()` stores the inputs/outputs of function calls. These may contain sensitive data, so you can disable capturing that data via `RunConfig.trace_include_sensitive_data`.

Similarly, Audio spans include base64-encoded PCM data for input and output audio by default. You can disable capturing this audio data by configuring `VoicePipelineConfig.trace_include_sensitive_audio_data`.

Custom tracing processors

The high level architecture for tracing is:

- At initialization, we create a global `TraceProvider`, which is responsible for creating traces.

- We configure the `TraceProvider` with a `BatchTraceProcessor` that sends traces/spans in batches to a `BackendSpanExporter`, which exports the spans and traces to the OpenAI backend in batches.

To customize this default setup, to send traces to alternative or additional backends or modifying exporter behavior, you have two options:

1. `add_trace_processor()` lets you add an **additional** trace processor that will receive traces and spans as they are ready. This lets you do your own processing in addition to sending traces to OpenAI's backend.
2. `set_trace_processors()` lets you **replace** the default processors with your own trace processors. This means traces will not be sent to the OpenAI backend unless you include a `TracingProcessor` that does so.

External tracing processors list

- Weights & Biases
- Arize-Phoenix
- MLflow (self-hosted/OSS
- MLflow (Databricks hosted
- Braintrust
- Pydantic Logfire
- AgentOps
- Scorecard
- Keywords AI
- LangSmith
- Maxim AI
- Comet Opik
- Langfuse
- Langtrace
- Okahu-Monocle