

Where a Replay Has to Hold

Seven places a rerun stops being the run you had, how FastAI Agent holds each one, and a run behind every claim.

Checked against FastAI Agent 1.89.0

Agent Replay already lets you *rerun* a past run: load its trace, fork, change the prompt or a tool, run again, compare (see the [Replay reference](#)).

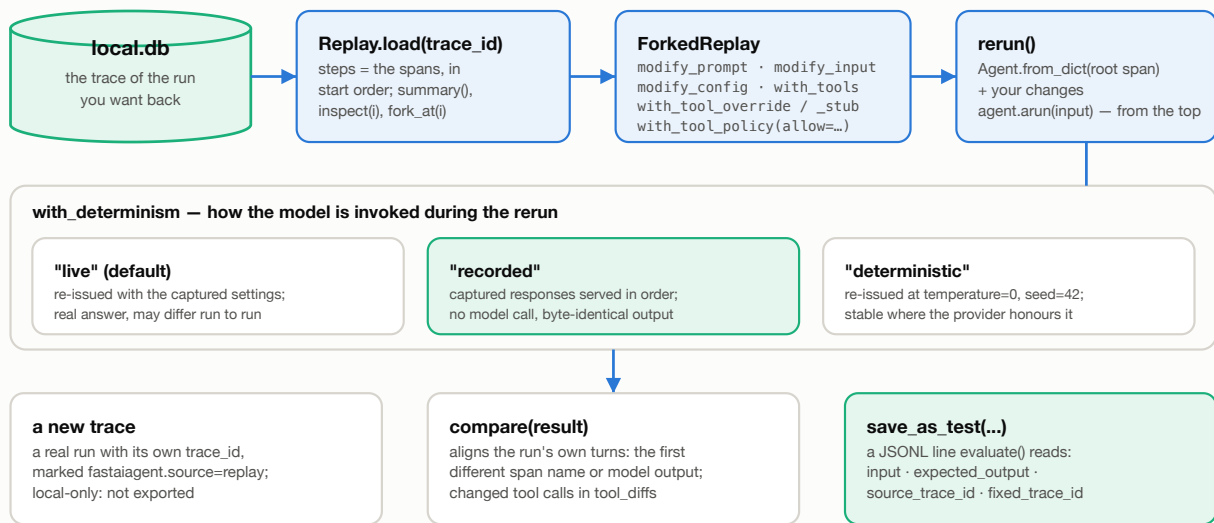
Running it again is not the same as reproducing it. A rerun makes one claim, *this is the run you had, with one thing changed*, and that claim is only as good as what the trace carried, what the model is allowed to say, what the tools are allowed to do, and what the rerun is allowed to touch. There are seven places it breaks:

- the trace doesn't carry enough to rebuild the agent;
- the model says something new;
- the tools do something real;
- the rerun needs a turn that was never recorded;
- the rerun reaches into production state;
- the fix isn't kept;
- the rerun is counted as production.

This page explains how Replay works, one of those places at a time. Each section has a diagram, the rule the SDK follows, a proof, and the code: the SDK source that implements the rule and the script that proves it. The proofs are a real run of a `gpt-4.1-mini` agent and its reruns, or scripts in [examples/replay/proofs/](#) that run offline against the published SDK, on the SDK's own `FunctionModel` and a throwaway `local.db`. The scripts run in CI, so what this page quotes can't drift from what the SDK does. Every output below came out of one of those runs.

First: a replay is a real run, built from a trace

One replay: load, fork, change one thing, run again, compare



A replay is a real execution, not a playback. What you override changes; everything else comes from the trace.

Code: `trace/replay.py` — `Replay` · `ForkedReplay.arerun` · `compare` · `ReplayResult.save_as_test`

A replay is a real execution, not a playback. What you override changes; everything else comes from the trace.

The vocabulary, in the order it happens:

- **Replay.load(trace_id)** reads the trace from `local.db` into **steps**: the spans, in start order. `summary()`, `inspect(i)` and `steps()` read them.
- **fork_at(step)** gives you a `ForkedReplay`. The step is the reference point for `compare()`; the rerun itself always starts from the top. Mid-trace resume, replaying the messages up to a step and continuing from there, is not what this does.
- **The modifiers**: `modify_prompt`, `modify_input`, `modify_config`, `with_tools` (replace every tool), `with_tool_override(name, tool)` and `with_tool_stub(name, returns=...)` (replace one), `with_tool_policy(policy, allow=[...])` (which of the original tools may really run), and `with_determinism(mode, on_miss=...)`.
- **The modes**: `"live"` re-issues the model calls with the captured settings; `"recorded"` serves the captured responses in order and never calls the model; `"deterministic"` re-issues at temperature 0 with a fixed seed.
- **rerun()** rebuilds the agent from the root span with `Agent.from_dict`, applies your changes, runs it, and returns a `ReplayResult` with the original output, the new output, the new trace id, and what it did with each tool.
- **compare(result)** aligns both traces' turns and finds the first that differs. **save_as_test(...)** writes the rerun as a case `evaluate()` reads.

You get all of it from one call:

```

replay = Replay.load(result.trace_id)
print(replay.summary())

forked = (
    replay.fork_at(0)
    .with_determinism("recorded", on_miss="error")
    .with_tool_policy("block", allow=["lookup_order"]) # only this tool really runs
)
rerun = forked.rerun()
assert rerun.new_output == rerun.original_output # no model call was made

fixed = (
    replay.fork_at(0)
    .modify_prompt("Use the tool to answer. Answer in French, one sentence.")
    .with_tool_policy("block", allow=["lookup_order"])
)
rerun = fixed.rerun() # live: a real call, a new answer
print(fixed.compare(rerun).diverged_at)

```

The rest of this page is what that rerun has to get right.

Code: `Replay` and `ForkedReplay` in [trace/replay.py](#). The full loop, with a real model, is [examples/04_agent_replay.py](#).

1 • Between the trace and the agent: the root span is the blueprint

A rerun that needs the original agent object, or a session kept alive, is not a rerun of a production failure; by the time you look, both are gone. The trace has to carry enough to rebuild the agent on its own, and it has to leave out the one thing that must never be in a trace.

Between the trace and the agent: the root span is the blueprint

```
root span · agent.shipping
agent.system_prompt "You answer..."
agent.llm.config {provider, model, ...}
agent.tools [{name, description,
parameters, replay_class}]
agent.guardrails [{name, guardrail_type, ...}]
agent.config {max_iterations, ...}
agent.input "Where is order 1042?"
agent.output "Order 1042 was..."
```

from_dict

system_prompt

verbatim — or your modify_prompt

llm_endpoint → LLMClient.from_dict

api_key never from the dict; resolved from the environment; base_url http(s) only

tools → the tool plan (1.89.0)

what you supply or allow (§3); every other tool is blocked

guardrails → Guardrail.from_dict

rebuilt, and they run again

config → AgentConfig

max_iterations, tool_choice, ...

NOT REBUILT — the rerun starts without:

memory nothing is read into the prompt, nothing is written back
checkpointer a pause cannot be held · plane agent_id managed policies never gate it
output_type a Python class is not in a trace; the response_format schema passes through

The proof rebuilt the agent with Agent.from_dict in the defining process: the tool was the live function, the guardrail was back, memory was None while the original's window held two messages, and no api_key to leak.

Code: trace/replay.py::_build_agent_dict / _build_tool_plan · agent/agent.py::Agent.from_dict · llm/client.py::LLMClient.from_dict

Five attributes on the root span become five arguments to the constructor. Credentials come from the environment; a rerun's tools come from you.

The rules:

- **The root span carries the blueprint:** `agent.system_prompt` as resolved for that run, `agent.llm.config`, `agent.tools` with each tool's schema and replay class, `agent.guardrails`, `agent.config`, plus `agent.input` and `agent.output`. They are captured on every run.
- **The api key is never in the trace and never read from it.** `LLMClient.to_dict` doesn't emit it; `from_dict` ignores one if present and warns, and resolves credentials from the environment. A `base_url` must be `http(s)`.
- **Agent.from_dict finds a tool's function in this process; a rerun does not rely on that.** `FunctionTool` registers itself by name on creation, and `from_dict` prefers the registered tool; in a process that never defined it, the tool is rebuilt as a schema with no function, with a warning. Since 1.89.0 a rerun rebuilds the agent *without* its tools and gives it only the ones you supply or allow; see section 3.
- **Guardrails are rebuilt** from their serialized form and run again on the rerun.
- **Not rebuilt:** memory, the checkpointer, the plane `agent_id`, and `output_type`. Each is a boundary of its own, in section 5.

Proof 1 runs an agent with a tool, a guardrail and memory, then reads the root span back and rebuilds from it:

```

the root span carries:
  agent.name          shipping
  agent.system_prompt You answer shipping questions.
  agent.llm.config     {"provider": "test", "model": "function-model"}
  agent.tools          [{"name": "lookup_order", "description": "Look an order up in the
shipping system.", ...
  agent.guardrails     [{"name": "no_secrets", "guardrail_type": "code", "position":
"output", "config": {}},...
  agent.config         {"max_iterations": 3, "tool_choice": "auto", "temperature": null,
"max_tokens": null,...
  api_key in agent.llm.config: False

```

```

Steps:
[0] agent.shipping
[1] memory.read
[2] llm.test.function-model
[3] tool.lookup_order
[4] llm.test.function-model
[5] guardrail.no_secrets
[6] memory.write
[7] memory.write

```

```

the rebuilt agent:
  llm          LLMClient(provider='test', model='function-model')
  tools        ['lookup_order']   the live function: True   replay_class=read_only
  guardrails   ['no_secrets']
  config       max_iterations=3
  memory       None   (the original's window holds 2 messages)

```

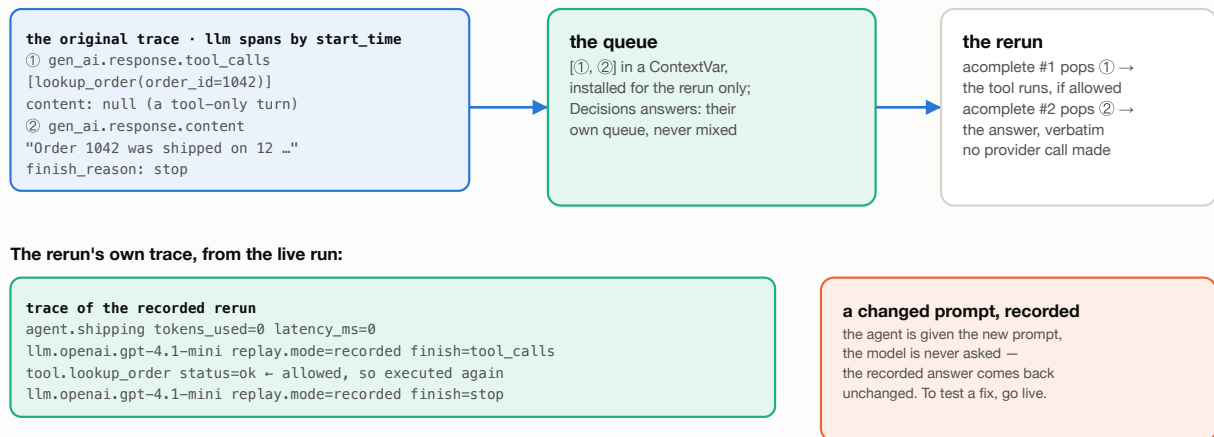
The tool came back as the very function that was defined, the guardrail came back, the config came back, and the rebuilt agent has no memory while the original's window holds the conversation.

Code: `_build_agent_dict` in [trace/replay.py](#); `Agent.from_dict` in [agent/agent.py](#); `LLMClient.from_dict` in [llm/client.py](#); `FunctionTool._from_dict` in [tool/function.py](#). `Replay.from_platform` maps a plane trace into the same shape. The proof is [proofs/proof_blueprint.py](#); the attribute list is in [Agent Reconstruction Attributes](#).

2 · Between the model and the record: a queue of captured answers

Rerunning the model gives you a new answer, which is what you want when testing a fix and exactly what you don't want in a regression suite. Recorded mode has to serve the model's words back in the right order, for every turn, without a network call.

Between the model and the record: a queue of captured answers



byte-identical output · identical with the original run · compare() → diverged_at=None

Needs: the captured responses. A trace with none — a plane-fetched trace with payloads stripped — raises ReplayError.

Code: trace/replay.py::_recorded_response_from_span / _all_llm_responses · llm/client.py::acomplete (_replay_recorded_response)

The captured responses form a queue. Each model call pops the next one and marks its span. The tool between the turns runs if you allowed it.

The rules:

- **Every captured response, in order.** Before the rerun, every span with `gen_ai.response.content` or `gen_ai.response.tool_calls` is read, sorted by start time, and installed as a queue in a ContextVar for that rerun only.
- **Each model call pops the front.** `acomplete` checks the queue first; when it has an entry, it returns it, marks the span `replay.mode=recorded`, and never calls the provider. A multi-turn tool loop replays turn by turn.
- **A tool-only turn is served too.** OpenAI sends `content: null` on a turn that only calls tools. Those spans are in the queue since 1.84.0, so the rerun reaches the tool between the two turns, and runs it if you allowed it (section 3). Before that, a recorded rerun jumped to the final answer and never reached a tool.
- **Decisions API answers have their own queue**, never handed to a chat turn or the reverse.
- **A changed prompt cannot change a recorded answer.** The rebuilt agent is given the new prompt, but the model is never asked. To test a fix, use live mode.
- **It needs the captured responses.** A trace with none, such as one pulled from the plane with payloads stripped, raises `ReplayError`.

Proof 2 is the live run, replayed with its read-only lookup allowed:

```

recorded : 'Order 1042 was shipped on 12 September via DHL with the tracking number 7788-2201.'
identical: True   tool calls so far: 2
tools      : {'lookup_order': 'live'}   exported=False
      agent.shipping                     tokens_used=0 latency_ms=1
      llm.openai.gpt-4.1-mini             tokens=None+None finish=tool_calls replay.mode=recorded
      tool.lookup_order                   runner.type=tool replay_class=read_only status=ok
      llm.openai.gpt-4.1-mini             tokens=None+None finish=stop replay.mode=recorded
compare(): status=ok diverged_at=None

```

Byte-identical, no tokens billed, the allowed tool called a second time, and both model spans marked recorded. The offline proof adds the prompt change:

```

modify_prompt under recorded mode:
  prompt the rerun was given: 'Répondez en français.'
  answer: 'Order 1042 was shipped via DHL on 12 September.'

```

The screenshot shows the 'Traces' interface for 'agent.shipping'. The timeline displays four spans: 'agent.shipping' (2ms), 'llm.openai.gpt-4.1-mini' (<1ms), 'tool.lookup_order' (<1ms), and another 'llm.openai.gpt-4.1-mini' (<1ms). The right panel shows the attributes of the first model span, including 'gen_ai.system': 'openai', 'gen_ai.request.model': 'gpt-4.1-mini', and 'gen_ai.request.tools' containing a function definition for 'lookup_order'.

The recorded rerun in the Local UI: four spans in 2 ms, no tokens, no cost.

```

{
  "required": {
    "order_id"
  }
}
]
{
  "replay.mode": "recorded",
  "gen_ai.response.finish_reason": "tool_calls"
}

```

Its first model span's attributes: `replay.mode` is `recorded`, and the finish reason is the captured `tool_calls`.

Code: `_recorded_response_from_span` and `_all_llm_responses` in [trace/replay.py](#); the pop is in `acomplete`, [llm/client.py](#). The offline proof is [proofs/proof_recorded.py](#); a recorded rerun with Decisions API calls is [examples/105_decision_replay.py](#). The modes are tabled in [Fidelity Guarantees](#).

3 · Between the record and the world: a rerun runs only the tools you give it

Recorded mode fixes the model's words. It does not fix the world. Whatever the recorded response asks a tool to do, the rerun is asked to do. Until 1.89.0 it did: the rebuilt agent called the real function in the process that defined it, and called a REST or MCP tool's endpoint from the trace, so a rerun of a refund agent refunded again. Since 1.89.0 the rerun runs only the tools you give it.

Between the record and the world: a rerun runs only the tools you give it

default: blocked

every tool you did not supply →
an error result naming it;
charges = [42.0] (not again)
blocked_tool_calls={'charge_card': 1}

live, allow=[...]

same process: the real function
runs → charges = [42.0, 42.0];
fresh process: no function;
REST / MCP: only when allowed

override / stub

with_tool_override(name, fn)
with_tool_stub(name, returns=...)
never registered globally;
with_tools([...]) replaces them all

Why the blocked rerun's output still matched: the next model turn is served from the recording whatever the tool returned.
So the rerun says it: blocked_tool_calls, a warning, and compare().tool_diffs with each changed tool call's disposition.

replay_class · on every tool span as fastaiagent.tool.replay_class

read_only

may be re-executed: a GET, a pure lookup

idempotent

recorded output injected; re-running is safe but pointless

side_effecting

the default · recorded output injected, never re-executed

never inferred: a GET RESTTool is side_effecting

until you say otherwise; the central Replay engine
reads the class to choose inject vs. execute;
the SDK's rerun blocks whatever you didn't allow

Rule: nothing a rerun calls runs for real unless you named it. with_tool_policy('error') raises instead of blocking.

Code: trace/replay.py::build_tool_plan · tool/_replay_blocked.py · tool/base.py (replay_class)

Default: blocked, and counted. Live: the real function, if this process has it. Override or stub: yours.

The rules:

- **Every tool you did not supply is blocked.** The model gets an error result that names the tool and how to allow it. The rerun counts the call in `blocked_tool_calls`, logs a warning, and reports every tool's disposition in `tool_dispositions`. Because the next model turn is served from the recording, a blocked rerun can still produce the recorded answer, which is why the rerun says what it blocked.
- **with_tool_policy("live", allow=[...]) runs the real thing.** "live" runs every function tool's registered implementation; `allow` names the ones that run under any policy. That works only in the process that defined the function. A fresh process has none (`no_function`), and the agent gets 'No function attached to this tool'.
- **REST and MCP tools run only when named in `allow`.** Their endpoint was rebuilt from the trace. A REST tool whose auth headers were redacted from the trace is never called, since its header holds `[redacted]`, not the credential; pass the live tool with `with_tool_override`.
- **with_tool_override(name, fn_or_tool) and with_tool_stub(name, returns=...) replace one tool; with_tools([...]) replaces them all.** Neither touches the global tool

registry. The Local UI's rerun lets you pick Block, Stub or Live per tool; `fastaiagent ui --agent path.py:agent` is how its tools get functions.

- `with_tool_policy("error")` raises instead of blocking, before anything runs in recorded mode. `FASTAIAGENT_REPLAY_TOOL_POLICY` sets the default per process.
- `replay_class` is recorded on every tool span as `fastaiagent.tool.replay_class`: `read_only`, `idempotent` or `side_effecting`. The default is `side_effecting`, and it is never inferred; a GET `RESTTool` is side-effecting until you say otherwise. The central Replay engine reads it to choose between injecting the recorded output and re-executing. The SDK's rerun does not: it blocks whatever you did not allow, whatever its class.

Proof 3 charges a card:

```
replay_class: charge_card='side_effecting' GET carrier_status='side_effecting'
tool span: fastaiagent.tool.replay_class='side_effecting'
original run: 'Done: charged 42.00.' charges=[42.0]
default rerun: 'Done: charged 42.00.' charges=[42.0]
  tool_dispositions={'charge_card': 'blocked', 'carrier_status': 'blocked'}
  blocked_tool_calls={'charge_card': 1}
  tool.status = error
live, same process: 'Done: charged 42.00.' charges=[42.0, 42.0]
  tool_dispositions={'charge_card': 'live', 'carrier_status': 'blocked'}
live, fresh process:
  output: 'Done: charged 42.00.'
  disposition: no_function
  tool.status = error | tool.error = 'No function attached to this tool'
with_tool_override: 'Done: charged 42.00.' charges=[42.0, 42.0]
  tool_dispositions={'charge_card': 'override', 'carrier_status': 'blocked'}
```

The default rerun charged nothing and said so: one blocked call. Allowing the tool charged the card a second time, in the process that defined it; in a fresh process there was nothing to run. The override charged nothing. Every one of them produced the recorded answer. **A recorded rerun that passes is not evidence that the tools behaved.** Read `blocked_tool_calls`, or `compare().tool_diffs`.

Code: `_build_tool_plan` in [trace/replay.py](#); the blocked stand-in in [tool/replay_blocked.py](#); `replay_class` in [tool/base.py](#). The proof is [proofs/proof_tools.py](#); [examples/70_tool_replay_class.py](#) runs the class rules as pytest, and [Tools → Replay safety](#) is the reference.

4 · Between the recording and the rerun: the miss, and the divergence

A recording is finite. A rerun with a changed tool or prompt can take a turn the original never took, and the honest options are to stop or to say loudly that you are going live. And once a rerun differs, you need to know where.

Between the recording and the rerun: the miss, and the divergence

a recording that ends early

```
original run: the second model call  
raised 'provider returned 503'  
trace: agent.shipping [ERROR]  
llm ← ① tool_calls captured  
tool.lookup_order  
captured model responses: 1
```

the rerun asks for ②

the queue is installed but empty:
replay is active, responses
exhausted — the miss

on_miss="live" · default

warning, then a real call —
billed, nondeterministic

on_miss="error"

ReplayError before any
provider call — for suites

compare(): where did the rerun leave the original?

live rerun with a French prompt, from the live run

```
step 0 agent.shipping 'Order 1042 was shipped on 12 Sept...' | 'La commande 1042 a été expédiée...' ← diverged_at=0  
step 1 llm.openai... (tool-call turn: no content) | (tool-call turn: no content)  
step 2 tool.lookup_order |  
step 3 llm.openai... 'Order 1042 was shipped on 12 Sept...' | 'La commande 1042 a été expédiée...'
```

- only the run's own turns are aligned: the root and its direct children, minus memory.* (a rerun has no memory)
- root first, so a rerun whose final answer differs diverges at step 0; a changed tool call goes to tool_diffs
- compare_status='rerun_failed' when the rerun's trace can't be loaded: 'couldn't tell', not 'no divergence'

Code: llm/client.py::acomplete (_replay_on_miss) · trace/replay.py::first_divergence / compare

An empty queue is a miss. One setting decides whether a miss is an error or a billed live call. The divergence is the first turn whose name or model output differs.

The rules:

- **A miss is an installed, empty queue.** The rerun made more model calls than the trace captured.
- **on_miss="error" raises before any provider call.** Use it for every suite: it can never spend tokens or drift.
- **on_miss="live", the default, warns and makes a real call.** Billed, nondeterministic, and loud since 1.48.0; before that it was silent.
- **compare()** aligns the run's own turns, root first: the root and its direct children, without memory.* spans (a rerun has no memory, and before 1.89.0 the missing spans shifted every index, so a byte-identical rerun of a memory agent "diverged"). The first turn whose span name differs, or whose agent.output or gen_ai.response.content differs, is diverged_at, an index into the full step list. The root carries the final answer, so a rerun whose answer differs diverges at step 0; a rerun whose answer matches but whose path differs diverges at the first changed span.
- **Tool changes are listed apart.** tool_diffs holds each aligned tool call whose status or result changed, with the rerun's disposition (blocked , stub , live , ...); tools_diverged_at is the first. A blocked tool does not move diverged_at.
- **compare_status="rerun_failed"** means the rerun's trace couldn't be loaded: couldn't tell, not no divergence.

Proof 4's original run died on its second model call, so its recording holds one response and a rerun needs two:

```

captured model responses: 1
on_miss='error' → ReplayError: determinism='recorded' ran out of captured LLM responses: the
rerun makes more LLM calls than the original trace. Use with_determinism('recorded',
on_miss='live') to allow falling through to live provider calls.
  warning: determinism='recorded' ran out of captured LLM responses; falling through to a
LIVE test call (billed, nondeterministic). Pass with_determinism('recorded', on_miss='error')
to fail instead.
on_miss='live' (default) → the live call to provider 'test' failed: LLMError

```

And the divergence, from the live run rerun with a French prompt:

```

live, new prompt: 'La commande 1042 a été expédiée le 12 septembre via DHL, avec le numéro de
suivi 7788-2201.'
compare(): status=ok diverged_at=0 (agent.shipping)
  step 0 agent.shipping      'Order 1042 was shipped on 12 September v' |
agent.shipping      'La commande 1042 a été expédiée le 12 se'
  step 1 llm.openai.gpt-4.1-mini '' |
llm.openai.gpt-4.1-mini ''
  step 2 tool.lookup_order   '' |
tool.lookup_order   ''
  step 3 llm.openai.gpt-4.1-mini 'Order 1042 was shipped on 12 September v' |
llm.openai.gpt-4.1-mini 'La commande 1042 a été expédiée le 12 se'

```

(The rerun allowed `lookup_order` , so the French answer was written from a real lookup.)

The path was the same, tool call and all, and the words changed; the first differing step is the root, because the root holds the answer.

The screenshot displays the 'Agent Replay' interface in the Local AI UI. The top section shows a trace for 'agent.shipping' with a duration of 1.73s, 4 spans, 444 tokens, and a cost of \$0.000136. The trace includes steps for 'llm.openai.gpt-4.1-mini' and 'tool.lookup_order'. Below the trace, a 'Rerun complete' section shows a comparison between the 'Original output' and the 'Rerun output'. The original output is 'Order 1042 was shipped on 12 September via DHL with tracking number 7788-2201.' and the rerun output is 'La commande 1042 a été expédiée le 12 septembre via DHL avec le numéro de suivi 7788-2201.' The comparison table shows that the rerun diverged at step 0. A green banner at the bottom right indicates 'Rerun complete'.

The same rerun started from the Local UI: Fork here, a new system prompt, Rerun with modifications. Original and rerun side by side, diverged at step 0.

Code: the miss is in `acomplete`, `llm/client.py`; `_first_divergence` and `compare` in `trace/replay.py`. The proof is `proofs/proof_miss.py`; `examples/38_replay_comparison.py` is the side-by-side walkthrough with a real model.

5 • Between a rerun and production: what a rerun cannot touch

A rerun is a real run. The question is what it is allowed to reach. The rebuilt agent is a stranger to production state by construction: it was built from attributes, not from your running objects.

Between a rerun and production: what a rerun cannot touch

memory

original run's prompt roles:
[system, user, assistant, user]
rerun's prompt roles: [system, user]
memory before: 4 · after: 4

guardrails

rebuilt from `agent.guardrails`
and run again on the rerun:
rerun spans end with
`guardrail.no_secrets`

a pause

a tool calls `interrupt(...)` →
`ReplayError: 'paused: refund needs
a human ... cannot hold a pause.
Override the tool ... to replay past it'`

and from the code, not the proof:

no plane `agent_id` → managed approval policies never gate a rerun; domain-wide plane guardrails still run
tools → only the ones you supply or allow run (§3) · a rerun never registers its agent

*A rerun is for reproducing a model's behaviour, not for re-running production side effects:
so it blocks every tool you did not give it. Allow only the tools that are harmless to run again.*

Code: `trace/replay.py::_arun_unpaused` · `agent/agent.py::Agent.from_dict` (no memory, no checkpoint)

No memory in, none out. Guardrails run again, domain-wide plane rules included. A pause has nowhere to be held. Managed approval policies do not apply.

The rules:

- **No memory.** The rebuilt agent reads nothing into its prompt and writes nothing back. The original agent's memory is untouched by a rerun.
- **Guardrails run again.** They were rebuilt from the trace; the rerun's trace ends with the same `guardrail.*` span.
- **No checkpoint, so no pause.** A tool that calls `interrupt()` during a rerun raises `ReplayError` naming what paused and how to replay past it. Before 1.78.0 the bare pause signal escaped.
- **No plane identity.** The rebuilt agent has no `agent_id` and never registers to get one, so managed approval policies never gate a rerun, and plane guardrails scoped to that agent do not apply. **Domain-wide plane guardrails still do:** they are enforced at the edge, like any local rule. A tool that writes runs in a rerun only when you supply or allow it (section 3).
- **Otherwise a normal run, marked as a rerun.** It is traced with its own trace id, and the trace says it is a replay (section 7).

Proof 5 gives the original agent a memory window and a guardrail, then reruns it, then overrides its tool with one that asks for approval:

```
original run: prompt roles ['system', 'user', 'assistant', 'user']  memory holds 4 messages
rerun:        prompt roles ['system', 'user']  memory holds 4 messages
rerun spans:  ['agent.refunds', 'llm.test.function-model', 'tool.refund', 'llm.test.function-
model', 'guardrail.no_secrets']
a tool that pauses → ReplayError: Replay of trace 118ea9588401c3afb16f65dde92f3090 paused:
refund needs a human. A replay rebuilds the agent without a checkpoint, so it cannot hold a
pause. Override the tool that paused with with_tool_override(name, tool) to replay past it.
```

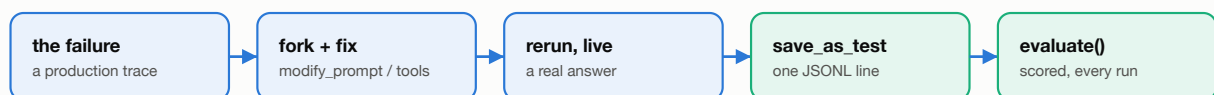
The original's second turn saw the conversation; the rerun saw only the system prompt and the question, and left the memory as it was. The guardrail ran. The pause was refused with its reason.

Code: `_arun_unpaused` in [trace/replay.py](#); `Agent.from_dict` in [agent/agent.py](#) takes no memory and no checkpoint. The proof is [proofs/proof_isolation.py](#). For the memory boundaries themselves, see [Agent Memory Breaks at the Boundaries](#).

6 · Between a fix and the future: the failure becomes a test

A rerun that proves a fix once is a demo. The point of replay is that the failure never comes back, which means the rerun has to end up somewhere a test runner reads.

Between a fix and the future: the failure becomes a test



```
the saved case · regression_tests.jsonl
input 'What is your refund policy?'
expected_output 'Refunds are issued within 14 days.'
trace_id 8b05b26f... ← the rerun
source_trace_id 09efdb5c... ← the failure
fixed_trace_id 8b05b26f...
fork_step 0
modifications {'prompt': 'You are a support agent. ...'}
```

what evaluate() reads
input and expected_output — the rest
is the paper trail back to the bug.
exact_match passed=True on the fix,
passed=False with the old prompt.
Any scorer works, including an
LLM judge.

*Free and deterministic suites: rerun with determinism='recorded', on_miss='error' — no model, no bill, no drift.
The regression-from-trace template runs this loop in CI against a real model; pytest --eval-baseline gates it.*

Code: `trace/replay.py::ReplayResult.save_as_test · eval (evaluate) · examples/regression-from-trace/`

One line per case, in the shape `evaluate()` already reads, with the trail back to the trace it came from.

The rules:

- **save_as_test** appends one JSONL line with `input` and `expected_output`, which `evaluate()` reads, plus the paper trail: the rerun's `trace_id`, the failing `source_trace_id`, `fixed_trace_id`, `fork_step` and the `modifications`.

- `evaluate()` runs the agent on the case and scores it with any scorer, `exact_match` or an LLM judge.
- **A suite can be free and deterministic:** rerun with `determinism="recorded"` and `on_miss="error"`, and no model is called.
- **A pytest gate keeps it running:** `pytest --eval-baseline` compares every run against a baseline (see [Agent CI](#)).

Proof 6 saves a recorded rerun and evaluates the agent against it, then evaluates an agent with the old prompt:

```
the saved case:
  input          'What is your refund policy?'
  expected_output 'Refunds are issued within 14 days.'
  trace_id       '8b05b26f25a17448566dbaed0abeef44'
  source_trace_id '09efdb5c25de85d392aad2293d77cb6e'
  fixed_trace_id  '8b05b26f25a17448566dbaed0abeef44'
  fork_step      0
  modifications   {'prompt': 'You are a support agent. Refunds are issued within 14 days.'}
  created_at      '2026-10-10T19:51:37.505832+00:00'
evaluate() on the saved case: exact_match passed=True score=1.0
evaluate() with the old prompt: exact_match passed=False score=0.0
```

The live run saved the same way, pointing back at its own trace:

```
saved: {'input': 'Where is order 1042?', 'expected_output': 'Order 1042 was shipped on 12
September via DHL with the tracking number 7788-2201.', 'trace_id': '3fc2738e...',
'source_trace_id': 'c3934819...', 'fixed_trace_id': '3fc2738e...', 'fork_step': 0,
'modifications': {}, ...}
```

Code: `ReplayResult.save_as_test` in [trace/replay.py](#). The proof is [proofs/proof_regression.py](#); the whole loop against a real model, with a broken tool to fix, is the [Regression from Trace](#) template, and [examples/62_replay_to_regression.py](#) is the single-file version.

7 · Between a rerun and the record of production: a rerun says what it is

A rerun's trace looks like a run of the agent, because it is one. Until 1.89.0 nothing said otherwise. It was exported to the platform as a production run of the agent, counted in Analytics, Overview and Agents, curated into eval datasets and mined for memory facts. A rerun also pushed its rebuilt agent definition to the platform, where it could overwrite the real one.

Between a rerun and the record of production: a rerun says what it is

the rerun's root span

```
fastaiagent.source = 'replay'
fastaiagent.replay.source_trace_id = c3934819...
fastaiagent.replay.determinism = 'recorded'
fastaiagent.replay.tool_policy = 'block'
fastaiagent.replay.modifications = ['prompt']
fastaiagent.replay.surface = 'python'
each tool span: ...replay.tool_disposition
```

local.db

rows written with
synced=1: already sent

not exported

the platform drain skips them;
add_exporter sinks drop them;
an older SDK on the same
store skips them too

never registered

a rerun does not push its
agent definition, exported
or not

production readers leave experiments out, and say what they left out

```
UI Analytics · Overview · Agents → a note: 'Excludes 12 experiment traces ($0.42)'
list_traces(exclude_sources=True) · fastaiagent eval curate · fastaiagent learn
opt back in: the 'Include experiments' toggle · --include-experiments
```

Opt in: `forked.with_export()` · `connect(export_experiments=True)` · `FASTAIAGENT_EXPORT_EXPERIMENTS=1`. Elsewhere: `with_trace_db(path)`.

Code: `trace/provenance.py` · `trace/storage.py::LocalStorageProcessor.on_start` · `trace/otel.py::EgressFilteredExporter`

Marked at the source, kept on the machine, left out of the numbers, and never registered.

The rules:

- **Every span is marked.** A rerun's spans carry `fastaiagent.source = "replay"`. Its root also carries `fastaiagent.replay.source_trace_id`, the fork step, the determinism mode, the tool policy, the *kinds* of modification (never the new text) and the surface (`python`, `cli`, `ui`). Each tool span carries its disposition.
- **Not exported.** The rows are written already marked as sent, so no exporter ships them, including an older SDK sharing the store, and `add_exporter` sinks drop them. Opt in with `with_export()`, `connect(export_experiments=True)` or `FASTAIAGENT_EXPORT_EXPERIMENTS=1`; the spans keep their marking.
- **Never registered.** A rerun does not push its agent definition, exported or not.
- **Left out of production readers**, which say what they left out: the Local UI's Analytics, Overview and Agents, `list_traces(exclude_sources=True)`, `fastaiagent eval curate` and `fastaiagent learn`.
- **A store of its own, if you like.** `with_trace_db(path)` writes the rerun elsewhere; `compare()` finds it there.

Proof 7 reruns a run and reads back what was written:

```
rerun root: fastaiagent.source='replay'
  replays: True
  determinism='recorded'
  tool_policy='block'
rerun.exported=False
original rows synced=[0]
rerun    rows synced=[1]
list_traces(): 2 traces    exclude_sources=True: 1 (the rerun left out: True)
with_trace_db: rerun in experiments.db: True    in local.db: False
compare(): status=ok
```

The original is waiting to be exported; the rerun never will be. The production listing has one trace, and the rerun written elsewhere is still comparable.

Code: [trace/provenance.py](#) ; the stamp is in `LocalStorageProcessor.on_start` , [trace/storage.py](#) ; the export filter is `_EgressFilteredExporter` in [trace/otel.py](#) . The proof is [proofs/proof_provenance.py](#) . The reader side is [Tracing → provenance](#).

If you build or buy replay, ask

1. **What does the rerun rebuild from?** The trace alone, or an object or session you have to keep alive?
2. **Under "recorded", what exactly is fixed?** The model's words? The tools? Both?
3. **What runs when a tool is called in a rerun, and who decided?** Your stub, the real function, or nothing?
4. **What happens when the rerun needs a turn the recording doesn't have?** An error, or a quiet live call?
5. **What can a rerun touch?** Memory, approvals, managed policies?
6. **Where does the fix go afterwards?** A screenshot, or a test?
7. **Does a rerun count as a run?** In your dashboards, your exports, your eval data, your agent registry?

FastAI Agent's answers are the seven sections above, and each has a run behind it.

What replay still won't do

- **The fork is a reference point; the rerun starts from the top.** The agent re-executes end to end with your changes applied, and `fork_at(step)` tells `compare()` where to look.
- **Recorded mode cannot test a fix.** The model never runs, so the recorded answer comes back whatever the prompt says. Use live mode to test, recorded mode to pin.
- **A tool that did something different is not a divergence.** `diverged_at` reads span names and model outputs; tool results are listed in `tool_diffs` , so check both.

- **A chain trace is not refused.** Replay rebuilds from the first parentless span, and a `chain.` `<name>` root has no agent blueprint, so the rerun is a default agent named `replayed-agent` on the default model, fed the captured responses. On a two-agent chain the offline check returned the first agent's answer under the second agent's name. Known, and open; use `Chain.aresume` and `Chain.afork` for chains, as [Concepts](#) says.
 - **`modify_state` raises `NotImplementedError`.** Local replay is read-only on state; `Agent.afork` and `Chain.afork` fork from a checkpoint with changed state.
 - **A fresh process has no tool functions unless you supply them,** through `with_tool_override`, `with_tool_stub`, `with_tools`, or `fastaiagent ui --agent`; allowing a tool there finds nothing to run.
 - **`@idempotent` does not protect a rerun.** Its cache lives in the checkpoint, and a rerun has none; an allowed `@idempotent` tool runs again.
 - **`replay_class` is not acted on locally.** The SDK's rerun blocks by default instead; choosing per call by class is the central engine's job.
 - **Streaming is replayed as its final text.** Chunks are not recorded.
 - **"deterministic" is as deterministic as the provider.** It forces temperature 0 and a seed; a provider that ignores the seed is stable, not identical.
 - **A trace without captured responses cannot replay recorded.** A plane trace fetched with payloads stripped raises `ReplayError`.
-

The point of all of it

A rerun is a claim about a run you can no longer observe, rebuilt from a record that had to be complete, with a model that is either asked again or quoted exactly, tools that run only by your choice, nothing of production in reach, and a trace that says it is a rerun. That is the whole design: not a smarter playback, but a rerun whose every difference from the original you chose.

The split between the SDK and the plane stays the same. Load, fork, rerun, compare and save are all in the open-source SDK, from a plain trace on your disk. Choosing per tool call between injecting a recorded output and re-executing, by `replay_class`, is the central Replay engine's job; the SDK records the class on every span so it can.

Ask your own replay the seven questions. The bugs aren't in the rerun; they're in what the rerun was allowed to touch.

See also

- [Concepts & Mental Model](#): the short version, in prose.
- [Replay reference](#): every method, and the Local UI flow.
- [Fidelity Guarantees](#): the modes, what is captured, and the known gaps.
- [Where a Trace Has to Hold](#): the record this page rebuilds from.

- [Regression from Trace](#): the loop as a template.