

A Durable Run Breaks at the Boundaries

Six places a paused or crashed run goes wrong between one process and the next, how FastAI Agent holds each one, and a proof you can run for every claim.

Requires FastAI Agent 1.88.0+

FastAI Agent already lets you see a run's progress: every turn and every tool call is a checkpoint row, and the Local UI's [checkpoint inspector](#) shows them as a timeline with a state diff between any two (see [Durability](#)).

Seeing the rows is not the same as trusting a resume. Durability is a promise: a process can die, or stop for a human, and another process can pick up exactly where it left off without repeating what was done. That promise has to hold wherever two things meet:

- one step and the next;
- the process that paused and the one that resumes;
- a crash and a rerun;
- a resume and a side effect;
- a run that ended and one that died;
- a laptop and a fleet, and the plane.

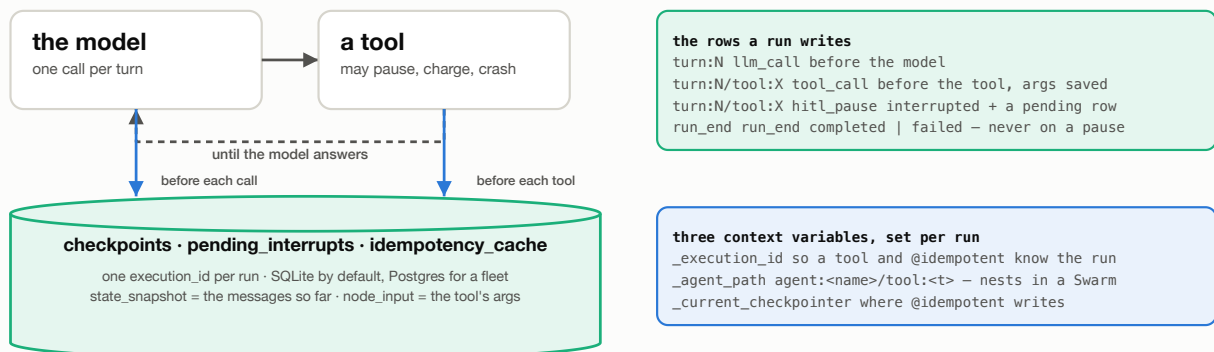
Those six boundaries are where durability goes wrong.

This page explains how FastAI Agent's durability works, one boundary 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 in [examples/durability/proofs/](#) and run against the published SDK: a real agent on the SDK's own offline model, real SQLite and Postgres, and real child processes that are killed where it matters. They 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. One of them turned up a defect in 1.87.0, fixed in 1.88.0; section 3 says what changed.

First: a durable run is one id and a few kinds of row

Durability is not a feature of the agent loop; it is a few rows the loop writes, keyed by one id, that another process can read.

A durable run is one id and a few kinds of row



`resume(execution_id)` reads the newest row: its `status` says what to do, its `node_id` says where.

✓ **Agent(checkpointer=...)** opts in; Chain checkpoints by default; Swarm and Supervisor nest under one id

✗ **no checkpointer** → nothing is written, and a tool's `interrupt()` raises instead of pausing

One `execution_id` per run. A row before each model call and before each tool. One more row, plus a pending row, for a pause. One marker at the end. Resume reads the newest row.

- **The id.** `execution_id` is minted once per run, or passed in, and placed in a context variable so every tool and every `@idempotent` function in the run reads the same one.
- **The rows.** `turn:N` before each model call; `turn:N/tool:X` before each tool, with the tool's arguments saved; `turn:N/tool:X` again, status `interrupted`, when a tool pauses; `run_end` when the run ends. Each carries the messages so far as `state_snapshot`.
- **The tables.** `checkpoints`, `pending_interrupts` and `idempotency_cache`, in SQLite by default or Postgres for a fleet.
- **The re-entry.** `resume(execution_id)` reads the newest row: its `status` says what to do, its `node_id` says where.

You opt in with one keyword:

```
agent = Agent(name="refund-bot", llm=llm, tools=[refund], checkpointer=SQLiteCheckpointer())
result = agent.run("Refund order 1042", execution_id="refund-1042")
# result.status == "paused" if the tool called interrupt(); the process can exit now.

# later, anywhere with the same store and the same tools:
result = agent.resume("refund-1042", resume_value=Resume(approved=True, metadata={"approver": "alice"}))
```

The rest of this page is what those rows have to get right.

Code: the row is `Checkpoint` in [chain/checkpoint.py](#); the store contract is [checkpointers/protocol.py](#); the context variables are in [chain/interrupt.py](#) and [chain/idempotent.py](#); the agent wires them in `_arun_core`, [agent/agent.py](#). The example is [examples/42_durability_hitl.py](#).

1 · Between one step and the next: what one run writes

A row is only useful if it was written before the thing that might kill the process. Where the rows go, and when, decides whether a crash has a re-entry point.

Between one step and the next — what one run writes

a run that completes (proof 1)

```
#0 turn:0 llm_call completed
#0 turn:0/tool:refund tool_call completed
#1 turn:1 llm_call completed
#2 run_end run_end completed
```

a run that pauses (proof 1)

```
#0 turn:0 llm_call completed
#0 turn:0/tool:refund tool_call completed
#0 turn:0/tool:refund hitl_pause interrupted
pending_interrupts: manager_approval {order, amount, balance}
```

- ✓ a row is written before the step, not after: a crash inside the step still has a re-entry point
- ✓ the pause row and the pending row commit together: the approvals UI never sees a half-suspended run
- ✗ a paused run has no run_end: a marker after an interrupted row would hide the pause from resume

what a row holds

state_snapshot: every message so far, the turn, the tool name and call id
node_input: the tool's arguments as dispatched · interrupt_context: the frozen snapshot
agent_path: agent:refund-bot/tool:refund — who wrote it, for a Swarm or Supervisor

The Local UI's checkpoint inspector shows these rows as a timeline with a state diff between any two.

A run that completes and a run that pauses, row by row. The pause row and the pending row commit together. The paused run has no end marker.

The rules:

- **A row is written before the step, not after.** The turn row goes down before the model is called; the tool row before the tool is dispatched, with its arguments. A crash inside either step still has a re-entry point.
- **A pause is two rows in one transaction:** the interrupted checkpoint and the pending_interrupts row the approvals UI reads. Nobody ever sees a half-suspended run.
- **A paused run has no run_end.** A pause is not an ending, and a marker written after an interrupted row would hide the pause from the guards that make a resume demand a Resume value.
- **Every row says who wrote it.** agent_path is agent:<name>/tool:<tool> for a plain agent and nests under a Swarm or Supervisor, so one execution_id can hold a whole topology.

Proof 1 runs a small refund that completes and a large one that pauses, on a scratch SQLite file:

```

— a run that completes: one small refund
status: completed · output: 'Refund for order 1042 is on its way.'
#0 turn:0                step=llm_call    status=completed
#0 turn:0/tool:refund     step=tool_call  status=completed
#1 turn:1                step=llm_call    status=completed
#2 run_end               step=run_end     status=completed

— a run that pauses: a large refund
status: paused · pending_interrupt: {'reason': 'manager_approval', 'node_id':
'turn:0/tool:refund', 'agent_path': 'agent:refund-bot/tool:refund'}
#0 turn:0                step=llm_call    status=completed
#0 turn:0/tool:refund     step=tool_call  status=completed
#0 turn:0/tool:refund     step=hitl_pause status=interrupted
pending_interrupts rows: [('run-large', 'manager_approval', {'order': '1042', 'amount':
50000, 'balance': 100})]
run-end rows for the paused run: []

```

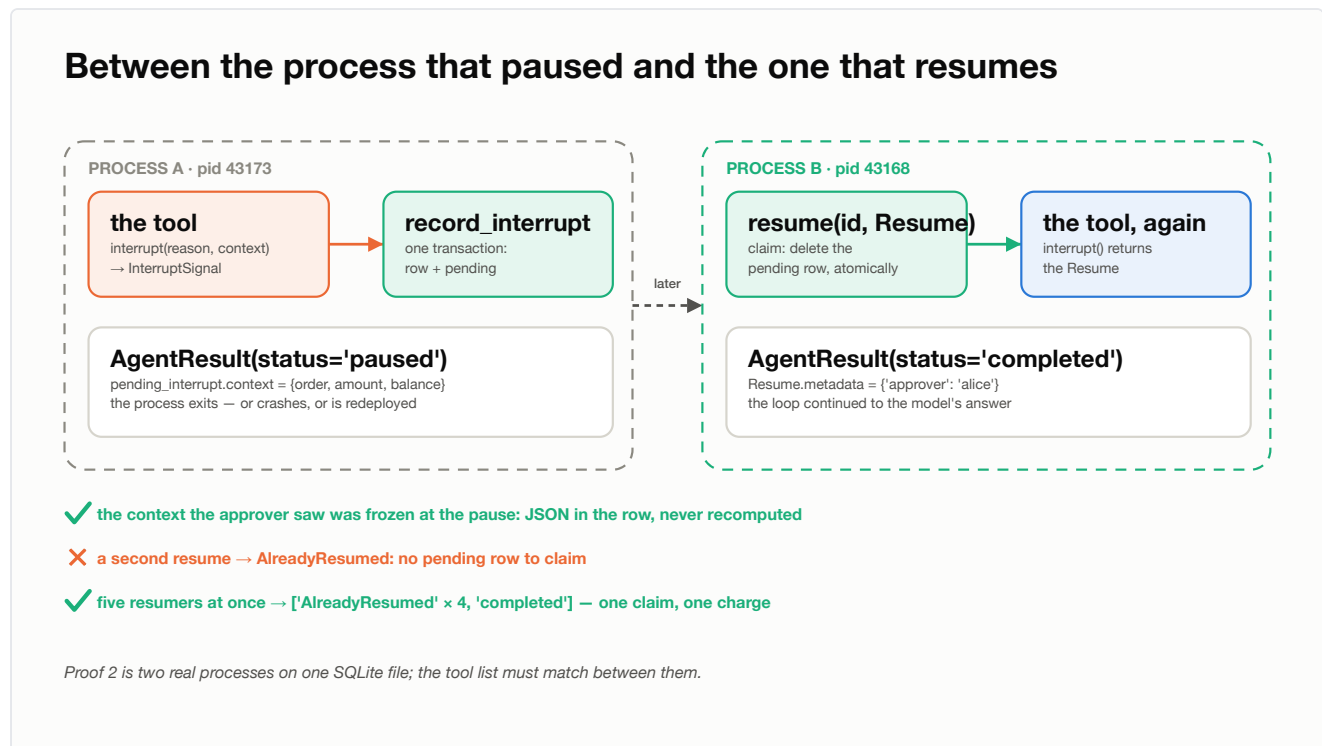
Two model turns and one tool call became four rows and a marker. The pause became one more row and a pending entry, and no marker.

The same rows in the Local UI: one per step, with its status, and the idempotency cache for the run.

Code: `_put_turn_checkpoint`, `_put_tool_checkpoint` and `_record_agent_interrupt` in [agent/executor.py](#); `record_interrupt` in [checkpointers/sqlite.py](#) and [postgres.py](#). The proof is [proof_1_rows.py](#); [tests/test_checkpointer_protocol.py](#) pins the store contract.

2 · Between the process that paused and the one that resumes

The whole point of a pause is that the process holding it can go away. The resume then happens somewhere else, later, from the rows alone, and it has to happen exactly once.



Process A pauses and exits. Process B claims the pending row, re-invokes the tool with the decision in scope, and finishes. A second resumer finds nothing to claim.

The rules:

- **`interrupt(reason, context)` raises on the first pass.** The executor records the pause and the run returns `status="paused"` with the context on it. The process can exit.
- **The context is frozen at the pause.** It is serialized into the row and never recomputed, so the approver's decision is about a snapshot, not about whatever the world looks like at resume time.
- **Resume claims the pending row atomically**, then re-invokes the suspended tool with the `Resume` in scope, so `interrupt()` returns it instead of raising. The loop continues to the answer.
- **A second resume finds no row** and raises `AlreadyResumed`. Under a race, exactly one caller claims the row; the others raise.
- **The tools must match.** Resume looks the suspended tool up by name on the agent it is called on; a missing tool raises `ChainCheckpointError` naming it.

Proof 2 pauses in a child process, resumes in the parent, resumes again, then races five threads for one pause:

```

— process A pauses and exits; process B resumes
child pid=43173 status=paused context={'order': '1042', 'amount': 50000, 'balance': 100}
this pid=43168 resumes with Resume(approved=True, metadata={'approver': 'alice'})
status: completed · output: 'Refund for order 1042 is on its way.' · charges: [{'order':
'1042', 'amount': 50000}]

— the approver's context was frozen at the pause
frozen in the checkpoint: {'order': '1042', 'amount': 50000, 'balance': 100} – JSON at pause
time, never recomputed

— a second resume is refused
AlreadyResumed: Agent execution 'refund-1' has no pending interrupt to claim – either it was
never suspended or anot ...

— five resumers race for one pause
child pid=43178 status=paused context={'order': '1042', 'amount': 50000, 'balance': 100}
outcomes: ['AlreadyResumed', 'AlreadyResumed', 'AlreadyResumed', 'AlreadyResumed',
"completed('Refund for o')"]
charges in this process: 2 – one per resumed run

```

Two processes, one SQLite file. Five threads, one claim, one charge.

Code: `interrupt`, `Resume` and `AlreadyResumed` in [chain/interrupt.py](#); the claim is `delete_pending_interrupt_atomic` in [checkpointers/sqlite.py](#); `Agent.aresume` in [agent/agent.py](#). The proof is [proof 2 pause resume.py](#); see [Suspending HITL](#) and [examples/customer-support-agent/](#).

3 · Between a crash and a rerun: resume re-enters at the newest row

A crash is a pause nobody asked for. The newest row says which step was in flight, and the rule is the same as for a pause: re-enter there, repeat nothing that was committed.

Between a crash and a rerun — resume re-enters at the newest row

killed inside the tool (exit 2)

newest row: turn:0/tool:refund · tool_call
resume → the tool runs with its saved args
model calls on resume: 1 (the next turn), charges: 1

killed waiting on the model (exit 3)

newest row: turn:1 · llm_call
resume → the model is re-issued at turn 1
model calls on resume: 1, charges: 0 — the tool is not re-run

two tools in one turn; the first pauses

turn 0: refund(1042, 50000) → pauses notify(1042) → never dispatched
resume: refund runs with the Resume; notify is NOT re-dispatched — resume re-enters at a row, never mid-turn
the model is shown '[no result: the agent was interrupted before this tool ran; it was not run on resume]'

✓ a row is written before the step, so every crash has a re-entry point; nothing committed is repeated

✗ the re-entered step runs from the top: wrap its side effects in @idempotent (next section)

✓ a tool re-invoked by crash recovery may pause: the pause is recorded like any other, and a Resume finishes the run — proof 3: paused, then completed with one charge (fixed in 1.88.0)

Proof 3 kills a real child process with `os._exit` at each point and resumes in the parent.

Killed inside the tool: the tool runs again from its saved arguments, the model is not called. Killed waiting on the model: the model is re-issued, the tool is not. A sibling tool call the pause left behind is not re-dispatched.

The rules:

- **A tool-boundary crash re-invokes the tool** with the saved arguments. The model is not re-called: the assistant's tool calls are already in the saved history.
- **A turn-boundary crash re-issues the model** at that turn, with the saved history. The tool that ran before it is not run again.
- **Sibling tool calls are not re-dispatched.** When a model asked for two tools in one turn and the first paused, resume answers the one it suspended on and tells the model the other did not run. Firing it on resume would run a side effect the model never saw a result for.
- **A re-invoked tool may pause.** If the process died before the tool reached its `interrupt()`, crash recovery re-runs it and it pauses now. The pause is recorded exactly as the loop would record it, and the run resumes from it with a `Resume`.
- **The re-entered step runs from the top.** That is why side effects need `@idempotent`, the next section.

Proof 3 kills a real child process with `os._exit` inside the tool, then while waiting on the model, and resumes each run in the parent:

```

— killed inside the tool (exit code 2)
#0 turn:0                step=llm_call    status=completed
#0 turn:0/tool:refund     step=tool_call  status=completed
resume → status=completed model calls=1 tool ran with saved args → charges=[{'order': '1042',
'amount': 20}]
#1 turn:1                step=llm_call    status=completed
#2 run_end                step=run_end     status=completed

— killed waiting on the model (exit code 3)
#0 turn:0                step=llm_call    status=completed
#0 turn:0/tool:refund     step=tool_call  status=completed
#1 turn:1                step=llm_call    status=completed
resume → status=completed model calls=1 charges=[] (the tool was not re-run)

— two tools in one turn; the first pauses
status: paused · notify ran during the first pass: []
resume → status: completed · notify ran on resume: []
tool results the model was shown at the next turn: ['{"approved": true, "charge": 1}', '[no
result: the agent was interrupted before this tool ran; it was not run on resume]']

— a tool re-invoked after a crash can pause
child exit code: 2
crash resume → status: paused · node: turn:0/tool:refund · pending: ['crash-then-pause']
approve      → status: completed · charges: [{'order': '1042', 'amount': 50000}]

```

The first resume made one model call, for the turn after the tool; the charge happened once. The second made one model call and no charge. The sibling `notify` never ran, and the model was told so in the tool result.

The last case is a large refund killed before its tool reached `interrupt()`. Crash recovery re-ran the tool, the tool paused, and the pause was recorded with its pending row; approving it finished the run with one charge.

!!! note "Changed in 1.88.0" Until 1.88.0 that last case raised `InterruptSignal` out of `resume()`: the tool loop converted the signal into a recorded pause, but the resume path's direct re-invocation did not, so nothing was checkpointed and every later resume re-ran the tool and raised again.

Code: the three resume shapes are `Agent.aresume` in [agent/agent.py](#); the sibling note is `balance_tool_messages` in [llm/message.py](#). The proof is [proof 3 crash.py](#); [tests/test_resume_crash_then_pause.py](#) pins the crash-then-pause case with a real killed process; see [Resume shapes](#) and [Sibling tool calls](#).

4 · Between a resume and a side effect: the re-executed step

A resumed step runs again from the top. Everything in it before the pause runs twice unless something remembers it already ran. That something is a cache keyed by the run.

Between a resume and a side effect — the re-executed step

the tool, as written

```
def refund(order, amount):  
    receipt = charge_card(order) # side effect  
    decision = interrupt(...) # pause here  
    return {approved, **receipt}
```

paused, then resumed

first pass: charge_card → interrupt raises
resume: charge_card → interrupt returns
plain function: fired 2× · @idempotent: fired 1×

idempotency_cache

(execution_id, key) → JSON result

key = sha256(qualname + args + kwargs), or key_fn=
a hit returns the stored JSON, the body never runs
a new execution_id is a miss: fired 1× again
outside any run: no cache, fired every time

✗ a return that is not JSON-serializable raises `IdempotencyError` inside the tool — refused, not silently uncached

✓ the cached value comes back as plain data (dicts, lists); return plain data from idempotent functions

Proof 4: the same tool paused and resumed, plain and wrapped: ['1042', '1042'] versus ['1042'].

A charge before the pause fires on both passes. Wrapped in `@idempotent`, the second pass returns the cached result and the body never runs. The cache is per run.

The rules:

- **A plain side effect fires again on resume.** The tool is re-invoked from the top, and nothing in the SDK knows which line is safe to repeat.
- **@idempotent caches by (execution_id, key).** The key is a hash of the function's name and arguments, or your `key_fn`. A hit returns the stored result and never runs the body.
- **The cache is per run.** A new `execution_id` is a miss. Outside any run, there is no cache and the body runs every time, which is what makes wrapped functions safe to unit-test.
- **A result that cannot be stored is refused.** The value is stored as JSON; a return that cannot be serialized raises `IdempotencyError` rather than silently going uncached.

Proof 4 pauses and resumes the same tool, plain and wrapped:

```
— the same tool, paused and resumed  
plain function      status=completed charge_card fired 2× → ['1042', '1042']  
@idempotent         status=completed charge_card fired 1× → ['1042']  
  
— the cache is per run  
@idempotent, new run status=completed charge_card fired 1× → ['1042']  
  
— outside a run there is no cache  
called twice outside any run → fired 2 ×  
  
— a value that cannot be stored is refused  
the tool raised inside the loop; the model was told: "Tool 'refund' failed: @idempotent  
function 'opaque' returned a non-JSON-serializable value of type 'object': Unable to s" ...
```

Same tool, same pause, same resume. The card was charged twice without the decorator and once with it.

Code: `chain/idempotent.py` ; the cache rows are `get_idempotent / put_idempotent` in each checkpoint. The proof is `proof 4 side effects.py` ; `tests/test_idempotent.py` pins the scoping; see [Side effects & idempotency](#).

5 • Between a run that ended and one that died: the marker

Until 1.65.0 a run that finished and a run that died the instant after its last step left the same rows: a `completed` checkpoint as the newest. Resuming a finished run re-ran it. The run-end marker is what tells the two apart, and it means the opposite thing to a resume and to a fork.

Between a run that ended and one that died — the marker

run_end · completed

resume → `AlreadyResumed`
fork → a new `execution_id`,
the original untouched

run_end · failed

the provider raised after the tool
resume steps over the marker and
re-issues the model: `completed`

no marker · paused

a pause is not an ending
resume needs a `Resume` value
prune never deletes it

Before the marker a finished run and a run that died after its last step were byte-identical, and resuming a finished run re-ran it — re-calling the model and re-firing every side effect not wrapped.

proof 5

```
done-1 last row: #2 run_end completed resume → AlreadyResumed fork → new id, rows 4 → 4
failed-1 last row: #2 run_end failed resume → completed, charges: 1 (the tool did not run again)
fork(checkpoint_id=<the marker>) → ChainCheckpointError: the marker is not a step
prune(older_than=0s) deleted 17 rows · still stored: ['paused-1'] · pending: ['paused-1']
```

✓ the marker sorts last under both orderings the SDK uses, and carries the final state so the plane's /latest is useful

✗ a tool that raises is not a failed run: its error is a tool result the model reads; only an escaped exception fails the run

`completed` : resume is refused, fork is the ordinary case. `failed` : resume steps over the marker. `Paused`: no marker, and prune leaves it alone.

The rules:

- A **completed** marker refuses a resume with `AlreadyResumed` : there is nothing to continue, and continuing would re-fire every side effect.
- A **failed** marker is stepped over. Crash recovery is the feature; a run that raised stays resumable, and resume re-enters at the last real row.
- A fork branches a finished run under a new `execution_id` , leaving the original untouched. A fork from the marker itself is refused: it is a tombstone, not a step.
- A tool that raises is not a failed run. Its error becomes a tool result the model reads. Only an exception that escapes the run writes a `failed` marker.
- Prune never deletes a pause. `completed` and `failed` rows older than the cutoff go; an `interrupted` row stays, because deleting it would orphan a human.

Proof 5 finishes a run, fails one by making the provider raise once, forks the finished one, and prunes:

```
— a finished run refuses to resume
last row: #2 run_end          step=run_end    status=completed
resume → AlreadyResumed: Agent execution 'done-1' already finished (completed) – there is
nothing to resu ...

— a run that raised stays resumable
run raised: RuntimeError · provider returned 503
last row: #2 run_end          step=run_end    status=failed
resume → status: completed · charges: [{'order': '1042', 'amount': 20}] (the tool did not run
again)

— a fork branches a finished run under a new id
fork → execution_id='40229800-d3c8-40f7-83fc-3706c4c6860c' status=completed · original rows:
4 → 4
fork from the marker → ChainCheckpointError: Checkpoint 'c9bab603-...' of execution 'done-1 ...

— prune keeps every pause
prune(older_than=0s) deleted 17 rows · still stored: ['paused-1'] · pending: ['paused-1']
```

The failed run re-entered after the tool, so the charge stayed at one. The fork left the original's four rows exactly as they were.

Code: `write_run_end`, `latest_resumable` and `latest_forkable` in [chain/checkpoint.py](#); `prune` in each checkpointer; `Agent.afork` in [agent/agent.py](#). The proof is [proof_5_run_end.py](#); [tests/test_run_end_checkpoint.py](#) pins the marker; see [When checkpoints are written](#).

6 · Between a laptop and a fleet, and the plane

The same `Agent` runs on SQLite or Postgres; only the checkpointer changes. Both implement one protocol, and the one operation that must be exactly-once is the claim. Connected, a checkpointer also keeps an outbox to the plane, and the plane never runs a step.

Between a laptop and a fleet, and the plane

SQLiteCheckpointer

one file · one process at a time
claim: SELECT + DELETE under a lock
race of 5 → 1 completed, 4 AlreadyResumed

PostgresCheckpointer

many workers · a schema per app
claim: DELETE ... RETURNING (MVCC)
race of 5 → 1 completed, 4 AlreadyResumed

your own backend

the Checkpointer protocol: ~12 methods
ReplicatedCheckpointer is optional
and separate, so nothing breaks

THE OUTBOX — when connected

write locally, synced=0

background drain → POST /checkpoints/ingest

2xx → synced=1

the plane serves

no age or count bound: an un-acked row for a paused run is never dropped · a payload refusal is quarantined,
not left to strand the queue · disconnected: fetch_unsynced() shows the rows, nothing leaves the machine

✓ `restore_from_plane()` pulls the latest row down so a normal resume proceeds on a machine that never ran it

✗ the plane serves and records; it never executes a step and never resolves a pause

Proof 6 ran both backends offline: five rows waited un-acked, none was sent, mark_synced emptied the outbox.

Two backends, one protocol, one claim each. The outbox replicates rows in the background and marks them synced only when the plane confirms. Disconnected, nothing leaves.

The rules:

- **SQLite claims under one lock** with a select and a delete in one transaction; one process at a time. **Postgres claims with `DELETE ... RETURNING`** under MVCC, so many workers can race and exactly one wins.
- **The protocol is small and the replication surface is separate.** `Checkpointer` is what the agent needs; `ReplicatedCheckpointer` is optional, so a custom backend without it simply does not replicate.
- **The outbox is write-driven and non-blocking.** A row is written locally as `synced=0`; a background drain posts un-acked rows to the plane and marks them synced only after a 2xx. There is no age or count bound: a paused run's rows are never dropped for being old.
- **One deliberate loss.** A row the plane refuses on payload grounds is quarantined with its reason rather than left to strand every later row behind it.
- **Disconnected, nothing leaves.** The rows wait. `restore_from_plane()` pulls the latest row down so a resume can proceed on a machine that never ran the original.

Proof 6 runs the pause, the race and the outbox on both backends, offline:

```

— sqlite: pause, race, claim
  protocol: Checkpointer=True Replicated=True
  outcomes: ['AlreadyResumed', 'AlreadyResumed', 'AlreadyResumed', 'AlreadyResumed',
'completed']
  claim    : SELECT + DELETE under one lock

— sqlite: the outbox, disconnected
  un-acked rows for this run: 5 (syncd only after the plane answers 2xx; none sent – not
connected)
  after mark_synced: 0 waiting

— postgres: pause, race, claim
  protocol: Checkpointer=True Replicated=True
  outcomes: ['AlreadyResumed', 'AlreadyResumed', 'AlreadyResumed', 'AlreadyResumed',
'completed']
  claim    : DELETE ... RETURNING

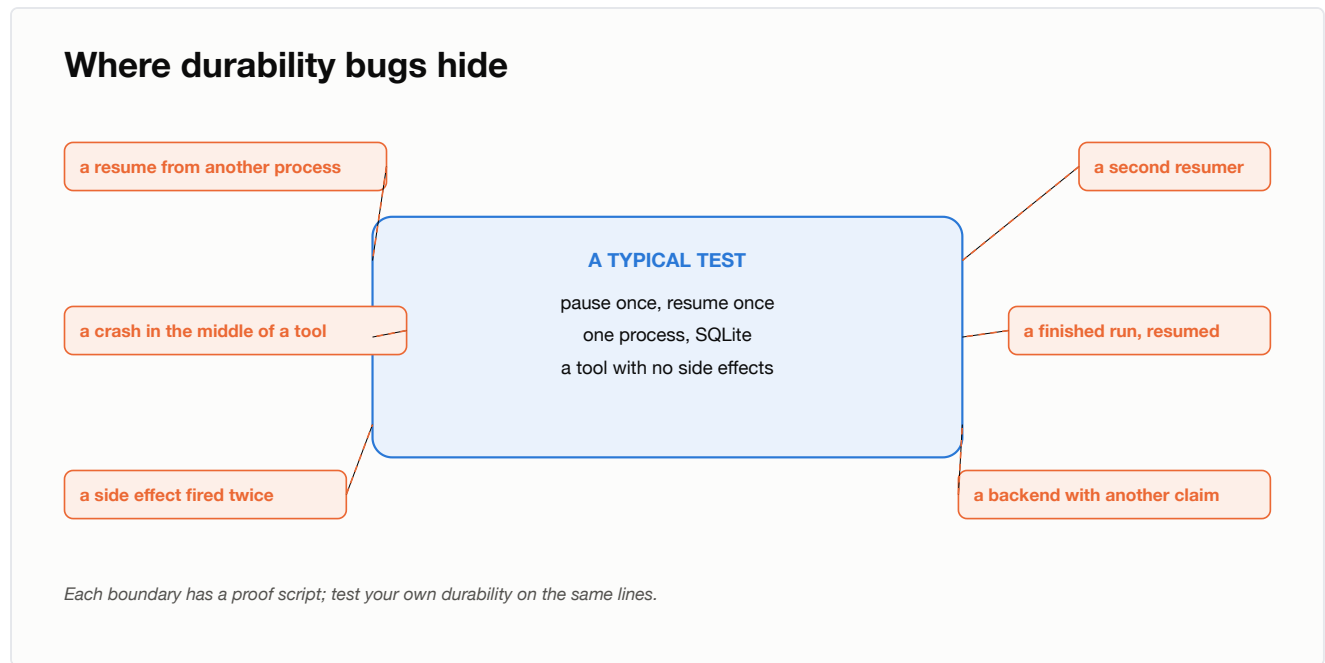
— postgres: the outbox, disconnected
  un-acked rows for this run: 5 (syncd only after the plane answers 2xx; none sent – not
connected)
  after mark_synced: 0 waiting

```

Same code, same outcome on both. Five rows waited un-acked on each backend and none was sent, because no plane was connected; that is the local-first half of the design, and the half this page could prove without one.

Code: [checkpointers/sqlite.py](#), [checkpointers/postgres.py](#) and the outbox in [checkpointers/platform_replica.py](#). The proof is [proof_6_backends.py](#) (Postgres when PG_TEST_DSN is set; `scripts/dev_backends.sh up` starts one); the backend suites are [tests/integration/test_postgres_checkpointer.py](#) and [tests/test_checkpoint_quarantine.py](#); see [Checkpointers](#), [Connected checkpoints](#) and [examples/86_connected_durability.py](#).

Where durability bugs hide



A typical test pauses once and resumes once, in one process, on SQLite, with a tool that does nothing. Durability breaks on the lines around that box.

If you build or buy durability, test the boundaries:

1. **Pause in one process and resume in another**, from the rows alone. Then resume again.
2. **Kill the process inside a tool**, and again while it waits on the model. Count the model calls and the side effects on resume.
3. **Put a side effect before the pause** and resume. Did it fire twice?
4. **Race five resumers** for one pause, on every backend you ship.
5. **Resume a run that finished**, and one that raised. Which one re-runs?
6. **Disconnect** and write a few rows. What is waiting, and did anything leave?

The proof scripts behind this page run each of those checks.

What durability still won't do for you

- **Writes are synchronous.** `aresume()` is async for symmetry; the store calls block.
- **Resume needs the same tools.** A fresh agent must register the tools the paused run used, by name.
- **A tool's error is not a failed run.** Only an exception that escapes the run writes a `failed` marker.
- **A re-entered step runs from the top.** `@idempotent` absorbs the side effects you wrap; it cannot find the ones you didn't.

- **Resume carries approved and metadata , nothing else.** A `data=` keyword is dropped silently; put payloads in `metadata` .
 - **No watchdog, no auto-resume.** A pause waits for a human, or for your scheduler. It never expires on its own.
 - **SQLite is one process at a time.** Two replicas need Postgres.
 - **The plane serves and records.** It never executes a step and never resolves a pause; your application is the approver.
-

The point of all of it

Durability is a promise about a process that is no longer there. That is why it has to be both visible and trustworthy: the rows make a run visible; this page walked the six boundaries where a resume has to hold.

The split between the SDK and the plane stays the same. The SDK writes the rows in your store, claims the pause in your process, and resumes on your machine. The plane keeps a durable copy and serves it back; it never runs a step. Everything on this page, except the plane itself, is in the open-source SDK.

Test your own durability the same way. The bugs aren't in the happy pause; they're on the boundaries.

See also

- [Concepts & Mental Model](#): the checkpoint record, when it's written, how resume decides, the atomic claim.
- [Durability](#) and [Quickstart](#).
- [Side effects & idempotency](#), [Checkpointers](#), [Multi-agent](#), [Connected checkpoints](#).
- [Agent Durability](#) and [Human-in-the-Loop](#).
- [Checkpoint inspector](#): the timeline and state diff in the Local UI.
- [Agent Memory Breaks at the Boundaries](#): the same approach for memory.