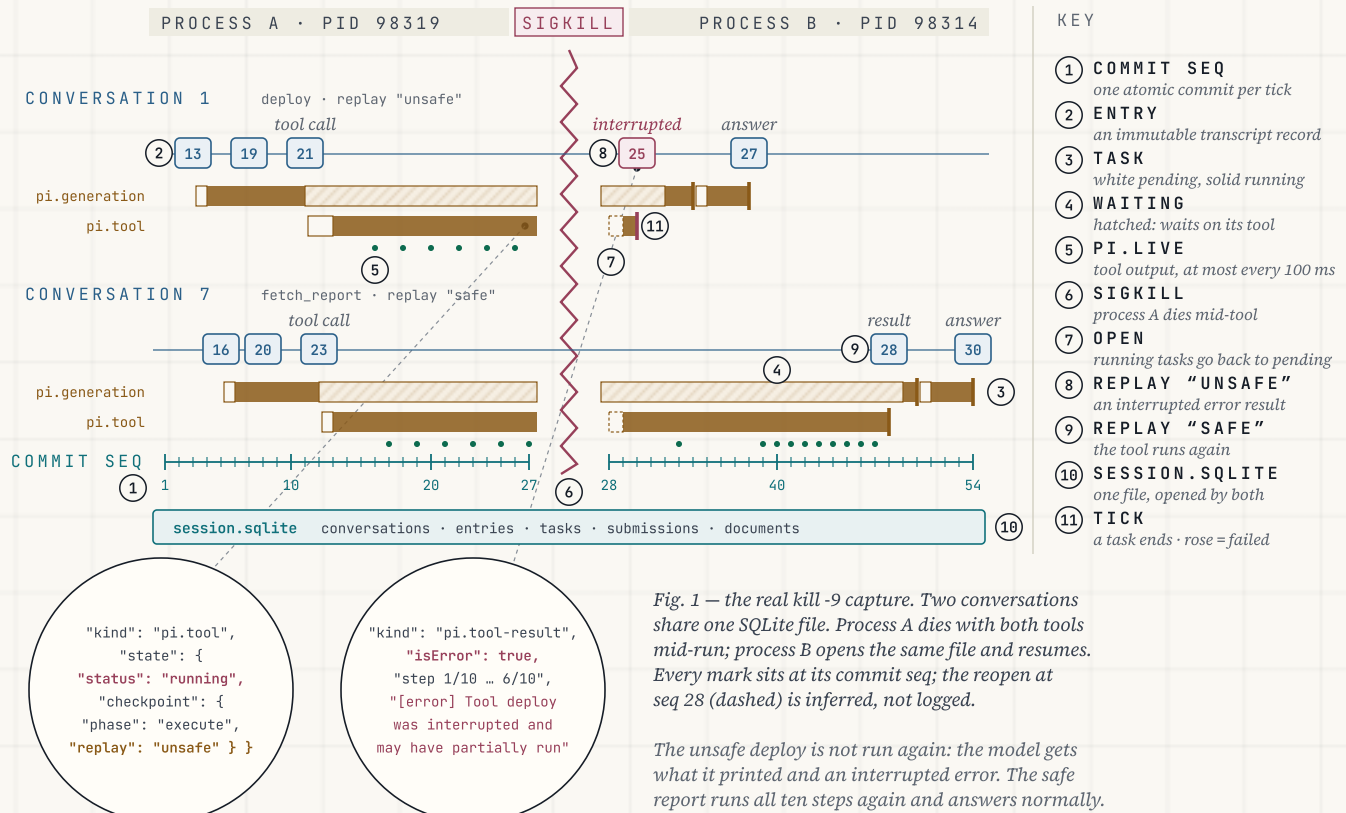


PLATE I · A CRASH, AND WHAT SURVIVES IT

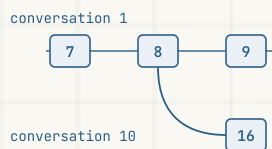
TWO CONVERSATIONS · ONE SQLITE FILE



TASK 22 AT THE KILL
open() sets its status to "pending"

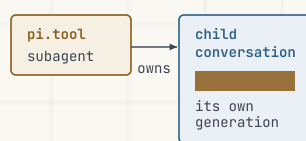
ENTRY 25 ON RESUME
written by task 22 at seq 30

FIG. 2 · FORKS



parent: { conversationId: 1, at: 8 }
A fork reads its parent's entries up to 8, then writes its own.

FIG. 3 · SUBAGENTS



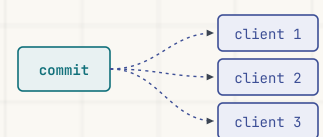
ownership: { kind: "task", taskId }
Abort the call and its child conversation is aborted too.

FIG. 4 · COMPACTION



context: [20, 18, ...]
The model sees the summary, then entry 18 onward.

FIG. 5 · LIVE WATCHERS



["a", [... "thinking", "..."]]
Every client applies the same ops, one frame per commit.

Pi Durable

Technical Manual

The durable agent harness from the first idea down to the stored rows: atomic commits, tasks that checkpoint every phase, tool replay after a crash, forks, subagents, compaction, live watchers, and the SQLite and JSONL storage underneath.

Contents

1 The Model

1.1	What Pi Durable is for	10
1.2	The architecture in one page	13
1.3	The core rule and its invariants	16
1.4	The life of one answered input	19

2 Sessions and Commits

2.1	Records and IDs	25
2.2	The commit and the mutation line	29
2.3	When a commit fails	33

3 Conversations and Context

3.1	Entries and the built-in kinds	37
3.2	From transcript to model context	40
3.3	Forks and owned conversations	43
3.4	Reset, handoff and heads	46

4 Documents

4.1	Defining a document	50
4.2	Access, creation and lifetimes	53
4.3	Bases, deltas and migrations	56
4.4	The built-in documents	60

5 Tasks

5.1	Tasks as state machines	63
5.2	Phases, steps and progress	67
5.3	The effect sandwich	70
5.4	The scheduler	73
5.5	Ownership and structured concurrency	76
5.6	Abort, fault and orphan	80

6 Runs

6.1	Submissions and the inbox	84
6.2	Run control and the live document	88
6.3	Generation	91
6.4	Tool calls and replay	94
6.5	Compaction	97

6.6	Usage and cost	100
7	Extensions and the Agent	
7.1	Extensions and the registry	103
7.2	The per-conversation agent	106
7.3	Hooks	109
7.4	Defining tools	112
7.5	The system prompt	115
7.6	Reload while running	118
7.7	Subagents	121
8	Watching	
8.1	Chord and document state	125
8.2	The conversation view and watch()	129
8.3	Agent events	133
8.4	The task graph and inspect()	137
9	Storage and Environment	
9.1	The storage contract	141
9.2	SQLite	144
9.3	JSONL	148
9.4	Memory, conformance and portability	152
9.5	The execution environment	155
10	In Practice	
10.1	Building a coding agent	159
10.2	Contracts that bite	163
10.3	Limits and non-goals	168
R	Reference	
R.1	Harness and Conversation API	172
R.2	Built-in kinds	179
R.3	Settings and defaults	180
R.4	Errors and reasons	183
R.5	Glossary	185
R.6	Sources	187
R.7	Index of figures	189

How to read this book

This is a map of one package at one commit, and every record, file and event in it was produced by running that package.

This book documents Pi Durable, the npm package `@earendil-works/pi-durable` at version 1.0.3, from repository `earendil-works/pi` at commit `5b6c792` (5 October 2026). It follows the harness from its one rule down to the rows in a SQLite file, and every record shown was captured from a real run.

Who this is for

You are a TypeScript engineer who has built or used an LLM agent: a loop that sends a transcript to a model, runs the tool calls it returns, and sends the results back. You may have used the Pi coding agent. You have not read the Pi Durable source or its spec. After this book you can design a task with the right phases and predict what survives a crash at any point in a run. You can also read a storage dump line by line, write a tool that is safe to rerun, and find out why a run is stuck.

EXPERIMENTAL

The README opens with one line of status: “The API changes without notice between releases.” Everything here is true of 1.0.3. Check the CHANGELOG before you rely on a detail in a later version.

How it was made

Nothing here was written from memory. Each section closes with a **Sources** line naming the files it was written from. Paths are relative to `research/pi/packages/durable/` unless they start with `research/`. The sources rank in this order of authority:

SOURCE	PATH	USED FOR
Source code	<code>src/**</code>	every type, default, field name and behaviour; cited as <code>src/harness/tool.ts</code>
Spec	<code>docs/spec.md</code>	every rule, quoted exactly; cited as <code>spec §5.2</code>
README, changelog	<code>README.md</code> , <code>CHANGELOG.md</code>	the public surface as the authors present it
Design notes	<code>docs/pico-v5-handoff.md</code> , <code>docs/pico-v5-chord-usage.md</code>	why a rule exists
Tests	<code>test/*.test.ts</code>	exact behaviour at the edges
Examples	<code>test/examples/NN-*.ts</code> , <code>research/runs/</code>	runnable code and its output; cited as <code>ex 24</code> , <code>run 24</code>
Capture kit	<code>research/capture/</code>	every record, file, event and transcript shown

The source wins over the spec. Where the two disagree, the text says so. The spec calls the system by its internal name, **Pico5**, and predates the code: §2.2 still says “Pico5 is not implemented yet”. This book uses the package’s name, Pi Durable, everywhere else. A claim that could not be traced to one of these files was deleted.

The capture kit

The records in this book come from scripts that drive the package and write what it stores. They live in `research/capture/` and run against the faux provider of `@earendil-works/pi-ai`, so they need no network and no API key. One script answers one input with one tool call and dumps every SQLite row and every JSONL file. Another kills a process in the middle of two tool calls

and records what a second process finds. Others record task transitions, watch operations, agent events, inbox steps and derived context. Each script runs from the package directory like an example, and a listing head names the capture file it was cut from.

Ids and commit numbers in a capture are stable from run to run. Timestamps, process ids and UUIDs are not. One thing can shift: a throttled progress commit can land one commit earlier or later, which moves later sequence numbers by one

`research/capture/README.md`.

Conventions

Methods, options, fields, kinds, statuses and reasons are set in `code`. A source cited mid-sentence is set as `spec §4` or `src/session/session.ts`. Cross-references such as [the effect sandwich \(p. 70\)](#) carry the printed page number.

Records are shown as the storage holds them: pretty-printed JSON, two-space indent, keys in stored order. A listing that cuts something marks the cut with `...` and says what was cut in the note below it.

The spec writes its rules as plain statements, without RFC 2119 capitals; a lowercase “must” in it is still a rule. The book quotes such rules word for word.

A first conversation you can run

This small app asks one question, closes its SQLite store, then reopens it and reads the saved answer. A fake model always replies “Paris.”, so you need no API key. Use Node 24, save the listing as `first-conversation.mjs`, and run it in a new folder.

```
$ npm init -y
$ npm install @earendil-works/pi-durable@1.0.3 \
  @earendil-works/pi-ai@1.0.3 @earendil-works/chord@1.0.3
$ node first-conversation.mjs
```

```
import { BACKGROUND_CONTEXT } from '@earendil-works/chord/context';
import { createModels } from '@earendil-works/pi-ai/models';
import {
  fauxAssistantMessage, fauxProvider,
} from '@earendil-works/pi-ai/providers/faux';
import { createRegistry, Harness } from '@earendil-works/pi-durable';
import { openNodeSqliteStorage } from '@earendil-works/pi-durable/storage/sqlite/node';

const context = BACKGROUND_CONTEXT;
const faux = fauxProvider();
const models = createModels();
models.setProvider(faux.provider);
faux.setResponses([fauxAssistantMessage('Paris.')]);
const options = { models, registry: createRegistry() };
const path = process.env.DEMO_DB ?? './conversation.db';

let harness = await Harness.open(await openNodeSqliteStorage(path), options, context);
const root = await harness.root(context, {
  agent: { model: { provider: 'faux', modelId: 'faux-1' } },
});
const input = await root.submit({
  type: 'input', content: 'Capital of France?', requestId: 'capital-1',
}, context);
const result = await input.wait(context);
if (result.status !== 'done') throw new Error('The input was not answered');
await harness.close(context);

harness = await Harness.open(await openNodeSqliteStorage(path), options, context);
const reopened = await harness.root(context);
const page = await reopened.entries({}, 10, undefined, context);
const answer = page.items.find((entry) => entry.id === result.answer);
if (answer?.kind !== 'pi.assistant') throw new Error('Saved answer missing');
console.log('saved answer:', JSON.stringify(answer.model[0]?.content));
await harness.close(context);
```

Complete example, tested against the pinned package build. The faux provider exercises the real harness and SQLite store; it does not call a remote model.

```
saved answer: [{"type":"text","text":"Paris."}]
```

The database keeps the conversation after this process exits. Run the file again: the same `requestId` returns the saved submission rather than asking twice. This checks storage and reopen; it does not simulate a crash during a tool effect.

Three things to notice: `models` supplies the model; `registry` holds extension code; `context` carries cancellation. `root` is the conversation you submit to. The remaining chapters explain what those calls save and how interrupted work is recovered.

`MemoryStorage`, used in later short snippets, keeps everything only in RAM. Use a persistent backend such as SQLite when you want to reopen saved work. Reinstall the same extension and task code before resuming unfinished work.

Sources: ex 14; `src/storage/sqlite/node.ts`; `src/harness/harness.ts`; `manual/examples/first-conversation.mjs`

Running the examples

The package ships runnable examples in `test/examples/`, numbered 00 to 31. The low numbers use the layers underneath the Harness; from 14 on they build real agents. Run one from the package directory; most use the faux provider and need no key

README §Examples:

```
$ node --conditions=source --experimental-strip-types test/examples/14-chat.ts
answer: [ { type: 'text', text: 'Paris.' } ]
transcript: [ 'pi.user', 'pi.system', 'pi.assistant' ]
```

The output of every example is kept in `research/runs/`, and the book cites it as `run 14`. Where a section shows a transcript, it is one of those files or a capture, never a reconstruction.

How the parts are ordered

The order runs from the idea to the bytes and then out to practice. Part 1 gives the whole model in plain words; each later part zooms in on one layer of it.

PART	WHAT IT COVERS
1 · The Model	What the harness is for, its layers, the core rule and its eight invariants, and one answered input from end to end. Read it once, completely.
2 · Sessions and Commits	Records and ids, the commit and the mutation line, and what happens when a commit fails.
3 · Conversations and Context	Entries and their kinds, how a transcript becomes model context, forks, resets and heads.
4 · Documents	Defining typed state, its scopes and lifetimes, bases and deltas, and the five built-in documents.
5 · Tasks	Tasks as state machines, the step rules, the effect sandwich, the scheduler, ownership, and abort.
6 · Runs	Submissions and the inbox, the live document, generation, tool calls, compaction, and usage.
7 · Extensions and the Agent	The registry, the per-conversation agent, hooks, tools, the system prompt, reload, and subagents.
8 · Watching	Chord state, the conversation view and <code>watch()</code> , agent events, and the task graph.
9 · Storage and Environment	The storage contract, SQLite, JSONL, conformance, and the execution environment.
10 · In Practice	Building a coding agent, the contracts that bite, and the limits and non-goals.
R · Reference	The API, built-in kinds, settings and defaults, errors and reasons, a glossary, the sources, and an index of figures.

Colour in figures

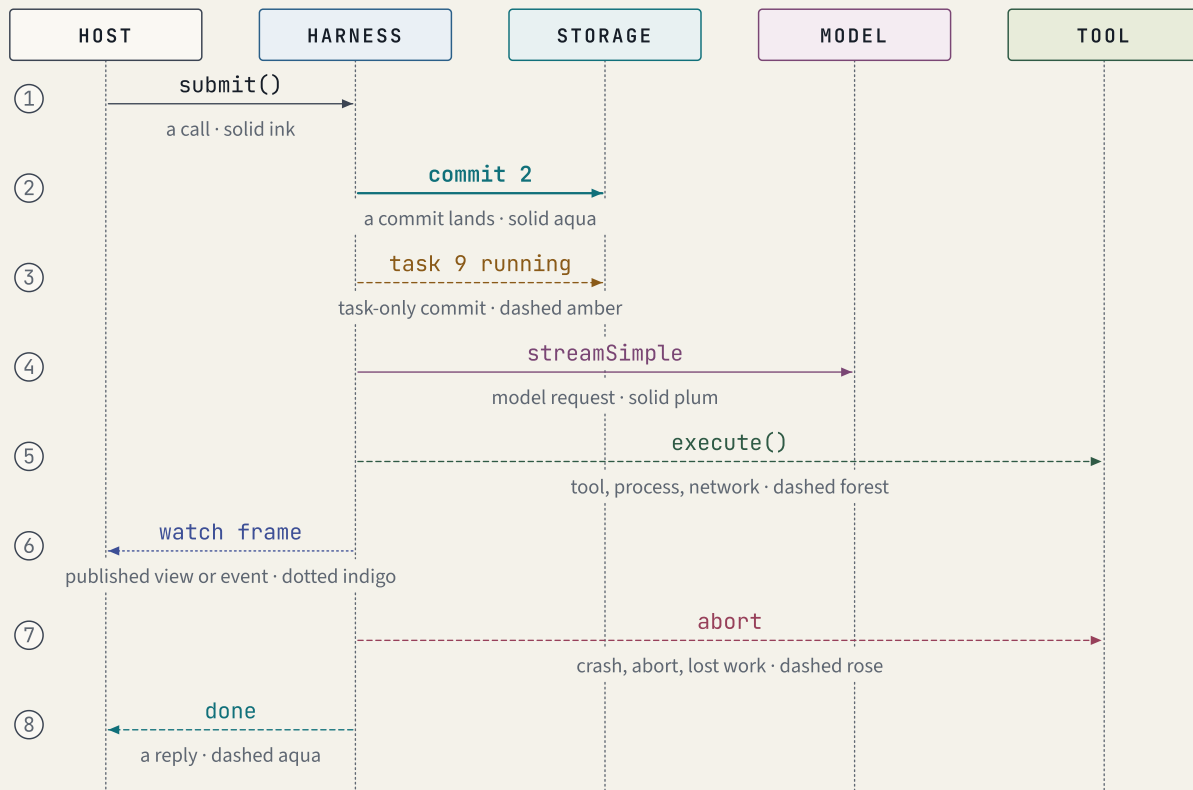
Every figure uses one colour code, and a colour means the same thing everywhere.

COLOUR	MEANS	COLOUR	MEANS
blue	conversations, entries, the transcript	plum	the model, generation, the provider
aqua	the Session, commits, storage records	forest	tools, the environment, external effects
violet	submissions and the inbox	indigo	views, watches, events, observers
green	documents and Chord state	rose	crash, abort, fault, error, lost work
amber	tasks, phases, checkpoints	grey	the host application; hatched, outside the active range

Sequence figures put each party on a vertical lifeline, with time running down the page. The arrow style says what kind of exchange it is. The arrow a figure is about is drawn heavier, with a bold label. Numbered circles in the left margin are step numbers; in figures of a captured run they are the commit's sequence number in storage.

FIG. 0.1 READING A SEQUENCE FIGURE

SEQUENCE



Every exchange in the book is drawn with the same eight arrow styles. Arrows into Storage are commits; an amber one is a commit whose only write is a task changing state. Labels follow the one-turn capture shown in 1.4 (p. 19) where it has the exchange.

Sources: package.json (name, version, engines); README.md (status line, §Examples); CHANGELOG.md §1.0.3; docs/spec.md (title, §1); research/capture/README.md; research/capture/scripts/sqlite-one-turn.ts; manual/AUTHORING.md (colour table)

The Model

An agent whose every visible step is a stored commit, so a process that dies mid-turn is just a pause.

1.1	What Pi Durable is for	10
1.2	The architecture in one page	13
1.3	The core rule and its invariants	16
1.4	The life of one answered input	19

01

What Pi Durable is for

An agent process can die at any instruction; Pi Durable makes every visible step a stored commit, so a death mid-turn is a pause, not a loss.

Your agent is halfway through a deploy tool when the laptop lid closes. When the process comes back, what did it finish? Pi Durable records what it knew before the interruption. The tool's replay policy decides whether to run it again or report that it may have partially run. This section shows what goes wrong without it, the three moves it makes, and what it deliberately does not do.

The problem: a turn has four kinds of state

Agent processes die for ordinary reasons: a deploy replaces them, the kernel kills them for memory, a laptop sleeps. A death in the middle of a turn catches four kinds of state in flight:

- the **transcript**, the messages so far;
- the **model call**, a response half streamed;
- the **tool call**, an effect half done;
- the **application state**, such as a todo list the tools were updating.

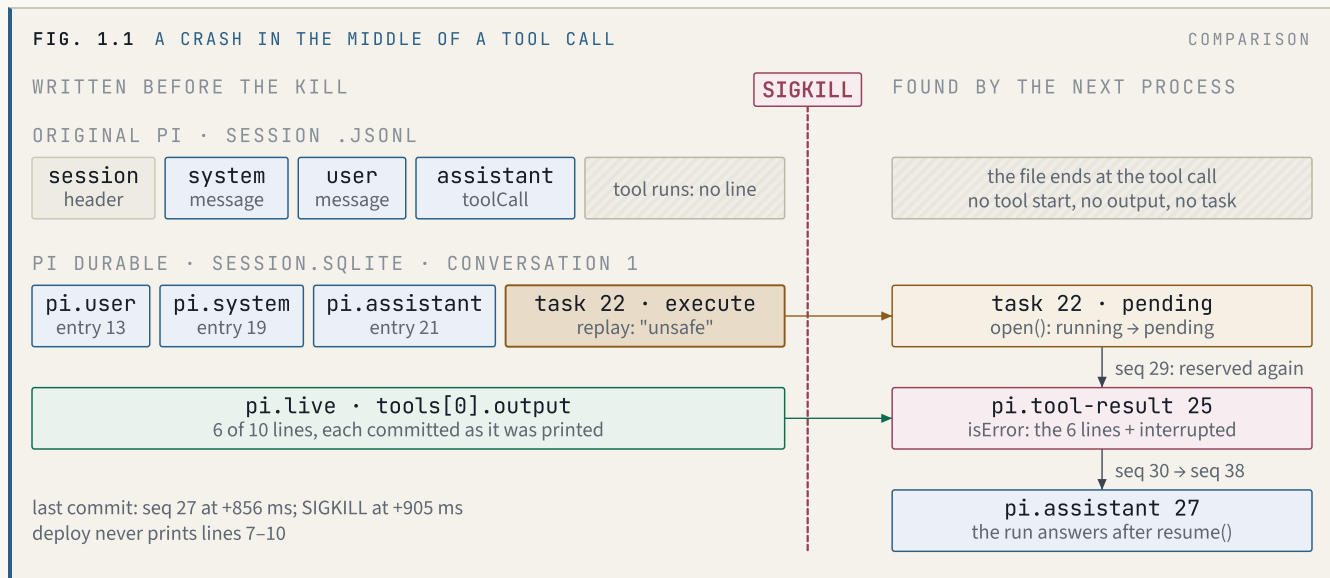
Each lives somewhere different. After the crash they disagree, and nothing records how far each one got. Earendil's announcement names the goal: agents that survive crashes, redeploys and outages, and resume interrupted work instead of restarting it earendil.com/posts/pi-durable.

What original Pi keeps

The Pi coding agent already persists its sessions. A session is a JSONL file under `~/.pi/agent/sessions/`. Each line is one JSON object with a `type`; the first is a session header, and the rest are entries that form a tree through `id` and `parentId` [coding-agent docs/session-format.md](#). The tree gives Pi in-place branching: `/tree` moves to an earlier entry without erasing the branch you leave [coding-agent docs/sessions.md](#).

Lines are appended as the agent loop produces them. A user message, an assistant message with its tool calls, and a tool result are each one `message` line [session-format.md §Entry Types](#). The loop records a response once the provider has finished streaming it, then runs each tool call and records the results [coding-agent docs/how-pi-works.md §Agent loop](#).

That design keeps the conversation. It does not keep the turn. If the process dies while a tool runs, the file ends at the assistant message that called it. No line says the tool started or what it printed. A half-streamed response is not in the file, and state an extension held only in memory is gone. The next process can read the conversation back, but it cannot tell whether the tool's effect happened.



A crash mid-tool leaves original Pi a file that ends at the call; Pi Durable reopens an interrupted task. Left of the dashed line is what was written before SIGKILL; right is what the next process finds. The Pi Durable rows are conversation 1 of the crash run in `research/capture/crash-*.txt`, where `deploy` printed 6 of its 10 lines. The original Pi row follows `session-format.md`.

Pi Durable's answer

Pi Durable is a harness: one open storage plus the machinery that runs agents on it. It makes three moves, and each closes one of the gaps above.

Everything visible is committed — “Conversations, model turns, tool calls, and your own state are committed to storage before anything is shown” [README](#). A half-streamed answer and a tool's running output are committed too, with a default minimum interval of 100 ms between progress commits. A crash loses progress since the last successful commit; storage delays and output pacing can make that gap longer [README](#) §[Watching a Conversation](#).

Work is tasks with checkpoints — a model call and a tool call each run as a **task**, a durable state machine that commits a checkpoint at every step. A tool task commits its intent, the final arguments and a replay policy, before `execute()` runs `src/harness/tool.ts`. On reopen the harness knows exactly which calls had started.

App state is documents in the same commit — a **document** is typed JSON stored next to the transcript. It changes in the same atomic commit as the entries, so a todo list can never be ahead of the message that changed it [README](#) §[Your Own State](#).

The crash run shows the result. A second process reopened the SQLite file. Opening alone turned the running tool task back to pending. On `resume()`, the `deploy` tool, which was not declared safe to rerun, got a `pi.tool-result` entry holding its six committed lines and an error:

tool result written on resume, excerpt `research/capture/crash-after.txt`

JSON

```
{
  "role": "toolResult",
  "toolCallId": "deploy-1",
  "toolName": "deploy",
  "content": [
    { "type": "text", "text": "deploy site: step 1/10 (pid 98319)\n...step 6/10 (pid 98319)\n" },
    { "type": "text",
      "text": "<harness>\n[error] Tool deploy was interrupted and may have partially run\n</harness>" }
  ],
  "isError": true,
  "timestamp": 1791211977057
}
```

The model message of entry 25. The first text block, cut here, holds all six output lines from the killed process. The entry's `data.diagnostics` carries the same error with code: "interrupted".

The model then read that result and answered, and the input settled done. In the same crash, a `fetch_report` tool declared `replay: "safe"` simply ran again from step 1 in the new process `research/capture/crash-README.md`. The next process did not guess. It read a checkpoint that said “this effect may have happened” and applied a policy chosen in advance. The effect sandwich (p. 70) and tool calls and replay (p. 94) take that apart.

What it is not

The scope is narrow on purpose. Three limits matter from the start.

One process per storage — “One process owns a storage at a time; there is no cross-process locking” `README §Storage`. A Harness is not a cluster scheduler. Supervising and restarting the process is the host’s job.

No multi-writer merge — the spec lists “CRDT/offline multi-writer merge” among its non-goals `spec §13`. Many clients may watch and steer one conversation, but every write goes through one Harness.

No rollback of the world — a commit is atomic in storage only. External effects “do not run inside the Session mutation transaction” `spec §1`, `invariant 4`. When the harness cannot know whether a payment went through, it says so and leaves the answer to an idempotency key or a remote record (limits and non-goals (p. 168)).

WHAT THIS MEANS FOR YOU

- Reopen the same storage and resume unfinished work. Safe tools can run again; unsafe tools report an interrupted result.
- Decide, per tool, whether running it twice is safe. That one choice decides what happens to a call that was cut off.
- Keep app state in documents, not in memory, so it can never disagree with the transcript.
- Keep one process per storage, and let your supervisor restart it.

Sources: `README.md` (intro, `§Concepts`, `§Watching a Conversation`, `§Your Own State`, `§Storage`); `docs/spec.md §1`, `§13`; `src/harness/tool.ts` (ToolTask phases `call`, `execute`); `research/capture/crash-README.md`, `crash-before.txt`, `crash-reopen-tasks.txt`, `crash-after.txt`; `research/pi/packages/coding-agent/docs/session-format.md`, `sessions.md`, `how-pi-works.md`; `earendil.com/posts/pi-durable` (Earendil, October 2026)

1. 2

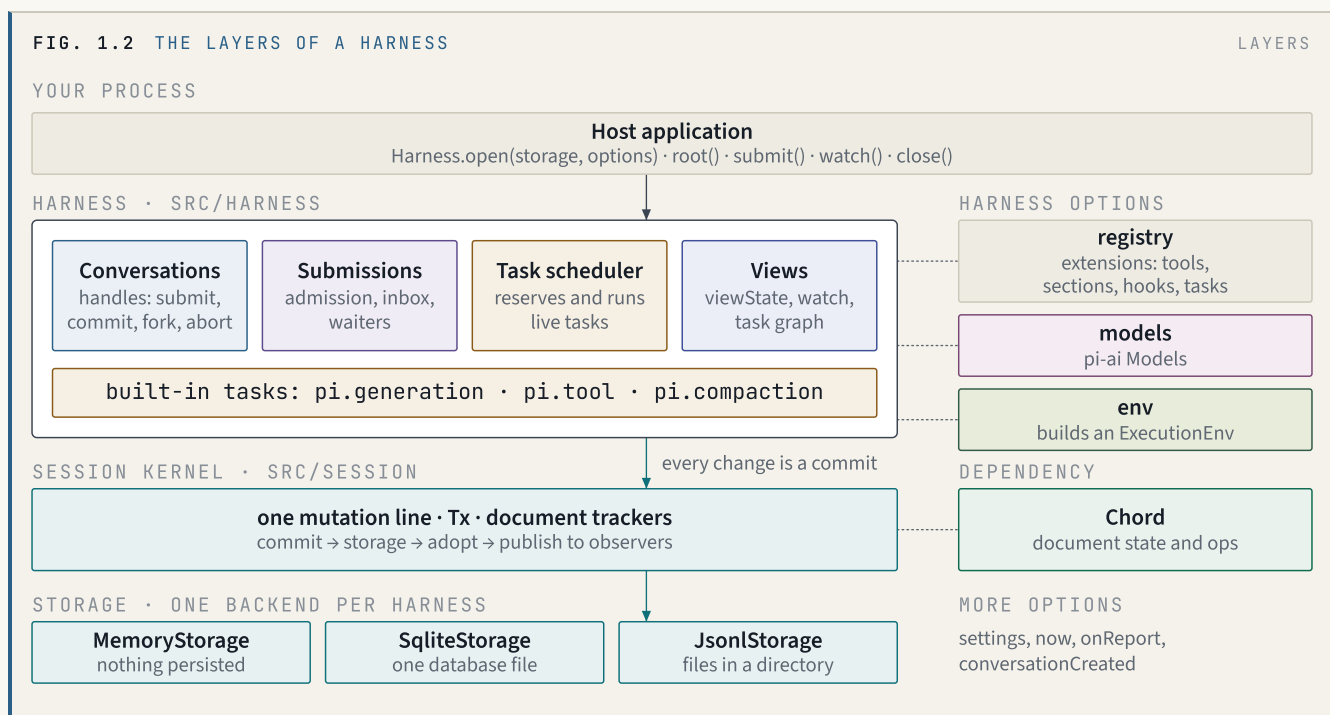
The architecture in one page

A Harness is a Session kernel with conversations, submissions and a task scheduler on top, one storage underneath, and everything else plugged in by the host.

Think of the package as a stack. Your program asks the Harness to do things; the Harness turns every request into a commit; one storage keeps the commits. This section names each layer, says which state each one owns, and shows what lives only in memory and is rebuilt when a process restarts.

Four layers

The package stacks four layers. Your process, the **host**, sits on top and calls the Harness. The Harness turns those calls into commits on one Session. The Session writes each commit to one storage. Code the harness runs but does not own, such as extensions, models and the execution environment, is passed in by the host when it opens the Harness.



The host drives a Harness, the Harness commits through one Session, and the Session writes one storage. The right column is HarnessOptions; Chord is a package dependency, not an option. After src/harness/harness.ts and src/session/session.ts.

Storage — a backend that stores records and documents and commits a batch of writes atomically. Three ship with the package [README §Storage](#); Part 9 covers the storage contract (p. 141).

Session — the kernel. It “owns one mutation line, conversations, entries, tasks, submissions, and documents” [spec §1](#). The **mutation line** is a queue that runs one commit at a time. The Session also keeps one tracker per loaded document and publishes each commit to its observers [src/session/session.ts](#). You can use it alone through `createSession()`; examples 00 to 05 do.

Harness — the Session kernel “extended with conversation handles and a registry” `src/harness/harness.ts`. It adds the handles you call, the submission machinery, the task scheduler, and the views.

Inside the Harness

Four components do the Harness’s work, each in its own file under `src/harness/`.

COMPONENT	FILE	OWNS
Conversation handles	<code>harness.ts</code>	nothing: a handle is an id plus methods; “compare handles by id” README §Concepts
Submissions	<code>submissions.ts</code> , <code>inbox.ts</code>	admission of inputs and writes, the per-conversation inbox, and the waiters behind <code>wait()</code>
Task scheduler	<code>scheduler.ts</code>	an in-memory mirror of every live task; reserving, running and aborting them
Views	<code>view.ts</code> , <code>task-graph.ts</code> , <code>events.ts</code>	<code>viewState()</code> , <code>watch()</code> , the task graph and agent events

The harness answers inputs with three built-in tasks. `pi.generation` prepares the prompt and calls the model. `pi.tool` runs one tool call. `pi.compaction` summarizes old context [spec §8](#). They “are not an extension and cannot be removed or replaced” `src/harness/registry.ts`, so always build the registry with `createRegistry()`; `Harness.open()` refuses one that lacks them.

What the host plugs in

`Harness.open(storage, options, context)` takes everything else as `HarnessOptions` [spec §2.2](#).

OPTION	IS	COVERED IN
<code>models</code>	the <code>pi-ai Models</code> interface; generation calls <code>models.streamSimple()</code>	generation (p. 91)
<code>registry</code>	the installed extensions: tools, prompt sections, hooks, wraps and tasks	the registry (p. 103)
<code>settings</code>	run policy read at every use and never stored	settings and defaults (p. 180)
<code>env</code>	builds the <code>ExecutionEnv</code> for each tool call and prompt render	the environment (p. 155)
<code>conversationCreated</code>	runs in every commit that creates or forks a conversation	access and creation (p. 53)
<code>now, onReport</code>	the clock, and a sink for errors the harness reports but survives	—

Two dependencies are not options: `@earendil-works/pi-ai` supplies the message types and the faux provider, and `@earendil-works/chord` supplies document state `package.json`. Every async call takes a `Chord Context`; cancelling it cancels the wait, never the work [README §Quick Start](#).

One call through the layers

Follow `root.submit(input, context)` down the stack to see how the layers divide the work.

1. The `Conversation` handle passes the input to the Submissions component `src/harness/submissions.ts`.
2. Submissions makes one commit. It appends the user entry, records the submission, creates the first generation task, and marks the conversation busy in its `pi.Live` document.
3. The Session writes that commit to storage, and only then tells its listeners about it `src/session/session.ts`.
4. The scheduler is one of those listeners. It sees the new task and claims it for running. Claiming is another commit `src/harness/scheduler.ts`.

- 5. The task runs outside any commit. It calls the model through `models`, and every result it wants to keep goes back in as a new commit.

No layer reaches around another, and the scheduler learns about new work only from published commits. The life of one answered input (p. 19) walks the chain with real records.

Memory and storage

The split between memory and storage is the key to the whole package. Storage holds the truth. Memory holds a working set that is always rebuilt from it:

Live tasks — the scheduler “mirrors every committed non-terminal task record” `src/harness/scheduler.ts`. On open it loads them all and turns any task left **running** by a dead process back to **pending**, in one commit.

Loaded documents — a cache of documents already read `src/session/session.ts`.

Waiters and watches — the promises behind `wait()` and the subscriptions behind views; the next process creates them again.

If the process dies, nothing in that list matters: open reads it all back. “Reopening the storage picks the work up where it stopped” `README`.

Entry points

The package exposes ten import paths. Node-specific code sits behind the paths that end in `/node`; the portable SQLite and JSONL cores also run on Bun or in Cloudflare Durable Objects `README` `$Storage`.

IMPORT	GIVES YOU
@earendil-works/pi-durable	Harness, createRegistry, defineExtension, defineTool, defineTask, defineDoc, entry tokens, built-in docs and tasks, MemoryStorage, createSession
.../storage/sqlite/node, .../storage/sqlite	openNodeSqliteStorage(file); the portable core over an async database facade
.../storage/jsonl/node, .../storage/jsonl	openNodeJsonlStorage(directory, context); the portable core over a FileSystem
.../storage/memory	MemoryStorage on its own
.../env, .../env/node	the ExecutionEnv interfaces; NodeExecutionEnv
.../tools	createReadTool(), createWriteTool(), createEditTool(), createBashTool(), and the CodingTools extension of all four
.../testing	registerStorageConformance(), registerEnvConformance(), storage benchmarks

From the `exports` map of `package.json`. The package requires Node 22.19.0 or later.

WHAT THIS MEANS FOR YOU

- Your code talks to the Harness and its conversation handles, never to storage.
- Anything you need after a restart must be in a commit; memory is a cache.
- Handles hold no state. After a restart, get a new one with `harness.conversation(id, context)`.

Sources: `package.json` (`exports`, `dependencies`, `engines`); `README.md` (`$Quick Start`, `$Concepts`, `$Storage`); `docs/spec.md` §1, §2.2, §8; `src/index.ts`; `src/harness/harness.ts` (`Harness.open`, `conversationCreated`); `src/harness/registry.ts` (`BUILTIN_TASKS`); `src/harness/scheduler.ts` (`TaskScheduler`, `open`); `src/session/session.ts` (`SessionImpl`); `test/examples/00-conversation.ts`

The core rule and its invariants

One sentence defines the system: commit everything together, and show nothing that is not committed. Eight invariants make that sentence true.

The whole design fits in one idea: nothing counts until it is committed. A **commit** is an atomic save: everything in it is stored together, or nothing is. If your screen shows it, a crash cannot take it back. This section states that rule, defines the terms the rest of the book uses, and explains the eight guarantees that make the rule hold.

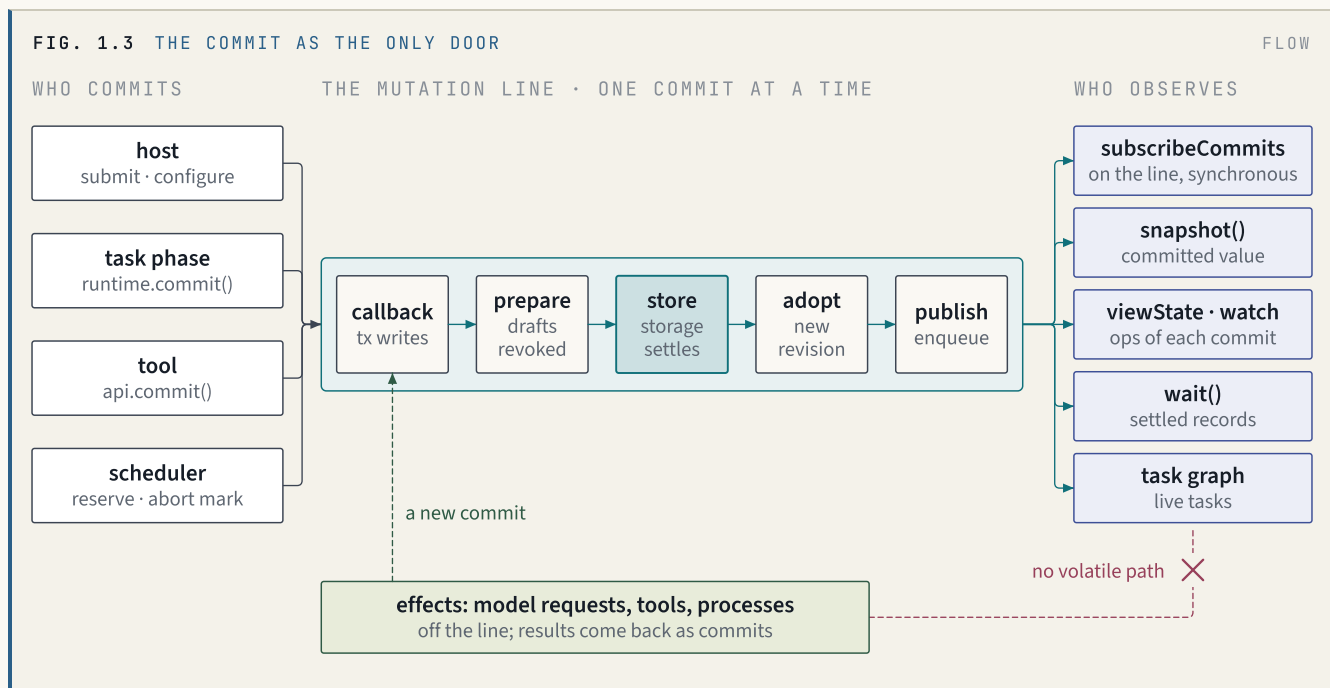
The rule

The spec states the whole design in two sentences, before any type:

CORE RULE

“A Session atomically commits immutable entries, full task records, and Chord-tracked documents. Only committed state is observable.”

The first sentence says what a **commit** holds: transcript records, task records and document changes, written as one unit. The second says that nothing else can be seen. A screen, a waiter, a watch and the scheduler itself all read committed state, and there is no side channel *spec, preamble*. Read it as a door: every change goes through the commit, and every observer sits behind it.



Every visible change passes through one commit on the mutation line; effects reach observers only as commits. Stage names follow spec §4: the line is held from the callback through publication.

The terms

The spec defines its nouns once, in §1. The rest of the book uses them exactly this way.

Session — owns one mutation line, conversations, entries, tasks, submissions, and documents.

Conversation — a transcript scope. It may fork another conversation.

Entry — an immutable transcript record.

Task — a durable state machine attached to one conversation.

Document — your application’s JSON state, saved with the conversation or task.

Definition — the TypeScript declaration that names a document and sets its initial value and storage rules.

Source — a read-only stream of saved document updates.

Turn — one assistant response and the tool calls it makes.

Run — the sequence of turns from an admitted input to the final answer. A conversation is busy while a run is active.

The eight invariants

Spec §1 then lists eight “required invariants”. Each one closes a specific failure. The table quotes them in short form; the paragraphs below say what each one buys.

#	INVARIANT (SPEC §1)	WITHOUT IT
1	One commit is atomic across all record and document writes.	a tool call stored without its task
2	A document update is published only after its storage commit succeeds.	a screen shows state a crash erases
3	All visible progress is durable; no volatile publication path.	streamed text that never existed after reopen
4	External effects do not run inside the mutation transaction.	one slow call stalls every conversation
5	Entries and IDs are immutable and never reused.	references point at the wrong record
6	Drafts are revoked when the callback settles; values are copied, strict JSON.	writes that bypass the commit
7	The line is held through settlement and adoption; user callbacks run later, off it.	a commit built on unadopted state
8	An uncertain storage failure is fatal to the open Session.	memory and disk silently diverge

1 · Atomic across records and documents

A commit is one call to `Storage.commit(writes)`, and it either stores every write or none `src/session/session.ts`. In the one-turn capture, commit 6 writes the assistant entry with the tool call, moves generation 9 to `waiting`, creates tool task 12, and updates two documents. Split across two writes, a crash between them would leave a tool call that no task will ever run.

2 and 3 · Publish after storage, and only then

The Session publishes a commit only after `Storage.commit()` has returned and the new revisions are adopted. Invariant 3 removes the alternative: there is no way to show something without committing it. Partial answers and tool output are therefore committed too, throttled by `settings.progress`. Each of those commits is a storage write; that is the price of a screen that never shows anything a crash can take back `src/harness/generation.ts (streamResponse)`.

4 · Effects stay off the line

“External model, process, tool, network, and human effects run outside it” [spec §4](#). Every conversation of a Harness shares one mutation line, so a model call awaited inside a commit would block every other commit until it returned. Worse, the commit’s content would depend on an effect it cannot undo. The tool task shows the pattern: commit the intent, run `execute()` off the line, then commit the result (the effect sandwich [\(p. 70\)](#)).

5 · Immutable entries, ids never reused

All record types share one id space. In the SQLite capture, conversation 1, documents 2 to 6, entry 7, submission 8 and task 9 come from one counter `research/capture/sqlite-rows.txt (record_ids)`. Records point at each other by id: tool task 12’s input is `{"assistant": 11, "callId": "call-1"}`, and submission 8’s answer is entry 15. If an id could be reused or an entry rewritten, those pointers would silently change meaning. Even an id taken by a commit that faulted stays used `research/capture/NOTES.md §6`.

6 · Drafts die with their callback

Inside a commit, `tx.doc()` returns a mutable draft. When the callback settles, the Session “synchronously prepares or aborts every open change”, and the draft is revoked. A test keeps a draft past its commit and checks that reading or writing it then throws `test/session-documents.test.ts, “revokes escaped drafts”`. Without this, a stray reference could change a document outside any commit, breaking the rule.

7 · The line covers storage and adoption

The line stays held until storage has settled and the new revision is adopted. The next commit therefore always starts from the state the previous one stored. Commit observers run on the line, but only to capture immutable values. Your watch and document-state callbacks run later, off the line, so a slow UI cannot stall the harness (the commit and the mutation line [\(p. 29\)](#)).

8 · An uncertain failure poisons the Session

A storage error that might have written something leaves memory unable to know what disk holds. The Session then refuses every later call with “Session is poisoned by a failed commit after storage admission; reopen it” `src/session/session.ts`. Only `StorageRejected` promises that nothing was written, and only it lets the Session continue. Preparation failures, such as a non-JSON value in a draft, happen before storage sees the batch and roll back normally (when a commit fails [\(p. 33\)](#)).

WHAT THIS MEANS FOR YOU

- Ask one question of any design: is the state you show or act on committed? If not, it does not exist yet.
- Change related things in one commit, so a crash can never split them.
- Never call a model, a tool or the network inside a commit callback.
- If storage fails in a way that may have written, close and reopen; do not retry.

Sources: `docs/spec.md (preamble, §1, §4)`; `src/session/session.ts (commit, poison check)`; `src/errors.ts (StorageRejected)`; `src/harness/generation.ts (streamResponse)`; `src/harness/tool.ts (call phase)`; `test/session-documents.test.ts (“revokes escaped drafts”, “poisons the Session after an uncertain Storage failure”)`; `research/capture/sqlite-rows.txt, sqlite-commits.txt, NOTES.md §6`

1.4

The life of one answered input

One question, one tool call and one answer take seventeen commits, and reading them in order is reading the whole harness.

A user asks “Count to 3.”, the model calls a `count` tool, and then it answers. That small exchange takes seventeen commits. This section walks through them in order, so you can read any commit log and say who wrote each line, why, and what a crash right there would leave behind.

The run

The README’s Quick Start is the smallest host. It opens a Harness over storage, gets the root conversation, submits an input, and waits for the answer:

Quick Start, abridged `README.md §Quick Start`

TS

```
const harness = await Harness.open(
  new MemoryStorage(), { models, registry: createRegistry() }, context
);
const root = await harness.root(context, {
  agent: { model: { provider: "openai", modelId: "gpt-6-sol" } },
});
const submission = await root.submit(
  { type: "input", content: "What is the capital of France?" }, context);
const settled = await submission.wait(context);
// settled.answer is the id of the pi.assistant entry that answered
await harness.close(context);
```

Imports, model setup and reading the answer back are cut. `wait()` resolves once the input is done or unanswered.

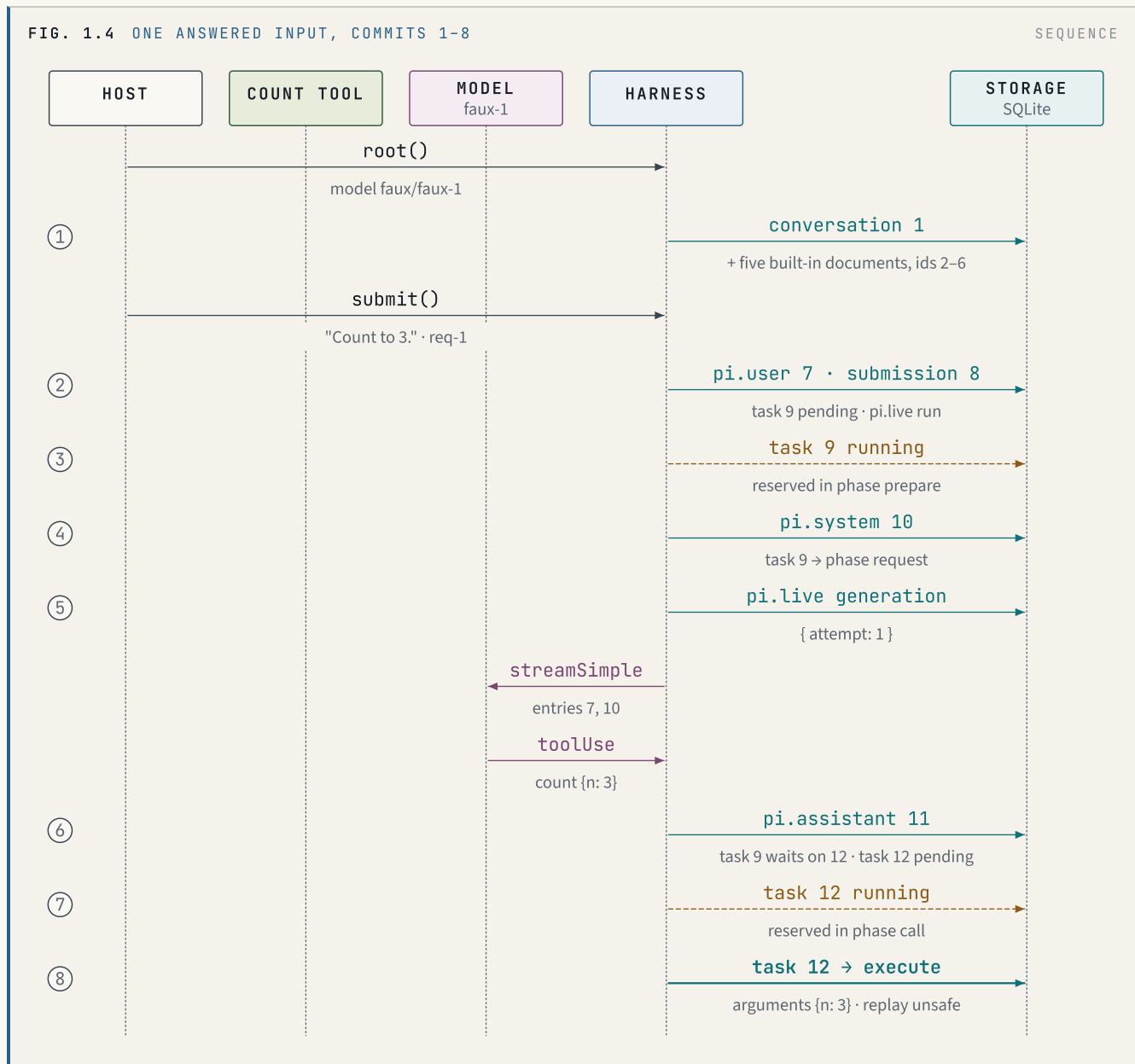
The capture runs the same steps over SQLite, so every commit can be read back as rows `research/capture/scripts/one-turn.ts`. The faux model calls a `count` tool with `{n: 3}`, then answers “I counted 1, 2, 3.”. The tool prints three lines with `api.output()` and returns `{}`. A `subscribeCommits()` listener logs every commit.

Two figures show the run. Each numbered arrow is one commit, and its number is the commit’s seq in storage.

Commits 1–2: a conversation, then an input

`root()` creates the root conversation in commit 1. The same commit creates its five built-in documents, ids 2 to 6: `pi.live`, `pi.inbox`, `pi.usage`, `pi.provider` and `pi.agent` `src/harness/harness.ts` (`conversationCreated`). The agent document stores the model choice, `faux/faux-1`. Opening the Harness also ran a commit, to reconcile running tasks, but it wrote nothing. A commit with no writes is discarded and uses no sequence number `src/session/session.ts`.

`submit()` is commit 2, and it is the one that makes the input durable. The conversation is idle, so admission does three things in one commit `src/harness/submissions.ts` (`admitSubmission`). It appends a `pi.user` entry, 7. It creates submission 8 with status `placed` and `entry: 7`. And it starts a run: it creates generation task 9 and sets `pi.live.run` to `{ "taskId": 9, "inputs": [8] }`. From here on the conversation is busy, because “busy” is defined as `pi.live.run` being present (run control (p. 88)). A crash after this commit loses nothing the user typed. Resubmitting with the same `requestId` returns submission 8 instead of a new one.



Commits 1-8 take the input from `submit()` to a tool call whose intent is durable. From `research/capture/sqlite-commits.txt`; document operations from `research/capture/extra-p1/sqlite-commit-ops.txt`.

Commits 3-6: the first generation

Commit 3 is the scheduler at work. It reserves task 9 by rewriting it from `pending` to `running`, without changing its checkpoint `{phase: "prepare", attempt: 1}` (the scheduler (p. 73)). Every task in this run starts with such a reservation commit.

The `prepare` phase renders the system prompt and tool list and commits them as `pi.system` entry 10, which holds the `preamble` section and the `count` tool. The same commit, 4, moves the task to `phase request` with the model, the thinking level and a `cutoff` of entry 10. The `request` phase first commits `pi.live.generation = {attempt: 1}` (commit 5), then calls `models.streamSimple()` off the line. The faux stream finished inside the 100 ms throttle window, so no partial was committed. A slower stream would add throttled partial commits here (generation (p. 91)).

The response stops with `toolUse`. Commit 6 is the turn's pivot. It writes the response as `pi.assistant` entry 11 and adds its usage to `pi.usage`. It creates tool task 12, owned by task 9. It lists the call in `pi.live.tools`. And it moves task 9 to `waiting` on [12] with policy `allSettled` `src/harness/generation.ts` (`startToolRound`). A crash after commit 6 reopens with the call recorded and its task pending, so the tool will run. A crash before it reopens in phase `request`, and the model call is simply made again.

Commits 7–8: intent before effect

Commit 7 reserves task 12. Its `call` phase then reads the call from entry 11, resolves the tool, validates the arguments and runs any `beforeTool` hooks. Then it commits its **intent**, commit 8, before running any tool code:

tool task 12 after commit 8 `research/capture/sqlite-rows-midrun.txt`

JSON

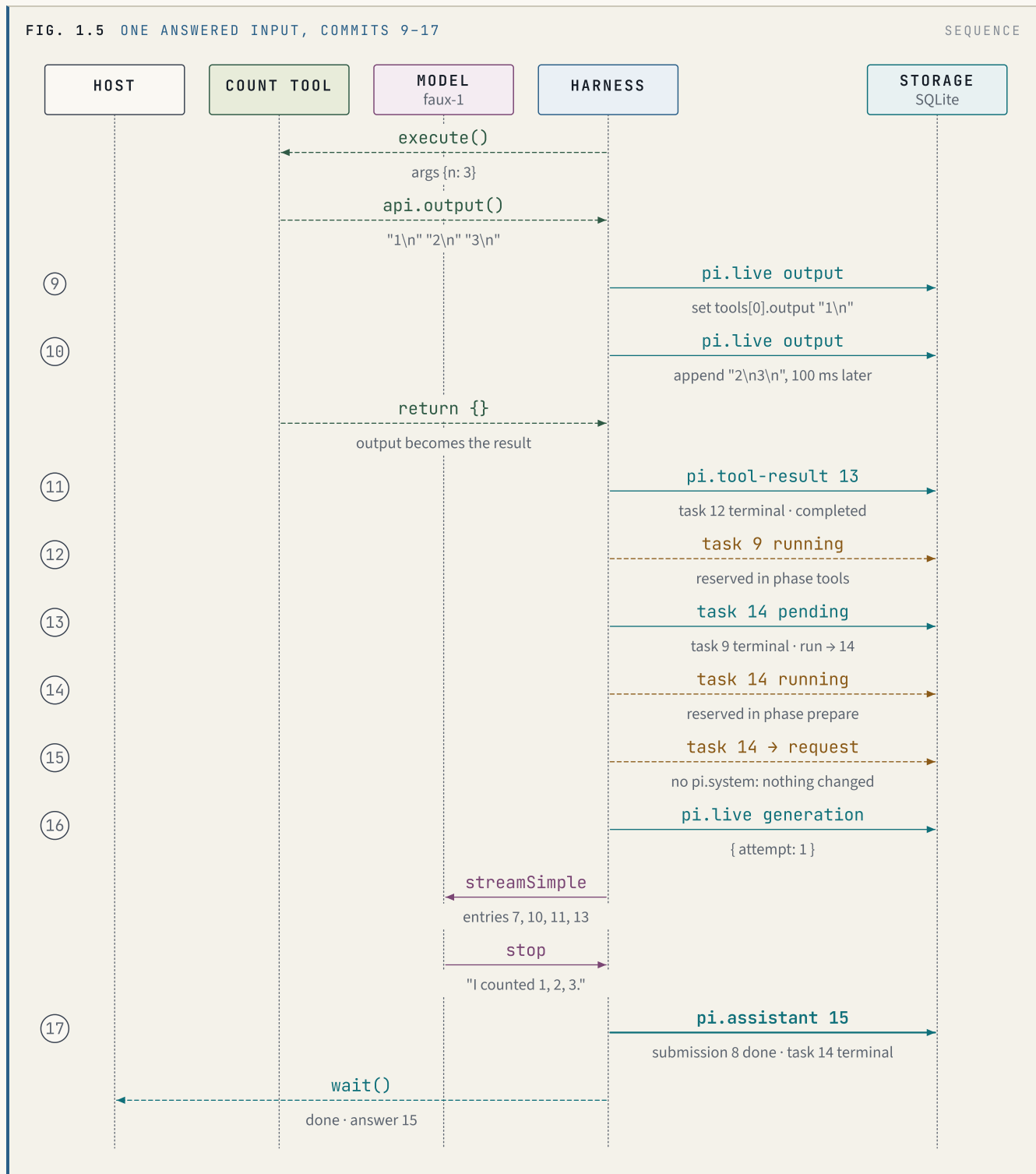
```
{
  "id": 12,
  "kind": "pi.tool",
  "input": { "assistant": 11, "callId": "call-1" },
  "owner": 9,
  "state": {
    "status": "running",
    "checkpoint": {
      "phase": "execute",
      "arguments": { "n": 3 },
      "replay": "unsafe"
    }
  }
}
```

The `tasks` row, read from a second connection while `execute()` ran; `conversationId`, `version` and two boolean flags are cut. `count` declares no `replay`, so the stored policy is `"unsafe"`.

This record is the reason a crash mid-tool is recoverable. The `execute` phase is reached on its own only by recovery. If a later process finds a task in it, the tool may have run, and the stored `replay` decides between running it again and writing an interrupted error result `src/harness/tool.ts`. That is exactly what happened to `deploy` in [1.1 \(p. 10\)](#).

Commits 9–11: output, then the result

The tool's output is committed as progress, with a default minimum interval of 100 ms. Commit 9 sets `pi.live.tools[0].output` to `"1\n"`; commit 10 appends `"2\n3\n"`. A watcher sees the output grow, and a crash would lose only output since the last successful progress commit. There is no fixed time limit on that gap. When `execute()` returns, commit 11 appends `pi.tool-result` entry 13 with content `"1\n2\n3\n"` and makes task 12 `terminal` with outcome `{status: "completed", result: {entryId: 13}}`.



Commits 9–17 carry the tool output, the result, a second generation and the answer. Amber arrows are task-only commits: reservations, and the move to request with no prompt change.

Commits 12–17: the second generation and the answer

Task 12 is terminal, so task 9's wait is over. Commit 12 reserves it in phase `tools`. That phase is where queued writes and steering messages sent during the tool round would be placed (the inbox (p. 84)). It then hands the run over: commit 13 creates

generation 14, moves `pi.live.run.taskId` to 14, clears the tool round, and ends task 9 completed with `result: {entryId: 11}`src/harness/generation.ts (`finishToolRound`).

Generation 14 repeats the first one's phases with one difference. Its `prepare` phase finds the prompt and tools unchanged, so it appends no `pi.system` entry, and commit 15 only moves the task to `request`. System entries are positional deltas, sent only when something changed (the system prompt (p. 115)). Commit 16 marks the attempt in `pi.live`, and the model answers with stop reason `stop`.

Commit 17 finishes everything at once. It appends the answer as `pi.assistant` entry 15 and settles submission 8 done with `answer: 15`. It makes task 14 terminal, deletes `pi.live.run` and `pi.live.generation`, so the conversation is idle again, and adds the second response's tokens to `pi.usage`. The Submissions component sees the settled record in the commit's publication and resolves the host's `wait()`.

The whole log

SEQ	RECORDS WRITTEN	DOCUMENTS	WRITER
1	conversation 1	create 2-6	root()
2	entry 7 <code>pi.user</code> ; submission 8 placed; task 9 pending	<code>pi.live</code>	<code>submit()</code>
3	task 9 running · prepare	—	scheduler
4	entry 10 <code>pi.system</code> ; task 9 · request	—	generation 9
5	—	<code>pi.live</code>	generation 9
6	entry 11 <code>pi.assistant</code> ; task 9 waiting; task 12 pending	<code>pi.live</code> , <code>pi.usage</code>	generation 9
7	task 12 running · call	—	scheduler
8	task 12 · execute	<code>pi.live</code>	tool 12
9, 10	—	<code>pi.live</code>	tool 12 progress
11	entry 13 <code>pi.tool-result</code> ; task 12 terminal	<code>pi.live</code>	tool 12
12	task 9 running · tools	—	scheduler
13	task 14 pending; task 9 terminal	<code>pi.live</code>	generation 9
14	task 14 running · prepare	—	scheduler
15	task 14 · request	—	generation 14
16	—	<code>pi.live</code>	generation 14
17	entry 15 <code>pi.assistant</code> ; submission 8 done; task 14 terminal	<code>pi.live</code> , <code>pi.usage</code>	generation 14

Every commit of `research/capture/sqlite-commits.txt`; the JSONL run has the same seventeen.

WHAT THIS MEANS FOR YOU

- The user's text is safe from commit 2; a resubmit with the same `requestId` cannot duplicate it.
- A tool's intent is committed (commit 8) before any of its code runs, so recovery always knows a call may have started.
- No commit leaves an answer without its submission settled, or a tool call without a task. Kill the process after any row and it reopens knowing what to do next.

Sources: README.md (§Quick Start, §Concepts "One answered input"); `research/capture/sqlite-commits.txt`, `sqlite-rows.txt`, `sqlite-rows-midrun.txt`, `jsonl-commits.txt`, `scripts/one-turn.ts`, `scripts/sqlite-one-turn.ts`; `research/capture/extra-p1/sqlite-commit-ops.txt` (`test/capture-p1-ops.ts`); `src/harness/harness.ts` (`root`, `conversationCreated`); `src/harness/submissions.ts` (`admitSubmission`); `src/harness/generation.ts` (`prepare`, `request`, `startToolRound`, `finishToolRound`, `answer`, `startRun`); `src/harness/tool.ts` (`call`, `execute`); `src/harness/scheduler.ts` (`#reserve`); `src/session/session.ts` (`#runCommit`)

Sessions and Commits

One mutation line, one atomic commit at a time, and nothing published until storage says it is stored.

2.1	Records and IDs	25
2.2	The commit and the mutation line	29
2.3	When a commit fails	33

02

2 . 1

Records and IDs

Five record types and one number space hold everything a Session stores, and two of the edges between records look alike but mean different things.

Open a Pi Durable database and you find only five kinds of record, each numbered from one shared counter. Learn the five shapes and the two ways one conversation can point at another, and you can read any storage dump.

Four tables and the documents

A **Session** stores four kinds of table record plus documents: “Tables are conversations, entries, tasks, and submissions” [spec §4](#). Documents are your app’s JSON state, with a small record of their own (documents (p. 53)). Every record is a plain JSON object, declared as a TypeScript type in `src/types.ts`. The figure shows all five and what each field is for.

FIG. 2.1 THE RECORD FAMILY

STRUCTURE

ConversationRecord 3.3	TaskRecord 5.1
id ConversationId; 1 is the root	id TaskId<R>
parent? { conversationId, at }: history	conversationId the conversation it belongs to
owner? { conversationId, taskId }: creator	kind · version definition name and version
	input original input, kept for life
	owner? TaskId of the owning task
	background outside idle waits and cascades
	abortRequested durable abort mark
	state TaskState: status and checkpoint
	memos? first-writer-wins values
EntryRecord 3.1	SubmissionRecord 6.1
id EntryId; Session-global, ordered	id SubmissionId
conversationId the conversation it was appended to	conversationId the target conversation
kind string discriminator	requestId? host dedupe key, per conversation
model? Message[] for the model	type input or write
data? JSON for the application	status queued, placed, done, unanswered
head? EntryId where context restarts	entry? EntryId of the placed entry
edits? ContextEdit[] on earlier entries	answer? EntryId of the answer (input)
byTaskId? TaskId of the task that wrote it	reason? detail? why it went unanswered
DocumentRecord 4.2	
id DocumentId of one incarnation	
kind · key? definition kind; family key	
scope session, conversation or task	
history? fork? conversation documents only	
createdAt Seq of the creating commit	
retiredAt? Seq of the retiring commit	

COLOURED OR ITALIC TEXT IS A REFERENCE

to a conversation or an entry

to a task

italic: a commit Seq, not a record

Five record types share one ID space, and their references are typed IDs. Field names are from `src/types.ts`; the numbers in the card heads are the sections that cover each type in depth.

The records differ in how they change:

- **Conversations and entries never change** after the commit that creates them (invariant 5, [the invariants \(p. 16\)](#)).

- **Tasks and submissions are replaced whole.** Each change writes a complete new record under the same ID `src/types.ts` (`TaskRecord`).
- **A document record** describes one document from creation to retirement. Its content is stored separately, as full copies and Chord operations ([bases and deltas](#) (p. 56)).

ConversationRecord — only `id`, an optional `parent` and an optional `owner`. A conversation's model, settings and application state are not in this record. They live in documents such as `pi.agent` ([the built-in documents](#) (p. 60)).

EntryRecord — one immutable transcript event. It has separate fields for the model (`model`), the application (`data`) and context control (`head`, `edits`); see [entries](#) (p. 37).

TaskRecord — identity (`kind`, `version`, `input`), ownership (`owner`, `background`), the durable abort mark and a `state union`; see [tasks as state machines](#) (p. 63).

SubmissionRecord — a union of `type` (`input` or `write`) and `status`. An input moves through `queued`, `placed`, `done` and `unanswered`. A write has no `placed` state, and only an input has an `answer`; see [submissions](#) (p. 84).

DocumentRecord — `kind`, optional family key, scope, and for conversation documents the history and fork policy. `createdAt` and `retiredAt` are commit sequence numbers stamped by storage.

IDs are plain numbers with a type label

Every record ID is a plain JavaScript number. TypeScript adds a label, called a **brand**, so you cannot pass an `EntryId` where a `TaskId` is expected. The brand exists only at compile time:

IDs, Seq and the root `src/types.ts`

TS

```
export type ConversationId = Id<"conversation">;
export type EntryId = Id<"entry">;
export type TaskId<Result = unknown> = Id<"task", Result>;
export type SubmissionId = Id<"submission">;
export type DocumentId = Id<"document">;

/** Strictly increasing sequence assigned to one atomic storage commit; gaps are permitted. */
export type Seq = number & { readonly [seqBrand]: "sequence" };

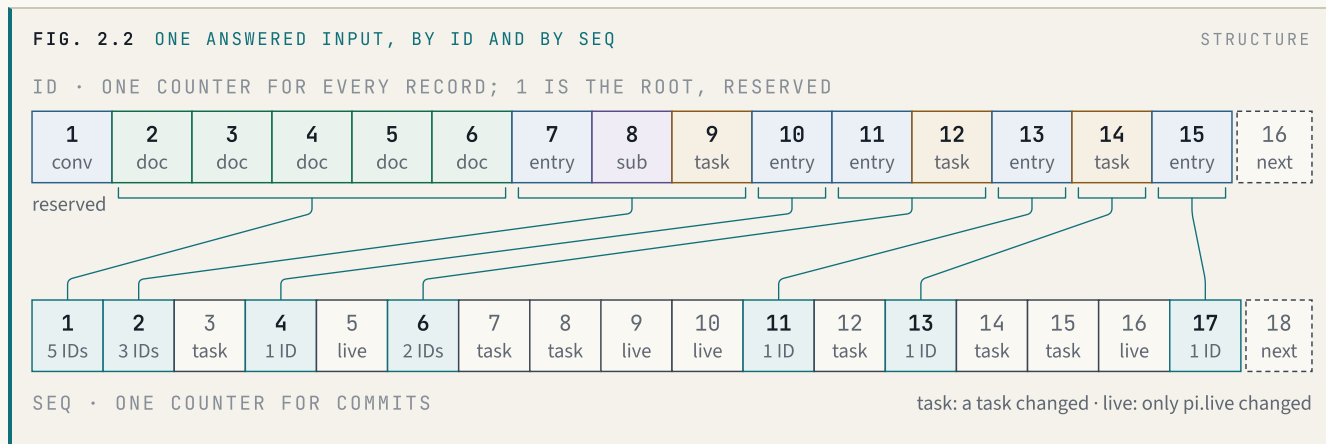
/** The root conversation always uses this reserved ID. */
export const ROOT_CONVERSATION_ID = 1 as ConversationId;
```

The declarations of the generic `Id` type and the two brand symbols are cut. A `TaskId` also carries its task's result type, which is how `waitForTask()` returns a typed outcome.

"IDs and sequences remain ordinary numbers in memory, JSON, JSONL, and SQLite" [spec §2](#). The package does not export a helper to brand a number. When your code receives an ID as JSON, from a URL or a UI message, cast it yourself with `as EntryId`, and check it at that spot.

One counter for every record

Conversations, entries, tasks, submissions and documents all take their IDs from one Session-wide counter, `Storage.mintId()` `src/types.ts`. So no two records share a number, whatever their type. The captured SQLite run makes this visible: the types interleave in the order the commits needed them.



Every record but the root draws its ID from one counter; commits are numbered separately. ID 1 is reserved for the root conversation, so commit 1 mints only the five document IDs 2–6. Each line joins a group of IDs to the commit that minted it; ten of the 17 commits minted none. The dashed cells are `next_id` and `next_seq` from `durable_metadata`. IDs and types from `record_ids` in `research/capture/sqlite-rows.txt`, commits from `research/capture/sqlite-commits.txt`.

Three facts follow from one counter.

IDs order entries across conversations — “Entry IDs are Session-global and ordered” [spec §2.1](#). A fork’s cap and a context range are both comparisons of entry IDs, even when the entries belong to different conversations ([forks \(p. 43\)](#)).

The root is reserved, not minted — `ROOT_CONVERSATION_ID` is 1, and `MemoryStorage` starts minting at 2 `src/storage/memory.ts`. A plain Session that creates its first conversation therefore gets ID 2 (`run 00`). The Harness creates conversation 1 lazily through `root()`.

A rolled-back commit burns its IDs — minting happens inside the callback, before the batch reaches storage. The transaction methods that create records are asynchronous “because remote storage may allocate globally unique numeric IDs durably” [spec §3.3](#). In the captured failure run, three commits that rolled back took IDs 3, 4 and 5, and the next entry that committed was 6 (`research/capture/extra-p23/commit-failures.txt`). The guarantee is only that an ID is “never reused after a committed write” (invariant 5).

A **Seq** is a second counter. It numbers commits, not records: storage gives one to each committed batch, and it “strictly increases between commits; gaps are permitted” [spec §2](#). The two counters run at different speeds. In the capture, 17 commits minted 14 IDs (2–15), and ten of those commits minted no ID at all; they only changed task states or `pi.live`. You meet a Seq on a document’s `createdAt` and `retiredAt`, on every commit publication, and on each stored entry, where it lets a document be read as of that entry ([rewindable history \(p. 56\)](#)).

Two edges: parent and owner

A `ConversationRecord` has two optional edges with the same shape. Both point at a conversation, and they mean different things. “Conversation history parenting and task ownership are separate” [spec §2](#):

	PARENT	OWNER
Shape	<code>{ conversationId, at: EntryId }</code>	<code>{ conversationId, taskId: TaskId }</code>
Set by	a fork at entry <code>at</code>	creation with <code>{ kind: "task", taskId }</code> ownership
Controls	inherited entries and historical documents	attribution, subtree abort, idle traversal
Caller supplies	the parent conversation and entry	only the task ID; the Session derives the conversation
After the task ends	—	stays recorded

From spec §2 and the field comments in `src/types.ts`.

The owner edge “is not an access-control capability” spec §2; it records who made the conversation. Creation always states ownership, either `{ kind: "ownerless" }` or `{ kind: "task", taskId }`, and the Session fills in the task’s conversation itself. A fork can carry both edges: it inherits history from one conversation while a task in another owns it.

Records have three more edges that use the same IDs. A task’s owner is a `TaskId`, which builds the task tree (ownership and structured concurrency (p. 76)). An entry’s `byTaskId` names the task whose commit appended it. A submission’s `entry` and `answer` name the user entry it placed and the assistant entry that answered it. The captured submission 8 ends as `{ status: "done", entry: 7, answer: 15 }` `research/capture/sqlite-rows.txt`.

WHAT THIS MEANS FOR YOU

- An ID is unique across all record types, but the number does not say which type it is. Keep the type with it, and let the brands catch mix-ups.
- Expect gaps. A failed commit burns the IDs it minted, and that is normal.
- Use `parent` to ask “where did this history come from” and `owner` to ask “which task made this”.

Sources: spec §1 (invariant 5), §2, §2.1, §3.3, §4; `src/types.ts` (`Id`, `Seq`, `ROOT_CONVERSATION_ID`, `ConversationRecord`, `EntryRecord`, `TaskRecord`, `SubmissionRecord`, `DocumentRecord`, `Storage.mintId`, `Storage.entry`); `src/index.ts`; `src/session/transaction.ts` (`conversation creation`); `src/storage/memory.ts` (`nextId`); `run 00`; `research/capture/sqlite-rows.txt`; `research/capture/sqlite-commits.txt`; `research/capture/extra-p23/commit-failures.txt`

2.2

The commit and the mutation line

Every write goes through one serialized callback; its writes are stored together or not at all, and observers see nothing until storage has said yes.

Every change to a Session goes through `commit()`. A commit is an atomic save: everything in it is stored together, or nothing is. Commits run one at a time, in a queue called the mutation line. This section shows what you may do inside a commit and when the rest of the program sees the change.

One way to write

`session.commit(change, context)` is the only write path. Your callback receives a transaction, `tx`. Everything it writes is stored as one batch when the callback returns, or thrown away if it throws `ex 00`. “One Session commit is atomic across all record and document writes” (invariant 1, [spec §1](#)). `Conversation.commit()` binds it to one conversation; a task’s `runtime.commit()` also stamps the task’s ID on every entry it appends.

a commit that reads, appends, edits a document and creates a task [docs/spec.md §4](#) TS

```
await session.commit(async tx => {
  const conversation = await tx.conversation(conversationId); // table read
  const live = await tx.doc(LiveDoc, conversationId);

  await tx.appendEntry(conversationId, message); // first table write
  delete live.generation; // document mutation remains valid
  await tx.createTask(Follow, {}, {
    ownership: { kind: "conversation" }, conversationId,
  }); // further table writes are fine
}, context);
```

The read comes first because it must; the document draft stays usable after the first write.

The mutation line

The **mutation line** is a queue that runs one commit at a time. A commit keeps the line from the moment its callback starts until storage has stored the batch and observers have been told: “through callback execution, preparation, storage settlement, committed baseline adoption, and publication enqueue” [spec §4](#). Meanwhile no other commit starts. The figure below follows one commit through those stages.

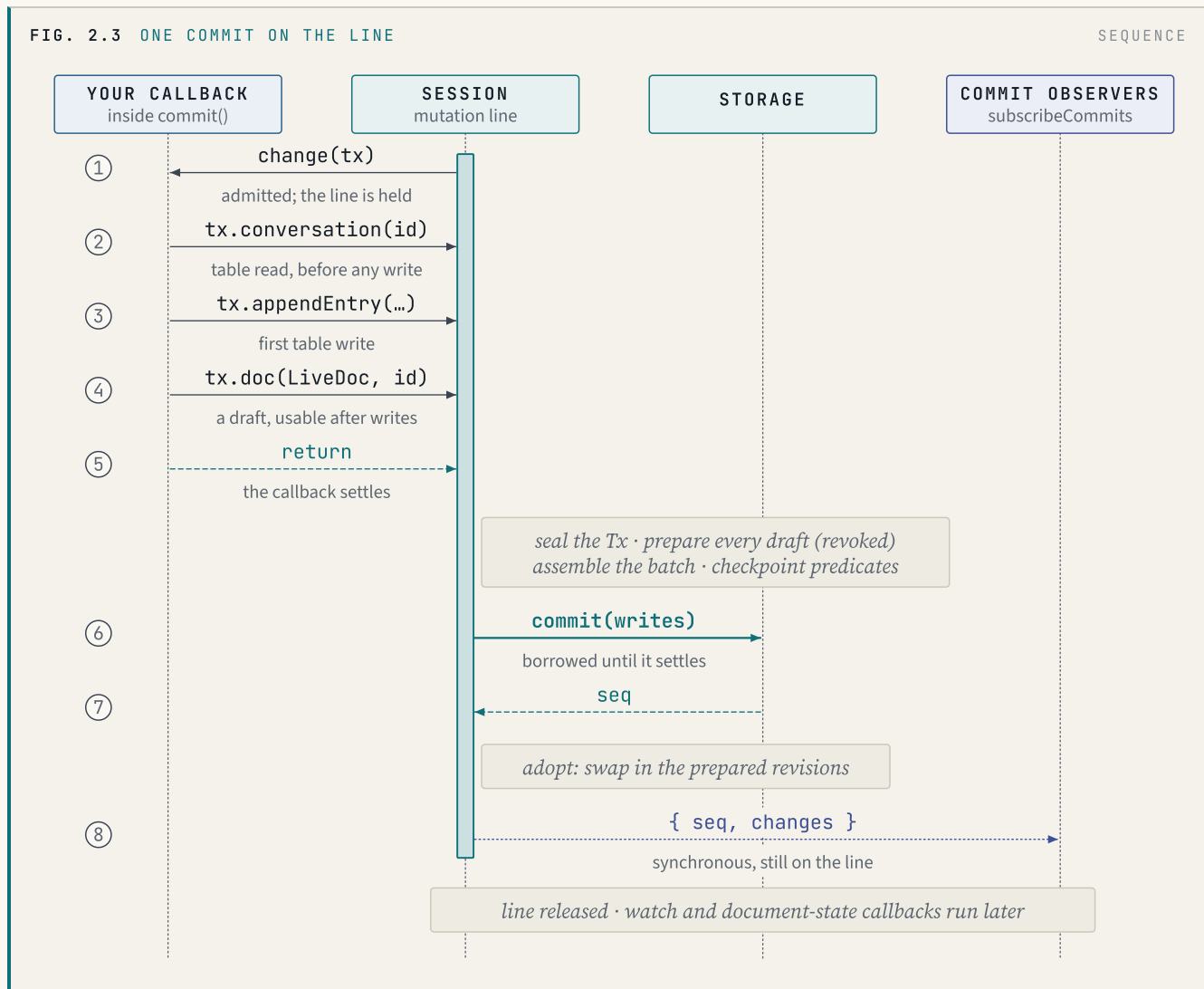
Steps 1–5: inside your callback

`commit()` rejects at once if the Session is closing or poisoned. Otherwise the callback runs with a fresh transaction, whose operations fall into three groups (table below).

The rule to remember is **read before you write**. “Any table read after the first table write throws `ReadAfterWrite`” [spec §4](#), with a message such as `Tx.conversation()` cannot read tables after the first table write. Table reads go straight to storage, and storage has not yet seen this transaction’s writes, so a read after a write could contradict them. You never need one: creation methods return the record they made.

GROUP	OPERATIONS	RULE
Table reads	conversation, entry, task, scanConversations, scanEntries, latestHeadMarker, scanTasks, submissionByRequest	only before the first table write
Table writes	createConversation, forkConversation, appendEntry, createTask, createSubmission, settleSubmission, placeSubmission	return the created record or ID
Docs	doc (get or create), retireDoc	usable before and after table writes

The Tx interface in `src/types.ts`; the rules are spec §4.



The line is held from admission until the publication is enqueued; callbacks run after. The aqua bar is the held line. After `src/session/session.ts` and `src/session/transaction.ts`.

Documents work differently. `tx.doc()` returns a **draft**, a mutable copy you edit like a plain object, and it sees your own changes. Values you assign “are copied by value and must be strict JSON”; an `undefined` or a `Date` fails the commit before anything is stored spec §1 (invariant 6); `test/session-documents.test.ts`. When the callback settles, every draft is revoked. A draft kept past the callback throws on its next use, and a `tx` method called late rejects with `Transaction has settled`.

Between steps 5 and 6: checking the batch

When your callback returns, the Session turns its work into one batch and checks it. A forgotten `await` is caught first: a `tx` call still pending fails the commit with `Session commit callback settled before its pending Tx operations`. Then drafts become final values and the batch is validated, for example that every owner task exists and is still live. Any failure here rolls the whole commit back ([when a commit fails \(p. 33\)](#)). A batch with no writes never reaches storage: nothing is stored and nothing is published.

Steps 6–8: storage, then everyone else

`Storage.commit(writes)` stores the batch atomically and returns its `Seq`. Once a commit has started, cancelling your `context` no longer stops it: “caller cancellation does not interrupt storage settlement or undo the commit” [spec §4](#). After storage succeeds, the Session swaps in the new document values and builds one publication, `{ seq, changes }`. It hands that to every `subscribeCommits()` listener, still on the line. Here is one, from the turn in which the model asked for a tool:

```
commit seq 6, as a commit observer saw it research/capture/sqlite-commits.txt
```

TEXT

```
commit seq=6
  entry 11 kind=pi.assistant byTask=9
  task 9 pi.generation status=waiting phase=tools on=[12] policy=allSettled
  task 12 pi.tool status=pending phase=call
  document pi.live (2) 2 op(s)
  document pi.usage (4) 1 op(s)
```

One line per change, summarized by the capture script. The assistant entry, the parent’s wait, the new tool task and two document changes became visible together, or not at all.

Commit observers run on the line “only to capture immutable state” (invariant 7). They “must not throw, block, or call Session APIs” [spec §4](#). Your watch and document-state callbacks run later, off the line ([Chord and document state \(p. 125\)](#)), so a slow screen never holds up the next commit.

Never await an effect inside a commit

“External model, process, tool, network, and human effects run outside it” [spec §4](#). Breaking that rule costs you twice.

The line stalls — while a callback awaits a network call, every other commit waits behind it: tool progress, other conversations, the scheduler.

The record lies — if the process dies after the effect but before the batch is stored, the effect happened and nothing records it. If the callback throws after the effect, the batch rolls back and the effect is still unrecorded.

The fix is to commit the intent, perform the effect outside any commit, then commit the outcome: [the effect sandwich \(p. 70\)](#). Every built-in tool and generation task works this way.

FIRE AND FORGET

Work started inside the callback and left running can still change the open transaction, and then fails once it settles. The spec calls this unsupported [spec §12](#). Await every `tx` call, and start background work only after `commit()` resolves.

WHAT THIS MEANS FOR YOU

- Do all table reads first, then writes. Documents can be read and changed at any point.
- Await every tx call, and keep the callback short and free of I/O.
- Put only JSON in drafts, and never keep a draft after the callback.
- React to changes in watches, not in commit observers.

Details

Reads stay cheap despite the single line. A document whose committed value is cached is read without queueing `src/session/session.ts` (snapshot). Storage borrows the batch only until `Storage.commit()` settles, and “every storage read returns a detached JSON value” `spec §4`, so nothing you hold aliases stored state. The batch check validates fork sources, owners and task replacements, retires the documents of tasks that end in this commit, and finally runs each document’s `checkpoint` when predicate to choose a full copy or a delta (bases and deltas (p. 56)) `src/session/transaction.ts`.

Sources: `spec §1` (invariants 1, 4, 6, 7), `§2.2`, `§3.3`, `§4`, `§12`; `src/session/session.ts` (commit, publication, snapshot); `src/session/transaction.ts` (Tx operations, settlement, batch assembly, adoption); `src/errors.ts` (ReadAfterWrite); `src/types.ts` (Tx, StorageWrite, CommitPublication, Storage.commit); `ex 00`; `research/capture/sqlite-commits.txt`; `research/capture/extra-p23/commit-failures.txt`

2.3

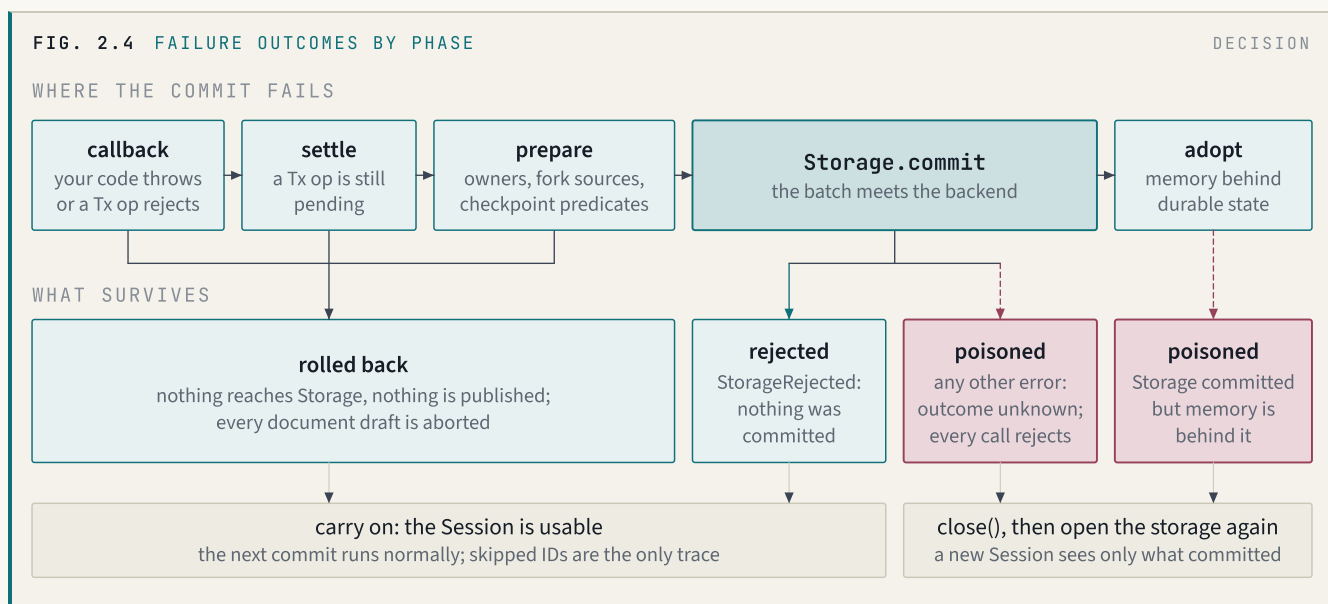
When a commit fails

Where a failure happens decides its outcome: before storage it is an ordinary rollback, and inside storage it is either a guaranteed rejection or a fatal uncertainty.

Most failed commits are harmless: nothing was stored, and you carry on. One kind is not. If storage fails in a way that might have stored the batch, the Session stops accepting work until you close it and open the storage again. This section shows how to tell the cases apart and what to do in each.

Three outcomes

A commit can fail in five places, and every failure ends in one of three outcomes: rolled back, rejected by storage, or poisoned. Invariant 8 states the split: “An uncertain storage failure is fatal to the open Session. It publishes nothing and must be reopened. Preparation and checkpoint failures occur before storage admission and roll back normally” [spec §1](#).



Failures before Storage admission roll back; an uncertain Storage result, or an adoption that fails after Storage committed, poisons the Session. The bottom row is what to do next in each case. Branches from [src/session/session.ts](#) and [src/session/transaction.ts](#).

The captured failure run shows each outcome on a plain Session over a `MemoryStorage` whose next commit can be made to fail:

```
$ node --conditions=source --experimental-strip-types test/capture-p23-context.ts
read after write -> ReadAfterWrite: Tx.conversation() cannot read tables after the first table write
callback throws -> Error: validation failed
empty commit -> ok "nothing written"
storage rejects -> StorageRejected: rejected before any durable effect
next commit -> ok {"kind":"app.note","data":"after rejection","id":6,"conversationId":2}
uncertain storage failure -> Error: disk vanished
next commit -> Error: Session is poisoned by a failed commit after storage admission;
                reopen it (cause: Error: disk vanished)
snapshot -> Error: Session is poisoned by a failed commit after storage admission; ...
close -> ok undefined
```

From `research/capture/extra-p23/commit-failures.txt`; the long poisoned line is wrapped and the second one cut. The rejection and the two failures before it burnt IDs 3–5.

Before storage: an ordinary rollback

Three of the five places come before storage: the **callback** itself, **settle** (a tx call still pending), and **prepare** (the batch check). A failure in any of them throws away every change and rethrows the original error `src/session/transaction.ts`. That covers your own exceptions, `ReadAfterWrite`, a non-JSON value in a draft, and a reference to a missing conversation or task (`Conversation N does not exist`).

The rollback is complete. Nothing is stored, nothing is published, and every document keeps its earlier value. A test fails a commit that changed two documents; both still read as before, and the next commit works normally `test/session-documents.test.ts`. The only trace is a gap in the ID sequence (records and IDs (p. 25)).

Inside a task, a rolled-back commit usually ends the task. If a phase handler lets the error escape, the task ends **faulted** (phases, steps and progress (p. 67)). In one captured scenario a task tried to create owned work in the same commit that finished it; the batch check rejected it with `Task owner 12 is completing`, and the task ended **faulted** with nothing from that commit stored `research/capture/NOTES.md` (note 6).

Inside storage: rejected or uncertain

Once the batch is handed to `Storage.commit()`, the Session can no longer assume anything. The storage contract defines the one safe exception: “`StorageRejected` means a batch was rejected before any durable effect and is guaranteed not to have committed. Session rolls such a batch back normally; unknown failures after Storage admission remain fatal because their commit state is uncertain” `spec §10`. Backends use it for failures they can prove had no effect. SQLite, for example, reports a failed fork copy as `Document copy N was rejected` `src/storage/sqlite/storage.ts`.

Every other storage error **poisons** the Session: it marks itself unusable. So does a failure just after storage succeeded, while the Session swaps in the new values, because then storage holds a commit that memory does not reflect `src/session/session.ts`.

A poisoned Session rejects everything that needs its state. Every later commit, `snapshot()`, `state`, `watch` and `line read` throws `Session is poisoned by a failed commit after storage admission; reopen it`, with the original error as its `cause` `src/session/session.ts`. Nothing from the failed batch is published, so no observer saw a value that may not exist `test/session-documents.test.ts`.

The Session does not retry because neither guess is safe. If the batch did commit, memory is behind storage; if it did not, applying it in memory would publish a change that does not exist. Only storage knows, and a new Session reads exactly what it holds. “Do not catch the error and continue using it” `spec §12`. JSONL storage also poisons itself after an uncertain append (JSONL (p. 148)).

ERROR	RAISED	SESSION AFTERWARDS
<code>ReadAfterWrite</code>	table read after a table write	usable
...settled before its pending Tx operations	callback returned early	usable
Transaction has settled	Tx or draft used after settlement	usable
<code>StorageRejected</code>	storage, before any durable effect	usable
any other storage error	storage, outcome unknown	poisoned
adoption error	after storage committed	poisoned
Session is closed	any call after <code>close()</code> began	closing

Messages from `src/errors.ts`, `src/session/session.ts` and `src/session/transaction.ts`. `ConversationBusy`, the third exported error class, is an admission result, not a commit failure ([submissions \(p. 84\)](#)).

Close, then reopen

Recovery starts with `close()`, which works even on a poisoned Session. It runs in a fixed order [spec §2.2](#):

Seal — admission closes at once. New commits and reads reject with `Session is closed`, and a Harness also rejects new conversation and submission operations.

Notify — `subscribeClose()` listeners run synchronously. The Harness stops watches and signals task invocations here, and every document state and watch keeps its last value.

Join — the Harness waits for task, tool and hook invocations to return, “including code that ignores its signal” ([the scheduler \(p. 73\)](#)).

Settle — commits admitted before the seal finish on the line. Then the line clears its listeners and cache and closes storage.

“Close writes no task outcome” [spec §2.2](#). A running task stays `running` in storage, exactly as if the process had died. Once `close()` resolves, “a new Harness may open the same Storage”.

Reopening is ordinary `Harness.open()` on the same storage. It finds what committed and nothing else, and turns surviving `running` tasks back to `pending` ([the scheduler \(p. 73\)](#)). Get new handles; the old ones are dead. A task reopened after its intent was committed may have performed its effect, the case [the effect sandwich \(p. 70\)](#) handles.

IF THE PROCESS DIES

A crash inside `Storage.commit()` is the uncertain case with no process left to poison. SQLite stores each commit in one SQL transaction, and JSONL publishes a commit marker only after its sidecars. Either way, reopening finds the batch entirely present or entirely absent ([SQLite \(p. 144\)](#), [JSONL \(p. 148\)](#)).

WHAT THIS MEANS FOR YOU

- Treat a thrown commit as “nothing happened” unless the Session is now poisoned.
- Never catch a poison error and carry on. Close the Harness and open the storage again.
- If you write a storage backend, throw `StorageRejected` only when you can prove nothing was stored.

Sources: [spec §1](#) (invariant 8), [§2.2](#) (close and reopen), [§4](#), [§10](#), [§12](#); `src/session/session.ts` (commit, poison checks, close); `src/session/transaction.ts` (failure and success settlement, batch assembly); `src/errors.ts`; `src/storage/sqlite/storage.ts` (document copy rejection); `test/session-documents.test.ts` (rollback and poison tests); `test/session-forks.test.ts` (`StorageRejected` rollback); `research/capture/extra-p23/commit-failures.txt`; `research/capture/NOTES.md` (notes 1 and 6)

Conversations and Context

A transcript of immutable entries; heads, edits and forks decide what the model actually sees.

3.1	Entries and the built-in kinds	37
3.2	From transcript to model context	40
3.3	Forks and owned conversations	43
3.4	Reset, handoff and heads	46

03

3.1

Entries and the built-in kinds

An entry is one immutable line of the transcript, with separate fields for what the model reads, what your code reads and where the model's context starts.

A chat transcript is a list of things that happened: a question, an answer, a tool result. Pi Durable stores each one as an **entry** and never edits it afterwards. After this section you can read any stored entry, tell who wrote it, and add your own entry kinds.

One record, three readers

A **conversation** is one transcript, and an **entry** is “an immutable transcript record” in it [spec §1](#). Each entry is read by three different parts of the system, and each reads its own field:

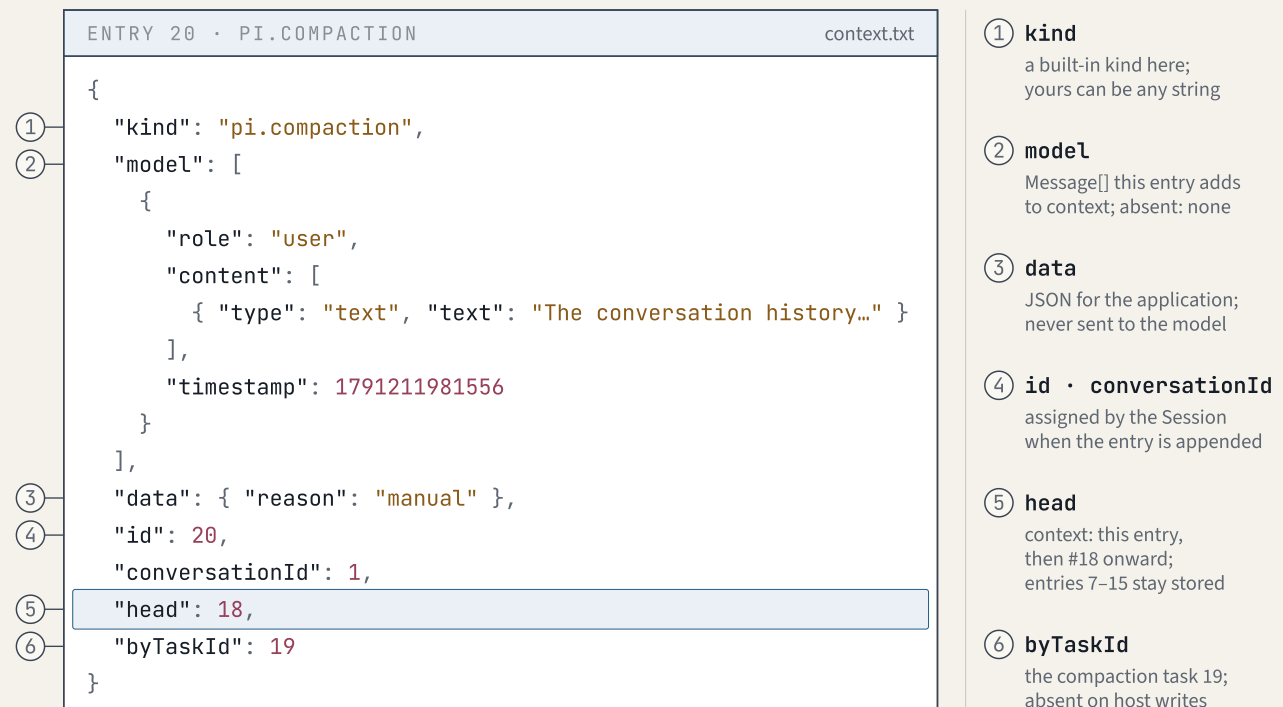
The model reads `model`: the messages this entry adds to the next request.

Your code reads `data`: JSON the model never sees.

Context building reads `head` and `edits`: where the model's context starts, and which earlier entries to hide or replace. Nothing stored is rewritten.

FIG. 3.1 ANATOMY OF AN ENTRY

ANATOMY



An entry keeps what the model sees, what the application sees and where context starts apart. Entry 20 of `research/capture/context.txt`, written by a manual compaction; the summary text is cut after three words.

FIELD	HOLDS	SET BY
<code>id</code>	an <code>EntryId</code> , ordered across the whole Session	the Session
<code>conversationId</code>	the conversation it belongs to	the Session
<code>kind</code>	a string naming what the entry is	the writer
<code>model?</code>	messages for the model; absent for display or bookkeeping	the writer
<code>data?</code>	JSON for views, extensions and your code	the writer
<code>head?</code>	the first entry of the model's context from now on	the writer; "self" means this entry
<code>edits?</code>	hide or replace earlier entries, in context only	the writer
<code>byTaskId?</code>	the task whose commit appended it	the Session

The fields of `EntryRecord`, paraphrased from its field comments in `src/types.ts`.

Entries are written in a **commit**, an atomic save: everything in it is stored together, or nothing is. You never set `id`, `conversationId` or `byTaskId`. `tx.appendEntry()` fills them in, and turns `head: "self"` into the new ID. `byTaskId` is therefore a reliable record of which task wrote the entry. Entries you append from the host with `Conversation.commit()` have none.

The built-in kinds

The Harness writes six kinds. Each is exported as a typed `Entry` token `src/entries.ts`. Only two of them carry `data` [spec §8.1](#).

KIND	MODEL	DATA	WRITTEN BY
<code>pi.user</code>	one user message	—	submissions
<code>pi.assistant</code>	one assistant message, any stop reason	—	generation
<code>pi.system</code>	one system message, as a change	—	generation, before a request
<code>pi.tool-result</code>	one tool result message	<code>{ diagnostics }</code>	tool tasks
<code>pi.reset</code>	none, or the handoff as a user message	—	<code>reset()</code> , a tool's handoff
<code>pi.compaction</code>	the summary as a user message	<code>{ reason }</code>	compaction

From [spec §8.1](#) and `src/entries.ts`. The tokens are `UserEntry`, `AssistantEntry`, `SystemEntry`, `ToolResultEntry`, `ResetEntry` and `CompactionEntry`.

`pi.assistant` records every response, failures included: “answers, failed attempts with their error text and usage, and converted partials with stop reason `aborted`” [spec §8.1](#). The transcript hides nothing, so every token spent stays visible. Context building then leaves the failed ones out of later requests ([context derivation \(p. 40\)](#)).

System entries are changes, not a header

A `pi.system` entry does not hold the whole prompt. It holds a change to the prompt and the tool list, at a point in the transcript. Here is the first one of a captured run:

entry 10, the first pi.system of a conversation research/capture/sqlite-rows.txt

JSON

```
{
  "kind": "pi.system",
  "model": [{
    "role": "system",
    "content": "",
    "sections": { "preamble": "You are a terse assistant." },
    "toolsAdded": [{ "name": "count", "description": "Count from 1 to n, ...", ... }],
    "timestamp": 1791211974788
  }],
  "id": 10, "conversationId": 1, "byTaskId": 9
}
```

Reordered and joined; the tool's description and parameter schema are cut.

Each named section is added or replaced, and `null` removes one. Tools are removed and added the same way. Replaying every system entry in order gives the current prompt and tool set [spec §2](#). The Harness writes a new `pi.system` entry only when the prompt or tools actually change, so a stable conversation has just one. The full story is in [the system prompt \(p. 115\)](#).

Your own kinds

Any non-empty string is a valid kind. `defineEntry()` turns it into a typed token, and the token types data when you append and read:

a typed application entry src/entries.ts · src/types.ts (Tx)

TS

```
const NoteEntry = defineEntry<{ text: string }>("app.note");

// No `model`, so the model never sees it.
await root.commit(
  (tx) => tx.appendEntry(NoteEntry, root.id, { data: { text: "user pinned b.md" } }),
  context,
);

// Undefined when entry `id` is absent or has another kind.
const note = await harness.commit((tx) => tx.entry(NoteEntry, id), context);
```

`app.note` is this book's example kind. The token's `is()` guard narrows any entry by its kind.

Choose fields by reader. What the model must see goes in `model`; what only your code needs goes in `data`. An entry with only `edits` is common: the captured `app.correction` replaces an earlier answer in context and adds no message of its own [research/capture/context.txt](#), stage 5.

WHAT THIS MEANS FOR YOU

- Put model-facing text in `model` and everything else in `data`.
- Never change an entry. Append a new one with `edits` instead.
- Use `byTaskId` to see which task wrote an entry when you debug.
- While a run is active, add entries with a write submission, not a raw append [spec §12](#) ([submissions \(p. 84\)](#)).

Sources: [spec §1](#), [§2](#), [§8.1](#), [§12](#); [src/types.ts](#) (`EntryRecord`, `EntryDraft`, `Tx.appendEntry`, `Tx.entry`); [src/entries.ts](#); [src/harness/types.ts](#) (`CompactionReason`); [src/session/transaction.ts](#); [research/capture/context.txt](#); [research/capture/sqlite-rows.txt](#)

From transcript to model context

The transcript records everything that happened; nine fixed rules turn it into a request the provider will accept, without rewriting anything stored.

A real transcript is messy. A reply was cut off by a crash. A tool call never got its result. Two results arrived out of order, with a user message between them. Sent as is, a provider would reject it. After this section you can predict, message by message, how committed history becomes model context, before request hooks change it.

Two readings of one transcript

The stored transcript only grows. The **context** is what the model sees, and it is computed from that history every time. `Conversation.context()` shows both readings side by side [src/harness/types.ts](#) (`ContextView`):

entries — the raw entries in the active range. A UI shows these.

messages — messages derived from committed history, before preparation updates and request hooks.

head — the entry that decides where the range starts, if any.

contributions — what each entry adds, before tool results are reordered. Useful when you debug.

The nine rules

The spec defines the derivation as nine ordered rules [spec §2.1](#). The right column points into the worked example below.

#	RULE, IN PLAIN WORDS	EXAMPLE
1	Find the newest entry that has a head.	#20
2	Context starts at that head, or at the beginning if there is none.	#10
3	Read every entry from there to the end.	#10–#20
4	Apply edits: the newest edit of an entry wins; <code>omit</code> hides it, <code>replace</code> swaps its messages.	#19 replaces #11
5	Put the head entry first, then the others in order.	#20 first
6	Keep every system message where it was written.	#10 kept
7	Move each call's results directly after the assistant message that made the call, in call order.	#16, #14 after #13
8	Add an error result for every call that has none; drop results with no call.	call-c
9	Leave out entries with no <code>model</code> , and assistant messages that stopped as <code>aborted</code> , <code>error</code> or <code>deferred</code> .	#12, #18

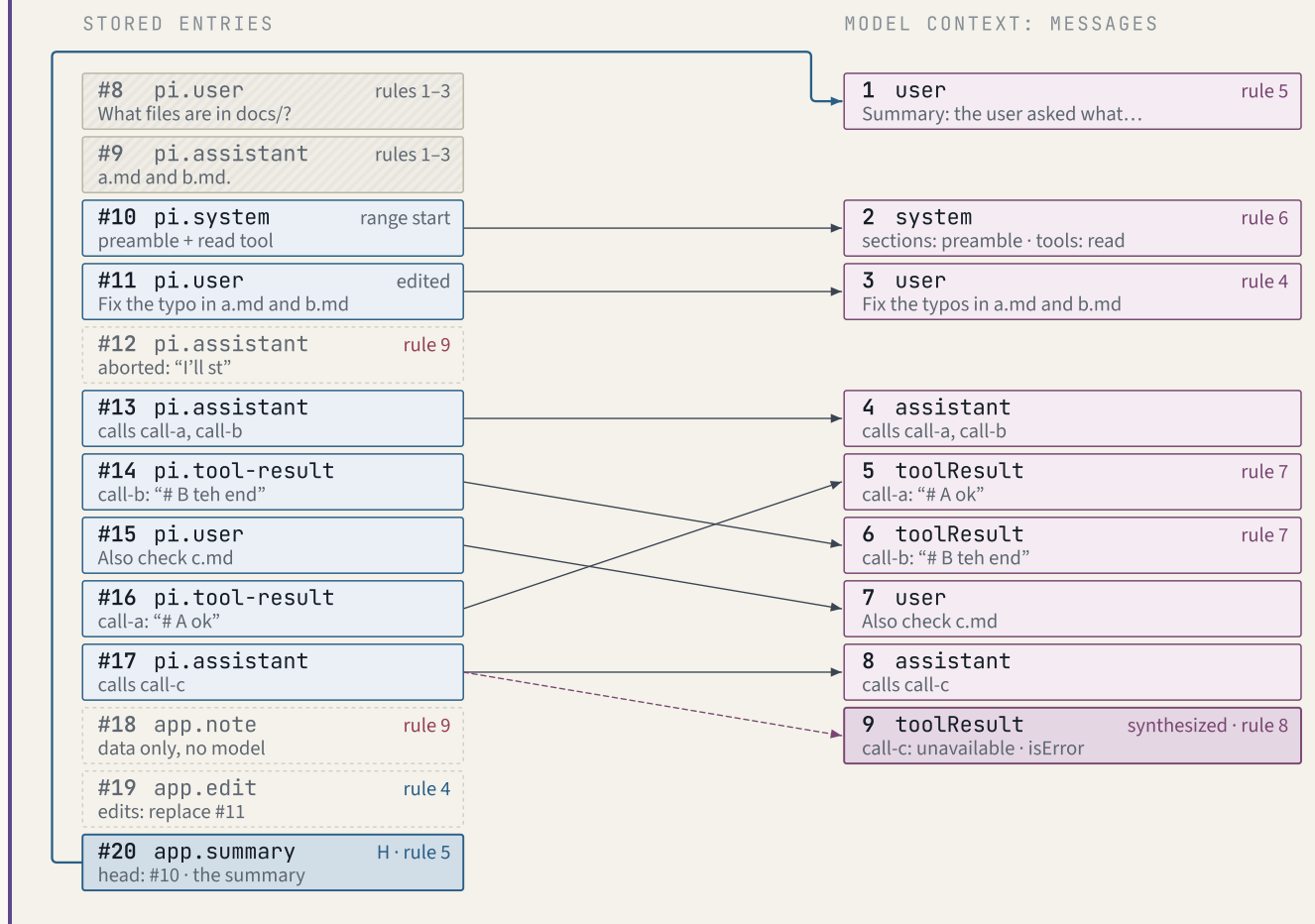
Paraphrased from [spec §2.1](#).

A worked example

This transcript was written entry by entry and read back with `context()`. An agent was asked to fix two files. One reply was aborted. Two reads came back out of order, with a user message between them. A third read never finished. The application then added a note, corrected the user's wording, and wrote a summary whose head skips the opening exchange.

FIG. 3.2 ONE TRANSCRIPT, TWO READINGS

FLOW



Context derivation reads a stored transcript as nine request messages. Entries #8–#20 and the derived messages are from research/capture/extra-p23/context-derivation.txt. Hatched entries lie before the head’s range; dashed ones are in the range but contribute no message. Message 9, drawn deeper, has no entry: derivation synthesizes it because call-c has no stored result.

The range (rules 1–3). The newest entry with a head is the summary #20, and its head is #10. So context covers #10 through #20. Entries #8 and #9 are still stored and still readable; they are just not sent.

Edits (rule 4). #19 replaces #11. Message 3 is the corrected text, “Fix the typos in a.md and b.md”, not the original. An edit only counts if it sits inside the range.

Order (rules 5 and 6). The summary #20 becomes message 1. The system entry #10 stays where it was, as message 2.

Left out (rule 9). The aborted reply #12 and the note #18 add nothing. Neither does #19, which only edits.

Tool results (rules 7 and 8). #13 called call-a, then call-b. Their results, #16 and #14, move up to sit right after #13, in call order and ahead of the user message #15. The last assistant message, #17, called call-c and nothing answered, so a result is made up for it:

message 9, synthesized for call-c research/capture/extra-p23/context-derivation.txt

JSON

```
{
  "role": "toolResult",
  "toolCallId": "call-c",
  "toolName": "read",
  "content": [{ "type": "text",
    "text": "Tool result unavailable: history ends before this call completed." }],
  "isError": true,
  "details": { "reason": "missing_result" },
  "timestamp": 1791300010000
}
```

The text and details are fixed in `src/harness/context.ts`. The timestamp is copied from the assistant message that made the call.

Why these rules exist

Each rule handles a state that a durable transcript can legitimately be in.

Crashes and aborts — an interrupted reply is stored as an aborted `pi.assistant` entry ([generation](#) (p. 91)). Rule 9 keeps its half-finished text, and any tool calls in it, out of later requests.

Interrupted tools — a tool task that fails or is abandoned writes no result. Rule 8 tells the model the call did not finish [spec §5.4](#).

Parallel tools and steering — results land in the order they finish, and a user can steer between them ([submissions and the inbox](#) (p. 84)). Providers require results right after their call; rule 7 restores that.

Forks — a fork taken between a call and its result inherits the call but not the result. Example 09 forks at an assistant message with two calls; the fork's context ends with two error results [run 09](#).

WHAT THIS MEANS FOR YOU

- To see what the model will get, call `context()` and read messages. Do not rebuild it from entries yourself.
- A crash never produces an invalid request, so you need no cleanup code for the transcript.
- Reading context never writes. Two reads of the same transcript give the same messages.

Sources: [spec §2.1](#), [§2.2](#) (ContextView), [§5.4](#), [§8.1](#); `src/harness/context.ts`; `src/harness/types.ts` (ContextView); `ex 09`; `run 09`; `research/capture/extra-p23/context-derivation.txt` (`script test/capture-p23-context.ts`)

Forks and owned conversations

A fork branches a conversation at one entry and shares the history before it; an owned conversation belongs to a task, so aborting the task reaches it.

Users want to go back to an earlier message and try something else, while keeping the original. Agents want to start a side conversation for a subagent. Both are new conversations. After this section you can create each kind and predict what it sees.

Four ways to get a conversation

CALL	GIVES YOU
<code>harness.root(context, options)</code>	conversation 1, created on first call; later calls write nothing
<code>harness.createConversation(options, context)</code>	a new, empty conversation; ownership is required
<code>conversation.fork(at, options, context)</code>	a branch that shares history up to entry at
<code>tx.createConversation(), tx.forkConversation()</code>	the same, inside any commit, such as a tool's

From spec §2.2 and `src/harness/harness.ts`. The options are `ownership`, `agent` and `init`.

Creation is one commit. It writes the conversation, its five built-in documents ([the built-in documents \(p. 60\)](#)), any `agent` setting, and whatever your `init(tx, id)` writes. Nothing is half-created [spec §2.2](#).

Creating a conversation and sending its first message are separate calls. If your process can crash between them, write a key in `init` so you can find the conversation again, then submit with a request ID so a retry is not sent twice [spec §2.2](#).

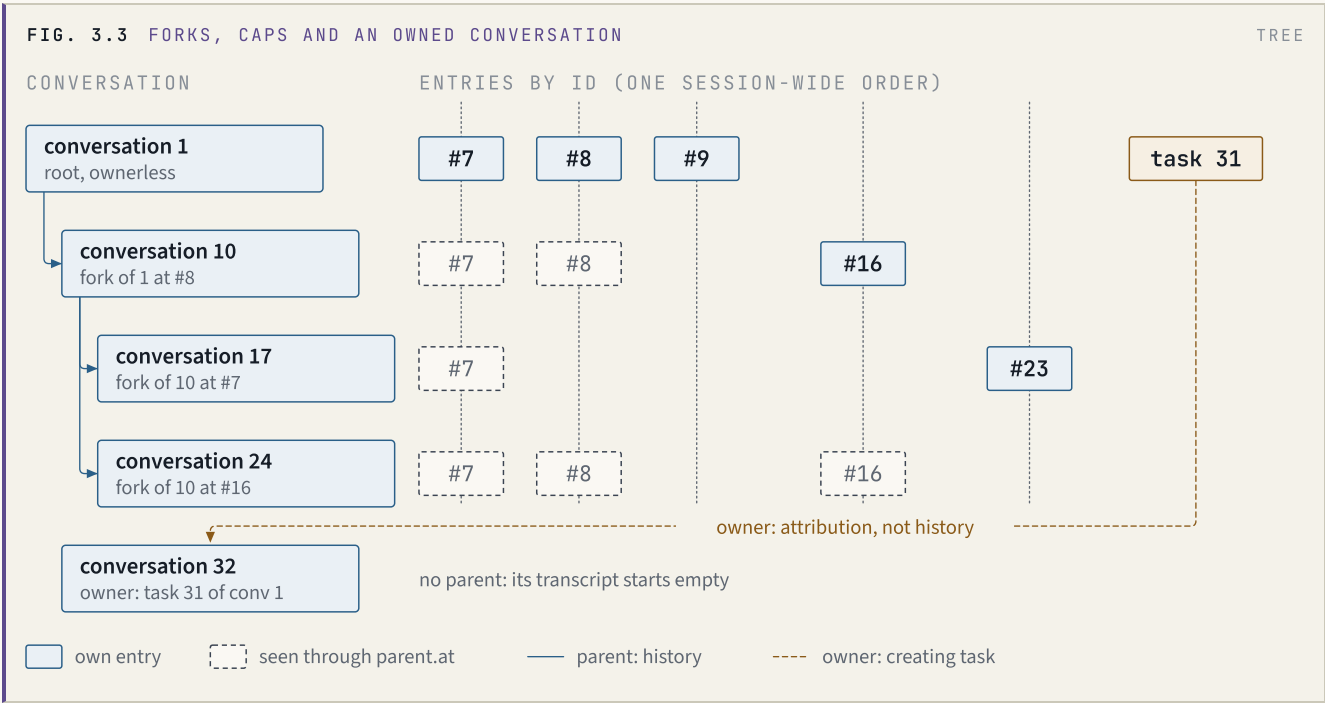
Forks

A **fork** is a new conversation that starts from one entry `E` of a parent. Its record stores `parent: { conversationId, at }`. Nothing is copied. The fork sees its own entries, then the parent's entries up to `E`, then the grandparent's up to its own cap, and so on [spec §2.1](#), [§3.7](#). Each ancestor's entries are cut off at the smallest cap on the way up.

```
$ node --conditions=source --experimental-strip-types test/capture-p23-context.ts
root 1: #9:"three" #8:"two" #7:"one"
fork 10: #16:"fork-only" #8:"two" #7:"one"
grand 17: #23:"grandchild-only" #7:"one"
fork2 24: #16:"fork-only" #8:"two" #7:"one"
grand.fork(#9) -> Entry 9 is not visible from conversation 17
```

From `research/capture/extra-p23/forks.txt`; entries newest first. Conversation 17 cannot see #9, so it cannot fork there.

Heads cross forks too. A fork taken after a compaction starts its context at the parent's summary `research/capture/context.txt`, stage 6. A fork taken between a tool call and its result gets an error result in its context ([context derivation \(p. 40\)](#)).



A fork sees its own entries, then each ancestor's entries up to the parent . at cap. Conversation 32 has no parent; task 31, which lives in conversation 1, owns it. Conversations, entries and visibility from research/capture/extra-p23/forks.txt.

What a fork inherits

STATE	IN THE FORK
entries	shared up to E, never copied
fork: "asOf" documents	a copy of the parent's value at E
fork: "current" documents	a copy of the parent's value now
fork: "initial" documents	nothing; created fresh on first use
pi.agent	the parent's agent settings as of E
pi.provider	a new provider session ID, never the parent's
live state, inbox, usage	empty
tasks and task documents	never copied
Session documents	shared, not copied

From spec §2.2, §3.7. Document fork policies are in [defining a document](#) (p. 50).

The capture shows the asOf rule. The root's agent instructions said "v1: be brief" when #8 was written and "v2: be thorough" later. The fork at #8 starts with v1, and it has its own provider session ID `research/capture/extra-p23/forks.txt`.

Owned conversations

An **owned conversation** belongs to a task. Create it with `{ kind: "task", taskId }` ownership; the record then stores `owner: { conversationId, taskId }`. The task can be created in the same commit:

a task and the conversation it owns, in one commit `test/capture-p23-context.ts`

TS

```
const owned = await harness.commit(async (tx) => {
  const taskId = await tx.createTask(Supervisor, null, {
    ownership: { kind: "conversation" }, conversationId: root.id, background: true,
  });
  const child = await tx.createConversation({ ownership: { kind: "task", taskId } });
  return { taskId, child };
}, context);
// child: {"id":32,"owner":{"conversationId":1,"taskId":31}}
```

The comment is the captured output in `research/capture/extra-p23/forks.txt`.

Ownership is about work, not history. The owned conversation starts empty. It gets a one-time copy of the agent settings of the owner's conversation; later changes there do not reach it [spec §2.2](#). What the owner edge does is connect the two for abort and idle waits. Aborting the task reaches the conversation's non-background work, and waiting for the task to go idle includes that work ([ownership and structured concurrency \(p. 76\)](#)). Subagents are built on exactly this ([subagents \(p. 121\)](#)).

The owner must be alive. A task that is finishing, finished or being aborted cannot gain a new conversation, even in the same commit [spec §3.3](#).

WHAT THIS MEANS FOR YOU

- Use a fork for “try again from here”. The original conversation is untouched.
- Use an owned conversation for work that should stop when its task stops, such as a subagent.
- Pick each document's fork policy on purpose; it decides what a branch starts with.

Sources: [spec §2](#), [§2.1](#), [§2.2](#), [§3.3](#), [§3.7](#); `src/harness/harness.ts` (`root`, `createConversation`, `fork`); `src/harness/agent.ts` (`createAgent`); `src/session/transaction.ts`; `src/session/forks.ts`; `ex 03`; `run 09`; `research/capture/extra-p23/forks.txt`; `research/capture/context.txt`

3 . 4

Reset, handoff and heads

A head moves the start of the model's context forward without deleting anything; reset, handoff and compaction all work this way.

Sometimes the model should forget. The context window is full, or the user wants a fresh start on the same task. Pi Durable never deletes the old messages; it just tells the model to start reading later. After this section you can start a fresh context, with or without a handoff note, and predict what the model sees next.

What a head is

A **head** is an entry field that says “the model's context starts here” `src/types.ts`. An entry that has one is a **head marker**. Only the newest marker counts, and context derivation puts it first (rules 1, 2 and 5 (p. 40)). Older entries stay in storage; they are just no longer sent `spec §2.1`.

There are two ways to point a head:

Back at a kept entry — a compaction summary heads the first entry it keeps. Context becomes “summary, then the kept tail” (compaction (p. 97)).

At itself — a reset heads itself. Context becomes the reset entry alone, plus whatever comes after it.

reset() and handoff

`Conversation.reset(handoff, context)` starts a new context. It submits a `pi.reset` entry with head: `"self"`. If you pass handoff text, the entry carries it as a user message `spec §2.2`.

reset(text) — the model's context becomes that one message. In stage 4 of the figure below: `"We were doing arithmetic for Ada. Continue from 5+5."`

reset(undefined) — the entry has no message, so context is empty until the next input `research/capture/extra-p23/heads.txt`.

A reset is queued like any other write. If the conversation is idle, it lands at once. If a run is active, it waits for the next **boundary**, a safe point between model and tool steps (submissions and the inbox (p. 84)). Landing during a tool round ends the run, and its inputs are reported `unanswered` with reason `reset`. Landing after an answer changes nothing for that answer `spec §6`. `reset()` returns once the submission is committed, not when the entry lands; watch the conversation to see it.

The figure follows one captured conversation through five stages: three questions, a compaction, one more turn, a reset, and a last turn with a correction.

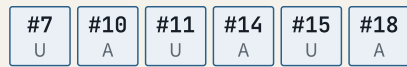
FIG. 3.4 THE ACTIVE RANGE MOVING WITH HEADS

TIMELINE

STAGE

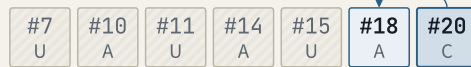
STORED ENTRIES OF CONVERSATION 1

- ① **three questions**
no head yet



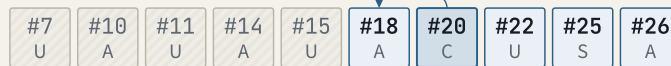
context().entries: #7 #10 #11 #14 #15 #18

- ② **root.compact()**
#20 sets head #18



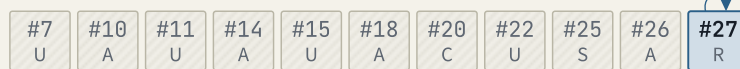
context().entries: #20 #18

- ③ **one more turn**
same head marker



context().entries: #20 #18 #22 #25 #26

- ④ **root.reset(...)**
#27 heads itself



context().entries: #27

- ⑤ **one more turn**
#34 edits #33



context().entries: #27 #29 #32 #33 #34

 in the active range head marker stored, outside context
U pi.user · A pi.assistant · S pi.system · C pi.compaction · R pi.reset · E app.correction

A head moves the start of context forward; every entry stays stored. Stages 1–5 of research/capture/context.txt; the context().entries lines are copied from it.

A tool can ask for the same thing. If its result's control carries handoff: string, then when the tool round finishes the run ends with a pi.reset entry holding that text, “exactly what reset(handoff) writes” spec §7.3. If several tools ask, the last in call order wins.

Stale head writes

A head may only move forward. A head write that points before the current start would bring back history that was cut, so it is refused as **stale**: it settles unanswered with reason stale and writes nothing spec §6. A reset heads itself, so it is never stale.

two head writes after a reset at #9 research/capture/extra-p23/heads.txt

TEXT

```
context entries: [#9 pi.reset, #11 pi.user]
```

```
write with head=#8 (before the start): status "unanswered", reason "stale"
```

```
write with head=#11 (inside the range): status "done", entry 13
```

```
context entries now: [#13 app.summary head=11, #11 pi.user]
```

```
stored entries (newest first), nothing deleted:
```

```
  [#15 pi.reset, #13 app.summary, #11 pi.user, #9 pi.reset, #8 pi.assistant, #7 pi.user]
```

The submission records are shortened to their status. #15 is a later plain reset().

The same rule decides between competing compactions: a summary that would move the start backwards settles `stale` (compaction (p. 97)).

What a head never does

Delete — every entry above is still stored and still returned by `entries()`. You can still fork at an old entry.

Change the provider ID — reset and compaction keep the conversation's `pi.provider` session ID [spec §2.2](#).

Lose the system prompt — after a head, the next request writes a fresh, complete `pi.system` entry (the system prompt (p. 115)).

WHAT THIS MEANS FOR YOU

- Use `reset(text)` to start fresh but tell the model where it left off.
- Write your own head entries through a write submission, not a raw `tx.appendEntry()`. Only the submission path checks for stale heads, and a raw write can leave a mounted view out of date [spec §12](#).
- If a write ends `stale`, a newer head already moved past its target. Recompute it from the current context.

Sources: [spec §2.1](#), [§2.2](#) (reset, `pi.provider`), [§6](#), [§7.3](#), [§7.4](#), [§8.1](#), [§12](#); `src/types.ts` (`EntryRecord.head`); `src/harness/harness.ts` (`reset`); `src/harness/submissions.ts`; `src/harness/inbox.ts`; `src/harness/generation.ts`; `src/harness/types.ts` (`ToolControl`); `research/capture/context.txt`; `research/capture/extra-p23/heads.txt`

Documents

Typed JSON state committed with the transcript, stored as Chord operations and occasional complete bases.

4.1	Defining a document	50
4.2	Access, creation and lifetimes	53
4.3	Bases, deltas and migrations	56
4.4	The built-in documents	60

04

4.1

Defining a document

A definition is a typed token that fixes a document's kind, version, owner and fork behaviour, and the stored record remembers every one of those choices.

An agent needs state that is not chat: a todo list, a plan, the user's settings. If that state were stored separately from the transcript, a crash could leave the two out of step. A **document** is that state, stored in the same **commit** as the entries. A commit is an atomic save: everything in it is stored together, or nothing is. After this section you can define one and choose how it behaves when a conversation is forked.

A document is one JSON object that changes commit by commit. Storage keeps each change either as a small list of edits or as the whole value (bases and deltas (p. 56)). The Harness keeps its own state in five built-in documents (the built-in documents (p. 60)), and you define yours with the same function, `defineDoc()`.

a per-conversation todo list README §Your Own State

TS

```
const Todos = defineDoc<{ items: string[] }>({
  kind: "app.todos",
  version: 1,
  scope: "conversation",
  history: "latest", // or "rewindable" to read old values with snapshotAsOf()
  fork: "initial", // what a fork starts with: "initial", "current", or "asOf"
  initial: () => ({ items: [] }),
});

await root.commit(async (tx) => {
  (await tx.doc(Todos, root.id)).items.push("write docs");
}, context);
```

The import line is omitted. "asOf" is only allowed together with `history: "rewindable"`; the compiler enforces it.

The definition is a token

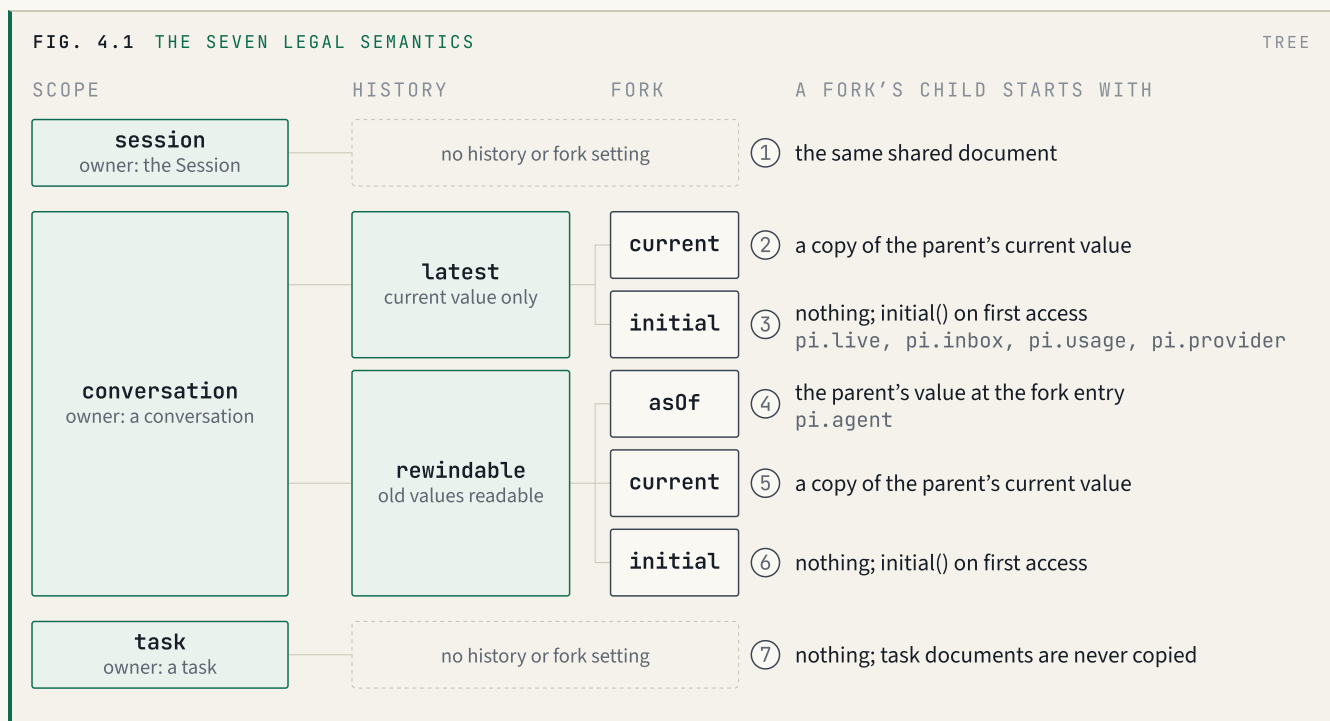
`defineDoc()` returns a token: a plain object that carries the definition and the value type. It checks only that `version` is a positive integer `src/documents.ts`. Nothing is registered. Every read and write passes the token, as in `tx.doc(Todos, root.id)`, and the token supplies the type, the initial value and the migration for that one call `spec §3.3`. Define each token once in a module and import it everywhere. Two different definitions of the same kind are unsupported, and nothing detects them `spec §12`.

FIELD	TYPE	WHAT IT DECIDES
kind	string	The stored name. It is part of the public protocol and cannot be renamed.
version	positive integer	Version of the value's shape; stored with every change.
scope	"session" · "conversation" · "task"	Who owns it and how long it lives.
history	"latest" · "rewindable"	Conversation only. Whether old values stay readable.
fork	"current" · "initial" · "asOf"	Conversation only. What a fork's copy starts with.
initial()	() => T	The value of a new document; must be a JSON object.
migrate?	(value, fromVersion) => T	Upgrades an older stored value when it is read (4.3 (p. 56)).
checkpointWhen?	(value, ops, info) => boolean	When to store the whole value instead of the edits (4.3 (p. 56)).

The definition fields, from `src/types.ts` (`CommonDocDefinition`, `DocumentSemantics`).

Scope, history and fork

Scope decides the owner and the lifetime. A `session` document is shared by every conversation and lives until you retire it. A `conversation` document belongs to one conversation. A `task` document belongs to one task and is retired automatically when the task ends `spec §3.1`. Only conversation documents choose a history and a fork policy, which gives seven legal combinations.



Scope fixes owner and lifetime; only conversation documents choose history and fork. The union `DocumentSemantics` of `src/types.ts`; fork results from `spec §3.7`. The right column names the built-in documents that use a combination.

history: "latest" keeps only the current value. "rewindable" keeps every stored change, so `snapshotAsOf(token, conversationId, entryId)` can return the value as it was when any entry was written. The cost is storage: old changes are never deleted (4.3 (p. 56)).

fork decides what a forked conversation starts with (`forks` (p. 43)). "current" copies the parent's value now. "asOf" copies the value at the fork entry, which only a rewindable document still has. "initial" copies nothing; the fork's first `tx.doc()` creates

a fresh one. In run 02, a notes document reads `after` `goodbye` in the parent, while a fork taken at an earlier entry starts with `after` `hello`.

Families

A **family** is one definition for many documents under one owner, told apart by a string key. Think “one review document per file”. `defineDocFamily()` takes the same fields plus `family: true`, and its initializer receives a **seed**:

```
one review document per file docs/pico-v5-chord-usage.md §2 TS

const ReviewDoc = defineDocFamily<ReviewState, ReviewInput>({
  kind: "app.diff-review", version: 1, family: true, scope: "conversation",
  history: "latest", fork: "current",
  initial: seed => ({ path: seed.path, patch: seed.patch, comments: [] })),
});

await tx.doc(ReviewDoc, conversationId, review.key, review.seed);
```

The key is part of the address; the seed is not. The seed is used once, when the member is first created, and ignored after that `spec §12`. In the capture, a second `tx.doc()` on key `src/a.ts` passed the seed `{"path": "IGNORED"}`, and the member kept `"path": "src/a.ts"`. To change a member, edit it.

The choices are stored

Each stored document records its kind, scope, history and fork policy, so existing data still makes sense when the code that defined it is missing `spec §3.2`. Every access checks the token against that record. If you change `fork` from `"initial"` to `"current"` in your code, existing documents do not change. They become unreadable through the new token, which throws `Document 4 (app.todos) does not match the supplied definition semantics src/documents.ts`.

WHAT THIS MEANS FOR YOU

- State that describes the transcript, like a plan: `rewindable + asOf`, so a fork from an old entry sees the matching plan. Example 27’s `app.plan` and the built-in `pi.agent` do this.
- State that belongs to the user, like a draft: `current`, so a fork takes it as it is now `docs/pico-v5-chord-usage.md §2`.
- State that belongs to one branch, like a sandbox or spend: `initial`. A copied spend counter would count twice, so “a fork starts at zero” `spec §8.6`.
- Treat kind, scope, history and fork as permanent. Only the value’s shape can evolve, through `version` and `migrate()`.

Sources: `src/documents.ts`; `src/types.ts`; `spec §1`, `§3.1`, `§3.2`, `§3.3`, `§3.7`, `§8.6`, `§12`; `README §Your Own State`; `docs/pico-v5-chord-usage.md §2`; `ex 02`, `ex 27`; `run 02`; `research/capture/extra-p4/documents.txt` (`test/capture-p4-documents.ts`)

4 . 2

Access, creation and lifetimes

Only `tx.doc()` creates a document; every read returns `undefined` for one that does not exist, and scope decides what ends a document's life.

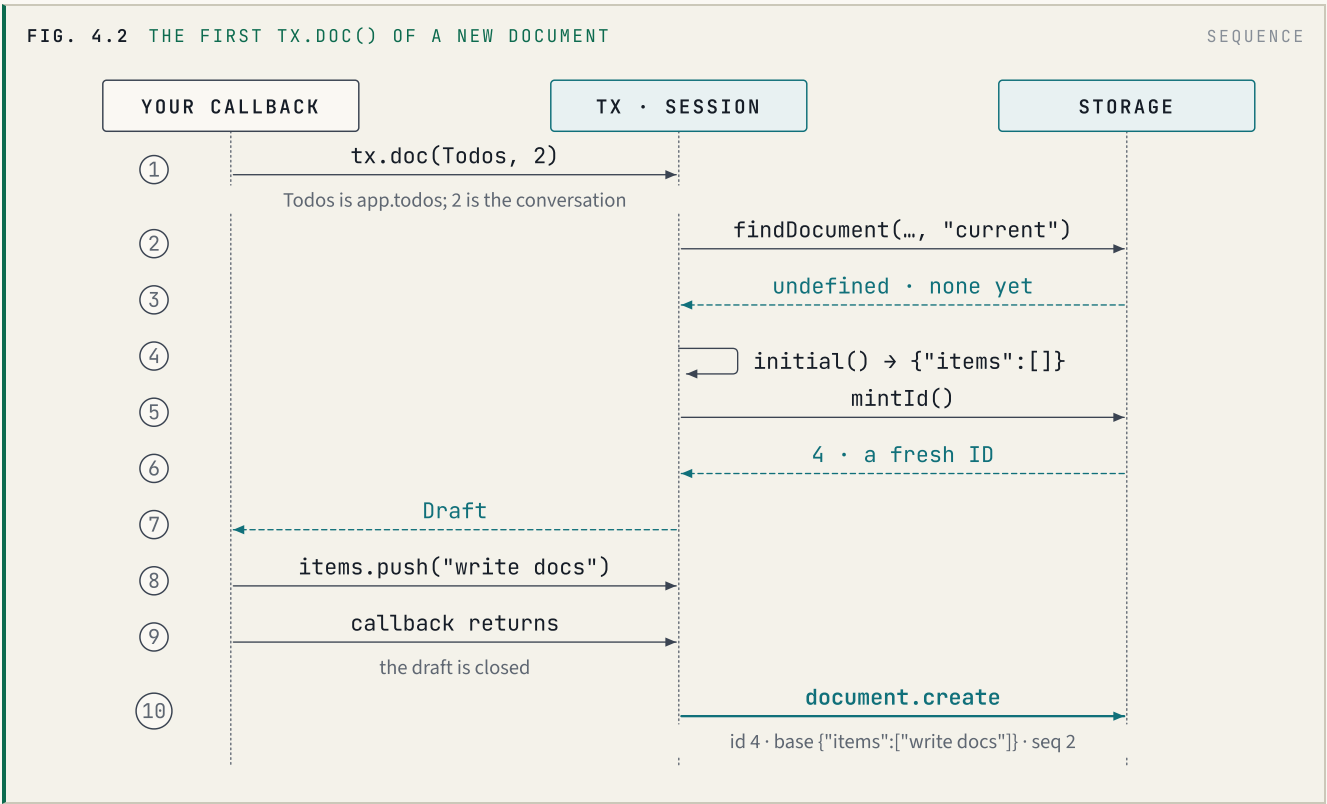
You never “create” a document explicitly. The first time a commit asks for it, it exists. Reads are different: they never create anything. After this section you can create, read and retire documents, and say when each one exists.

tx.doc() is get-or-create

Inside a commit, `tx.doc()` returns a **draft**: an editable view of the document for this commit only. If the document does not exist yet, the same call creates it. The arguments follow the token's scope:

SCOPE	SINGLE DOCUMENT	FAMILY MEMBER
session	<code>tx.doc(Doc)</code>	<code>tx.doc(Doc, key, seed)</code>
conversation	<code>tx.doc(Doc, conversationId)</code>	<code>tx.doc(Doc, conversationId, key, seed)</code>
task	<code>tx.doc(Doc, taskId)</code>	<code>tx.doc(Doc, taskId, key, seed)</code>

The `Tx.doc` overloads in `src/types.ts`. Reads take the same owner and key, without the seed, plus a `Context`.



Creation is a write: the first `tx.doc()` of a missing document stores its whole value in that commit. From `src/session/transaction.ts`; ID 4, seq 2 and the value are from `research/capture/extra-p4/documents.txt`.

The figure follows one creation. The Session finds no current document (steps 2–3), so it runs `initial()`, asks storage for a fresh ID, and hands your code a draft (steps 4–7). Asking again in the same commit returns the same draft [spec §3.3](#). Your edits

go into the draft (step 8). When your callback returns, the draft is closed, and the commit stores the whole final value, `{"items": ["write docs"]}`, not the empty initial one (steps 9–10). Other readers see it only after the commit succeeds (the [commit](#) (p. 29)).

READS THAT WRITE

`tx.doc()` on a missing document is a write, even if your code only reads the draft. Use `snapshot()` to look without creating. A draft kept after its commit is dead: assigning to it “throws: the draft was revoked” [spec §3.4](#).

Reads never create

“Snapshot, state, and watch lookup return `undefined`” for a document that does not exist [spec §12](#). Treat `undefined` as the default value. In example 11, a prompt section shows nothing while the todo list is missing, and the tool’s `tx.doc()` creates the list on its first write [ex 11](#).

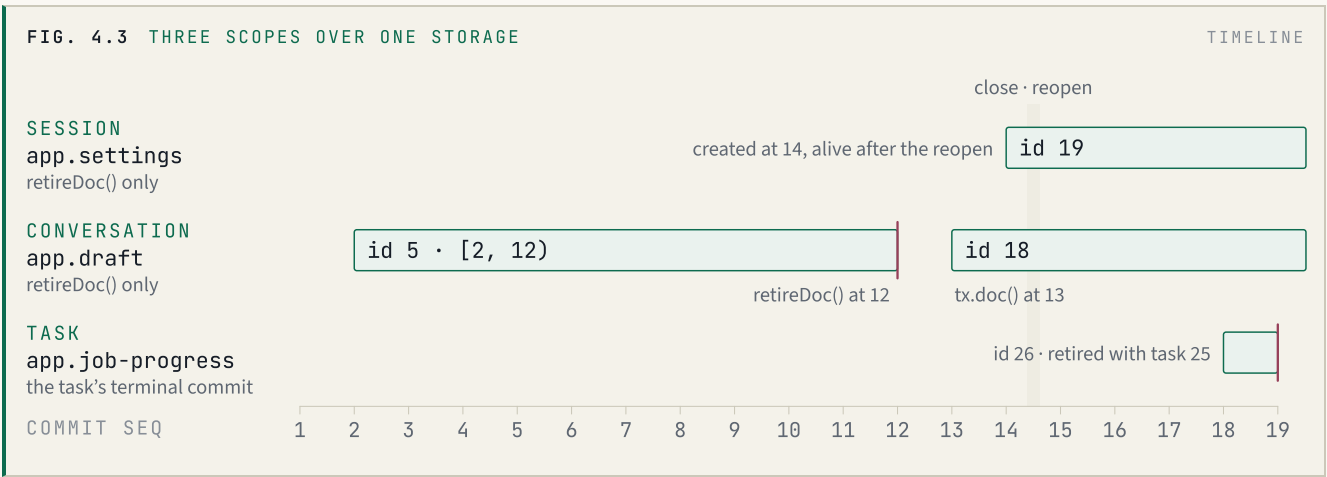
METHOD	RETURNS
<code>snapshot()</code>	the current value
<code>snapshotAsOf()</code>	the value when an entry was written; rewindable documents only
<code>documentState()</code>	a read-only live state (8.1 (p. 125))
<code>watchDoc()</code>	a stream of committed changes (8.2 (p. 129))

The read methods. `snapshot()` and `snapshotAsOf()` are available on the Session, the Harness, tasks, tools and hooks (`src/types.ts`, `src/harness/types.ts`).

A snapshot is shared, not copied, and it is not frozen. If your code changes it, later readers see the change [spec §3.4](#). Copy it before you modify it.

What ends a document

Each create-to-retire life of a document is an **incarnation**, with its own ID. Retiring a document and creating it again gives a new ID; IDs are never reused [spec §3.2](#). The capture retired `app.draft` (ID 5) at commit 12 and created it again (ID 18) at commit 13. Watchers see a retirement as the value `null`.



An incarnation lives from its creating commit up to its retiring commit; scope decides what retires it. One SQLite file: a plain Session wrote seq 1–14, then the file was reopened as a Harness for seq 15–19. `research/capture/extra-p4/documents.txt` and `document-publications.txt`.

Session and conversation — end only when your code calls `retireDoc()`. Nothing in the Harness does. The Session document `app.settings` survived a close and reopen with `{"theme": "dark"}`.

Task — ends in the commit that finishes its task [spec §3.3](#). Task 25's last commit wrote `done = 2` and finished the task, so that last edit was never readable. Afterwards `snapshot()` returns `undefined` and `tx.doc()` throws `Task 25 is terminal`.

Documents every conversation gets

A Harness creates its five built-in documents in every commit that creates or forks a conversation, through a creation hook `src/harness/harness.ts`. You can add your own with `HarnessOptions.conversationCreated(tx, conversation)`, which runs right after the built-ins in the same commit. Use it for documents every conversation must have. Use the `init` option of `createConversation()` or `fork()` for data of one call. If either throws, the whole creation fails. A plain Session has no hook and no built-in documents.

WHAT THIS MEANS FOR YOU

- Use `snapshot()` to look, `tx.doc()` to change.
- Handle `undefined` from every read; it means “not created yet” or “retired”.
- Anything that must outlive a task goes into its result, an entry or a longer-lived document, in the task's final commit [spec §12](#).
- Kinds starting with `pi.` belong to the Harness. Nothing enforces it, but reusing one collides with Harness behaviour [spec §12](#).

Sources: `src/session/transaction.ts` (`doc`, `retireDoc`); `src/session/session.ts`; `src/types.ts` (`Tx.doc`, `DocumentReader`); `src/harness/harness.ts` (`conversationCreated`); `src/harness/types.ts` (`HarnessOptions`); [spec §2.2](#), [§3.2–3.4](#), [§12](#); [ex 11](#); [research/capture/extra-p4/documents.txt](#), [document-publications.txt](#) ([test/capture-p4-documents.ts](#))

4.3

Bases, deltas and migrations

Storage keeps a document as occasional bases, whole copies of the value, plus the small edits between them; your definition decides when to write a base.

Saving the whole document on every change wastes space. Saving only the edits makes reads slower and slower, because each read replays every edit. Pi Durable stores mostly edits and, now and then, a whole copy called a **base**. After this section you can read a document's rows in a storage dump, choose when bases are written, and change a document's shape safely.

Two kinds of record

Every stored change is one of two things `src/types.ts (DocumentContent)`:

Base — the whole value. Written when the document is created, when its version changes, and when your `checkpointWhen()` returns `true`.

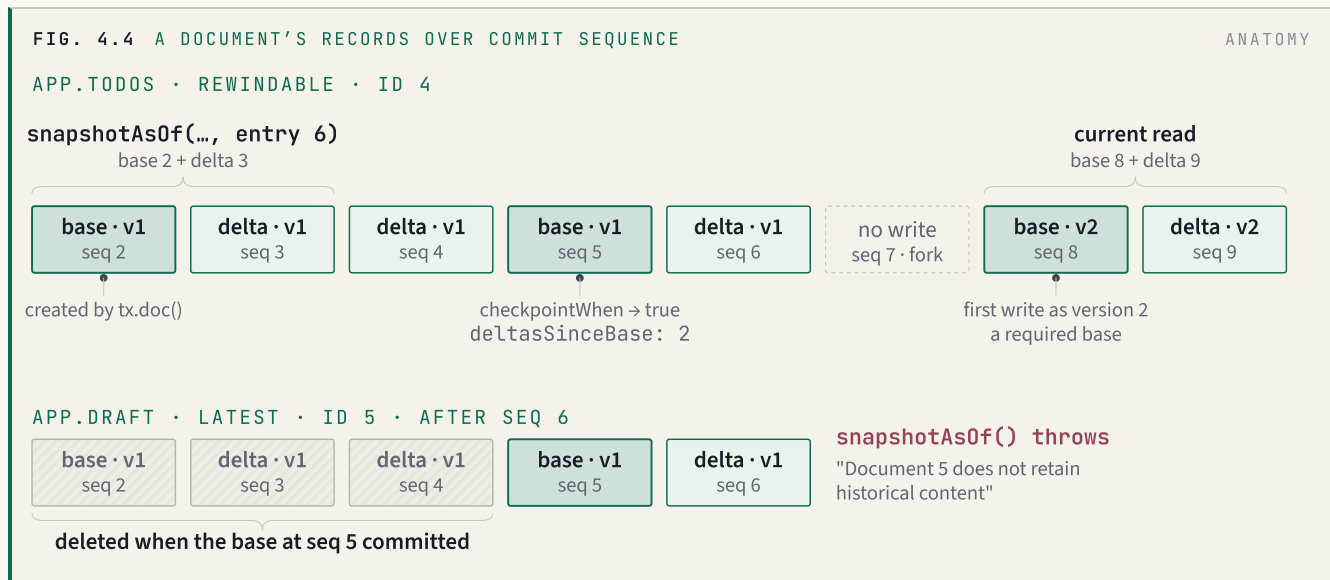
Delta — the list of edits this commit made, as Chord operations. Written for every other change. Chord generates them from ordinary draft edits; you never write them by hand.

A commit that changes nothing writes nothing. Both kinds record the definition version they were written under. The capture ran five commits against two documents with the same rule, `info.deltasSinceBase >= 2`. `app.todos` (ID 4) is rewindable; `app.draft` (ID 5) is latest.

document_revisions after five commits					research/capture/extra-p4/documents.txt	TEXT
document_id	seq	kind	version	content		
4	2	base	1	{ "items": ["write docs"] }		
4	3	delta	1	[["p", ["items"], 1, 0, ["fix build"]]]		
4	4	delta	1	[["p", ["items"], 2, 0, ["ship"]]]		
4	5	base	1	{ "items": ["fix build", "ship"] }		
4	6	delta	1	[["p", ["items"], 2, 0, ["celebrate"]]]		
5	5	base	1	{ "text": "Dear team, we shipped." }		
5	6	delta	1	[["s", ["text"], "Done."]]		

SQLite rows, one per stored change. "p" is an array splice (path, index, delete count, items) and "s" sets a value.

A read finds the newest base and applies every delta after it, in order. ID 5 has no rows before seq 5, because a base on a latest document lets storage delete everything older.



A read starts at the newest base and replays the deltas after it; only rewindable documents keep older records. Rows from research/capture/extra-p4/documents.txt. Seq 7 is a fork, which wrote nothing for ID 4; seq 8 and 9 come from the migration below.

The checkpoint predicate

`checkpointWhen(value, ops, info)` is asked once for each ordinary change. If it returns `true`, that change is stored as a base. `info.deltasSinceBase` is the number of deltas since the last base, not counting this change; the Session tracks it, so the check costs no storage read. In the capture the rule saw 0, then 1, then 2, so seq 5 became a base. If the predicate throws, the whole commit fails.

Two styles work well. A count, such as `info.deltasSinceBase >= 31`, caps how many deltas a read replays. A state test, as in `pi.live`, takes a base whenever the document is back to a small resting value. The choice affects storage only: watchers and views receive the same edits either way (8.1 (p. 125)).

NO CHECKPOINT, NO LIMIT

Without `checkpointWhen`, a document stores one base at creation and deltas forever. Reads get slower and storage “can grow without bound while the document is live” [spec §12](#). Storage never takes a base on its own [spec §3.5](#).

What a base lets storage delete

For a latest, Session or task document, a base makes every older record useless, and all three storage backends delete them. Retiring such a document deletes all its records. A rewindable document keeps everything, because old values must stay readable [spec §3.5](#).

That is what makes `snapshotAsOf()` work. On `app.todos`, the value as of entry 6 (written at seq 3) is base 2 plus delta 3: `{"items":["write docs","fix build"]}`. On `app.draft` the same call fails with `Document 5 does not retain historical content`.

Versions and migrations

To change a document’s shape, raise `version` and add `migrate()`. Every record stores its version, and each read compares it with the token’s [spec §3.6](#):

STORED VERSION	WHAT HAPPENS
same as the token's	the value is used as stored
older, token has migrate	<code>migrate(value, storedVersion)</code> returns the current shape
older, no migrate	throws <code>requires migration from version 1</code>
newer than the token's	throws <code>has newer version 2 than 1</code>

From `src/documents.ts` and spec §3.6.

The capture changed `app.todos` from a list of strings to a list of objects:

```

a version-2 token test/capture-p4-documents.ts TS

const TodosNext = defineDoc<TodosV2>({
  kind: "app.todos", version: 2,
  scope: "conversation", history: "rewindable", fork: "asOf",
  initial: () => ({ items: [] }),
  migrate: (value, fromVersion) => {
    if (fromVersion !== 1) throw new Error(`no migration from ${fromVersion}`);
    return { items: (value.items as string[]).map((text) => ({ text, done: false })) };
  },
});

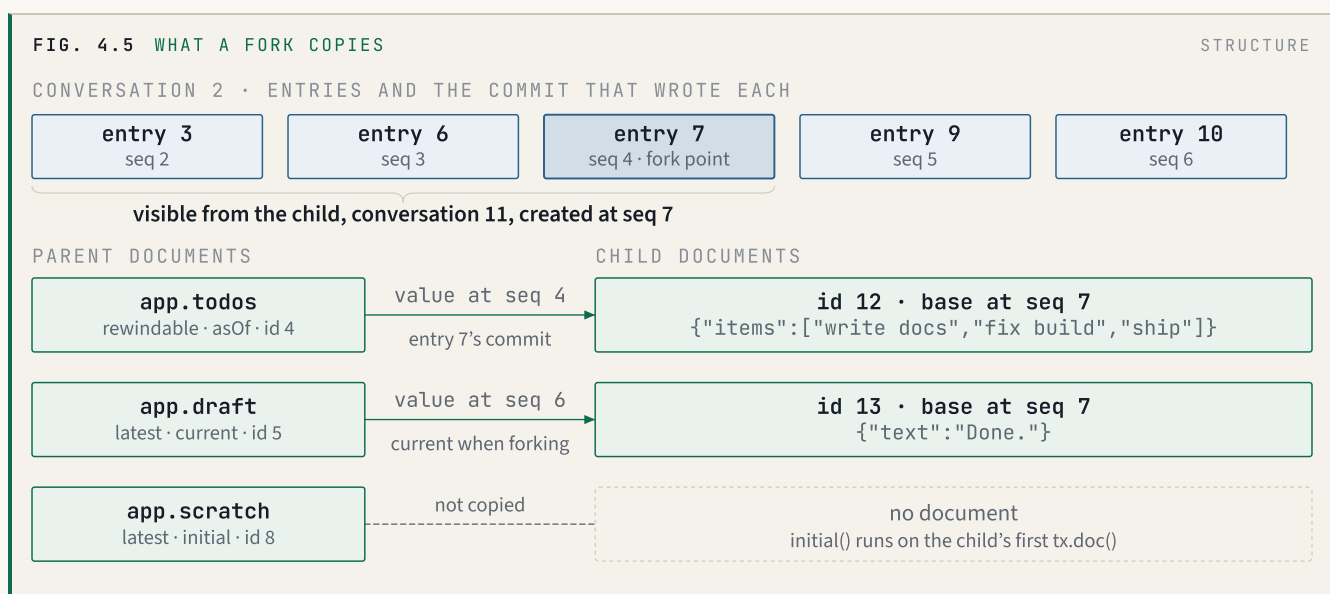
```

Kind, scope, history and fork are unchanged; only the shape and the version move. The capture's `checkpointWhen` line is omitted.

Migration happens when a document is used, never in bulk. A `snapshot()` through the new token returned the migrated value and wrote nothing. The first write through it stored a version-2 base (seq 8). From then on, the old version-1 token fails with `has newer version 2 than 1`, so code rolled back past that write can no longer open the document. The old version-1 records stay, and `snapshotAsOf()` through the new token migrates them on the fly.

Forks copy documents

Forking conversation 2 at entry 7 copied each of its documents by that document's own policy:



A fork copies each conversation document by its own policy, into a new incarnation with one base. IDs, sequences and values from `research/capture/extra-p4/documents.txt`; the policies are those of spec §3.7.

Each copy is a new, independent document with a new ID and one base `spec §3.7`. Storage makes the copy itself; a remote backend does it server-side. The copy keeps the stored version; the child migrates on its own first use. One restriction follows from copying committed values: a commit that forks may not also change one of the parent's `current` documents. It fails with `Cannot fork conversation 2 while changing its current-policy documents`. Commit the parent's change first `src/session/transaction.ts`.

WHAT THIS MEANS FOR YOU

- Give every long-lived document a `checkpointWhen`. A delta count, like the spec's `info.deltasSinceBase >= 31`, is the simplest rule.
- Change a document's shape with a new `version` and `migrate()`. Never rename its kind: the new name finds nothing and creates an empty document beside the old one. A rename needs an explicit copy and retirement `spec §12`.
- Choose `rewindable` only when you need old values. It never frees space.

Sources: `src/types.ts` (`DocumentContent`, `CheckpointInfo`); `src/documents.ts`; `src/session/transaction.ts`; `src/session/session.ts` (`snapshotAsOf`); `src/session/forks.ts`; `src/storage/sqlite/storage.ts`; `spec §3.2`, `§3.5-3.7`, `§12`; `docs/pico-v5-handoff.md §10`; `test/session-checkpoints-migrations.test.ts`; `research/capture/extra-p4/documents.txt`, `document-publications.txt` (`test/capture-p4-documents.ts`)

The built-in documents

Five conversation documents hold what the Harness needs between commits: what the agent is, who the provider sees, what is running, what is waiting, and what it cost.

The Harness stores its own state the same way you store yours: in documents. A UI reads them to show a spinner, a queue or a bill. After this section you can read the docs block of any conversation view and say what writes each part.

The five documents are defined with the same `defineDoc()` as yours, and their tokens are exported: `AgentDoc`, `ProviderDoc`, `LiveDoc`, `InboxDoc` and `UsageDoc` `src/index.ts`. All five belong to one conversation, are created with it, and appear in its view under `docs[kind]` (the conversation view (p. 129)). Your own documents are not mounted there; watch them with `watchDoc()`.

KIND	HOLDS	HISTORY · FORK	BASE WHEN	SEE
pi.agent	model, tools, instructions	rewindable · asOf	every change	7.2 (p. 106)
pi.provider	the provider session ID	latest · initial	every change	6.3 (p. 91)
pi.live	what is running now	latest · initial	nothing runs	6.2 (p. 88)
pi.inbox	queued inputs	latest · initial	the inbox is empty	6.1 (p. 84)
pi.usage	tokens and cost	latest · initial	every change	6.6 (p. 100)

From the definitions in `src/harness/agent.ts`, `provider.ts`, `live.ts`, `inbox.ts` and `usage.ts`. All five are version 1.

After one tool-using turn, the root conversation’s view held these values:

the docs block after one turn research/capture/view-final.json JSON

```
"docs": {
  "pi.agent": { "model": { "provider": "faux", "modelId": "faux-1" } },
  "pi.live": {},
  "pi.inbox": { "items": [] },
  "pi.provider": { "sessionId": "01a10c8d-c25d-72c3-83b8-8dc6cf418d86" },
  "pi.usage": { "models": { "faux/faux-1": { "input": 83, "output": 31, ... } }, "tools": {} }
}
```

The usage entry is cut after two counters. An idle conversation has an empty `pi.live` and an empty inbox.

At birth and in a fork

Every new conversation gets all five in its creating commit. Four of them always start the same way. Only `pi.agent` depends on how the conversation was made `spec §2.2; src/harness/agent.ts (createAgent)`:

DOCUMENT	NEW, OWNERLESS	NEW, OWNED BY A TASK	FORK AT ENTRY E
pi.agent	{}	a copy of the owner conversation’s agent	the parent’s agent as of E
pi.provider	a new UUIDv7, never copied		
pi.live	{}: nothing running		
pi.inbox	{ items: [] }: nothing queued		
pi.usage	{ models: {}, tools: {} }: zero spend		

Copies are one-time: “later owner changes do not reach it” `spec §12`.

Each document

pi.agent — what was chosen for this conversation: model, thinking level, extensions, tools, instructions and working directory. It stores names, never code `src/harness/types.ts` (`AgentState`). `configure()` changes it, and a tool result can add or remove tools. How names become a running agent is in [the per-conversation agent](#) (p. 106).

pi.provider — `{ sessionId }`, the ID the provider sees, used for prompt-cache affinity. Reset, compaction, model changes and reopen keep it `spec §2.2`. A conversation stored before 1.0.2 gets one on its first request `src/harness/provider.ts`; `CHANGELOG.md`.

pi.live — run control and progress. `run` is present exactly while the conversation is busy; generation, tools and compactions show what is in flight `src/harness/live.ts`.

pi.inbox — `{ items }`, the inputs and writes waiting for the next boundary, in order `src/harness/inbox.ts`.

pi.usage — this conversation’s spend, keyed by `provider/modelId` and by tool name. Failed and aborted attempts count too `spec §8.6`. `Harness.usage()` adds up every conversation.

`pi.live` shows how a good checkpoint rule keeps storage small. It writes a base whenever nothing is running, so its edits never pile up past one tool round:

pi.live records while a tool runs			research/capture/jsonl-dir-midrun/doc-2.jsonl	TEXT
seq 6	base	{ "run": { "taskId": 9, "inputs": [8], "tools": [{ "callId": "call-1", "name": "count", "taskId": 12, "status": "pending" }] }		
seq 8	delta	[["s", ["tools", 0, "status"], "running"]]		
seq 9	delta	[["s", ["tools", 0, "output"], "1\n"]]		
seq 10	delta	[["a", ["tools", 0, "output"], "2\n3\n"]]		

Other fields of each line are omitted. At seq 6 the tool was still `pending`, so nothing was running and the rule chose a base. After the run, the file held a single line: the base `{}` at seq 17 `research/capture/jsonl-dir/doc-2.jsonl`.

WHAT THIS MEANS FOR YOU

- Build UI state from the view’s docs block: `pi.live` for spinners, `pi.inbox` for queues, `pi.usage` for cost.
- Change `pi.agent` only through `configure()`. Do not write the other four; the Harness owns them.

Sources: `src/harness/agent.ts`; `src/harness/provider.ts`; `src/harness/live.ts`; `src/harness/inbox.ts`; `src/harness/usage.ts`; `src/harness/harness.ts`; `src/harness/types.ts` (`AgentState`); `src/index.ts`; `spec §2.2`, `§6`, `§8.2`, `§8.6`, `§12`; `CHANGELOG.md 1.0.2`; `research/capture/view-final.json`, `jsonl-dir-midrun/doc-2.jsonl`, `jsonl-dir/doc-2.jsonl`

Tasks

Durable state machines that checkpoint every phase, own their children, and abort from the bottom up.

5.1	Tasks as state machines	63
5.2	Phases, steps and progress	67
5.3	The effect sandwich	70
5.4	The scheduler	73
5.5	Ownership and structured concurrency	76
5.6	Abort, fault and orphan	80

05

5.1

Tasks as state machines

A task is work cut into phases, and its stored record is the only thing about it that survives a crash.

Say your agent is halfway through charging a card when the laptop lid closes. When the process comes back, something has to say how far it got. In Pi Durable that something is a task. After this section you can write a task, name its five states, and read a task record in storage.

A task is a list of phases

A *task* is a unit of work that the Harness commits as it goes. You split it into *phases*, named steps such as *prepare* and *charge*. When a phase finishes its part, it commits a *checkpoint*: a small JSON value that names the next phase and holds whatever that phase needs. A *commit* is an atomic save: everything in it is stored together, or nothing is. If the process dies, the next process reads the last checkpoint and runs that phase again. Nothing else carries over. Variables and closures are gone; only the stored record remains.

All work in Pi Durable runs this way, including the Harness's own model calls, tool calls and compactions. You write a task with `defineTask()` and register it in an extension's `tasks` list. The Harness finds the code by the task's name each time it starts the task (extensions and the registry (p. 103)).

a two-phase payment task test/examples/12-tasks.ts

TS

```
type PaymentState = { phase: "prepare" } | { phase: "charge"; key: string };
const Payment = defineTask<{ amount: number }, PaymentState, { receipt: number }>({
  name: "example.payment",
  version: 1,
  initial: () => ({ phase: "prepare" }),
  phases: {
    prepare: async (task, runtime, taskContext) => {
      const checkpoint = { phase: "charge", key: `payment-${task.id}` } as const;
      await runtime.commit(() => ({ status: "running", checkpoint }), taskContext);
    },
    charge: async (task, runtime, taskContext) => {
      const key = task.state.checkpoint.key;
      if (!payments.has(key)) payments.set(key, task.input.amount * 100);
      const receipt = payments.get(key)!;
      const outcome = { status: "completed", result: { receipt } } as const;
      await runtime.commit(() => ({ status: "terminal", outcome }), taskContext);
    },
  },
  abort: async (_task, runtime, taskContext) => {
    const outcome = { status: "aborted" } as const;
    await runtime.commit(() => ({ status: "terminal", outcome }), taskContext);
  },
});
registry.install(defineExtension({ name: "payments", tasks: [Payment] }));
```

Two inline state literals are hoisted into `checkpoint` and `outcome` constants to fit the page. Run 12 prints `{ status: 'completed', result: { receipt: 500 } }`.

Read it top to bottom. `initial` gives the first checkpoint. `prepare` picks an idempotency key and commits it in the checkpoint `{ phase: "charge", key }`. `charge` reads the key back from `task.state.checkpoint`, not from a variable, because after a

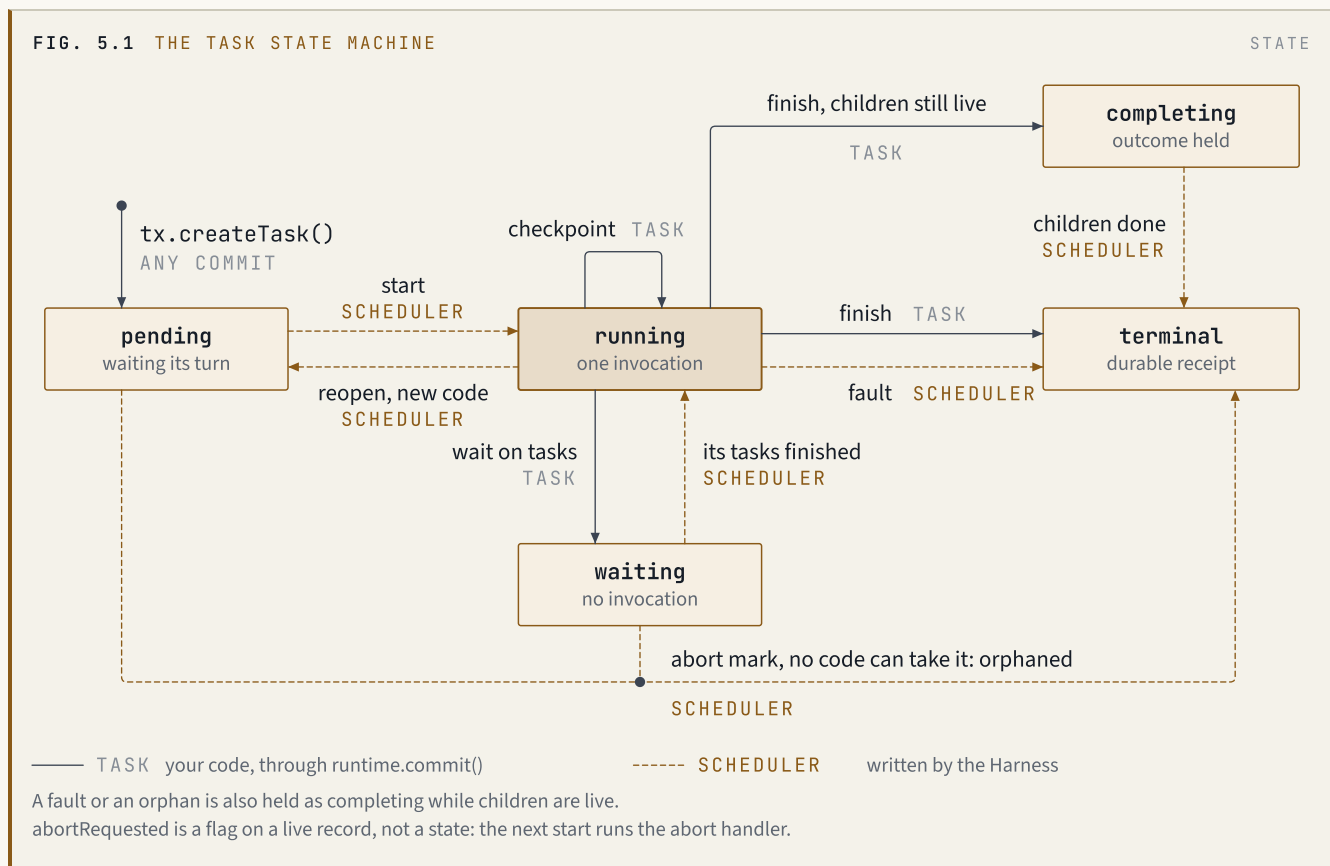
restart the variable would be gone. It then commits the final outcome. `abort` runs only if someone aborts the task.

FIELD	WHAT IT IS
<code>name</code>	The task's kind, stored in every record. Renaming it strands stored tasks of the old name.
<code>version</code>	Stored with every record. A newer definition must migrate older records (the scheduler (p. 73)).
<code>initial(input)</code>	The first checkpoint, committed when the task is created.
<code>phases</code>	One handler per phase name. Each handler sees the task narrowed to its own phase.
<code>abort</code>	Runs after an abort and must commit a final outcome (abort, fault and orphan (p. 80)).
<code>migrate?</code>	Upgrades the input and checkpoint stored by an older version.
<code>hooks?</code>	Hooks other extensions may attach to this task (hooks (p. 109)).

`TaskDefinition` in `src/types.ts`. Its type parameters are the input `I`, the checkpoint `S`, the result `R` and the hooks `H`.

Five states

A task is always in one of five states `src/types.ts`. Three are live. `pending` means it is waiting its turn to run. `running` means code is executing it now. `waiting` means it is paused until other tasks finish. The other two hold a result. `completing` means the result is decided but work the task started is still running. `terminal` means it is finished for good, and the record is now a receipt.



A task has five states; your code commits checkpoints, waits and outcomes, and the Harness does the rest. Edges from `src/harness/scheduler.ts`. An abort-marked waiting task also leaves `waiting` early to run its abort handler; an abort-marked `pending` or `waiting` task that no code can take ends orphaned.

Your code moves a task only through `runtime.commit()`. A phase can commit a new checkpoint and stay `running`, start a wait, or finish with `terminal`. It can never set `pending`. Only the Harness does that, when it reopens after a crash or hands a task to newly loaded code. The Harness also writes faults, orphans and abort marks: “The scheduler owns reservation, reconciliation, handover, faults, and orphaning; `abortTask()` owns abort marks” [spec §5.1](#). If a phase finishes while tasks it started are still running, the Harness stores it as `completing` instead ([ownership \(p. 76\)](#)).

The record

A task’s whole state lives in one JSON record, and every commit replaces the record whole. Here is an order from the captured shop run. It has charged a card, started three packing tasks, and is waiting for them.

```
order task 7, waiting (commit seq 7) research/capture/task-transitions.txt JSON

{
  "id": 7,
  "kind": "shop.order",
  "abortRequested": false,
  "state": {
    "status": "waiting",
    "checkpoint": { "phase": "decide", "chargeId": "ch_a_1", "packs": [8, 9, 10], ... },
    "on": [8, 9, 10],
    "policy": "failFast"
  },
  "memos": { "chargeId": "ch_a_1" }
}
```

Elided: `conversationId`, `version`, `input`, `background`, and the checkpoint’s key. Task 7 has no `owner` field because its conversation owns it.

state — the status, plus a checkpoint while the task is live or an outcome once it has one.

on, policy — the tasks it waits for, and what happens if one fails ([ownership \(p. 76\)](#)).

memos — small stored values where the first write wins ([phases and progress \(p. 67\)](#)).

abortRequested — the durable abort flag.

owner — set only on a child task.

`conversationId` and `owner` never change. A commit that tries to move a task to another conversation rejects with `Task N cannot change conversations` [src/session/transaction.ts](#).

When the order finishes, its record shrinks to a receipt. The checkpoint and memos are dropped, and the state keeps only the outcome: `{"status": "terminal", "outcome": {"status": "completed", "result": {"chargeId": "ch_a_1", "boxes": [...]}}` [capture/task-transitions.txt](#). Terminal records are kept forever, so keep results small. Put large data in an entry or a document, and return only its ID [spec §12](#).

Outcomes

STATUS	FIELDS	WRITTEN BY	MEANING
completed	result	the task	success; the typed R
failed	error, result?	the task	an expected failure, such as a declined card
aborted	reason?, result?	its abort handler	aborted, and the task cleaned up
orphaned	reason	the Harness	aborted while no code could take it
faulted	error	the Harness	a bug: a throw, or a phase that made no progress

`TaskOutcome<R>` in `src/types.ts`. Every error is plain JSON, `{ message, detail? }`, never a runtime Error.

What a phase can call

Each phase handler receives `(task, runtime, context)`. `task` is the stored record. `runtime` is the toolbox for this one run of the task, and it stops working when the run ends `src/types.ts`.

PURPOSE	MEMBERS
Commit	<code>commit(change, context)</code> , <code>memo(name, candidate, context)</code>
Read committed state	<code>memo(name)</code> , <code>getTask</code> , <code>outcomes</code> , <code>entry</code> , <code>context</code> , <code>snapshot</code> , <code>snapshotAsOf</code> , <code>watchDoc</code>
Wait	<code>waitForTask</code> , <code>sleep(until)</code>
Environment	<code>agent()</code> , <code>hooks</code> , <code>registry</code> , <code>settings</code> , <code>models</code> , <code>env()</code> , <code>conversation()</code>
Identity and clock	<code>taskId</code> , <code>conversationId</code> , <code>signal</code> , <code>now()</code> , <code>report()</code>

`TaskRuntime` in `src/types.ts`.

`runtime.commit()` is the one way to commit. Your callback receives a transaction and the current record. It may append entries, write documents and create child tasks, and it may return the task's next state. All of it lands in one atomic commit, so a checkpoint can never disagree with the data it describes. Entries committed this way record the task as their author (by `taskId`) [spec §5.1](#). The commit rejects if the run has ended, the Harness is closing, the task is no longer `running`, or the task carries an abort mark.

WHAT THIS MEANS FOR YOU

- Put everything the next phase needs in the checkpoint. Variables do not survive a restart.
- Treat `name` as permanent. When the checkpoint's shape changes, raise `version` and add `migrate`.
- Commit a checkpoint and the data it describes in the same `runtime.commit()`.
- Keep outcomes small; store bulk data in a document or an entry.

Sources: [spec §1](#), [§5.1](#), [§5.3](#), [§12](#); `src/tasks.ts`; `src/types.ts` (`TaskDefinition`, `TaskState`, `TaskRecord`, `TaskOutcome`, `TaskRuntime`); `src/harness/scheduler.ts`; `src/session/transaction.ts`; [ex 12](#), [run 12](#); [research/capture/task-transitions.txt](#).

5.2

Phases, steps and progress

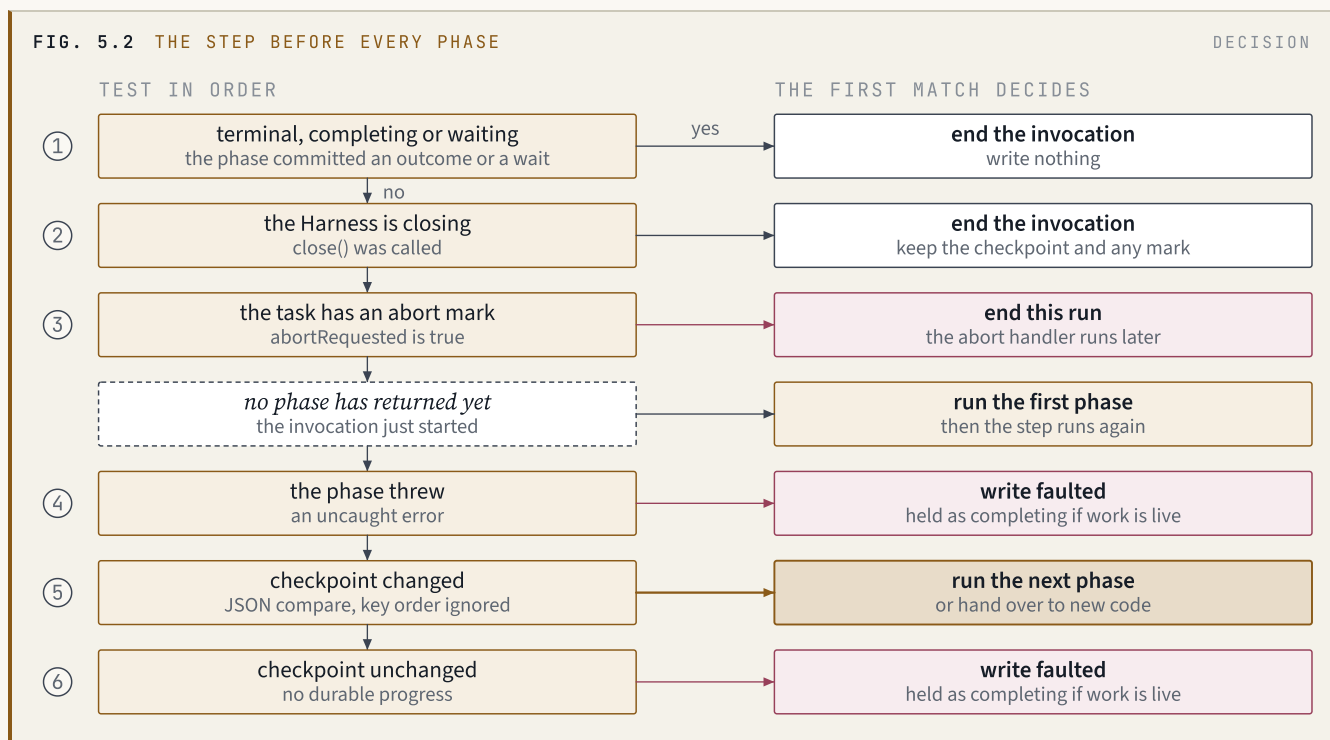
A phase that commits nothing new would run forever, so the Harness stops the task with a fault instead.

A task runs one phase after another. Between phases the Harness makes a quick check: is the task finished, aborted or shutting down, and did the last phase commit progress? After this section you can predict that check for any phase, design phases that always make progress, and use memos and `sleep` safely.

One run, many phases

When the Harness starts a task, it creates an *invocation*: one in-memory run of the task, with its own abort signal. An invocation runs phase after phase until the task finishes, waits or is stopped. Committing a checkpoint does not end it. The next phase starts right away, in the same process.

Before each phase the Harness runs a *step*, a short check that reads the committed record and decides whether to go on. The step is itself a commit on the Session's single queue of writes (the commit (p. 29)), so it sees every commit the phase made before returning. A commit that reaches the queue after the step is rejected with `invocation has ended test/harness-tasks.test.ts`.



Before each phase the step applies the first matching rule; no progress is a fault. Rules from spec §5.1, implemented in `src/harness/scheduler.ts`.

The six rules

Before the first phase only rules 1–3 apply, because nothing has returned yet. After a phase returns, the first rule that matches wins:

1. **Finished, completing or waiting: stop.** The phase committed an outcome or a wait. There is nothing more to run.

2. **The Harness is closing: stop.** The checkpoint and any abort mark stay for the next process. Closing never writes an outcome.
3. **Abort mark: stop this run.** A separate abort run starts later, once the task's own child work has finished ([abort \(p. 80\)](#)).
4. **The phase threw: write `faulTted`** with the error's message. Like every outcome, it is held as `completing` while the task's child work is still live (ownership [\(p. 76\)](#)).
5. **Checkpoint changed: continue** with the next phase, or hand the task to newly loaded code ([the scheduler \(p. 73\)](#)).
6. **Checkpoint unchanged: write `faulTted`**, "because no durable progress was made", held as `completing` in the same way while child work is live.

The order matters in two cases. A phase that commits a new checkpoint and then throws is still faulted, because rule 4 comes before rule 5. A phase that commits a final outcome and then throws keeps its outcome, because rule 1 comes first. Any commit it attempts after that rejects with `Task N is terminal test/harness-tasks.test.ts`.

Why no progress is a fault

A phase that ends with the checkpoint it started with would run again with the same input, forever. So the step compares the two checkpoints as JSON, ignoring key order. If they are equal, it faults the task with `Task test.idle phase run returned without durable progress test/harness-tasks.test.ts`. Three patterns trip this rule:

- The phase returns without committing anything.
- The phase commits entries or documents but returns no new state from `runtime.commit()`. Only the checkpoint counts as progress.
- The phase commits a copy that equals the old checkpoint.

A phase may run itself again, as long as the checkpoint changes. Example 13's ticker has a single phase, `tick`, and commits `{ phase: "tick", n: n + 1 }` on each pass. The phase name stays the same, the checkpoint does not, so rule 5 continues.

Memos: remember a small decision

A *memo* is a small named value stored on the task record. The first write wins. `runtime.memo(name, candidate)` commits the candidate only if the name is still empty, and returns the stored winner either way. `runtime.memo(name)` only reads [spec §5.2](#). Two writes of "a" and "b" at the same time both get "a" back `test/harness-tasks.test.ts`. Memos survive restarts and are dropped when the task finishes. Committing a memo does not count as progress.

Use a memo to keep one effect inside a phase from repeating. Example 13 prints each tick once, even when its phase runs twice:

a memo guards an effect inside one phase `test/examples/13-recovery.ts`

TS

```
tick: async (task, runtime, taskContext) => {
  const n = task.state.checkpoint.n;
  if ((await runtime.memo(`printed-${n}`, taskContext)) === undefined) {
    await runtime.memo(`printed-${n}`, true, taskContext);
    console.log(`tick ${n}`);
  }
  // ...commit { phase: "tick", n: n + 1 }, or terminal once n reaches task.input.to
  await runtime.sleep(Date.now() + 50, taskContext);
},
```

The checkpoint commit and a test-only hook are elided.

```
$ node --conditions=source --experimental-strip-types test/examples/13-recovery.ts
tick 1
tick 2
closed; saved checkpoint: { status: 'pending', checkpoint: { phase: 'tick', n: 2 } }
  memos: { 'printed-1': true, 'printed-2': true }
tick 3
tick 4
tick 5
after reopen: { status: 'completed', result: 'counted to 5' }
```

Run 13, with the memos wrapped onto a second line. The first process closed after printing tick 2 but before committing `n: 3`. The second process reran tick 2, found the memo `printed-2`, and skipped the print.

Sleeping

`runtime.sleep(until)` waits until the Harness clock reaches `until`. A sleep is not durable. Closing the Harness cancels it, and the next process restarts the phase from its checkpoint. If a deadline must survive a restart, put it in the checkpoint, as example 24's payment does with its `at` field.

Details

- The clock is the Harness's `now` option, `Date.now` by default. Long sleeps are split into timers of at most 2,147,483,647 ms, and the clock is checked again after each one. A sleep rejects when the run is stopped or the caller's context is cancelled `src/harness/scheduler.ts`.
- Hooks share the task's memo names, so prefix yours. Names such as `toString` never resolve to inherited properties `spec §5.2`; `test/harness-tasks.test.ts`.

WHAT THIS MEANS FOR YOU

- End every phase by changing the checkpoint, starting a wait, or finishing.
- Entries and documents are not progress. Return a new state from `runtime.commit()`.
- Use memos for small decisions, such as an ID or a chosen branch. Bulk progress belongs in a document `spec §12`.
- Keep deadlines in the checkpoint, not in a sleep.

Sources: `spec §5.1`, `§5.2`, `§12`; `src/harness/scheduler.ts`; `test/harness-tasks.test.ts` (task phases, task runtime); ex 13, run 13; ex 24.

5.3

The effect sandwich

Commit what you are about to do, do it, then commit what happened; a crash in the middle leaves a clear question, not a silent gap.

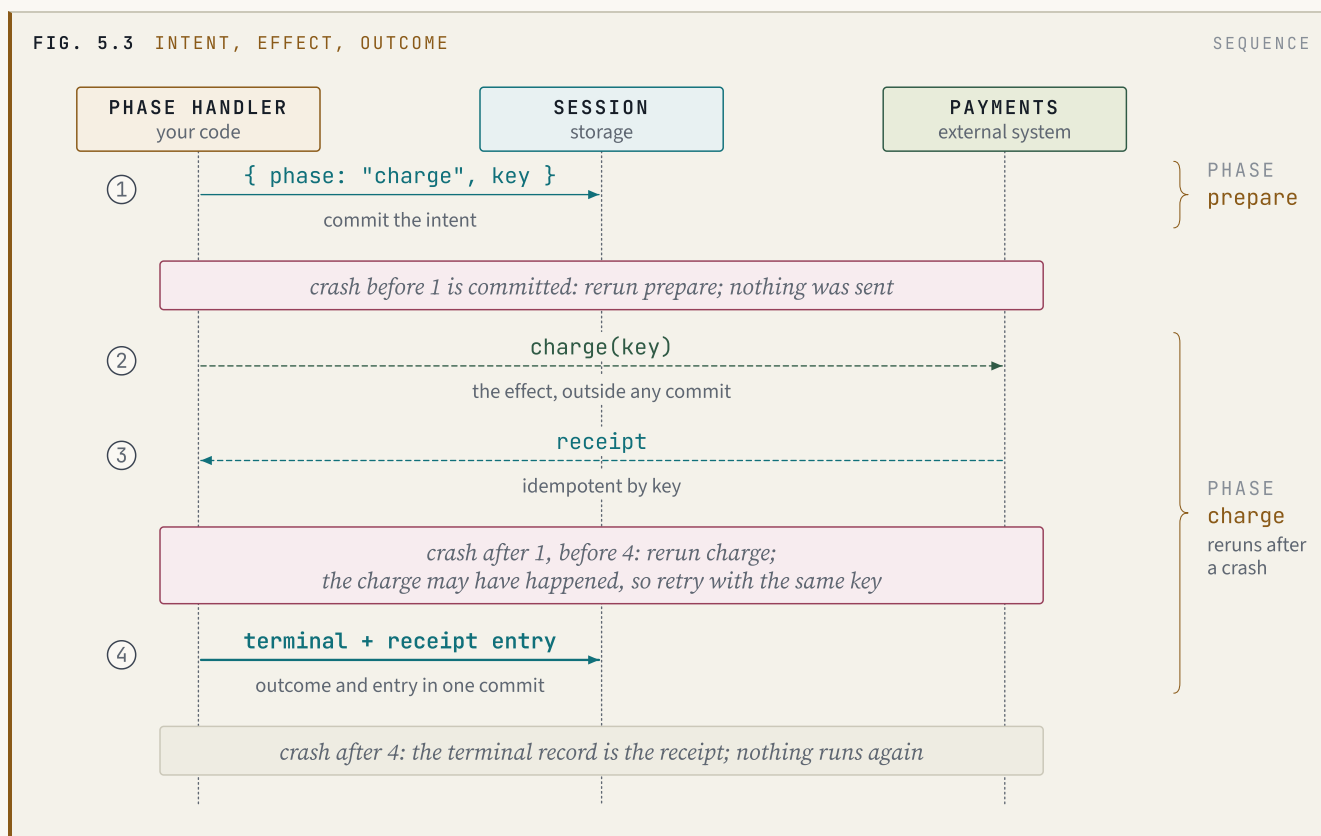
The Harness can commit its own records atomically, but it cannot make a card charge, a deploy or an email atomic with them. If the process dies right after the charge, did the customer pay? After this section you can split any outside action into phases so that every crash point has a defined, safe recovery.

Why effects need their own phase

An *effect* is anything that changes the world outside storage: a payment, an API call, a file on another machine. Effects may not run inside a commit [spec §1, invariant 4](#). A commit holds the Session's write queue, so waiting on the network there would stall every other writer ([the commit \(p. 29\)](#)). An effect therefore always happens between two commits, and the process can die between them. The spec's answer is a fixed shape:

```
commit intent phase
perform external effect
commit outcome or next phase
```

spec §5.2, verbatim.



Commit the intent, perform the effect, commit the outcome; a crash in the middle reruns the effect phase. The phases of spec §5.1's payment example. Step 4's entry and final state land in one commit.

The spec’s own example has three parts. `prepare` creates an idempotency key and commits it as the next checkpoint. `charge` performs the effect with that key, then appends a receipt entry and finishes the task in the same commit. The abort handler undoes the charge and commits `aborted`.

intent, effect, outcome docs/spec.md §5.1

TS

```
// Intent, effect, outcome.
prepare: async (task, runtime, context) => {
  await runtime.commit(() => ({ status: "running",
    checkpoint: { phase: "charge", key: newKey() } })), context);
},
charge: async (task, runtime, context) => {
  const receipt = await payments.charge(task.state.checkpoint.key); // idempotent by key
  await runtime.commit(async (tx, current) => {
    const entry = await tx.appendEntry(current.conversationId, receiptEntry(receipt));
    return { status: "terminal",
      outcome: { status: "completed", result: { entryId: entry.id } } };
  }, context);
},
// The abort handler decides the outcome; returning without one faults the task.
abort: async (task, runtime, context) => {
  await payments.cancel(task.state.checkpoint);
  await runtime.commit(() => ({ status: "terminal",
    outcome: { status: "aborted", reason: "user" } })), context);
},
```

Two long lines are wrapped; nothing else is changed.

Every crash point, and what reopening finds

When a new process opens the storage, every task that was `running` goes back to `pending` with its checkpoint (the scheduler (p. 73)). The checkpoint, not the process, says where the task stood:

CRASH	REOPEN FINDS	WHAT RUNS
before the intent is committed	checkpoint prepare	prepare again; no charge was sent
after the intent, before the charge	checkpoint charge	the charge, for the first time
after the charge, before the outcome	checkpoint charge	the charge again, with the same key
after the outcome is committed	terminal	nothing

The middle two rows look the same to the new process. In the spec’s words: “Reopening in an intent phase means the effect may have happened.”

The Harness cannot know whether the request reached the payment service before the process died, so your phase must be correct either way. The spec names three ways: “The phase handler retries safely, polls an external handle, or records interruption” spec §5.2.

Retry safely — send an idempotency key created in the intent phase and stored in the checkpoint. In the recovery test the transfer service is called twice, across a close and reopen, yet applies the transfer once, because both calls carry `transfer-N` test/harness-tasks-recovery.test.ts.

Poll a handle — when the remote system returns an operation ID, commit it before waiting, then ask the remote system for the result. The generation task does this for deferred model calls (generation (p. 91)).

Record interruption — when neither is possible, commit an outcome that says the effect may or may not have happened. Tool calls not marked `replay`: "safe" do this (tool calls and `replay` (p. 94)).

The built-in tasks use the same shape

The Harness's own tasks follow the sandwich, which is why their recovery is predictable. The `pi.tool` task commits the intent { `phase: "execute", arguments, replay` } just before running the tool. `replay` comes from the tool and defaults to "unsafe" `src/harness/tool.ts`. In a normal run the same phase goes on to run the tool and commit the result, so the `execute` phase "is reached only by recovery". There it reruns the tool only if both the stored and the current policy say `safe`. Otherwise it commits the call as failed with `Tool NAME was interrupted and may have partially run`.

The `pi.generation` task polls. When a provider answers with a deferred handle, the task commits a `poll` checkpoint with the `handle` and a `pollAt` time. A crash anywhere in that loop reopens at a checkpoint that still names the remote operation, and the abort handler can cancel it with the stored handle `src/harness/generation.ts`.

NO ROLLBACK

Pi Durable never undoes an outside effect. A failed commit rolls back only the Harness's own records (when a commit fails (p. 33)). Undoing a charge is up to the remote system: its idempotency keys, its cancel endpoint, its operation handles (limits and non-goals (p. 168)).

WHAT THIS MEANS FOR YOU

- **Create the key in the intent phase.** A key created inside the effect phase changes on every rerun and defeats deduplication. Example 12 derives it from the task ID; the shop capture commits "key": "charge-a" at seq 4, before any charge.
- **Commit the outcome and its consequences together.** A terminal commit atomically writes the outcome, appends result entries, retires the task's documents and settles any submissions it answers `spec §5.3`. Two commits would leave a window where the receipt exists but the task is still live.
- **Guard a second effect in the same phase with a memo.** The shop order's `charge` phase calls the service only if memo `chargeId` is empty (seq 5, then seq 6). A crash between the call and the memo still repeats the call, so the call itself must tolerate that.

Sources: `spec §1` (invariant 4), `§5.1` (payment example), `§5.2`, `§5.3`, `§12`; `test/harness-tasks-recovery.test.ts` (resumes an intent/effect/outcome task interrupted after its intent); `ex 12`; `research/capture/task-transitions.txt` (scenario A, seq 4-6); `src/harness/tool.ts`; `src/harness/generation.ts`.

5.4

The scheduler

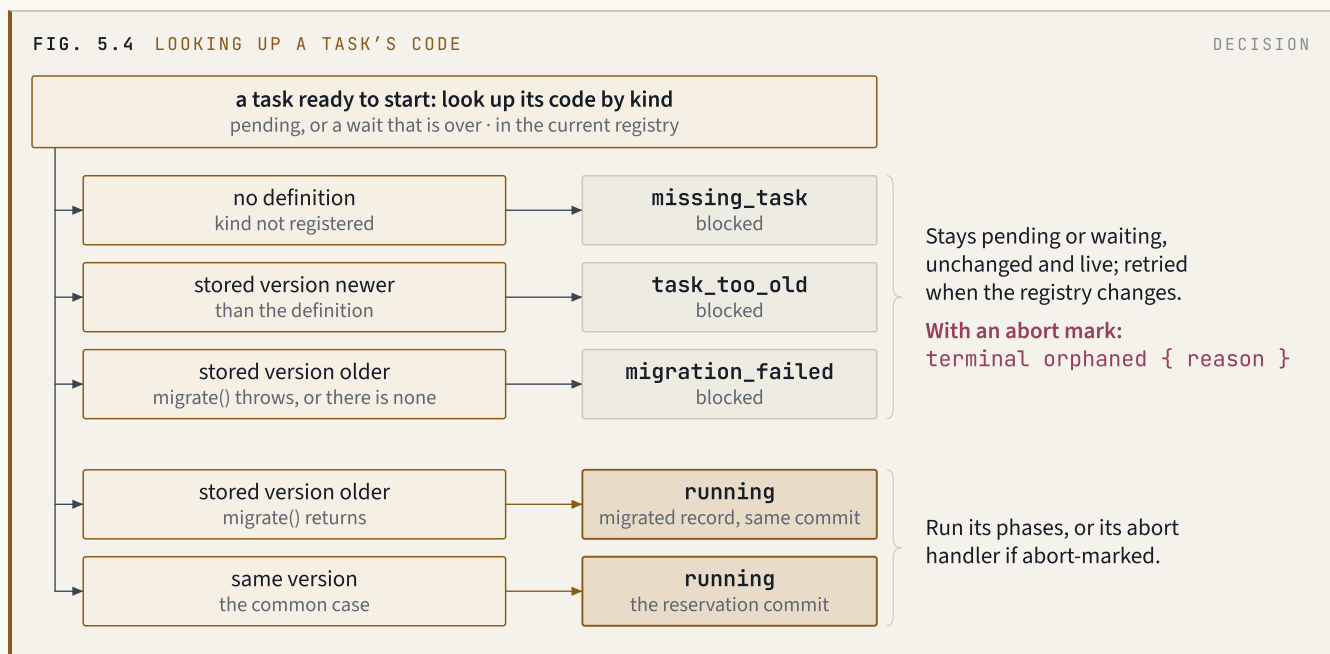
One part of the Harness decides which task runs next, and it never ends a task just because its code is missing.

Restarts, upgrades and missing extensions all raise the same question: which tasks may run now, and with which code? The scheduler answers it. After this section you can say what a restarted process does with each unfinished task, why a task can stay `pending` forever, and what happens when you reload an extension under a running task.

Starting tasks

The *scheduler* is the part of the Harness that starts tasks. Whenever something changes, such as a task being committed, an extension being installed, or a call to `resume()`, it looks for tasks that can start. A task can start when nothing is running it, it is not `completing`, and it is not waiting on a live task. For an abort-marked task, “waiting” means waiting for its own child work, because cleanup runs from the bottom up `src/harness/scheduler.ts`.

Starting a task is called *reservation*. In one commit, the scheduler looks up each startable task’s code by its kind, sets the task to `running`, and creates its invocation. A `waiting` task goes straight to `running`. If the record carries an abort mark, the invocation runs the abort handler instead of the phases `spec §5.4`.



The scheduler looks up a task’s code by kind; a task no code can take stays pending, and only an abort ends it, as orphaned. The table of `spec §5.4`, implemented in `src/harness/scheduler.ts`. A migrated record is committed together with `running`.

Blocked tasks

A task is *blocked* when no registered code fits it. A blocked task has not failed. Its record stays unchanged, it still counts as unfinished work for idle waits, and the scheduler tries again whenever the registry changes. Blocked is “derived runtime state, not a persisted task status” `spec §5.4`.

REASON	WHEN	WHAT UNBLOCKS IT
missing_task	no code with that name is registered	registering it
task_too_old	the stored version is newer than the code	registering code of that version or newer
migration_failed	the code is newer, and migrate is missing or throws	registering a different definition

From spec §5.4 and src/harness/scheduler.ts.

In the recovery test, a task with a missing definition stays `pending` until `addTask()` registers it, and then completes `test/harness-tasks-recovery.test.ts`. Use `harness.inspect()` to find blocked tasks. It reports `{ kind: "blocked", reason }` for each one (the task graph and `inspect()` (p. 137)).

Migration runs when the task is reserved. One callback, `migrate(input, checkpoint, fromVersion)`, upgrades records from every older version. If it throws, the error is reported once through `onReport`, and the scheduler tries again only after a different definition is registered for that kind. A newer definition without `migrate` blocks with `Task test.versioned version 2 has no migration from 1` `test/harness-tasks-recovery.test.ts`.

NEVER ENDED FOR MISSING CODE

“The Harness never terminalizes a task merely because registry code is missing or incompatible, neither at open nor later” spec §5.4. Only an abort ends a blocked task, as orphaned (`abort`, `fault` and `orphan` (p. 80)).

After a crash: reopening

When you open a Harness on existing storage, the scheduler makes one cleanup commit. Every task left `running` goes back to `pending` with the same checkpoint and memos. Nothing runs yet: the open “dispatches nothing” `src/harness/scheduler.ts`. Handlers start after `resume()`, or after a call that needs progress such as `waitForTask()` or `submit()` spec §2.2. Open also schedules a second commit that applies any abort marks and final outcomes a crash left half done.

STATUS AT CRASH	OPEN WRITES	THEN
pending	nothing	starts after resume
running	pending; checkpoint, memos and mark kept	reruns from the checkpoint
waiting	nothing	resumes when its on is finished
completing	nothing	finishes once its work below is done
terminal	nothing	never runs again

From `src/harness/scheduler.ts` and spec §5.1, §5.5.

a `pi.tool` task after `SIGKILL`, then after open `research/capture/crash-reopen-tasks.txt`

JSON

```
// the SQLite row left by the killed process
{ "id": 24, "kind": "pi.tool", "owner": 18, ...
  "state": { "status": "running",
    "checkpoint": { "phase": "execute", "arguments": { "target": "weekly" },
      "replay": "safe" } } }

// after Harness.open(), before resume(): only the status changed
"state": { "status": "pending", "checkpoint": { ... unchanged ... } }

// its owner, generation 18, stays "waiting" with "on": [24]
```

The process was killed mid-tool. Elided (...): `conversationId`, `version`, `input`, `background` and `abortRequested`, all unchanged by open.

Reloading code under a running task

A running invocation keeps the code it started with. At each phase boundary the step checks the registry again. If a different definition for the kind is now registered and can take the task (the same version, or a newer one with `migrate`), the task *hands over*. The step sets it back to `pending` with its checkpoint, memos and abort mark, and the next reservation runs it under the new code [spec §5.4](#). The handover test logs `old:a start, old:a end, new:b start`, so old and new code never overlap [test/harness-tasks-recovery.test.ts](#). If the new code cannot take the task, the old code keeps running and the problem is reported once. “A handler that never settles never hands over.” [Reload while running \(p. 118\)](#) shows this in practice.

Closing

`close()` stops new work, rejects pending `waitForTask()` and idle waits with a closed error, signals every invocation, and waits for them to return. It writes no outcome and sets no mark, so all work resumes on reopen [spec §5.1](#). A handler that ignores its signal keeps `close()` waiting [spec §12](#).

Details

- The scheduler keeps an in-memory copy of every live record, updated as each commit lands. Each pass reserves every startable task in one commit.
- If storage rejects the reservation commit, the reservations are undone in memory, the error goes to `onReport`, and the next pass tries again. The test task still runs exactly once [test/harness-tasks.test.ts](#). If the process dies during that commit, the reopened task is still `pending` and nothing ran [test/harness-tasks-recovery.test.ts](#).

WHAT THIS MEANS FOR YOU

- After a crash, every running task restarts at its last checkpoint. Write each phase so it can run again.
- A task that sits `pending` forever is usually blocked. Check `harness.inspect()`.
- When a checkpoint’s shape changes, raise `version` and ship a `migrate` that accepts every older version.
- Let handlers honour their signal, so `close()` and handover are quick.

Sources: [spec §2.2](#), [§5.1](#), [§5.4](#), [§12](#); [src/harness/scheduler.ts](#); [test/harness-tasks.test.ts](#) (task scheduling); [test/harness-tasks-recovery.test.ts](#) (blocked tasks, definition handover, task crash recovery); [research/capture/crash-reopen-tasks.txt](#).

5.5

Ownership and structured concurrency

Work a task starts belongs to it: the task cannot finish before that work does, and aborting the task aborts the work.

A checkout starts four card payments. It should not report done while a payment is still in flight, and if the customer cancels, the payments should be aborted too. Ownership gives you both. After this section you can create child tasks, wait for them with the right policy, and predict when a task holds as `completing`.

One ownership tree

Every task has an *owner*, set when it is created and never changed. You pass it as `ownership` in `tx.createTask(task, input, { ownership })` spec §5.5:

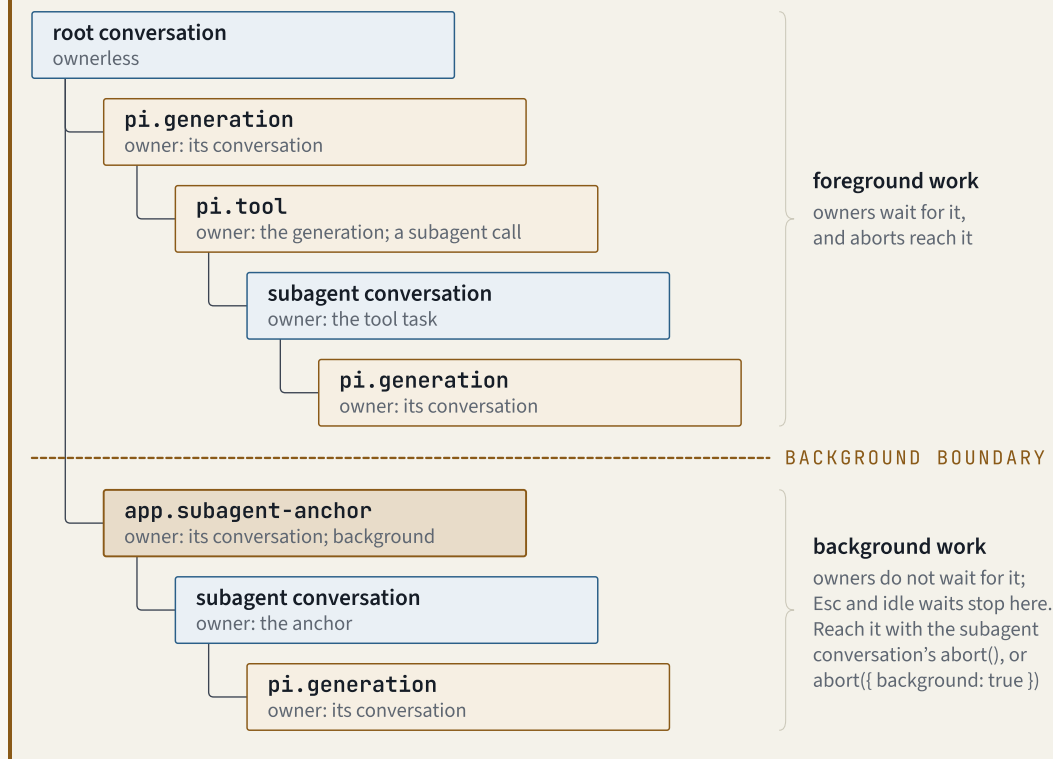
`{ kind: "conversation" }` — a top-level task, owned by its conversation.

`{ kind: "task", taskId }` — a *child task*, owned by another task and always in that task's conversation.

Conversations can have owners too. A subagent's conversation is owned by the tool task that created it (owned conversations (p. 43)). Together, tasks and conversations form one tree. Forks play no part in it.

FIG. 5.5 THE OWNERSHIP TREE

TREE



Tasks and conversations form one ownership tree, and a background task cuts off the subtree below it. The foreground subagent of `README $Abort` and Subagents and ex 22, beside the background anchor of ex 23. A generation owns its tool tasks (`README $Concepts`).

The owner must be live when the child is created: not `completing`, not `terminal`, not `abort-marked`. The check uses the state the commit will leave behind `src/session/transaction.ts`. So a task cannot create a child in the same commit that finishes it. The shop capture’s scenario D tries exactly that. The commit rejects with `Task owner 12 is completing`, the phase throws, and the task faults at seq 27. Create children in one commit, and finish in a later one.

Foreground and background work

A task’s *owned work* is everything below it in the tree: its children, the tasks in conversations it owns, and so on down. The spec calls this the *ordinary* owned work, and it stops at background tasks. A *background* task is a conversation-owned task created with `background: true`. Ordinary waits and aborts skip it “and its complete owned subtree” [spec §5.4](#). Idle waits use the same rule: `Conversation.waitForIdle()` returns once no live foreground task remains below the conversation.

Waiting on tasks

A phase waits by returning `{ status: "waiting", checkpoint, on, policy }`. The task then has no invocation. When every task in `on` has finished, it resumes at `checkpoint` and reads their outcomes, in order, with `runtime.outcomes(ids)`. Example 24’s checkout creates four payments and waits on them in the same commit:

```
create children and wait, in one commit test/examples/24-child-tasks.ts TS

pay: async (task, runtime, taskContext) => {
  await runtime.commit(async (tx) => {
    const payments: TaskId<PaymentResult>[] = [];
    for (const card of task.input.cards) {
      const ownership = { kind: "task", taskId: task.id } as const;
      payments.push(await tx.createTask(Payment, { card }, { ownership }));
    }
    return { status: "waiting", checkpoint: { phase: "decide", payments },
      on: payments, policy: "failFast" };
  }, taskContext);
},
```

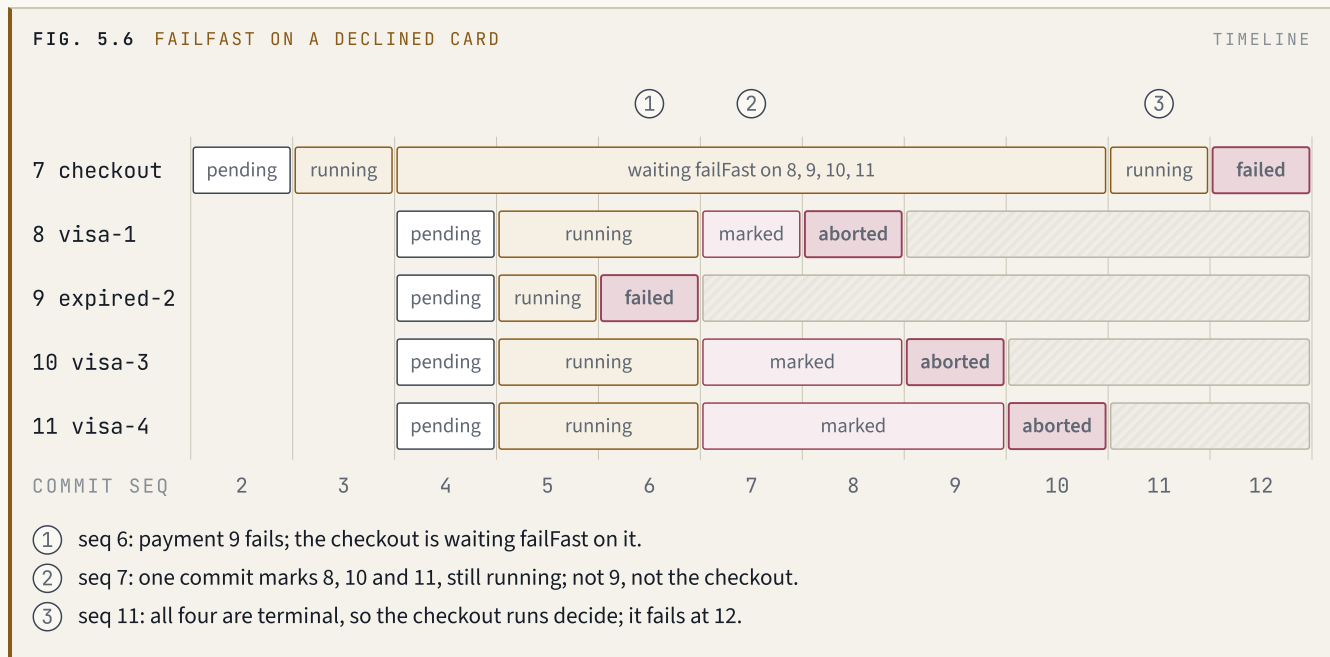
The ownership literal is hoisted to fit the page. The decide phase reads `runtime.outcomes(payments)` and commits `completed` only if every payment completed.

The rules for `on` are checked when the wait is committed, and breaking one faults the task [src/harness/scheduler.ts](#):

- `on` may not name a missing task, the task itself, or one of its owners, “which could never finish first”.
- `failFast` may name only tasks the waiter owns. `allSettled` may name any task, even one already finished.
- An empty `on` resumes at the next scheduling pass.
- An abort handler cannot wait at all.

`allSettled` or `failFast`

The policy decides what happens to the other tasks when one fails. With `allSettled`, nothing happens: the waiter resumes when the last task ends, whatever the outcomes. With `failFast`, “the first task in `on` that holds or ends with a non-completed outcome gives every other live task in `on` an abort mark, in the scheduler’s next reconcile commit” [spec §5.5](#). The waiter itself is not aborted. It resumes and decides its own outcome.



One failed payment aborts its live siblings in the next commit; the checkout itself is never abort-marked. Example 24's first scenario, recorded per commit in `research/capture/extra-p5/checkout-transitions.txt`. Each bar is a committed state that lasts until the next commit rewrites it; a terminal record is never rewritten (hatched). marked is running with an abort mark.

Read the figure left to right. At seq 6, payment 9 (`expired-2`) fails. At seq 7, a single commit puts an abort mark on payments 8, 10 and 11, and their abort handlers refund the cards. By seq 10 all three are aborted. At seq 11 the checkout resumes in `decide`, reads `aborted`, `failed`, `aborted`, `aborted`, and commits `failed` at seq 12.

completing: the owner waits for its work

If a task finishes while its owned work is still running, it is committed as `{ status: "completing", outcome }`. The result is fixed, but the task is not done. Once nothing live remains below it, the scheduler commits the final `terminal` record `spec §5.5`. The shop order shows it end to end:

SEQ	TASK	STATUS	WHAT HAPPENED
13	7 order	running	checkpoint finish; creates receipt child 11 in the same commit
14	11 send-receipt	running	started
15	7 order	completing	finish commits completed; child 11 is still live
16	11 send-receipt	terminal	completed, "sent"
17	7 order	terminal	the scheduler's final commit; <code>waitForTask(7)</code> resolves

Scenario A, `research/capture/task-transitions.txt`.

A held outcome follows five rules `spec §5.5`:

- It is final. No phase or abort handler runs again, and the task is never restarted or migrated.
- Aborting a `completing` task only marks it, which aborts the work below. The final record keeps the held outcome.
- A held outcome other than `completed` counts as cancellation intent, so the work below is aborted first. Such a held failure also triggers `failFast` before its own work drains `test/harness-structured.test.ts`.

- Writes split at the hold. Entries and documents from the finishing commit land at once. The terminal state, the retirement of the task's documents, and its waiters wait for the final commit.
- Waiters, idle waits and `inspect()` all treat a `completing` task as live.

HOLDS THAT SURPRISE

Foreground work an extension starts in a subagent's conversation holds the calling tool, and with it the whole run. Asking a subagent more questions while its tool is completing extends the hold. Create work that should not hold its owner as conversation-owned and `background` [spec §12](#).

WHAT THIS MEANS FOR YOU

- Make a task the owner of any work its result depends on.
- Use `failFast` for all-or-nothing children, and `allSettled` when you want every result.
- Create children in one commit and finish in a later one.
- Use a background, conversation-owned task for work that must not hold its caller.

Sources: [spec §5.4](#), [§5.5](#), [§12](#); [README §Concepts](#), [§Abort and Subagents](#), [§Child Tasks](#); [src/types.ts](#) (`TaskOwnership`, `TaskOptions`, `JoinPolicy`); [src/session/transaction.ts](#); [src/harness/scheduler.ts](#); [test/harness-structured.test.ts](#); [ex 22](#), [23](#), [24](#); [run 24](#); [research/capture/task-transitions.txt](#); [research/capture/extra-p5/checkout-transitions.txt](#).

5.6

Abort, fault and orphan

An abort is a committed mark that flows down the tree; each task's own handler decides how to clean up, after everything below it has finished.

Aborting work that has touched the outside world takes more than stopping it: a charged card needs a refund. Pi Durable lets each task decide how to clean up, and runs the cleanups in a safe order. After this section you can write an abort handler that compensates correctly, read an abort in storage, and tell from a record whether any cleanup ran.

How an abort works

You abort a task with `harness.abortTask(id)`. It does not stop the task on the spot. It commits a flag on the record, `abortRequested`, called the *abort mark*. Five steps follow [spec §5.4](#):

1. Commit `abortRequested` on the task.
2. Signal the running invocation, if there is one, and wait for it to return.
3. Wait until the task's owned work has finished.
4. Start a new invocation that runs the abort handler.
5. The abort handler commits a final outcome.

`abortTask()` does steps 1 and 2 and returns "marked". It returns "terminal" for a task that has already finished, and rejects for an unknown ID. It does not wait for the cleanup; use `waitForTask()` to see the final outcome [spec §2.2](#). If you stop waiting by cancelling your own context, the mark is already committed and the abort goes ahead [test/harness-tasks.test.ts](#).

Once the mark is committed, the running phase can no longer commit anything. Every `runtime.commit()` and memo write rejects with `Task N has a durable abort mark`, and the next step ends the run ([rule 3 of the step \(p. 67\)](#)). The abort handler runs later, in a new invocation. It still sees the record's memos, and it runs once [test/harness-tasks.test.ts](#).

The abort handler must decide the outcome, usually `aborted`. Returning without one faults the task with `Abort handler of task N returned without a terminal outcome`, and a throw faults it with the thrown message. The handler cannot wait and cannot create child tasks, because its own task is marked [src/harness/scheduler.ts](#); [spec §5.5](#). To compensate, act inline, or create background conversation-owned tasks and wait for them with `runtime.waitForTask()` [spec §12](#).

The abort flows down, cleanup runs up

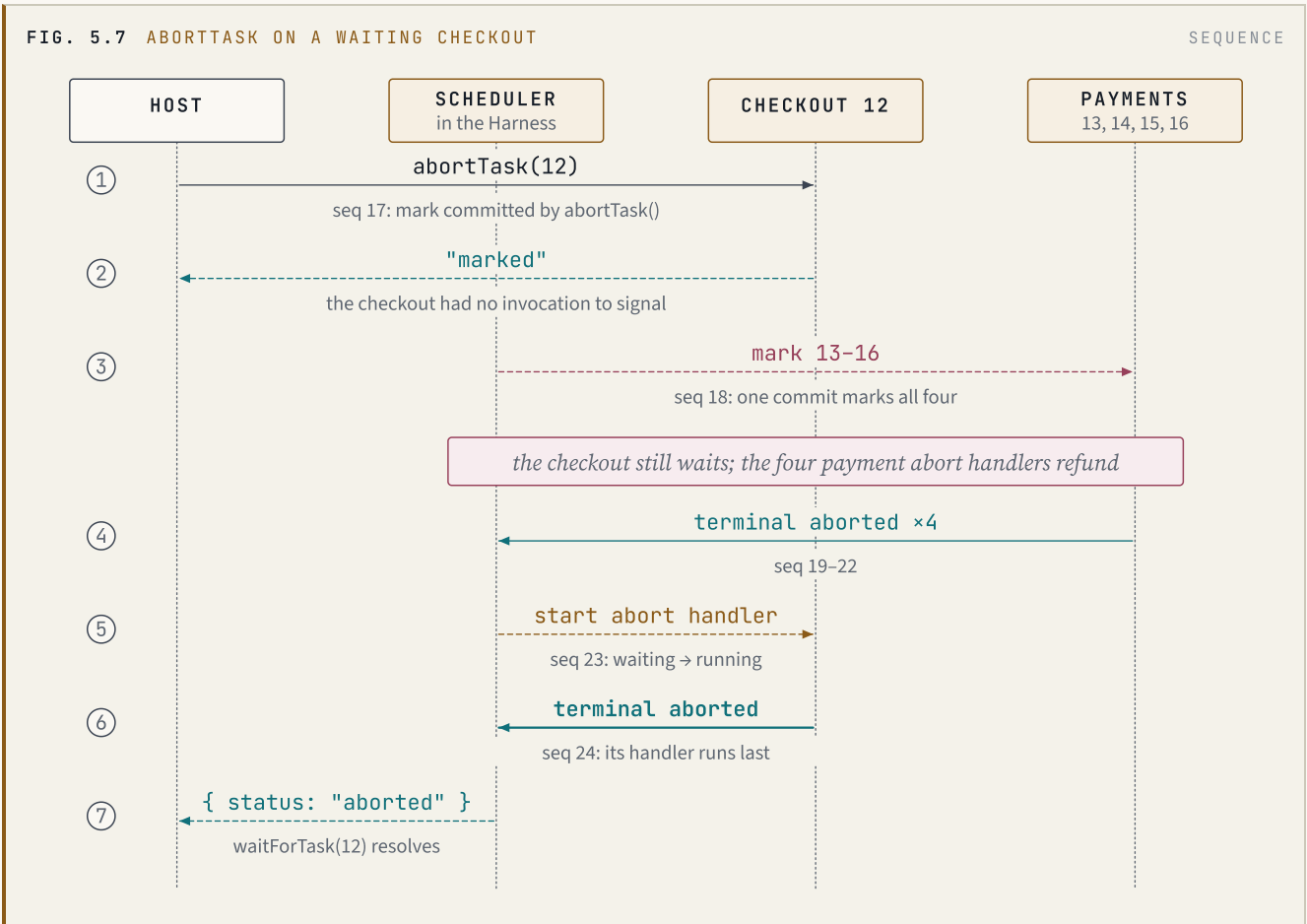
An abort spreads to the work below. A live task passes it on when it carries an abort mark, or when it holds a `completing` outcome other than `completed`. The spec calls either one *cancellation intent*, and it “cascades idempotently to foreground-owned work” [spec §5.4](#). The scheduler commits the children's marks in a separate commit, “and again at open, so a crash in between loses nothing”. Example 24's aborted checkout runs the whole protocol in eight commits, seq 17 to 24, shown in the figure below.

The order is guaranteed. A marked task's abort handler cannot start while its owned work is live, so “an abort handler sees final outcomes below it”. While it waits, `inspect()` shows the checkout as `{ kind: "waiting", on: [child] }`. In the three-level test, each handler logs that the level below is already `terminal` [test/harness-structured.test.ts](#).

The cascade has limits [spec §5.4](#):

- It stops at a background task that has no cancellation intent of its own.

- A finished owner never passes an abort on. New work in a finished subagent’s conversation, such as a user asking it a follow-up, “runs normally”.
- Tasks a waiter names in `on` but does not own are neither marked nor awaited.
- Each conversation the cascade reaches is treated like `Conversation.abort()`: its queued inputs become `unanswered` with reason `aborted`, and queued writes stay (submissions and the inbox (p. 84)). The aborted task’s own conversation keeps its queue.



An abort mark flows down in its own commit, and abort handlers run from the leaves up. Scenario 2 of `research/capture/extra-p5/checkout-transitions.txt`; seq numbers are the captured commit order. Run 24 prints the four refunds before `checkout aborted`.

Orphaned, faulted, aborted

OUTCOME	TASK CODE THAT RAN	EFFECTS IT STARTED
aborted	the task’s own abort handler	compensated, as far as the handler could
orphaned	none: no code could take the task	“may remain uncleaned”
faulted	a phase that threw or made no progress, or an abort handler that committed no outcome	unknown; nothing compensated

From spec §5.4. `failed` is not here: it is an outcome the task chose, like `completed`.

Orphaned. A blocked task cannot run its abort handler, because that code is missing or cannot take the task. So aborting it ends it as `orphaned`, with the blocked reason as `reason`. The recovery test aborts a task with a missing definition before `resume()` and reads `{ status: "orphaned", reason: "missing_task" }`, with the task’s documents already retired `test/harness-tasks-recovery.test.ts`. Only an abort orphans a task. Missing code alone never does.

Faulted. Faults come from task bugs, malformed provider data, or a commit that storage rejected cleanly (`StorageRejected`). An uncertain storage failure stops the Session instead and writes no outcome (when a commit fails (p. 33)). If storage rejects the fault itself, the task stays `running` and runs again at the next reservation `test/harness-tasks.test.ts`.

Crashes during an abort

The committed abort mark is all that matters, so every crash point recovers from it `test/harness-tasks-recovery.test.ts`:

CRASH	AFTER REOPEN
while the mark is being committed	no mark; the run resumes
after the mark, before the run returned	pending with the mark; only the abort handler runs
while the abort handler runs	a new abort invocation runs it again
while its outcome is being committed	the abort handler runs again
after the final outcome	nothing runs; <code>abortTask()</code> returns "terminal"

The task `crash` `recovery` suite. An abort handler can run more than once, so it must be safe to repeat, like any effect phase (the effect sandwich (p. 70)).

Details: cleanup for the built-in tasks

For the two outcomes the Harness writes itself, `faulted` and `orphaned`, it runs one cleanup hook in the commit that makes the outcome final `src/harness/live.ts`; `spec §5.4`. Only three kinds use it:

- pi.generation** — when it is the current run’s task, a committed partial reply becomes an aborted `pi.assistant` entry, the run’s inputs become `unanswered` with reason `faulted` or the blocked reason, and the run ends (run control (p. 88)).
- pi.tool** — marks the tool call done without a result entry. The run continues, and the missing result is filled in when the model context is built (context (p. 40)).
- pi.compaction** — clears the compaction status (compaction (p. 97)).

WHAT THIS MEANS FOR YOU

- Write abort handlers that are safe to run twice. A crash can make them run again.
- Always commit a final outcome from the abort handler; returning without one faults the task.
- Compensate inline, or through background tasks. An abort handler cannot create child tasks or wait on them.
- Treat `orphaned` and `faulted` as “nothing cleaned up”. Check the outside world yourself.

Sources: `spec §2.2`, `§5.3`, `§5.4`, `§5.5`, `§12`; `src/harness/scheduler.ts`; `src/harness/live.ts` (`settleSchedulerOutcome`); `test/harness-tasks.test.ts` (task abort); `test/harness-tasks-recovery.test.ts`; `test/harness-structured.test.ts` (abort order); `test/harness-ownership.test.ts`; `ex 24`, `run 24`; `research/capture/extra-p5/checkout-transitions.txt`.

Runs

*Submissions, the inbox, generation, tool calls and compaction:
the built-in tasks that turn an input into an answer.*

6.1	Submissions and the inbox	84
6.2	Run control and the live document	88
6.3	Generation	91
6.4	Tool calls and replay	94
6.5	Compaction	97
6.6	Usage and cost	100



6.1

Submissions and the inbox

Every message is committed the moment you send it; a busy conversation queues it, and a boundary decides when it lands.

A user types a second message while the agent is still answering the first. Should it interrupt, wait, or be refused? And if the process dies a second later, is the message lost? Pi Durable answers with one rule: a message is committed before anything else happens to it. A *commit* is an atomic save: everything in it is stored together, or nothing is. After this section you can predict what happens to a message sent at any moment, and choose how a busy conversation should treat it.

Two kinds of submission

A *submission* is the durable record of something you handed a conversation. There are two kinds. An *input* is a user message; it may start a *run*, the model calls and tool calls that answer it. A *write* adds an entry to the transcript without asking the model anything, such as a note from your app.

`Conversation.submit()` writes the record in one commit and returns a `Submission` handle. The handle has `status()`, `wait()` and `abort()`.

SubmissionDraft `src/harness/types.ts`

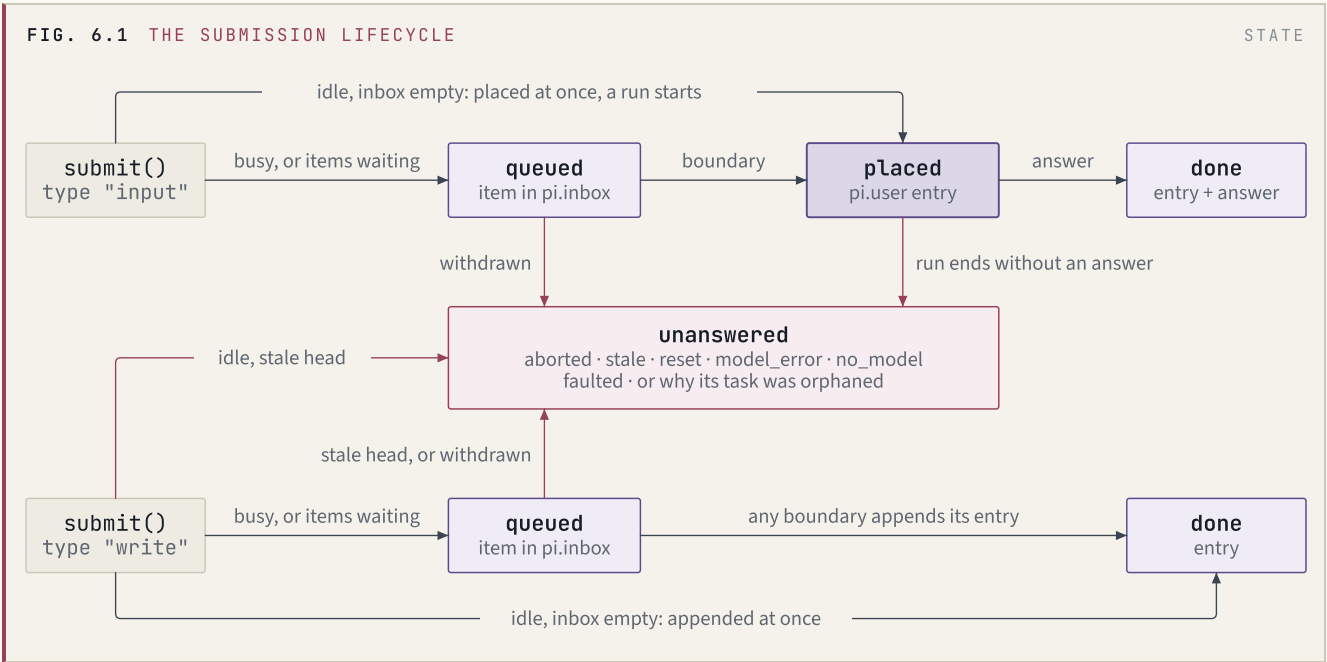
TS

```
type SubmissionDraft = { requestId?: string } & (
  | { type: "input"; content: UserInput;
      whenBusy?: "steer" | "followUp" | "reject" }
  | { type: "write"; entry: EntryDraft }
);
```

Simplified: the source marks every field `readonly` and adds `never` fields that keep the two branches apart.

Where a submission goes

A submission is in one of four states. It is `queued` while it waits. An input is `placed` once its message is in the transcript and a run is answering it. It is `done` when it has its entry and, for an input, the answer. It is `unanswered` when it ended without one; a `reason` says why. `wait()` resolves only at `done` or `unanswered`.



A submission is **queued**, **placed** or **settled**; only an **input** is ever **placed**. From `src/harness/submissions.ts`, `inbox.ts` and `live.ts`; reasons from spec §6 and §8.3. Rose arrows end without an answer.

Which path it takes depends on one question: is the conversation *busy*? A conversation is busy while a run is in progress (run control (p. 88)).

WHEN YOU SUBMIT	WHAT HAPPENS
idle, nothing queued	an input is placed and starts a run; a write is appended and is done
busy, or other items already queued	the submission is queued in the inbox
busy, with <code>whenBusy</code> : "reject"	nothing is written; <code>submit()</code> throws <code>ConversationBusy</code>
a <code>requestId</code> already used	nothing is written; you get the existing submission back

From `src/harness/submissions.ts` and the admission table of spec §6. A write that would undo a reset is the one exception; see Details below.

The `requestId` check makes retries safe. If a submit times out or your process restarts, send it again with the same ID and you get the same submission, not a second copy. The ID is scoped to one conversation.

The inbox

Queued submissions wait in `pi.inbox`, a small built-in document, one per conversation (the built-in documents (p. 60)); a fork starts with an empty one. Each item has a *mode* that says when it may land.

steer — “tell the agent as soon as its current tool calls finish”. The message joins the run in progress.

followUp — “handle this after the current answer”. The message starts the next run. This is the default `whenBusy`.

write — a queued write. It lands at the next chance, before any user message.

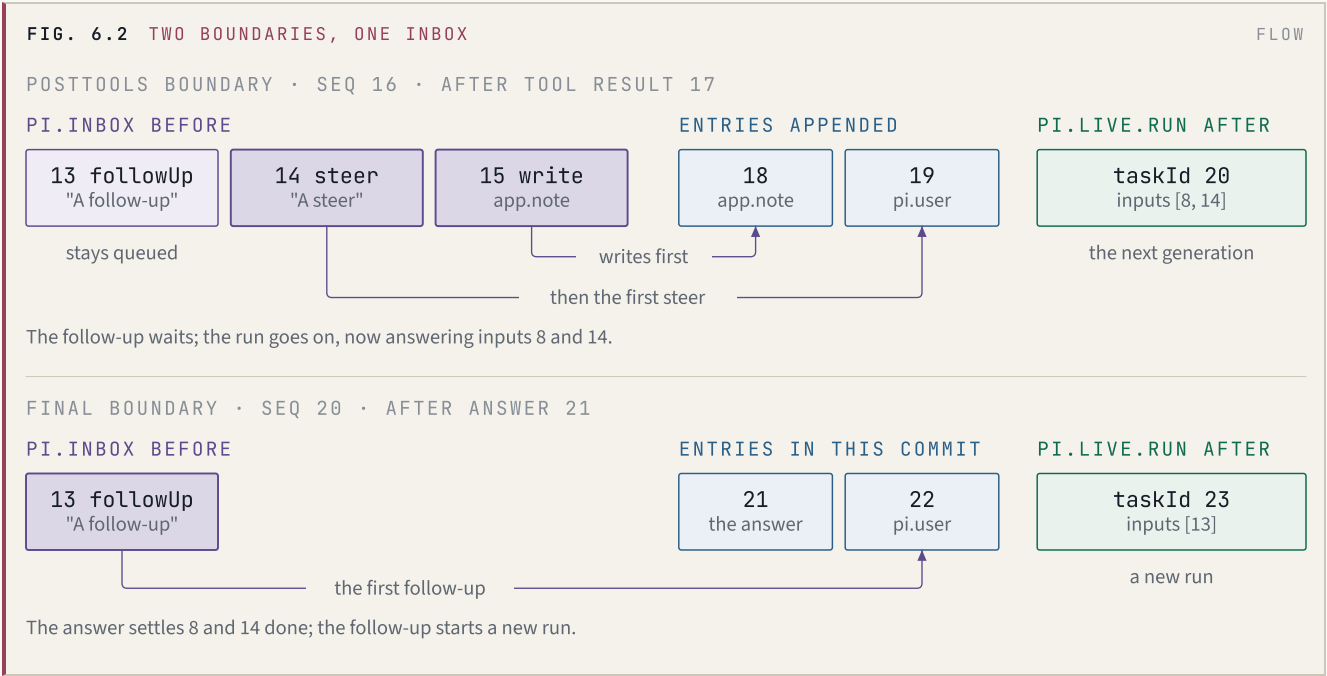
Boundaries: when queued items land

A *boundary* is the commit in which queued items enter the transcript. There are two. A `postTools` boundary runs after a round of tool calls when the run is going on. A `final` boundary runs when a run ends successfully.

BOUNDARY	WRITES	STEERS	FOLLOW-UPS	THE USER MESSAGES IT TAKES
postTools	all	the first	none	join the current run
final	all	the first	the first	start one new run

spec \$6; src/harness/inbox.ts. Set `steeringMode` or `followUpMode` to "all" to take every item of that mode instead of the first (both default to "one-at-a-time"). A queued reset turns a `postTools` boundary into a `final` one (see Details below). A submit to an idle conversation that still has items waiting queues behind them and runs a `final` boundary at once (src/harness/submissions.ts).

A boundary places writes first, then user messages, each in the order they were submitted. So a message queued before a reset or a compaction summary lands after it, and the model reads it in the new context.



A `postTools` boundary takes writes and steers; only a `final` boundary takes follow-ups. A tool call holds the first run while a follow-up, a steer and a note are queued; each connector joins a queued item to the entry it became. Answer 21 is appended in the same commit as the final boundary, not taken from the inbox. IDs, sequence numbers and `pi.live.run` values from `research/capture/inbox-steps.txt`.

Withdrawal, failure and crashes

`Submission.abort()` withdraws a submission that is still queued. It settles as `unanswered` with reason `aborted`, and the call returns "aborted". Once placed it is too late: the call returns "already_placed", or "settled" if it has already finished. `Conversation.abort()` withdraws every queued input but keeps queued writes.

Only a successful run applies the `final` boundary. A run that fails or is aborted settles its own inputs as `unanswered` and leaves the inbox alone.

AFTER A FAILED RUN

Queued follow-ups stay in the inbox, and their `wait()` does not resolve, until something else is submitted or you withdraw them. The next submission queues behind them, so the older follow-up runs first (`test/harness-inbox.test.ts`).

A crash changes none of this: every record and every inbox item is already committed. After reopening, a queued follow-up still runs once the interrupted run finishes. Only the in-memory waiters are lost. Get a new handle with `harness.submission(id)` and wait on that.

Details: writes that move the head

A write can set a `head`, which moves where the model's context starts (`reset`, `handoff` and `heads` (p. 46)). A head that points earlier than the current start would bring back history that a `reset` or `summary cut`. Such a write is *stale*: it settles `unanswered` with reason `stale`, whether it was submitted while idle or reached a boundary. A `reset` (`head: "self"`) is never stale. If a `postTools` boundary takes a `reset`, it acts as a `final` boundary, and the current run's inputs settle with reason `reset`, because their context was cut before an answer. Compaction relies on the same rule (`compaction` (p. 97)).

WHAT THIS MEANS FOR YOU

- Pass a `requestId` on any submit you might retry.
- Pick `whenBusy` per message: `steer` for a correction the agent should see mid-run, `followUp` for the next question, `reject` when your UI should refuse.
- After a failed run, resubmit or withdraw queued follow-ups; do not wait on them forever.
- After a restart, get handles back with `harness.submission(id)`.

Sources: `spec §6`; `src/harness/submissions.ts`; `src/harness/inbox.ts`; `src/harness/types.ts` (`SubmissionDraft`, `Submission`, `QueueMode`); `src/types.ts` (`SubmissionRecord`); `src/harness/agent.ts` (`resolveSettings`); `src/harness/generation.ts`; `README §Busy Conversations`; `test/harness-inbox.test.ts`; `test/harness-submissions.test.ts`; `ex 20`; `run 20`; `research/capture/inbox-steps.txt`

6.2

Run control and the live document

One small document says whether the conversation is busy and shows the answer and tool output as they form; a crash loses only progress since the last successful commit. The gap can exceed the configured interval.

While an agent works, two questions matter. Is the conversation busy? And what should the screen show right now: the half-written answer, the tool's scrolling output? Both answers live in one document, `pi.live`. It is committed like everything else, so a UI that connects late, or a process that restarts, sees the same picture. After this section you can read `pi.live` and say whether the conversation is busy, what the user sees, and what a crash loses.

The shape

`pi.live` is a built-in document, one per conversation. Every field is optional; an idle conversation's value is `{}`.

LiveData, simplified `src/harness/live.ts`

TS

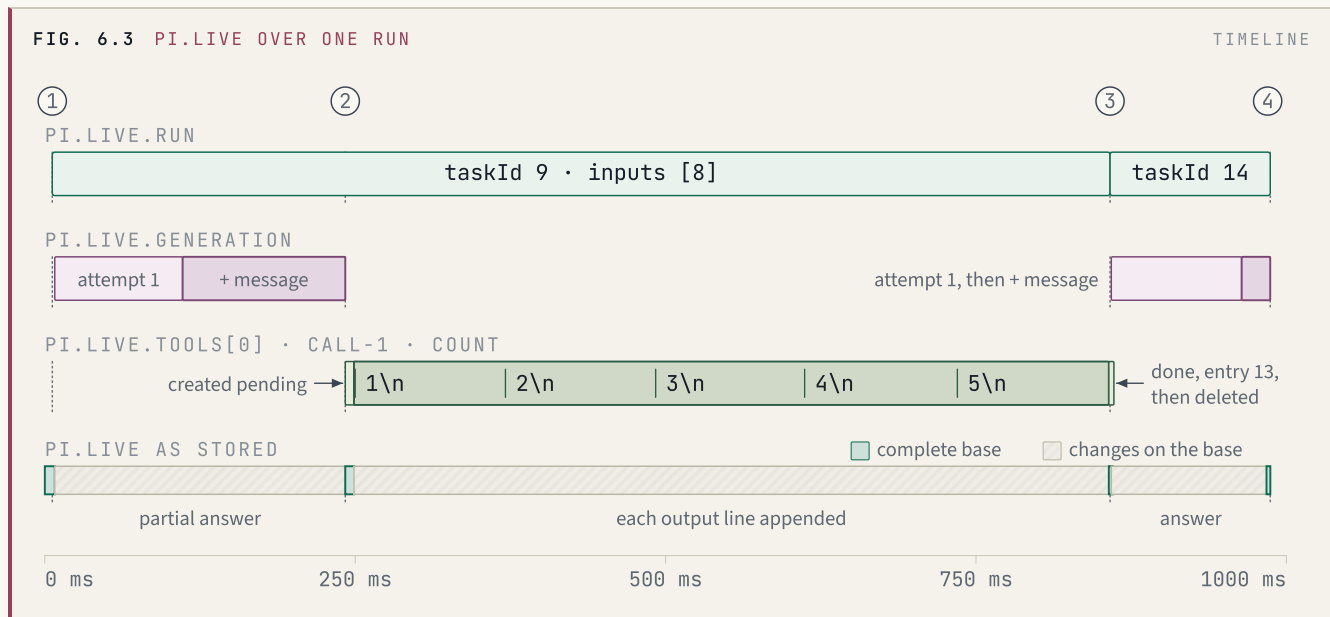
```
type LiveData = {
  run?: { taskId: TaskId; inputs: SubmissionId[] }; // present while busy
  generation?: { // the model call in progress
    attempt: number;
    message?: AssistantMessage; // the answer so far
    retry?: { at: number; error: string }; // waiting to retry
    deferred?: { pollAt: number }; // provider answers later
  };
  tools?: ToolSlot[]; // this round's tool calls
  compactions?: CompactionStatus[]; // see 6.5
};
```

The stored `message` is the JSON form of an `AssistantMessage`. Each `ToolSlot` shows one call: its `callId`, `name`, a status of `pending`, `running` or `done`, the output so far, and the result entry once done.

Run control: the busy flag

`run` is the lock. The conversation is busy exactly while `run` is present, and `submit()` checks nothing else ([submissions \(p. 84\)](#)). `inputs` lists the submissions this run will answer. `taskId` names the *generation*, the task making the current model call, that is responsible for answering them ([generation \(p. 91\)](#)).

A run is a chain of generations, one per turn. Between turns the lock is handed over, never released: the old generation creates the next one and rewrites `taskId` in the same commit. A steer that lands at that moment is appended to `inputs`. When the run ends, one commit settles every listed input and deletes `run`, `generation` and `tools`.



pi.live holds run control for the whole run and live progress only while something runs. One input, one `count` tool call that prints five lines, one answer; times from `research/capture/extra-p6/live-run.txt`. (1) submitting sets run; (2) the model asks for the tool, so `generation` gives way to `tools`; (3) the round hands `run` to generation 14; (4) the answer clears everything.

Live progress, and what it costs

`generation` and `tools` exist so a UI can show the answer and the tool output as they form. They are ordinary committed state, published like any other document (the conversation view (p. 129)). Every update is a commit, so updates are throttled. `settings.progress` has two intervals, both 100 ms by default:

partialIntervalMs — how often the streaming answer is committed. In the capture, the first partial lands 103 ms after the request starts.

outputIntervalMs — how often tool output is committed. The first line after a quiet spell is committed at once; later writes are spaced at least one interval apart, and further apart when the output is large.

A host with remote storage can raise both intervals and show progress in larger steps. The document stays small: whenever nothing is running, it is stored as a complete snapshot, so its change history never grows past one model call or one round of tools (`bases` and `deltas` (p. 56)).

What a crash leaves

A crash loses answer text or tool output that was not yet committed. The gap since the last successful commit can exceed the configured progress interval. In the crash capture the process was killed about 50 ms after its last output commit, so the `deploy` slot still holds all six lines it had printed.

conversation 1, pi.live after SIGKILL research/capture/crash-reopen-tasks.txt JSON

```
{
  "run": { "taskId": 15, "inputs": [14] },
  "tools": [{
    "callId": "deploy-1", "name": "deploy", "taskId": 22,
    "status": "running",
    "output": "deploy site: step 1/10 (pid 98319)\n ... step 6/10 (pid 98319)\n"
  }]
}
```

Lines 2–5 of the output are elided. The conversation is still busy; nothing answers input 14 until recovery decides what happened.

Recovery turns this progress into transcript, never into silence. A committed partial answer becomes an `aborted` assistant entry, so the transcript and UI retain what it had started to say. Later model requests exclude that aborted answer. A running tool’s committed output becomes the body of an `interrupted` result, or is cleared before a safe rerun (tool calls and replay (p. 94)).

Details: who sets and clears each field

FIELD	SET WHEN	CLEARED WHEN
run	a run starts	the run ends, on every path
generation	a model request starts, waits to retry, or waits for a deferred answer	the response is handled, or the run ends
tools	the model asks for tools	the round finishes, or the run ends
compactions	a compaction starts	it finishes

spec §8.2; src/harness/live.ts, generation.ts, tool.ts, compaction.ts. If the scheduler gives up on a run task (`faulted` or `orphaned`, `abort`, `fault` and `orphan` (p. 80)), the same commit settles its inputs `unanswered` and clears its fields.

WHAT THIS MEANS FOR YOU

- Drive your busy indicator from `run` and your streaming view from `generation` and `tools`.
- On slow or remote storage, raise `settings.progress` intervals.
- A crash loses uncommitted progress, with no fixed time bound, while the saved run survives.

Sources: spec §6, §8.2, §12; src/harness/live.ts; src/harness/generation.ts; src/harness/tool.ts; src/harness/output.ts; src/harness/agent.ts (DEFAULT_PROGRESS_POLICY); src/harness/types.ts (ProgressPolicy); CHANGELOG 1.0.3; test/harness-live-deltas.test.ts; test/harness-generation.test.ts; research/capture/extra-p6/live-run.txt; research/capture/inbox-steps.txt; research/capture/crash-before.txt; research/capture/crash-reopen-tasks.txt

6.3

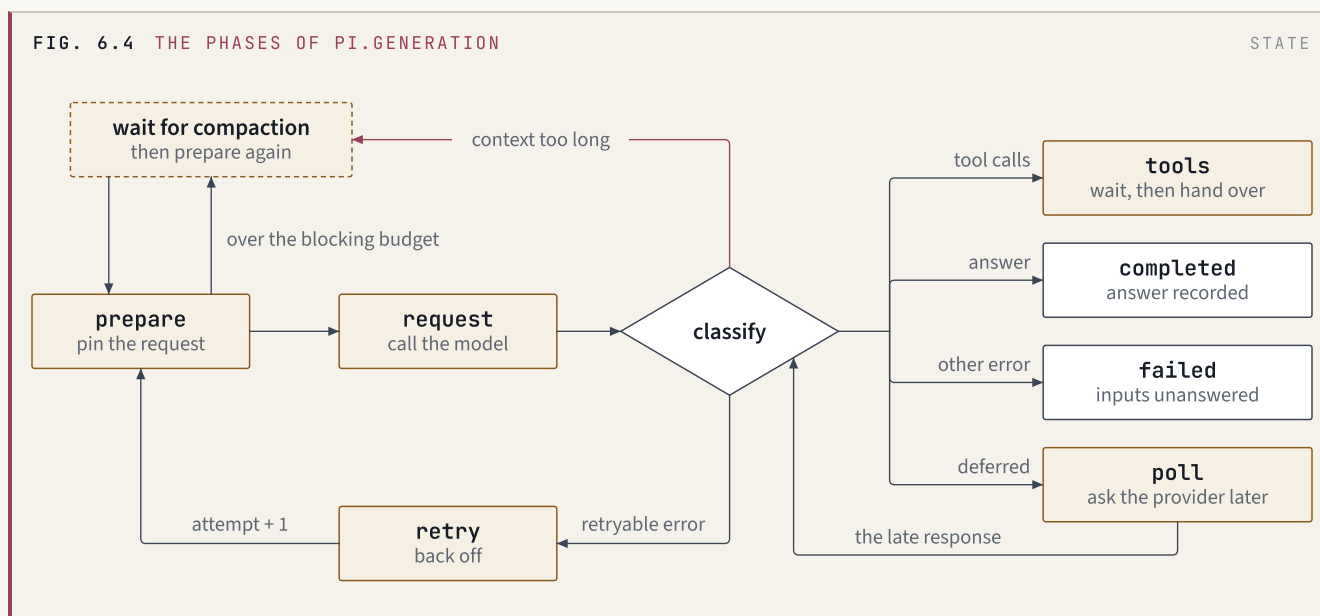
Generation

*Each model call saves its model, options and transcript cutoff before sending.
Recovery rebuilds that context and runs request hooks again.*

Calling the model is the slow, costly, failure-prone step of an agent. The provider may return a 503, the context may be too long, or the process may die mid-stream. Pi Durable runs each model call as a durable task called a *generation*. It commits the model, options and transcript cutoff. After a crash it rebuilds the request from those records, then reruns request hooks. After this section you can follow a model call through its phases, read its retry state, and say what a crash in each phase costs.

One generation, one turn

A generation is a `pi.generation` task (phases and steps (p. 67)). It makes one model call, decides what happens next, and owns the tool calls of its turn. A run is a chain of generations passing run control from one to the next (run control (p. 88)).



One generation prepares, calls the model, classifies the response, and ends or loops. From `src/harness/generation.ts`. Amber boxes are the task's phases; `request` and `poll` call the provider. White boxes are how the task ends. The dashed box is the task waiting on a compaction it owns, after which it prepares again with the same attempt. After the tool round the task completes and a new generation takes the run. Not drawn: an abort, which can end the task in any phase, and a missing model, which fails it with `no_model` in `prepare`, `request` or `poll`.

PHASE	WHAT IT DOES	ITS CHECKPOINT HOLDS
prepare	builds the system prompt and tool list; checks the compaction budgets	the attempt number
request	calls the model and streams the answer	the model, thinking level, stream options and <code>cutoff</code>
retry	waits out the backoff	the time to wake, <code>until</code>
poll	waits for a provider that answers later	the provider's handle and <code>pollAt</code>
tools	waits for the round's tool calls	the assistant entry and the tool task IDs

The `GenerationCheckpoint` union in `src/harness/generation.ts`. Every task starts at `{ phase: "prepare", attempt: 1 }`.

Prepare, then send

`prepare` reads the conversation's agent and settings (the per-conversation agent (p. 106)). Without a usable model the run fails with reason `no_model`. It renders the system prompt (the system prompt (p. 115)) and checks whether the context needs compacting first (compaction (p. 97)). Then one commit moves to `request` and pins the model, options and transcript cutoff. `cutoff` is the newest transcript entry the request includes.

Recovery rebuilds the committed context through `cutoff`, with the saved model and options. It then reruns `beforeRequest` hooks: their message replacements are not saved and can change on replay. A changed agent's model and options apply from the next `prepare`, which every retry passes through.

`request` builds the messages, lets `beforeRequest` hooks adjust them (hooks (p. 109)), and streams the response. The partial answer is committed on the throttle of the live document (p. 88). Each request carries a `sessionId`: one ID per conversation, kept in the built-in `pi.provider` document. Providers use it for prompt caching. It survives restarts, retries, resets and model changes; a fork gets a new one.

What the response decides

Every response is appended as a `pi.assistant` entry and its cost recorded (usage (p. 100)), failures included. Later requests leave failed and aborted messages out (transcript to context (p. 40)). The stop reason picks the next step.

THE MODEL RETURNED	NEXT
tool calls (<code>toolUse</code>)	run the tool round (tool calls (p. 94))
an answer (<code>stop</code> or <code>length</code>)	answer the inputs; run the <code>final</code> boundary
an error: context too long	compact once, then prepare again
an error the provider may recover from	wait, then retry
any other error, or <code>aborted</code>	inputs settle unanswered with <code>model_error</code> ; the run fails
deferred: "come back later"	poll after <code>pollAfterMs</code> , 5000 ms by default

`src/harness/generation.ts`. A `toolUse` response with no calls counts as an answer. The too-long rescue needs compaction enabled and a place to cut; without them, or on a second overflow, the run fails with `model_error`.

An answer can be extended. Before it settles the inputs, `onYield` hooks may return `{ continue }` with more user content. The run then goes on with that message, and the inputs stay open. This applies only when no queued user message or reset is landing at the same boundary.

Retries and late answers

The retry policy is `settings.retry`. By default it allows 3 retries after the first attempt, starting at 2000 ms and doubling, capped at 60 000 ms. pi-ai decides which errors are retryable. A too-long context is never retried this way; only a compaction can make it fit.

a 503, a 50 ms backoff, then the answer

research/capture/extra-p6/retry.txt

TEXT

```
seq 4 task 9 {"phase": "request", "attempt": 1, "cutoff": 7, ...}
seq 6 entry 10 pi.assistant, stopReason "error"
      task 9 {"phase": "retry", "attempt": 1, "until": 1791211854633}
      pi.live generation.retry = {"at": ..., "error": "503 service unavailable"}
seq 7 task 9 {"phase": "prepare", "attempt": 2}
seq 8 task 9 {"phase": "request", "attempt": 2, "cutoff": 10, ...}
seq 9 entry 11 pi.assistant; submission 8 done; task 9 completed
```

Run with `settings.retry.baseDelayMs = 50`; some commits are elided. The second request's `cutoff` includes the error entry, which the context then leaves out.

Both waits survive a restart. `retry` sleeps until the committed `until`, so a reopened process finishes the remaining backoff rather than starting over. `poll` serves providers that answer later: it sleeps until `pollAt`, asks again, and handles the result like a streamed one. Aborting during `poll` also asks the provider to cancel the request.

What a crash in each phase costs

CRASH DURING	WHAT RUNS AGAIN
prepare	preparation, from the same transcript
request, nothing streamed yet	the saved context, model and options; request hooks run again
request, partial answer committed	the partial becomes an aborted entry; the saved context is rebuilt and request hooks run again
retry or poll	the rest of the wait
tools	the end of the round (tool rounds (p. 94))

`test/harness-generation-recovery.test.ts` and `test/harness-tools-recovery.test.ts`. If the pinned model is gone after a restart, the run fails with `no_model`.

An interrupted request may be paid for twice: the provider may have done the work before the crash, and the resend does it again. A committed partial carries its own usage, so the ledger counts what was streamed before the crash.

WHAT THIS MEANS FOR YOU

- A resend keeps its saved model and options, but reruns request hooks. Keep hook output stable if you need identical messages.
- Tune `settings.retry;maxRetries` counts retries after the first attempt.
- Budget for the occasional double charge after a crash mid-request.

Sources: spec §8, §8.3; `src/harness/generation.ts`; `src/harness/provider.ts`; `src/harness/agent.ts` (`DEFAULT_RETRY_POLICY`); `src/harness/types.ts` (`GenerationHooks`, `ConversationRetryPolicy`); `research/pi/packages/ai/src/utils/retry.ts`; `README §Persist and Resume`; `test/harness-generation.test.ts`; `test/harness-generation-recovery.test.ts`; `research/capture/extra-p6/retry.txt`

6.4

Tool calls and replay

Every tool call commits its intent before it runs; after a crash, only a tool declared safe to repeat runs again, and every other one is reported as interrupted.

Your agent calls a `deploy` tool, and the process is killed halfway through. On restart, should it deploy again? Running it twice could be a disaster; quietly forgetting it is just as bad. Pi Durable never guesses. It commits the fact that the call is about to run, and after a crash it reruns only tools you declared safe to repeat. Every other call is reported to the model as interrupted. After this section you can decide whether a tool should be `replay: "safe"` and predict what the model sees after a crash.

One task per call

Each tool call the model makes becomes its own `pi.tool` task, owned by the generation that received the call. The task has two phases.

ToolTaskCheckpoint `src/harness/tool.ts`

TS

```
type ToolTaskCheckpoint =
  | { phase: "call" }
  // committed just before execute(): the final arguments and the replay policy
  | { phase: "execute"; arguments: JsonObject; replay: "safe" | "unsafe" };
```

A call goes through these steps:

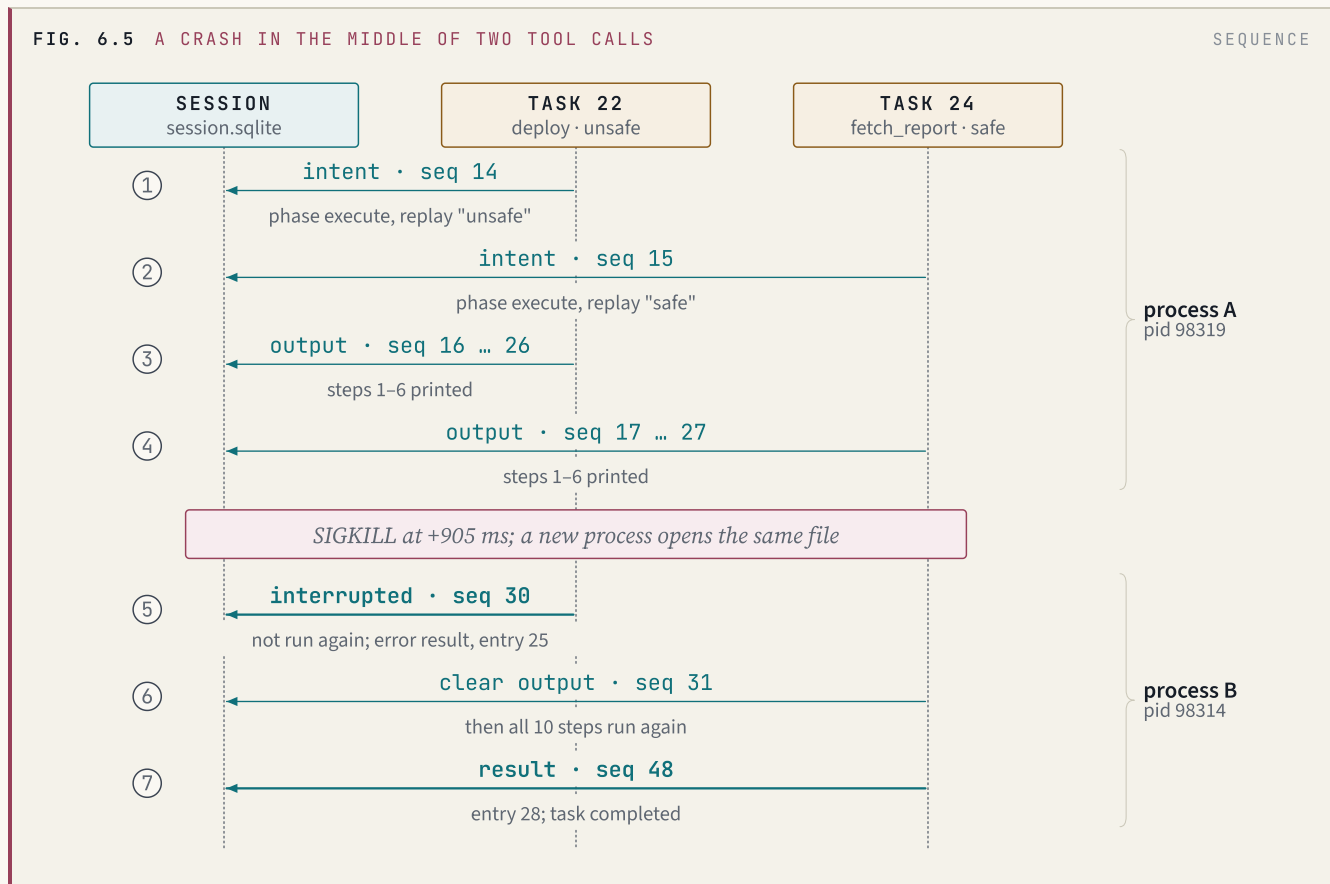
1. Find the tool by name and check the arguments against its schema.
2. Run the `beforeTool` hooks, which may block the call or change its arguments (hooks (p. 109)).
3. **Commit the intent:** phase `execute`, with the final arguments and the tool's `replay` policy, `"unsafe"` unless the tool says otherwise.
4. Run `execute()`, then the `afterTool` hooks.
5. Commit the result: one `pi.tool-result` entry, and the task ends.

This is the effect sandwich of [Part 5 \(p. 70\)](#): intent, effect, outcome. A call that fails before step 3 never touches the outside world, so it gets an error result at once. Whatever happens, the run goes on with a result entry: the model sees the error and decides what to do.

Replay after a crash

A tool task found in `execute` after a restart knows one thing: the effect may have started. The rule is simple. The tool runs again only when *both* the stored policy and the tool's current policy are `safe`. Otherwise the task commits an `interrupted` error result, built from the output the tool had already reported, and does not run the tool.

The capture runs two slow tools in two conversations, kills the process while both are printing, and resumes in a new process. `deploy` sets no `replay`, so it is `unsafe`; `fetch_report` declares `replay: "safe"`.



After a crash mid-call, a safe tool runs again and an unsafe one is reported as interrupted. Sequence numbers from research/capture/crash-before.txt and crash-after.txt; the time is on the test parent's clock. In the capture the two tools' output commits interleave.

The model in conversation 1 then receives this result:

```

entry 25, the interrupted deploy research/capture/crash-after.txt JSON
{
  "role": "toolResult", "toolCallId": "deploy-1", "toolName": "deploy",
  "content": [
    { "type": "text", "text": "deploy site: step 1/10 (pid 98319)\n ..." },
    { "type": "text", "text":
      "<harness>\n[error] Tool deploy was interrupted and may have partially run\n</harness>" }
  ],
  "isError": true,
  "timestamp": 1791211977057
}

```

The message inside the entry's `model` array. The first text part holds all six committed output lines; lines 2–6 are elided. The entry also stores the diagnostic, code `interrupted`, in `data.diagnostics`.

Task 22 ends **failed**, and the run goes on: the next generation answers “Tool deploy FAILED”. In conversation 7, the safe tool first clears its old output, then runs all ten steps in the new process and completes with entry 28. Both inputs end **done**.

Choosing a policy

`replay: "safe"` is a promise that running the effect twice is harmless: a read, an idempotent write, or a write with an idempotency key the remote side honours (the effect sandwich (p. 70)). Anything else should stay unsafe and let the model

an honest “may have partially run”. Three details make the rule stricter than it looks:

Both policies — a tool changed to unsafe is not rerun, and neither is a call stored as unsafe whose tool later became safe.

A missing tool — a tool no longer available to the conversation counts as unsafe.

A new environment — a rerun builds its environment again from the conversation’s current working directory, which may have changed (the execution environment (p. 155)).

A crash before step 3 is cheaper. The task is still in `call`, so the whole call runs once from the start, `beforeTool` included.

Tool rounds

A *tool round* is all the calls in one model response. They run in parallel by default, and one failing call never cancels the others. The round runs one call at a time when `settings.toolExecution` is `"sequential"` or any called tool declares `executionMode: "sequential"`. When every call has finished, the generation commits once. A tool result can steer that commit (defining tools (p. 112)):

terminate — end the run without another model call, but only if every call in the round asks for it.

handoff — reset the context to a handoff text and end the run (reset and handoff (p. 46)).

addTools — offer more tools from the next turn.

Otherwise a `postTools` boundary runs and a new generation takes the run. If the run is aborted mid-round, calls that never started get an `aborted` result.

Details: the error results the Harness writes

DIAGNOSTIC CODE	WHEN	TASK ENDS
<code>tool_unavailable</code>	the tool was not offered, or no longer exists	<code>completed</code>
<code>invalid_arguments</code>	the arguments fail the schema, before or after <code>beforeTool</code>	<code>completed</code>
<code>blocked</code>	a <code>beforeTool</code> hook blocks or throws	<code>completed</code>
<code>tool_error</code>	<code>execute()</code> throws, or its environment cannot be built	<code>failed</code>
<code>interrupted</code>	a crash mid-call without a safe policy	<code>failed</code>
<code>aborted</code>	the call is aborted	<code>aborted</code>

`src/harness/tool.ts` and `generation.ts`. Each result has `isError: true`; the last three keep the output reported so far. A task that ends `failed` or `aborted` also aborts any conversations the call owns (ownership (p. 76)).

WHAT THIS MEANS FOR YOU

- Mark a tool `replay`: `"safe"` only if running it twice is harmless.
- Report progress with output as you go: it is what the model sees if the call is interrupted.
- A `beforeTool` hook that must decide the same way after a crash should store its decision in a memo (memos (p. 67)).

Sources: spec §7.3, §8.4, §8.5, §12; `src/harness/tool.ts`; `src/harness/generation.ts`; `src/harness/agent.ts`; `src/harness/types.ts` (`ToolRegistration`, `ToolControl`, `ToolHooks`); README §Tools; `test/harness-tools.test.ts`; `test/harness-tools-recovery.test.ts`; `research/capture/crash-README.md`; `research/capture/crash-before.txt`; `research/capture/crash-reopen-tasks.txt`; `research/capture/crash-after.txt`

Compaction

A compaction replaces the oldest part of what the model sees with a summary; nothing is deleted, and the summary that cuts furthest wins.

Long conversations outgrow the model’s context window. Compaction asks the model to summarize the oldest part, then sends the summary in its place. The full history stays in storage; only what the model sees gets shorter. After this section you can set the token budgets, tell which compactions make a run wait, and predict which of two competing summaries wins.

What a compaction writes

A compaction is a `pi.compaction` task. It picks a prefix of the context, asks the model to summarize it, and adds one summary entry. The entry’s `head` is the first entry kept word for word, so the model’s context now starts at the summary and continues from there ([heads \(p. 46\)](#)). In this capture, `compact()` ran after three answered questions, while a fourth answer was being written.

summary entry 25 research/capture/extra-p6/compaction.txt JSON

```
{
  "kind": "pi.compaction", "id": 25, "head": 14,
  "model": [{
    "role": "user",
    "content": [{ "type": "text", "text": "The conversation history before this point was
      compacted into the following summary:\n\n<summary>\n## Goal\nPlan a week in
      Lisbon.\n</summary>" }],
    ""
  ]},
  "data": { "reason": "manual" },
  "byTaskId": 21
}
```

The summary text is wrapped onto three lines; the timestamp and `conversationId` are elided. The conversation was busy, so the summary waited in the inbox as write submission `compaction:22` and the running generation, task 21, placed it.

Three ways to start one

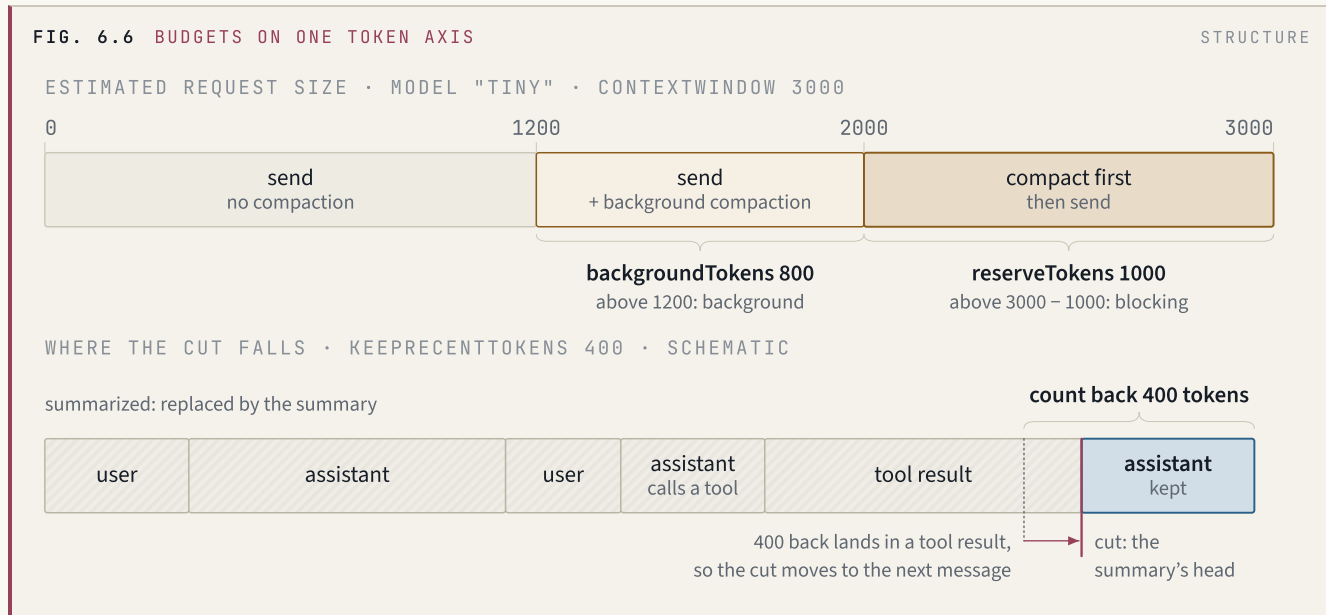
STARTED BY	REASON	RUN WAITS?	HOW THE SUMMARY LANDS
you, with <code>Conversation.compact()</code>	manual	no	as a write submission
generation, background budget	threshold	no	as a write submission
generation, blocking budget	threshold	yes	appended directly
generation, context too long	overflow	yes	appended directly

spec §8.7; `src/harness/compaction.ts`.

The last column is a safety rule. A running generation must not have its transcript change under it. A blocking compaction may append directly because its own generation is the one waiting for it. Every other summary is submitted like any write and lands at the next boundary, or at once if the conversation is idle ([submissions \(p. 84\)](#)). A non-blocking compaction never makes the conversation busy; it shows in `pi.live.compactions` while it runs.

When generation compacts

In `prepare`, generation estimates the size of the request it is about to send and compares it with two budgets from `settings.compaction`.



Two budgets decide when generation compacts; `keepRecentTokens` decides where the cut falls. Numbers from ex 25, a test model with a 3000-token window. The defaults are `reserveTokens` 16384, `backgroundTokens` 32768 and `keepRecentTokens` 20000. The lower row is schematic: counting back lands in a tool result, so the cut moves to the next message and fewer than 400 tokens stay.

Blocking — above `contextWindow` - `reserveTokens`, generation starts a compaction and waits for it, then prepares again and sends the request whatever the outcome.

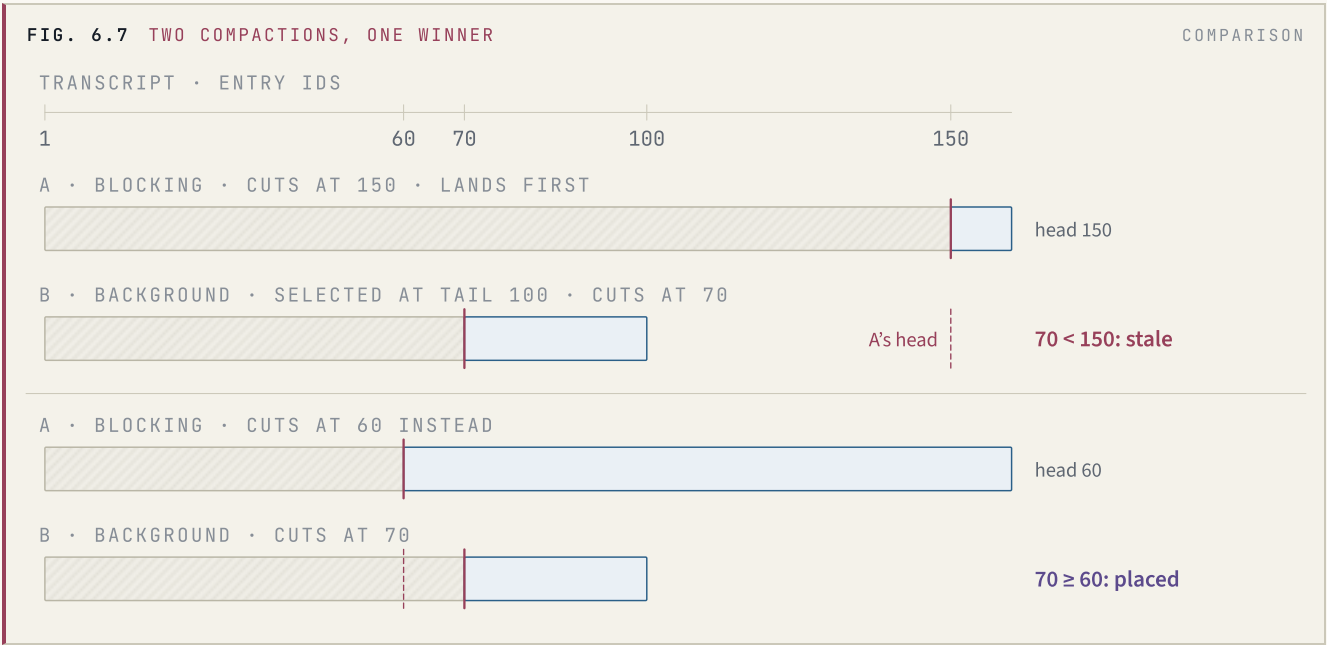
Background — above that line minus `backgroundTokens`, generation starts a compaction in the background and sends the request without waiting. `backgroundTokens`: 0 turns this off.

The cut is chosen by counting back from the newest entry until `keepRecentTokens` are kept. It then lands on the next user or assistant message, never on a tool result, so a kept assistant message keeps its tool results. If the whole context is smaller than `keepRecentTokens`, nothing is compacted.

A too-long context gets one rescue. When the provider rejects a request as too long, generation compacts and prepares the same attempt again. A generation compacts at most once, so a second overflow fails the run with `model_error`. Run 25 shows this: its last turn compacts on the blocking budget, then hits an overflow and fails.

When summaries compete

Compactions never coordinate; several can run at once. The stale-head rule of [the inbox \(p. 84\)](#) alone picks the winner. A summary whose cut is older than the context's current start settles `stale`. One that cuts at or after it is placed, whenever it started.



Compactions never coordinate; a summary that cuts before the context’s current start is stale. The worked example of spec §6. Hatching marks what each summary replaces; the dashed line is the context’s start once A has landed.

Details

The summary request — no tools, the transcript with tool results cut to 2000 characters, and any extra instructions you passed to `compact()`. Only a clean stop with text counts as a summary. Retryable errors retry under `settings.retry`. A `beforeCompact` hook can decline the compaction or supply its own summary ([hooks \(p. 109\)](#)).

Aborting — `Conversation.abort()` (the Esc key) aborts a manual compaction. A background compaction keeps going unless you pass `{ background: true }` or call `abortTask()`. A blocking one stops with its generation.

Edits during compaction — an app edit or head placed while a compaction runs can be lost or undone when the summary lands [spec §12](#).

WHAT THIS MEANS FOR YOU

- The defaults suit large windows. For a small window, keep `keepRecentTokens` well below the blocking line, or the context can overflow before anything is compacted.
- Call `compact()` any time; it never blocks the run.
- Place app edits before compacting, not during.
- Every summary attempt is paid for, even one that ends stale.

Sources: [spec §6](#), [§8.3](#), [§8.7](#), [§12](#); [src/harness/compaction.ts](#); [src/harness/generation.ts](#); [src/harness/live.ts](#) (`CompactionStatus`); [src/harness/agent.ts](#) (`DEFAULT_COMPACTION_POLICY`); [README §Compaction](#); [docs/pico-v5-handoff.md §20](#); [test/harness-compaction.test.ts](#); [ex 25](#); [run 25](#); [research/capture/extra-p6/compaction.txt](#)

Usage and cost

Spend is recorded in the same commit as the response that caused it, so the ledger and the transcript can never disagree.

How much has this conversation cost so far? `pi.usage` answers it. The ledger is updated in the same commit as each model response, so it matches the transcript exactly, even after a crash. After this section you can read the ledger, say which attempts it counts, and total spend across conversations without counting anything twice.

The ledger

`pi.usage` is a built-in document, one per conversation, with two maps. `models` is keyed by "provider/model". `tools` is keyed by tool name, for tools that pay for their own model calls. Each value is a pi-ai Usage record, the same shape every assistant message carries. A fork starts at zero.

pi.usage after one tool call and one answer research/capture/extra-p6/live-run.txt JSON

```
{
  "models": {
    "faux/faux-1": {
      "input": 74, "output": 15, "cacheRead": 54, "cacheWrite": 75, "totalTokens": 218,
      "cost": { "input": 0, "output": 0, "cacheRead": 0, "cacheWrite": 0, "total": 0 }
    }
  },
  "tools": {}
}
```

The sum of two responses: the tool-calling one (118 tokens) and the final answer. The test provider reports zero cost.

Token counts — `input`, `output`, `cacheRead`, `cacheWrite` and `totalTokens`, always present.

Optional counts — `cacheWrite1h` (cache writes with one-hour retention) and `reasoning` (output spent thinking), when the provider reports them.

cost — the same buckets plus `total`, priced by pi-ai for the model.

The ledger is a running sum. Per-turn spend is still in the transcript: each assistant entry carries its own `usage`.

What gets counted

Usage is recorded in the commit that appends the response, never later.

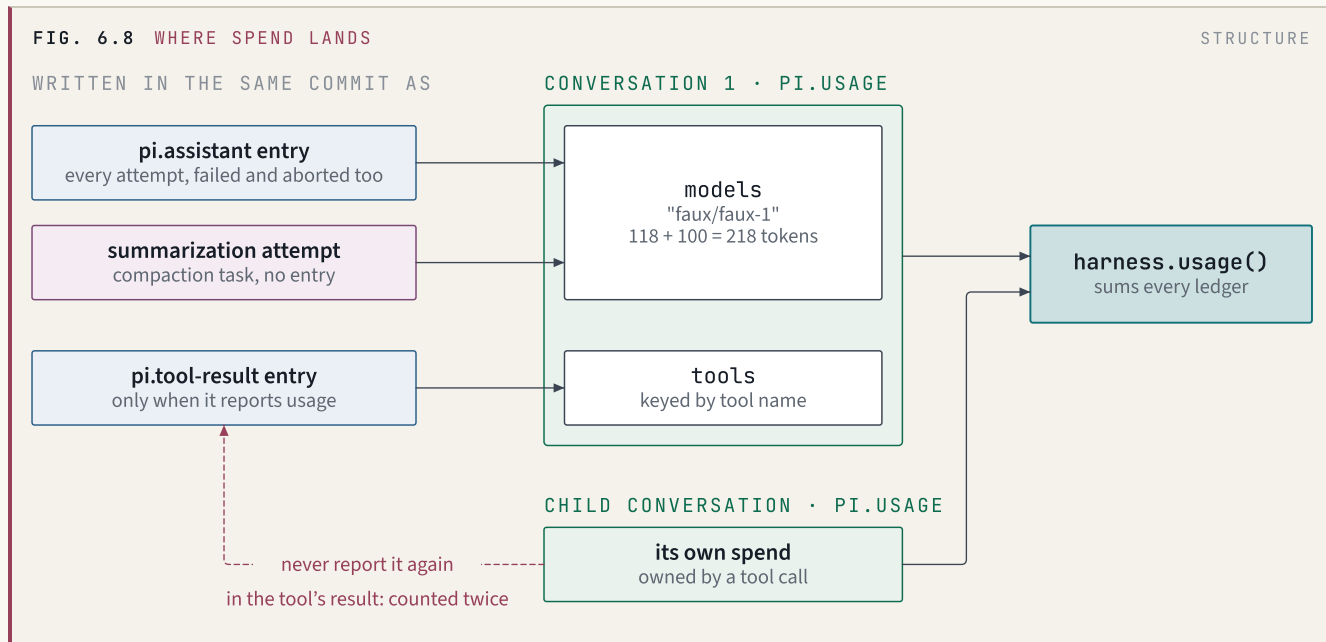
SPEND	GOES TO
every model response: answers, tool calls, failed attempts	models
a partial answer committed before a crash or abort	models
every summary attempt, even one that fails or ends stale	models
a tool result that reports usage	tools[name]

spec §8.6; src/harness/usage.ts, called from generation.ts, tool.ts and compaction.ts.

Failed and aborted attempts count, because the provider charged for them. An attempt cut off before anything was committed has no response, so nothing is counted for it; the resend is counted when it answers.

Totals across conversations

Each conversation counts only its own spend. `harness.usage()` adds up every conversation's ledger into one total; nothing stores that total. For other totals, such as one subagent tree, sum the ledgers yourself over `scanConversations()`.



Spend is recorded once, in the conversation that made it, in the commit that records the response. Writers from spec §8.6; the `models` total from research/capture/extra-p6/live-run.txt. The dashed path is the double count of spec §12.

DOUBLE COUNT

A tool that runs a subagent in its own conversation must not report that conversation's spend in its result's usage. The child's `pi.usage` already counts it, so `harness.usage()` would count it twice (subagents (p. 121)).

A UI reads the ledger like any other document, through the conversation view's `docs["pi.usage"]` (the conversation view (p. 129)).

WHAT THIS MEANS FOR YOU

- Read per-conversation spend from `pi.usage`, and the grand total from `harness.usage()`.
- Report tool usage only for spend the tool paid for directly.
- Tool names are data: read the maps with `Object.entries()`, not property access, in case a tool is named `__proto__`.

Sources: spec §8.6, §12; `src/harness/usage.ts`; `src/harness/generation.ts`; `src/harness/tool.ts`; `src/harness/compaction.ts`; `src/harness/harness.ts` (`usage`); `README §Usage and Cost`; `test/harness-inbox.test.ts` (describe "usage"); `test/harness-compaction.test.ts`; `research/capture/extra-p6/live-run.txt`

Extensions and the Agent

Code lives in a registry the process owns; conversations store only names, so the code can change while agents run.

7.1	Extensions and the registry	103
7.2	The per-conversation agent	106
7.3	Hooks	109
7.4	Defining tools	112
7.5	The system prompt	115
7.6	Reload while running	118
7.7	Subagents	121

07

7.1

Extensions and the registry

Your code lives in a registry inside the process; storage only ever records the names of the extensions a conversation wants.

An agent needs tools, a system prompt and a few guard rails. You also want to ship a fix to one of them without stopping the agent, and to restart after a crash without losing anything. Pi Durable gets both by keeping all code in memory and storing only names. After this section you can package your code as an extension, install it, replace it, and say what a restart needs.

The bundle

An **extension** is a named bundle of your code. It is a plain object with a `name` and up to five lists. `defineExtension()` only gives the object its type; it checks nothing `src/harness/define.ts`. Every list is matched to the rest of the system by a string name, never by object identity.

The **registry** is where extensions meet the Harness. `createRegistry()` returns an object your application owns. It already holds the three built-in task definitions, `pi.generation`, `pi.tool` and `pi.compaction`, which cannot be removed or replaced. `Harness.open()` rejects a registry that lacks them, so always start from `createRegistry()` `src/harness/registry.ts`; `src/harness/harness.ts`.

three extensions, one registry README §Extensions · spec §7.1

TS

```
import { createRegistry, defineExtension, hook, section, ToolTask }
  from "@earendil-works/pi-durable";
import { CodingTools } from "@earendil-works/pi-durable/tools";

const Coding = defineExtension({
  name: "coding",
  sections: [
    section("preamble", () => "You are a concise coding assistant.", { tag: false }),
    section("cwd", (input) => input.env?.cwd),
  ],
});

const Permissions = defineExtension({
  name: "permissions",
  hooks: [hook(ToolTask, {
    beforeTool: (call) => (isDangerous(call) ? { block: "Needs approval" } : undefined),
  })],
});

const registry = createRegistry();
registry.install(CodingTools); // "coding-tools": read, write, edit, bash
registry.install(Coding);
registry.install(Permissions);
```

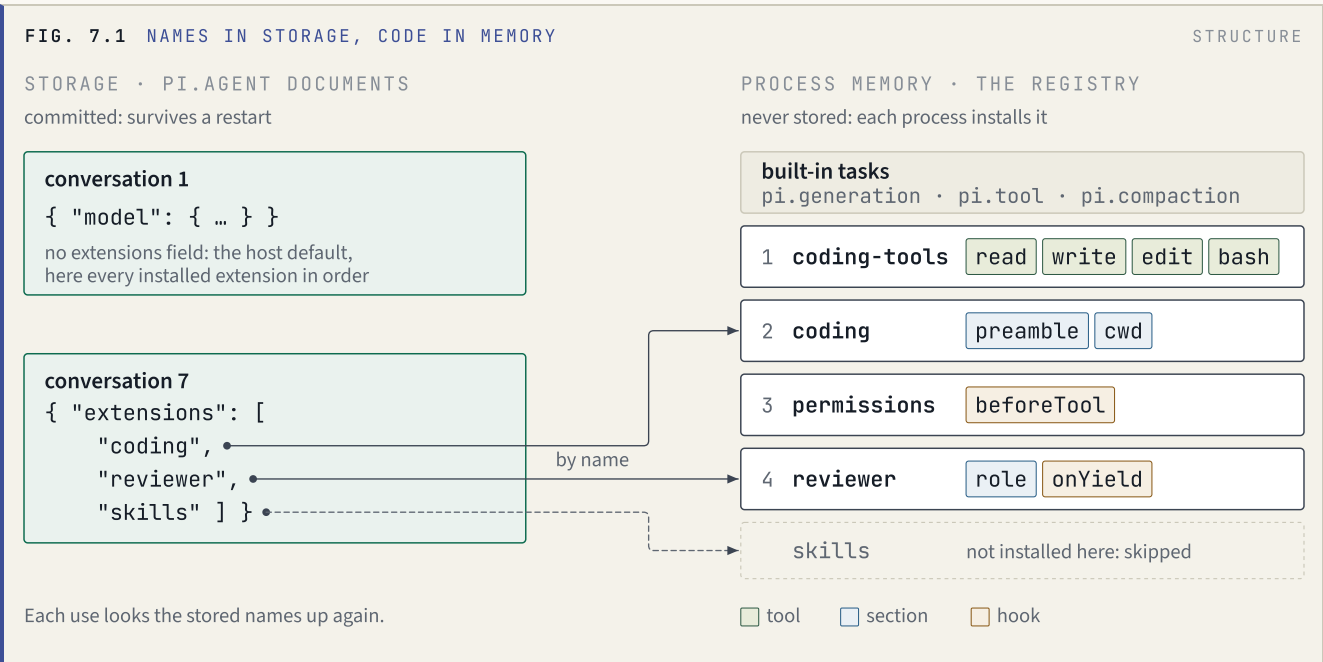
Install order matters. By default every conversation uses every installed extension in install order, so `coding-tools`' tools come before anything `coding` adds.

FIELD	HOLDS	MATCHED BY	SECTION
tools	tools the model can call	tool name	7.4 (p. 112)
sections	pieces of the system prompt	section key	7.5 (p. 115)
hooks	handlers the built-in tasks call	task name	7.3 (p. 109)
wraps	decorators for another extension’s tool or section	tool name or section key	7.2 (p. 106)
tasks	your own durable task definitions	task name	5.4 (p. 73)

The `Extension` interface, `src/harness/types.ts`.

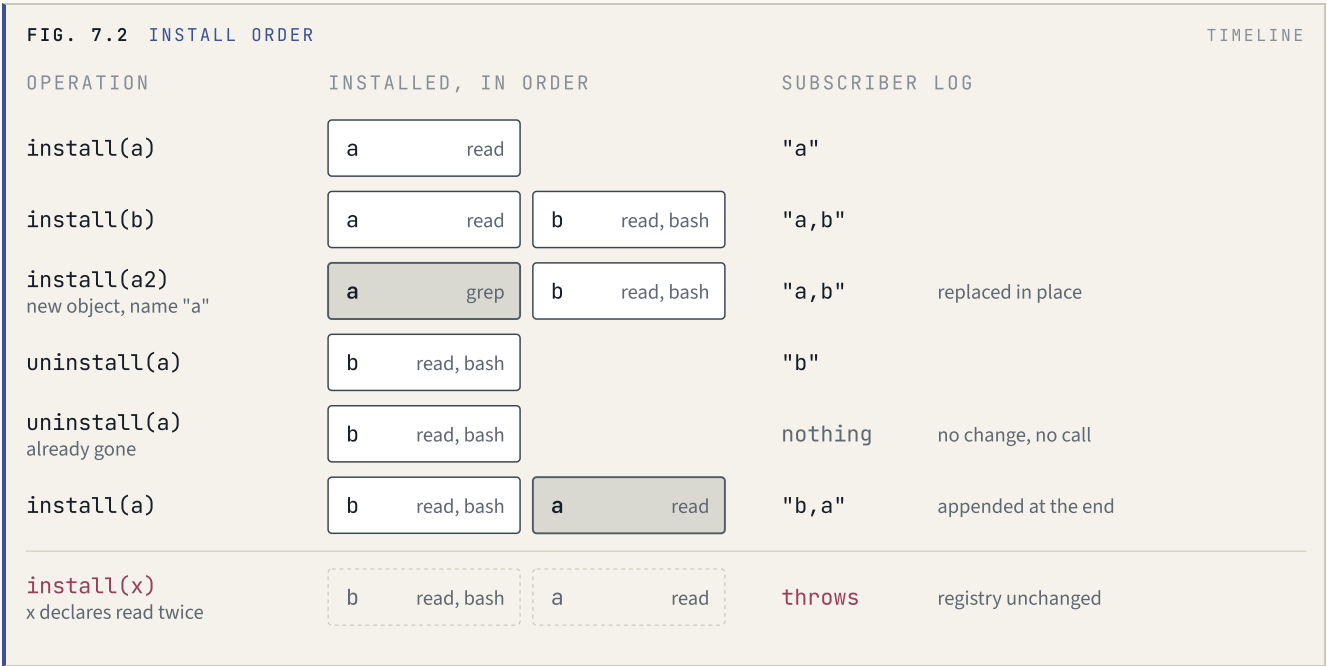
Names in storage, code in memory

Nothing in the registry is written to disk. In the spec’s words, the registry “is process-local, may outlive a Harness, and is not persisted” [spec §7.1](#). A conversation records which extensions it wants as a list of names in its `pi.agent` document (the per-conversation agent [\(p. 106\)](#)). Whenever the Harness needs code, it looks those names up in the registry.



Install, replace, uninstall

The registry has two operations. `install(extension)` adds a new name at the end, or replaces the extension that already has that name, in the same position. `uninstall(extension)` removes whatever holds that name. Each change takes effect immediately, and “an extension is the unit of reload” [spec §7.1](#).



A same-name install replaces in place; uninstalling and installing again moves the extension to the end. The operations and the subscriber’s log [“a”, “a, b”, “a, b”, “b”, “b, a”] come from one case of `test/harness-registry.test.ts`. The last row is a separate case of the same file: an extension that declares `read` twice is rejected. Uninstalling an absent name and a rejected install publish nothing.

Position is precedence. When two extensions define a tool with the same name, the later one wins, and later hooks run later ([hooks \(p. 109\)](#)). Because a replacement keeps its position, reloading an extension never changes who wins.

An install that would leave the registry invalid throws and changes nothing. Tool names and section keys must be unique within one extension; across extensions, a repeated tool name is how one extension overrides another. Section keys are lowercase (`^[a-z][a-z0-9_-]*$`) and never `instructions`, which is reserved. Task names must be unique across the whole registry. Work already running never sees the registry change under it ([reload while running \(p. 118\)](#)).

WHAT THIS MEANS FOR YOU

- Install every extension before `Harness.open()`, in the order you want precedence.
- Never rename an extension, tool or task that conversations already use: stored names stop matching.
- To ship a fix, install the new object under the same name. Install and uninstall call no cleanup code, so free shared resources yourself, and only once old calls finish [spec §12](#).

Sources: `src/harness/registry.ts`; `src/harness/define.ts`; `src/harness/types.ts` (`Extension`, `Registry`, `RegistrySnapshot`); `src/harness/harness.ts`:410–435; [spec §7.1](#), [§2.2](#), [§12](#); [README §Extensions](#); `test/harness-registry.test.ts`; `research/capture/sqlite-rows.txt`

The per-conversation agent

Each conversation stores its choices as names; the Harness turns them into real tools and prompt sections each time it needs them.

Different conversations need different setups: the main chat gets every tool, a read-only reviewer gets two, a subagent gets its own instructions. The **agent** of a conversation is that setup: its model, thinking level, extensions, tools, prompt sections, instructions and working directory. After this section you can configure one and predict exactly which tools the model will be offered.

Stored choices, resolved agent

The agent exists in two forms. The stored form, `AgentState`, lives in the conversation's `pi.agent` document and holds only names. A field you never set follows the host's defaults. The resolved form, `Agent`, is what the Harness builds from the stored names, the current registry and the host settings whenever it needs real objects. The stored document forks and rewinds with its conversation (the built-in documents (p. 60)).

FIELD	STORED AS	WHEN UNSET
model	{ provider, modelId }	no model; a request fails with <code>no_model</code>
thinkingLevel	a pi-ai level	"off"
extensions	a list of names, or { add?, remove? }	the host default: every installed extension
tools	a list of names, or { remove }	every tool of the selected extensions
instructions	a string	none; when set, rendered as the last prompt section
cwd	a path	none; passed to the host's environment factory (9.5 (p. 155))

`AgentState` and its defaults, `src/harness/types.ts` and `src/harness/agent.ts` `resolveAgent()`.

`Conversation.configure(change, context)` edits the stored agent. Each field follows one rule: "a given field replaces the stored one, `null` clears it, `undefined` changes nothing" `src/harness/agent.ts`. You may pass extension and tool objects; they are stored as their names. The standalone `configure(tx, id, change)` does the same inside a larger **commit**, an atomic save: everything in it is stored together, or nothing is. Example 7 shows what each call stores.

configure, then read back ex 07 · research/runs/07-configuration.txt TS

```
// default tools: [ 'read', 'write', 'grep' ]
await root.configure({
  model: { provider: "anthropic", modelId: "claude-sonnet-4-5" },
  thinkingLevel: "high",
  tools: [write, read], // objects in, names stored
}, context);
// stored: { model: {...}, thinkingLevel: 'high', tools: [ 'write', 'read' ] }

await root.configure({ extensions: { remove: [Search] }, tools: null }, context);
// without search: [ 'read', 'write' ]

await root.configure({ extensions: [Files, Search] }, context);
registry.uninstall(Files); // files uninstalled: [ 'grep' ]
registry.install(Files); // files reinstalled: [ 'read', 'write', 'grep' ]
```

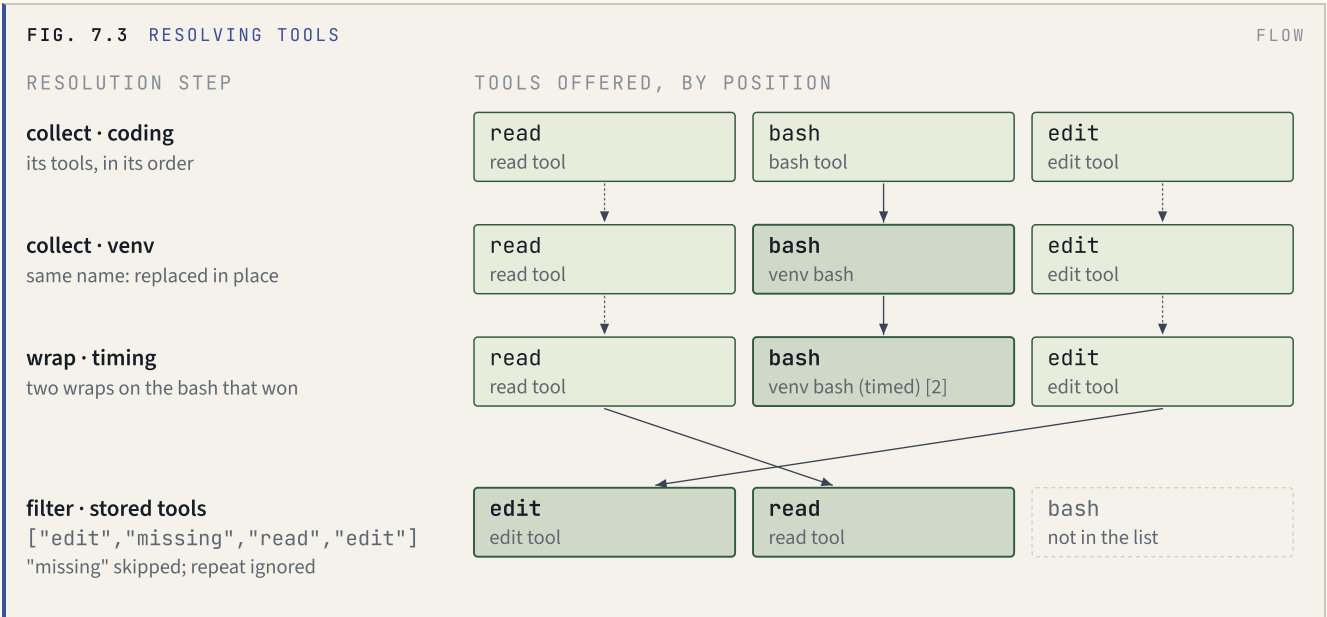
The comments are the example's printed output. `null` clears the tool filter, so every tool of the selection is offered again.

Because a field is replaced whole, `{ remove }` lists do not add up. Removing `edit` and then removing `bash` leaves only `bash` removed: “edit is offered again” [spec §2.2](#). A UI toggle must read the current list and write the new one. Configuring appends nothing to the transcript; the next model request picks up the change.

From names to tools

Resolution turns stored names into the tools and sections a request uses. It runs the same four steps every time and writes nothing:

1. **Pick extensions.** A stored list selects exactly those names, in its order. Otherwise start from the host default (every installed extension, in install order), then apply `add` and `remove`. Names that are not installed are skipped.
2. **Collect tools** in extension order. A later tool with the same name replaces the earlier one in place.
3. **Apply wrappers** to whichever tool or section holds the target name.
4. **Filter tools.** A stored list keeps exactly those names, in its order; `{ remove }` drops names. Sections are collected the same way, with `instructions` last.



Resolution collects tools in extension order, replaces same names in place, wraps the winner, then filters. Extensions, descriptions and filter from `test/harness-registry.test.ts`. The last row shows what the stored filter keeps from the row above it.

Two failures are quiet on purpose. A wrapper that throws, or that renames its target, removes that tool for this resolution and is reported to `HarnessOptions.onReport`; a call to it gets a `tool_unavailable` result. A wrapper whose target is not selected does nothing, so a timing wrapper is safe where there is no `bash` [spec §7.1](#).

When a change takes effect

The Harness resolves the agent again at each decision, never once per run: “each reader resolves once per decision” [spec §7.1](#). Host settings work the same way. `HarnessOptions.settings` holds Harness-wide policy (the default selection, retry, compaction, tool execution mode, queue modes). It is never copied or stored, so a settings object with getters “follows the user’s preferences without a Session write” [spec §2.2](#). The defaults are in [settings and defaults \(p. 180\)](#).

WHO	READS	WHAT	WHEN
a model request		model, tools, prompt, stream options	once, while preparing that request
a tool call		the called tool's code	when the call starts, and again on recovery
any task phase		hooks	once per phase, from that phase's registry snapshot
a tool round		parallel or sequential execution	once per round
an input boundary		steering and follow-up queue modes	at each boundary
tools and prompt sections		the environment for the current cwd	at each use

Condensed from spec §7.1, “Who resolves what, and when”.

So a change made during a turn reaches the next model request. A call the model already made resolves its tool when it runs. If you removed that tool in between, the call gets `tool_unavailable`; if you changed `cwd`, the call sees the new directory `README §Per-Conversation Agent`.

Copies at creation

A new conversation starts with a copy of someone’s agent, or with nothing `src/harness/agent.ts createAgent()`:

Fork — the parent’s agent as it was at the fork point (`forks` (p. 43)).

Task-owned — a copy of every stored field of the owning task’s conversation, `instructions` included. Later changes to the owner do not reach it.

Ownerless — empty, `{}`: every field follows the host.

The `agent` option of `root()`, `createConversation()` and `fork()` then applies its change in the same commit.

COPIED LISTS

`{ add, remove }` “always edits the host default selection”, never a copied list. A child that copied the list `[ "a", "b" ]` and then receives `{ remove: [ b ] }` ends up with the host default minus `b`, not `[ "a" ]`. To edit a copied list, read the resolved agent’s `extensions` and write the new list `spec §2.2, §12`.

WHAT THIS MEANS FOR YOU

- Leave fields unset when the conversation should follow the host; set them only to differ.
- Use a stored list when a conversation needs an exact, stable tool set, such as a reviewer.
- Expect configuration changes to apply at the next model request, not mid-request.

Sources: `src/harness/agent.ts`; `src/harness/types.ts` (`AgentState`, `AgentChange`, `Agent`, `HarnessSettings`, `Settings`); `src/harness/scheduler.ts`; `spec §2.2, §7.1, §12`; `README §Per-Conversation Agent`, `§Settings`; `ex 07` and `run 07`; `test/harness-registry.test.ts`

Hooks

A hook is a question a built-in task asks your extensions just before it commits a decision; your answer shapes what gets written.

You often want to steer the agent without rewriting it: block a dangerous shell command, trim old messages before a request, or ask the model to keep going when its answer is weak. Hooks do this. After this section you can write each hook, predict how several extensions’ answers combine, and keep a human approval stable across a crash.

Who gets asked

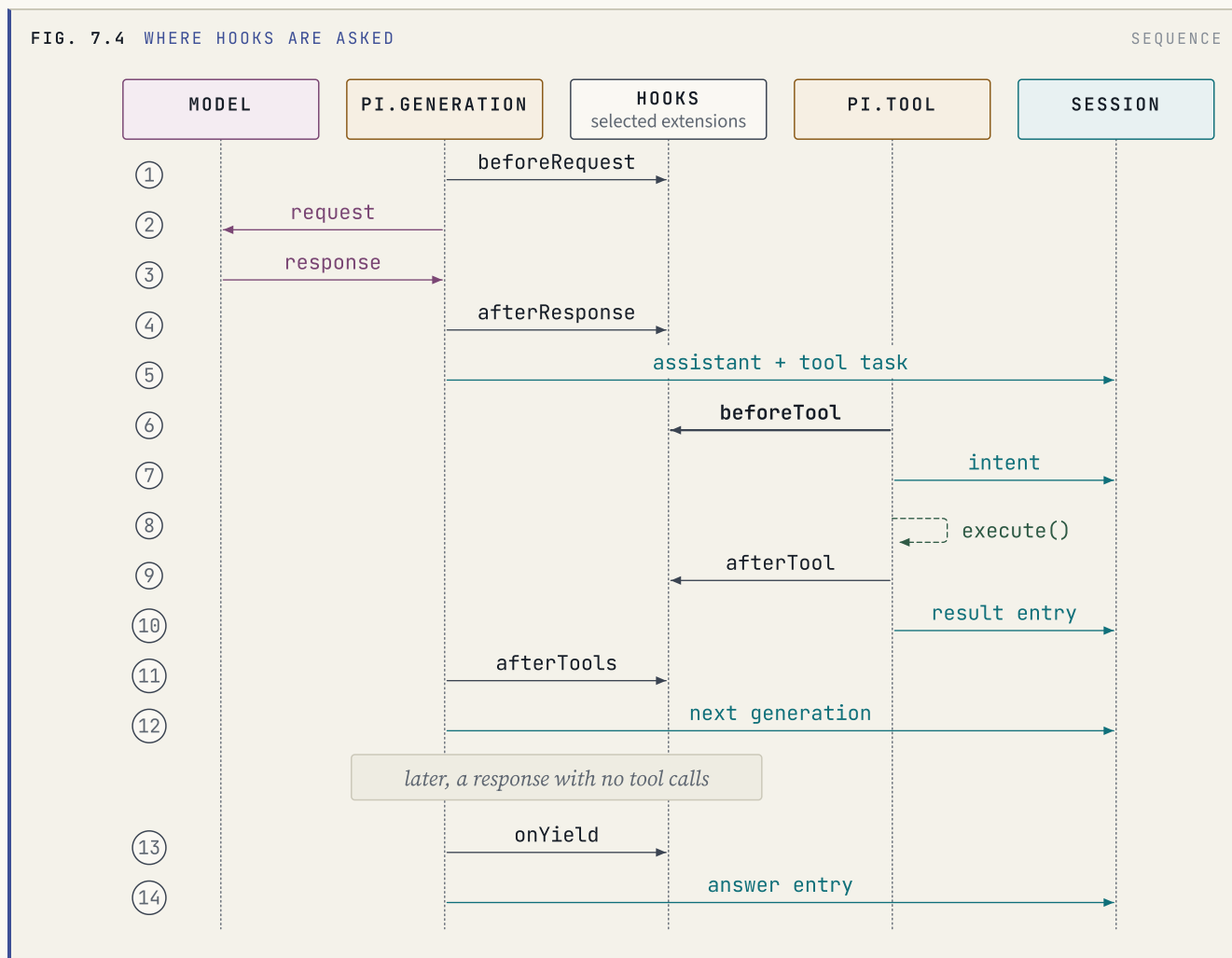
An extension registers handlers with `hook(task, handlers)`, for example `hook(ToolTask, { beforeTool })`. The task object gives type checking; only its name is kept, so “hooks survive a reload of their task” [spec §7.2](#).
A task asks only the hooks of extensions that its conversation selects. That is the whole scoping model: “To limit a hook to some conversations, select its extension only there” [README §Hooks](#). There is no global hook. The flip side matters for safety: a conversation whose extension list leaves out your permissions extension runs without its guard [spec §12](#).

The seven hooks

The generation task asks four hooks, the tool task two and the compaction task one. When several extensions answer, each hook combines the answers in its own way.

HOOK	ASKED BY	WHAT IT CAN DO	IF IT THROWS
beforeRequest	generation	replace the messages of this one request; each handler gets the previous one’s result	reported, ignored
afterResponse	generation	observe every model response	reported, ignored
onYield	generation	answer a final answer with { continue }; the first wins	reported, ignored
afterTools	generation	observe a finished round of tool results	reported, ignored
beforeTool	tool	rewrite the arguments, or block the call; the first block wins	the call is blocked
afterTool	tool	replace the result, handler by handler	reported, ignored
beforeCompact	compaction	decline, or supply a summary; the first decision wins	reported, ignored

GenerationHooks, ToolHooks and CompactionHooks, [src/harness/types.ts](#); [spec §7.2](#); [src/harness/tool.ts](#). “Reported” means passed to `HarnessOptions.onReport`. Handlers run in extension order. A throwing guard blocks: it fails closed.



Every hook is a question a built-in task asks before the commit it informs. One tool-using turn and its final answer, from `src/harness/generation.ts` and `tools.ts`; the guard, `beforeTool`, is in bold. `beforeCompact` is asked after compaction picks its range and before it summarizes ([compaction \(p. 97\)](#)).

Make decisions crash-safe with memos

A hook can run twice: “a crash before the consuming commit may rerun them” [spec §7.2](#). That is harmless for a pure check, but a human approval must not be asked again or change. Store it with `api.memo(name, ...)`, which keeps the first value written: a rerun that tries to record “denied” gets “approved” back `test/harness-tools.test.ts`. Hooks share the asking task’s memo names, so prefix yours ([phases, steps and progress \(p. 67\)](#)). To show status in a UI, commit it to a document ([the conversation view \(p. 129\)](#)).

Keep state in documents

Selecting an extension decides whether a conversation has a behavior. A document holds that behavior’s state. Plan mode in [spec §7.2](#) is the pattern: the extension is selected everywhere, and its `beforeTool` reads the conversation’s own plan-mode document.

a hook driven by an extension document spec §7.2

TS

```
const PlanModeDoc = defineDoc<{ enabled: boolean }>({
  kind: "app.plan-mode", version: 1, scope: "conversation",
  history: "latest", fork: "current", initial: () => ({ enabled: false }),
});
export const PlanMode = defineExtension({
  name: "plan-mode",
  hooks: [hook(ToolTask, {
    // An absent document means plan mode is off.
    beforeTool: async (call, api, context) =>
      (await api.snapshot(PlanModeDoc, api.conversationId, context))?.enabled
        && writes(call)
        ? { block: "Plan mode: read-only" }
        : undefined,
  })],
});
// /plan toggles this conversation's state; no selection change.
await root.commit(async (tx) => {
  (await tx.doc(PlanModeDoc, root.id)).enabled = true;
}, context);
```

Extensions get no setup step when a conversation is created, so a hook treats a missing document as “off”. Example 27 builds the other variant, a mode switched by changing the selection ([building a coding agent \(p. 159\)](#)).

Details

Arguments are checked twice — against the tool’s schema before `beforeTool` and after it. A rewrite that breaks the schema gives `invalid_arguments`; a block gives `blocked`.

`beforeTool` is not rerun after intent — once the call’s intent is committed, recovery reuses the stored arguments ([tool calls and replay \(p. 94\)](#)).

Hook work holds the task — a task a hook creates is owned by the asking task and keeps it open until it ends spec §12.

WHAT THIS MEANS FOR YOU

- Put guards in `beforeTool`; a throw blocks, which is the safe default.
- Select a guard’s extension in every conversation that needs it, including subagents that use a stored extension list.
- Store any decision a human made in a memo, and any mode in a document.

Sources: `src/harness/define.ts hook()`; `src/harness/types.ts (HookApi, GenerationHooks, ToolHooks, CompactionHooks)`; `src/harness/scheduler.ts`; `src/harness/agent.ts`; `src/harness/tool.ts`; `src/harness/generation.ts`; spec §7.2, §7.3, §12; README §Hooks; `test/harness-tools.test.ts`

7.4

Defining tools

A tool is a schema, a replay policy and an `execute()` function; the Harness turns what it reports into exactly one result entry.

A tool is how the model acts on the world: reading a file, running a command, calling an API. The hard part is a crash halfway through. After this section you can define a typed tool, stream its output to a UI, choose its replay policy, and read the result entry it leaves behind.

A tool in eleven lines

`defineTool()` types a tool from its `TypeBox parameters` schema. The Harness checks the model's arguments against that schema before `execute()` runs, so `args.n` below really is a number.

the count tool README §Tools · research/capture/scripts/one-turn.ts TS

```
import { Type } from "@earendil-works/pi-ai";

const count = defineTool({
  name: "count",
  description: "Count from 1 to n",
  parameters: Type.Object({ n: Type.Number() }),
  execute: async (args, api) => {
    for (let i = 1; i <= args.n; i++) api.output(`${i}\n`);
    return {};
  },
});
registry.install(defineExtension({ name: "count", tools: [count] }));
```

`execute()` returns no content, so the text it wrote with `api.output()` becomes the result.

Only the standard pi-ai tool fields (name, description, parameters) are sent to the model. Your own fields, such as a prompt snippet, stay in your code (the system prompt (p. 115)). Four optional fields set policy:

FIELD	DEFAULT	MEANING
replay	"unsafe"	may a call cut short by a crash run again on restart?
executionMode	"parallel"	"sequential" makes the whole round of calls run one at a time
prepareArguments	none	repairs arguments before they are checked, such as a JSON string where an array belongs
outputLimits	50 KiB, 2,000 lines, keep the head	bounds the output kept for the result

ToolRegistration, src/harness/types.ts; limits from src/truncate.ts. The execution mode default comes from settings' `toolExecution`.

Choose a replay policy

Every tool author must answer one question: if the process dies while my tool runs, may it run again? Before running a call, the tool task writes the final arguments and the policy to disk. On restart it reruns the call only if both the stored and the current tool say "safe". Otherwise it writes an `interrupted` error result with whatever output was saved. Declare `safe` only when running twice with the same arguments is harmless. The full story, with a captured crash, is in [tool calls and replay](#) (p. 94).

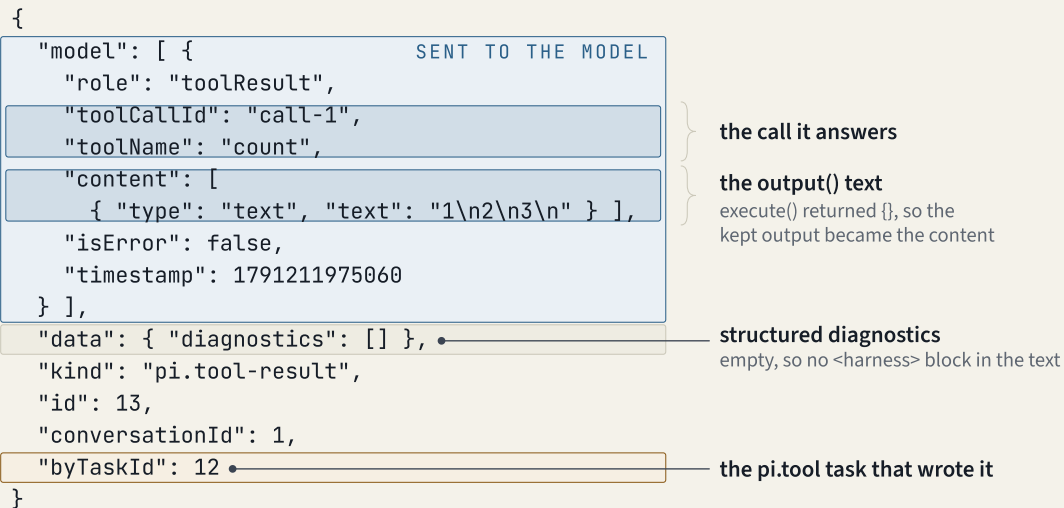
From result to entry

Every field of the returned result is optional: `content`, `isError`, `details`, `diagnostics`, `usage` and `control`. The tool task fills the gaps, runs the `afterTool` hooks, bounds long text (adding a truncated warning), and writes one entry. When there are diagnostics, the text the model sees ends with a `<harness>` block, one line per diagnostic, and the entry's `data` keeps them as a list.

FIG. 7.5 ANATOMY OF A RESULT ENTRY

ANATOMY

ENTRY 13 · RESEARCH/CAPTURE/SQLITE-ROWS.TXT



A tool-result entry stores exactly what the model saw, plus the structured diagnostics. Entry 13 of `research/capture/sqlite-rows.txt`, the `count` tool's answer to `count { "n": 3 }`, keys in captured order.

Errors come in two kinds, and the difference matters. A result with `isError: true` is an ordinary answer: the task completes. A throw from `execute()` also becomes an error result, with code `tool_error` and the partial output kept, but it ends the tool task `failed`, which aborts anything the call started `src/harness/tool.ts`. Errors the Harness writes itself carry a code that says why: `tool_unavailable`, `invalid_arguments`, `blocked`, `interrupted` or `aborted`.

Two result fields act on the turn. `usage` records the tool's own spend (`usage` and `cost` (p. 100)). `control` can offer more tools (`addTools`), end the run when every result of the round asks (`terminate`), or hand off to a fresh context (`handoff`, `reset`, `handoff` and `heads` (p. 46)).

What `execute()` receives

`execute(args, api, context)` gets an `api` object made for this one call. Every operation on it fails once the call has ended. The parts you will use most:

`env` — files and processes for the conversation's current `cwd`, or undefined when the host configured none (the execution environment (p. 155)).

`output(chunk)` — append running text, like `stdout`. If the result has no `content`, the kept text becomes the content.

details(value, context) — publish structured progress for a UI; the last value becomes the result’s `details` if the result has none.

diagnostic(d) — attach a remark for the model, such as a warning, separate from the tool’s data.

commit, memo, createTask — commits, memos and child tasks, as in any task.

conversation(id) — a handle to a conversation this call created ([subagents \(p. 121\)](#)).

Output, details and diagnostics reach the UI while the tool runs, about every 100 ms through the live document. The model sees none of it until the result entry is written ([run control and the live document \(p. 88\)](#)).

Built-in tools and wrappers

`@earendil-works/pi-durable/tools` ships `read`, `write`, `edit` and `bash`, together as the `CodingTools` extension (`coding-tools`). Nothing installs them for you. They work only through `api.env`, and all four are `replay-unsafe`. `bash` keeps the tail of long output, points to a file with the full output, and fails on a nonzero exit or a timeout `src/tools/`.

To change a tool for some conversations, override or wrap it rather than copy it. A later extension’s tool with the same name replaces the earlier one, and `wrapTool(tool, wrapper)` decorates whichever tool won. A wrapper “never captures a base, so reloading the base keeps its wrappers” [spec §7.3](#). Example 30 gives one conversation a `bash` that activates a `virtualenv` and times every `bash` call in both [run 30](#).

WHAT THIS MEANS FOR YOU

- Leave `replay unsafe` unless a rerun is harmless; record anything random in a memo first.
- Return `isError: true` for expected failures; throw only when the call’s own work should be aborted.
- Use `diagnostic()` for remarks to the model, not the tool’s output.

Sources: `src/harness/define.ts`; `src/harness/types.ts` (`ToolRegistration`, `ToolExecutionApi`, `ToolExecutionResult`); `src/harness/tool.ts`; `src/truncate.ts`; `src/tools/index.ts`, `bash.ts`, `env.ts`; [spec §7.3](#), [§8.5](#); [README §Tools](#); [ex 30](#), [run 30](#); [research/capture/sqlite-rows.txt](#)

7.5

The system prompt

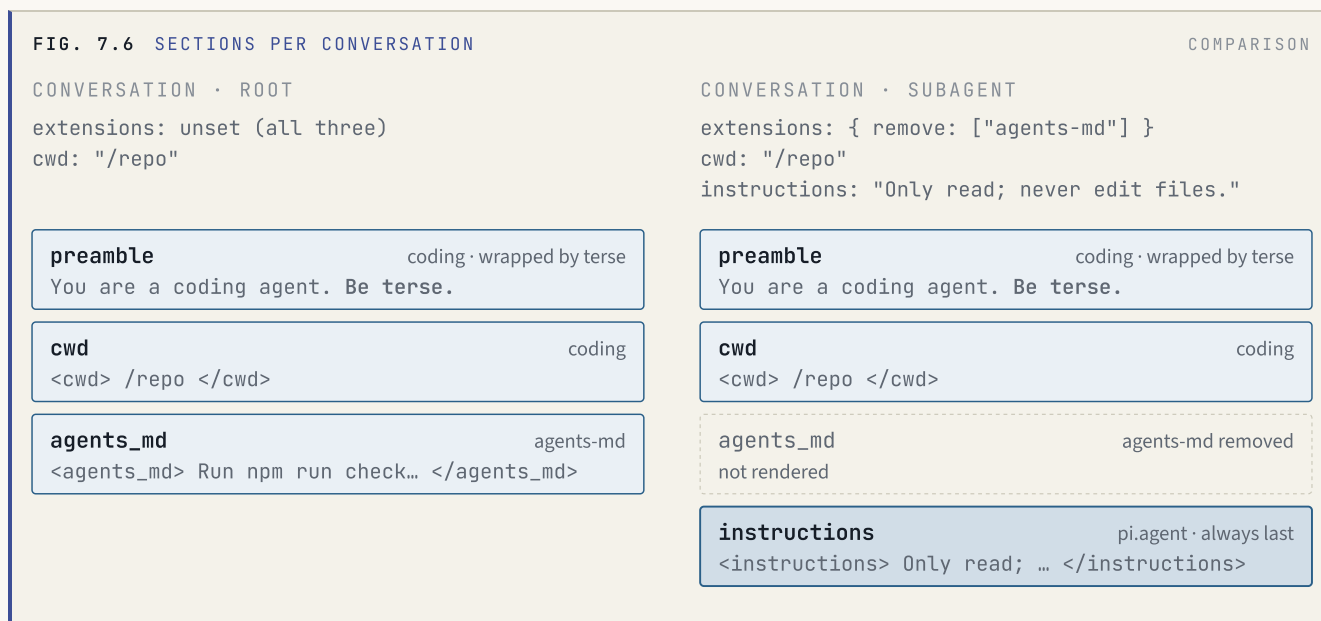
Nothing stores the prompt; each request renders it from sections, and only what changed is added to the transcript.

Providers cache the start of a prompt, so rewriting the system prompt on every turn wastes money and time. Pi Durable builds the prompt from small named pieces and, when one piece changes, sends only that piece. After this section you can write prompt sections and keep a conversation's prompt cache warm.

Sections

A **section** is a named piece of the system prompt. `section(key, render)` builds one. `render` returns a string, or `undefined` to leave the section out. By default the text is wrapped in tags, `<key>...</key>; { tag: false }` sends it as is. The selected extensions' sections and the agent's `instructions` are the only source of prompt text [spec §7.4](#).

A render function gets the resolved `agent` (including the tools this request offers), the `env` for this request (a `cwd` section renders `input.env?.cwd`), and `read` for committed document reads. Sections come in extension order, a later same-key section replaces an earlier one in place, `wrapSection()` decorators apply, and `instructions` come last. Example 15 shows all of this on two conversations.



Sections render in extension order, wrappers apply, deselected extensions drop out, and instructions come last. Rendered values from `research/runs/15-system-prompt.txt`; line breaks inside each tag are shown as spaces.

A section that throws does not fail the request. It keeps the text it last showed, if any, and the error is reported. Sections should not care where they run: “no section checks whether it runs in a subagent”. A conversation's prompt differs only through its selection, its `instructions`, its environment and its documents [spec §7.4](#).

Only the difference is sent

The transcript is the record of what the model saw, so the prompt lives there too. Before each request, the generation task replays the earlier system entries to learn what the model already has. It renders the sections it wants now and, if they differ, appends one `pi.system` entry with just the changes. The entry holds a `sections` patch plus `toolsAdded` and `toolsRemoved` (entries and the built-in kinds (p. 37)).

the first request's baseline research/capture/sqlite-rows.txt, entry 10

JSON

```
{
  "model": [ {
    "role": "system",
    "content": "",
    "sections": { "preamble": "You are a terse assistant." },
    "toolsAdded": [
      { "name": "count", "description": "Count from 1 to n, ...", "parameters": { ... } } ],
    "timestamp": 1791211974788
  } ],
  "kind": "pi.system",
  "id": 10,
  "conversationId": 1,
  "byTaskId": 9
}
```

The first request of a conversation writes the full baseline: one untagged `preamble` and the count tool (declaration elided). It follows user entry 7, the message it answers. This capture's preamble differs from example 15's, which the two figures in this section use.

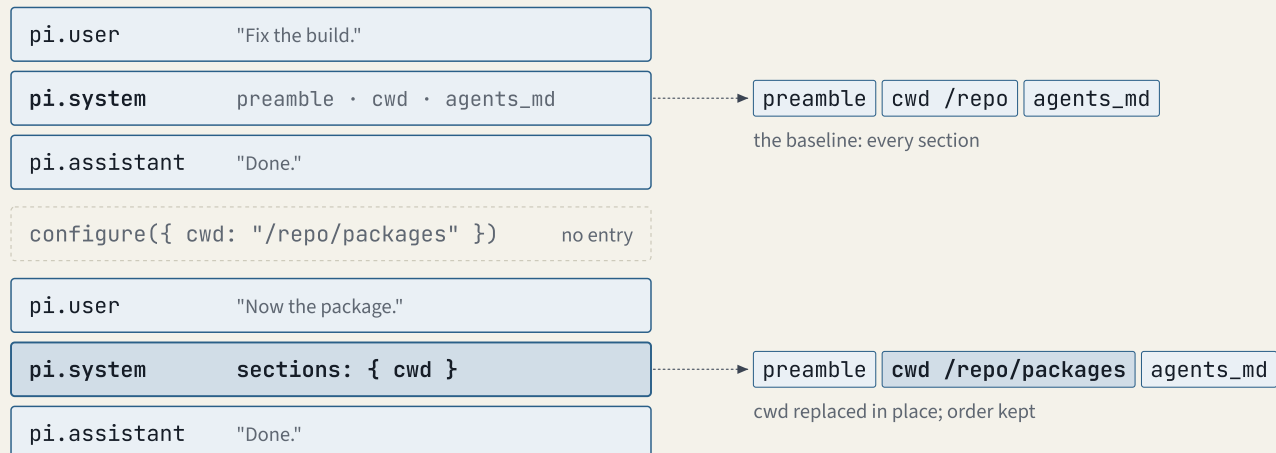
Replay applies system entries in transcript order. A string adds or replaces a section in place, `null` removes it. So after a `cwd` change, the next request appends a one-key patch, and the rest of the prompt keeps its place and its cache.

FIG. 7.7 THE DELTA

STRUCTURE

ROOT TRANSCRIPT, IN ORDER

PROMPT AFTER REPLAY



A later request appends only the section that changed, at its place in the transcript. The root conversation of ex 15; the second `pi.system` entry is { sections: { cwd: '<cwd>\n/repo/packages\n</cwd>' } } in research/runs/15-system-prompt.txt.

Example 27 shows tools changing the same way. Leaving plan mode appends sections: { plan_mode: null }, adds write, edit and bash, and removes submit_plan, while read keeps its place run 27.

Details

After a compaction or reset — the old baseline may lie outside the active context, so the next request writes a fresh full baseline, even if nothing changed. The capture kit saw `sections: {}` written this way for a conversation with no sections

`research/capture/NOTES.md`, note 18.

Reordering — a patch cannot move a section. When the desired order differs, the Harness writes two entries: one removing every section, one adding them all back in order `src/harness/prompt.ts`.

Missing tools — a request offers only tools that resolve. A tool whose extension disappeared is listed in `toolsRemoved`, and added back once it resolves again `spec §2.2`.

History is authoritative — rendered text stored in old entries stays as it was, even if the renderer changes.

WHAT THIS MEANS FOR YOU

- Keep renderers deterministic: an embedded timestamp changes the text, appends a delta and breaks the provider's cache `spec §7.4`.
- Give long-lived conversations a stable extension list; the host default moves whenever you install something `spec §12`.
- Add tools later through a filter or `control.addTools`: that appends one small delta and keeps the earlier prefix intact.

Sources: `src/harness/prompt.ts`; `src/harness/define.ts` `section()`, `wrapSection()`; `src/harness/generation.ts`; `src/harness/types.ts` (`PromptInput`, `PromptSection`); `spec §7.4`, `§2.2`, `§12`; `README §System Prompt`; `ex 15`, `27` and `runs 15`, `27`; `research/capture/sqlite-rows.txt`; `research/capture/NOTES.md`

7.6

Reload while running

Installing a new version under the same name swaps the code for all future work at once; work already running finishes on the code it started with.

You fix a bug in a tool while an agent is mid-turn. You want the fix live now, without killing the turn, and without a half-old, half-new tool call. After this section you can reload an extension in a running Harness, say which work sees the old or new code, and avoid the two reload bugs the registry cannot catch.

Reload is an install

There is no reload API. You install the new extension object under the same name. It replaces the old one in place, and “the Harness keeps running; no close or reopen is required” [spec §7.5](#). Because the position is kept, the extension’s tools, hooks and sections keep their precedence. Example 31 reloads a tool while a call to it is running, then restarts the process.

reload mid-call, then restart ex 31 (abridged)

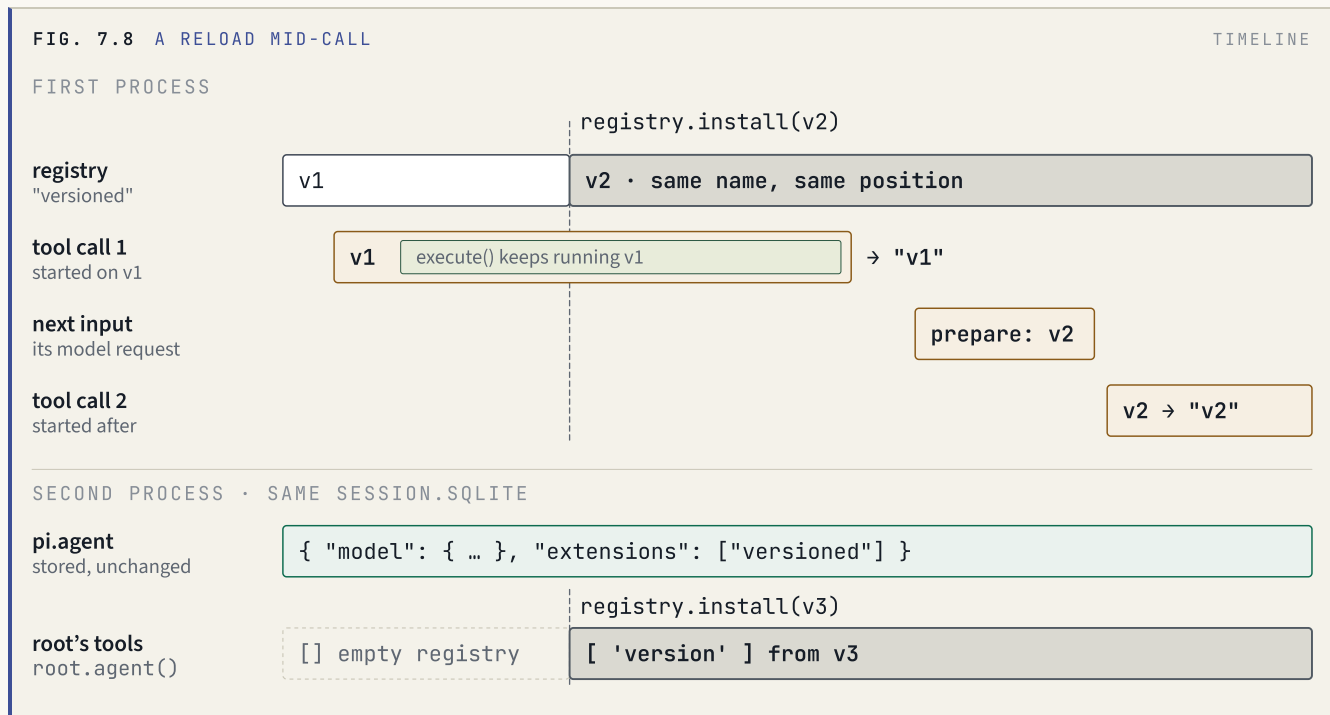
TS

```
const submission = await root.submit({ type: "input", content: "Which version?" }, context);
await running;                                     // execute() of v1 has started
registry.install(loadVersioned("v2"));              // same name "versioned"
release();
await submission.wait(context);                     // call running during the reload: v1
await (await root.submit({ type: "input", content: "And now?" }, context)).wait(context);
                                                    // next call: v2

await harness.close(context);

registry = createRegistry();                        // second process: nothing installed
harness = await open(registry);
root = await harness.root(context);
// after restart, before install: []
registry.install(loadVersioned("v3"));
// after install: [ 'version' ]
```

The comments are `research/runs/31-reload-and-restart.txt`. Both processes use one SQLite file; the conversation selected the extension by name.



Running work keeps the code it took; the next request or call uses the replacement. Example 31: the call started under v1 and returns “v1”; the next request is prepared under v2 and its call returns “v2”. After the restart the stored selection is unchanged and binds to v3. Dashed lines mark the installs.

Who sees which code

Three rules decide it:

A phase keeps its code — each task **phase** (one handler run between checkpoints) works from a registry snapshot taken when it starts. Its hooks and agent stay fixed until it ends `src/harness/scheduler.ts`.

The next phase sees the new code — each new phase takes a fresh snapshot. One run may use different versions across its phases: “nothing requires one run to see a single registry state” `spec §7.1`.

A tool call is pinned until it finishes — the call finds its tool, runs its hooks, executes and writes its result in one phase, “so nothing separates resolution from settlement” `src/harness/tool.ts`.

The tests pin both halves. A tool replaced mid-call still returns “v1”. A section and a `beforeRequest` hook reloaded while a tool runs show `mode: v1` in the first request and `mode: v2` in the second `test/harness-tools.test.ts`. Replacing code “never signals or interrupts” running work `spec §7.5`.

Task definitions hand over

Your own task definitions reload by name too. When a running task finishes a phase and a different definition is now installed for its kind, the task is handed to the new code if it can take it: the same version, or a newer one with a `migrate` function. Otherwise the old code keeps running it and the mismatch is reported. Versioning and migration are in [the scheduler](#) (p. 73).

Installing also unblocks. A task left blocked because its extension was missing at open, or because its stored version was too new, runs as soon as a fitting definition is installed. `harness.inspect()` shows which tasks are blocked ([the task graph and inspect\(\)](#) (p. 137)).

What a reload does not touch

Nothing on disk changes. A reload commits nothing, emits no event and rewrites no transcript. A conversation that uses the extension gets a system delta at its next request, but only if the rendered sections or tool declarations differ (the system prompt (p. 115)).

A restart is a reload from an empty registry. After the second process opens the file, the root's agent has no tools until the extension is installed again, then it has the new code's `version` tool `run` 31. Pending tasks from the extension resume once it is installed. Never open the new Harness before the old one's `close()` has resolved: two instances must not own one Session at once.

TWO FOOTGUNS

Early cleanup: replacing an extension “only stops new use”. Closing a database pool or a child process right after `install()` breaks calls still running on the old object. **Moving tasks:** moving a task definition from one extension to another “leaves a gap between the two installs in which its pending tasks are blocked” `spec` §7.5, §12.

WHAT THIS MEANS FOR YOU

- Reload by installing under the same name; there is nothing else to call.
- Bump a task's version when the meaning of its stored input or checkpoints changes.
- If old code ignores cancellation and never finishes, `close()` still waits for it. Only running extensions in a separate worker or process lets you kill them `spec` §7.5, §13.

Sources: `src/harness/registry.ts`; `src/harness/scheduler.ts`; `src/harness/tool.ts`; `spec` §7.1, §7.5, §12, §13; README §Reload; ex 10, 31 and runs 10, 31; `test/harness-tools.test.ts`

7.7

Subagents

A subagent is an ordinary conversation owned by the work that created it; ownership and request IDs make it abortable and crash-safe without a subagent API.

A subagent is a second agent, with its own transcript, that your agent hands a job to: “research this”, “review that diff”. You build one from conversations, ownership and a tool. After this section you can write the two patterns the package ships, and predict what an abort, an idle wait and a crash do to each.

The foreground pattern

A foreground subagent lives inside one tool call. The tool creates a child conversation owned by its own tool task, sends it the job, waits for the answer and returns it. Each durable action is keyed, so a rerun after a crash finds what the first attempt made.

the subagent tool ex 22 (abridged)

TS

```
defineTool({
  name: "subagent", replay: "safe",
  parameters: Type.Object({ task: Type.String() }),
  execute: async (args, api, ctx) => {
    const child = await api.commit(async (tx) => {
      const found = await tx.scanConversations({ ownerId: api.taskId }, 1);
      const existing = found.items[0];
      if (existing !== undefined) return existing.id;
      const created = await tx.createConversation({
        ownership: { kind: "task", taskId: api.taskId },
      });
      await configure(tx, created.id, { extensions: { remove: [Subagent] } });
      return created.id;
    }, ctx);
    await api.details({ conversationId: child }, ctx);
    const handle = (await api.conversation(child, ctx))!;
    const request = { type: "input", content: args.task,
      requestId: `subagent:${api.taskId}` } as const;
    const settled = await (await handle.submit(request, ctx)).wait(ctx);
    // ... throw unless settled is an answered input (elided)
    const text = await answerText(api, settled.answer, ctx);
    return { content: [{ type: "text", text }] };
  },
})
```

Run 22 prints the child’s assistant: 2, 3, and 5. and the parent’s The subagent says: 2, 3, and 5.

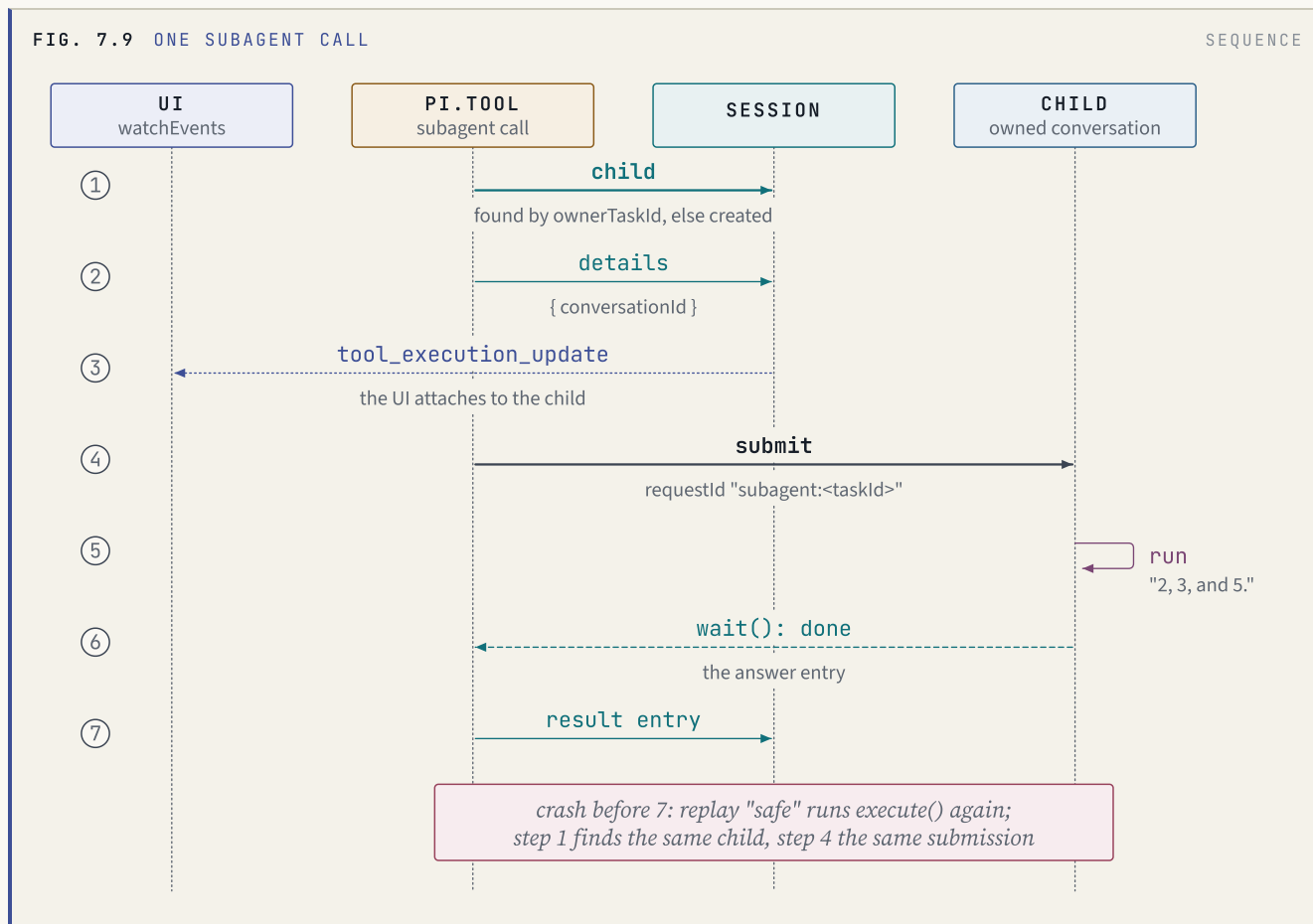
Each part answers a crash question:

Find before create — the child is owned by the tool’s task, so a rerun finds it with `scanConversations({ ownerId })` instead of making a second one.

One commit creates and configures — the child starts with a copy of the parent’s agent ([copies at creation \(p. 106\)](#)); the same commit removes the subagent extension so the child cannot start subagents itself.

A request ID from the task — on a rerun, `submit()` with the same ID returns the first submission instead of sending the job twice. Never use the provider’s call ID: it “is not unique across a Session” [spec §7.3](#).

Details point the UI at the child — the UI sees `{ conversationId }` in a `tool_execution_update` event and attaches to the child’s events ([agent events \(p. 133\)](#)).

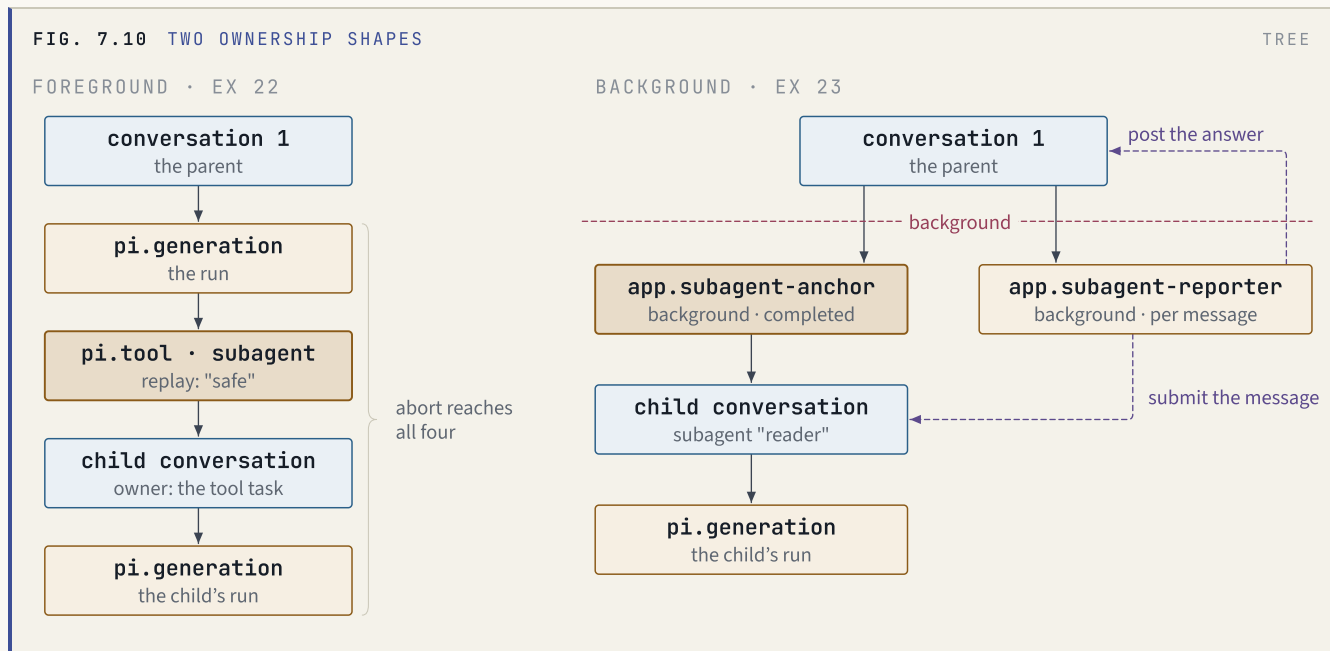


A replay-safe subagent call finds its child by owner and its submission by request ID. The steps of ex 22’s tool and UI. Step 1 is one commit; step 4 submits the job, step 5 is the child’s own run, and step 6 is the tool’s `wait()`.

The handle from `api.conversation()` works only while the call runs and accepts only input. Never use it inside a `commit()` callback: Session operations cannot nest (the commit and the mutation line ([p. 29](#))).

Ownership decides abort and idle

Owning the child makes it part of the call’s work. “Aborting the call aborts the child”, and so does the call failing; “the parent is idle only once the child is” [README §Abort and Subagents](#). A tool task that owns running work stays open until that work ends (ownership and structured concurrency ([p. 76](#)), [abort, fault and orphan \(p. 80\)](#)).



A foreground child belongs to the call; a background child hangs off a background anchor. Arrows point from owner to owned. Left: ex 22. Right: ex 23, where the anchor completes at once and a reporter task delivers each message to the child and posts the answer back to the parent. The dashed line is the background boundary: the parent's Esc and idle waits stop there.

The background pattern

A background subagent outlives the parent's turns: the user can message it, stop it and list it later. Example 23 puts this behind one subagent tool with the actions `spawn`, `send`, `stop` and `status`. A `spawn` commits three things at once. An **anchor** task, `app.subagent-anchor`, is a background task of the parent conversation; it completes at once and stays open while the child works. The child conversation is owned by the anchor, without the subagent extension and with its own `instructions`. The parent's `app.subagents` document maps the subagent's name to the child; a fork of the parent starts with it empty.

Each message travels through a background **reporter** task, `app.subagent-reporter`. It submits the message to the child, waits, and posts the answer to the parent as a follow-up input. Both submissions use request IDs derived from the reporter's ID, "so the child gets the message once and the parent gets the answer once" [spec §7.3](#). Run 23 closes the Harness while the subagent works; after the restart the parent still receives `reader: Moby Dick`.

The background flag is a wall. The parent's Esc and idle waits stop at the anchor, so they leave subagents alone. `abort(context, { background: true })` still reaches through. The `stop` action aborts the child directly: its current answer ends, queued messages are dropped, and the subagent stays usable, as any finished subagent does [spec §5.4](#).

WHAT THIS MEANS FOR YOU

- Key every durable action: find the child by owner, and derive request IDs from a task ID.
- Pick foreground for a job inside one answer, background for an agent the user talks to later.
- Never report the child's spend as the tool's usage; it would count twice ([usage and cost \(p. 100\)](#)).

Sources: [spec §7.3](#), [§5.4](#), [§12](#); [README §Abort and Subagents](#); [ex 22](#), [23](#) and [runs 22](#), [23](#); [src/harness/harness.ts](#); [src/harness/agent.ts](#); [src/harness/tool.ts](#)

Watching

Every screen is a function of committed state: views, watches, agent events and the task graph.

8.1	Chord and document state	125
8.2	The conversation view and watch()	129
8.3	Agent events	133
8.4	The task graph and inspect()	137

08

Chord and document state

An observer sees a document as a series of read-only values, each with the exact edits that produced it, and never sees anything storage does not hold.

A panel that shows a document must stay in step with it. When a tool updates a task list, the panel should change. After a crash, it should show exactly what was committed. A *commit* is an atomic save: everything in it is stored together, or nothing is. Pi Durable gives observers only committed changes: each one is a new read-only value plus the short list of edits that produced it. After this section you can attach a document to a UI and say what it shows when the document is retired, migrated, or the process dies.

What an observer receives

Documents are built on Chord ([@earendil-works/chord](https://github.com/earendil-works/chord)), a small library for replicated JSON state. You need two of its ideas to observe a document ([bases and deltas](#) (p. 56)):

Revision — an immutable JSON value. Every commit that changes the document produces a new one.

Operations — the exact edits from one revision to the next, written as small tuples. A copy elsewhere applies them to stay in step.

Observers only read. Pi Durable is the only writer, and an observed state has no methods that change it [spec §9.1](#). Each operation is a tuple: a one-letter name, then a path of keys and array indices into the value.

TUPLE	MEANING
["r", value]	Replace the whole value
["s", path, value]	Set a property or array element
["d", path]	Delete a property or array element
["a", path, text]	Append to a string
["t", path, count]	Remove count UTF-16 code units from a string's front
["p", path, index, remove, items]	Splice an array
["m", path, permutation]	Reorder an array: <code>new[i] = old[permutation[i]]</code>

From [research/pi/packages/chord/src/delta/README.md](https://research.pi/packages/chord/src/delta/README.md). The same change can be written with different tuples, and a large edit may arrive as one s or r. Only the resulting value is guaranteed.

Two ways to observe one document

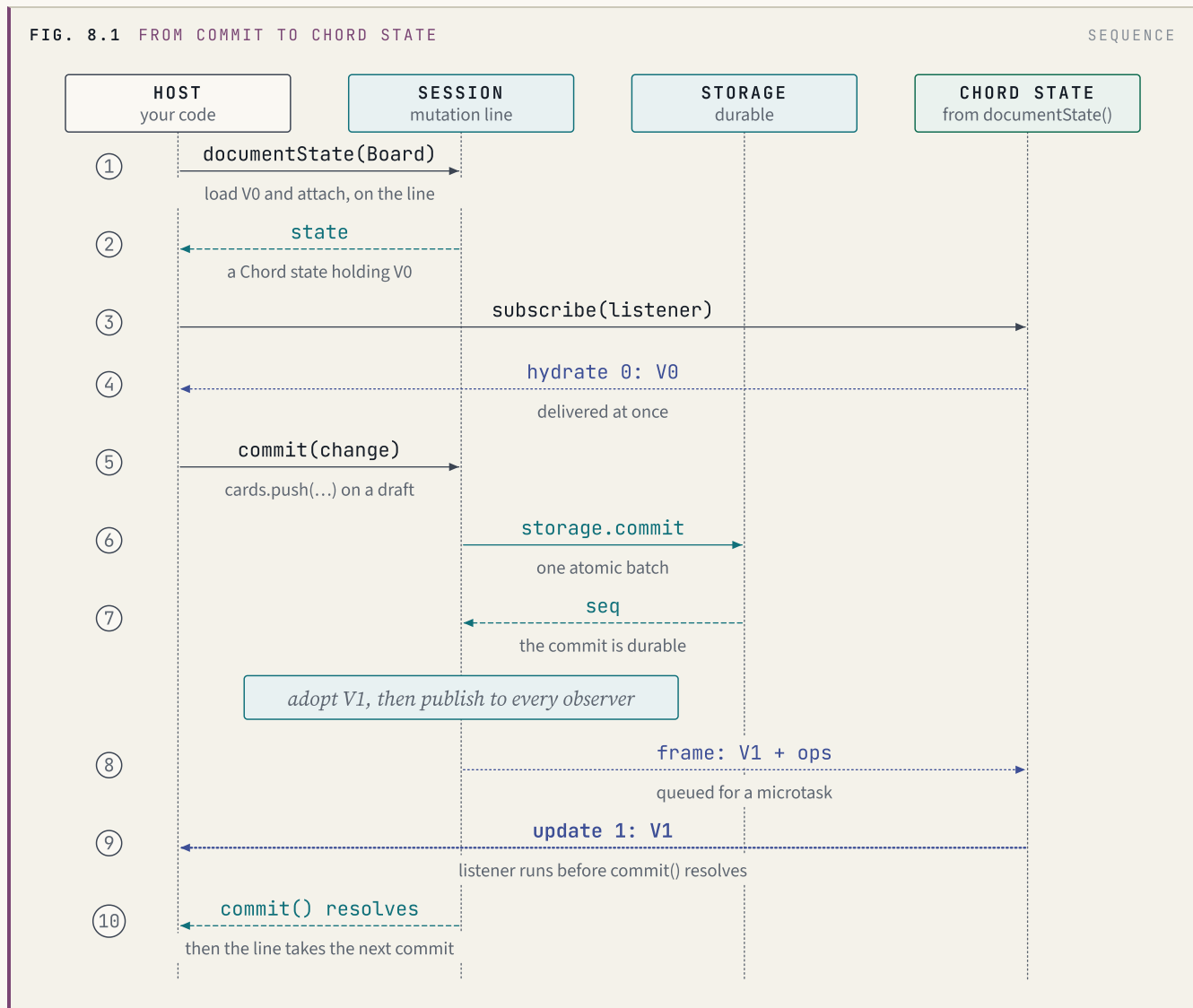
DOCUMENTSTATE()

A read-only state you subscribe to. It starts with the current value and calls your listener after each commit. Use it to drive a UI.

WATCHDOC()

A stream of frames, one per commit: the new value, its operations and the commit's `Context`. Your callback handles one frame at a time. Use it to forward changes in order ([the watch contract](#) (p. 129)).

Both take the same arguments as `snapshot()`, and two rules hold for both. Observing never creates: an absent document returns `undefined`; only `tx.doc()` creates one ([access and creation](#) (p. 53)). And an observer follows one *incarnation*, the stored copy it attached to. If that document is retired and created again, the observer does not follow it.



A document state sees a commit only after storage holds it; your listener runs in a microtask before `commit()` resolves. Step 1 runs on the Session's mutation line, so no commit can slip between the first value and the attachment; `subscribe()` then delivers that value at once as `hydrate 0`. The Board document is the one captured in `research/capture/extra-p8/doc-observation.txt` (next figure). `src/session/session.ts`, `src/session/observation.ts`, `chord/src/services/state.ts`.

This is the book's core rule seen from the read side. Nothing reaches an observer until storage holds the commit. Your listener runs in a microtask after the commit's work on the line, before `commit()` resolves. Only the asynchronous part of a listener runs off the line: a slow synchronous listener does delay the next commit, so hand long work to a promise `spec §1`. Attaching waits behind earlier commits, so the first value already includes all of them. You never receive operations for a revision you have not seen `spec §9.1`.

documentState() in use

Example 4 creates a conversation document with the text "first", attaches a state and commits one change.

a document state, from first value to update `test/examples/04-chord-state.ts`

TS

```
const notesState = await session.documentState(Notes, chat.id, context);
const stopNotes = notesState.subscribe((value, _deliveryContext, delivery) => {
  console.log("Chord notes:", delivery.kind, delivery.sequence, value);
});
await session.commit(async (tx) => {
  (await tx.doc(Notes, chat.id)).text = "published through Chord";
}, context);
```

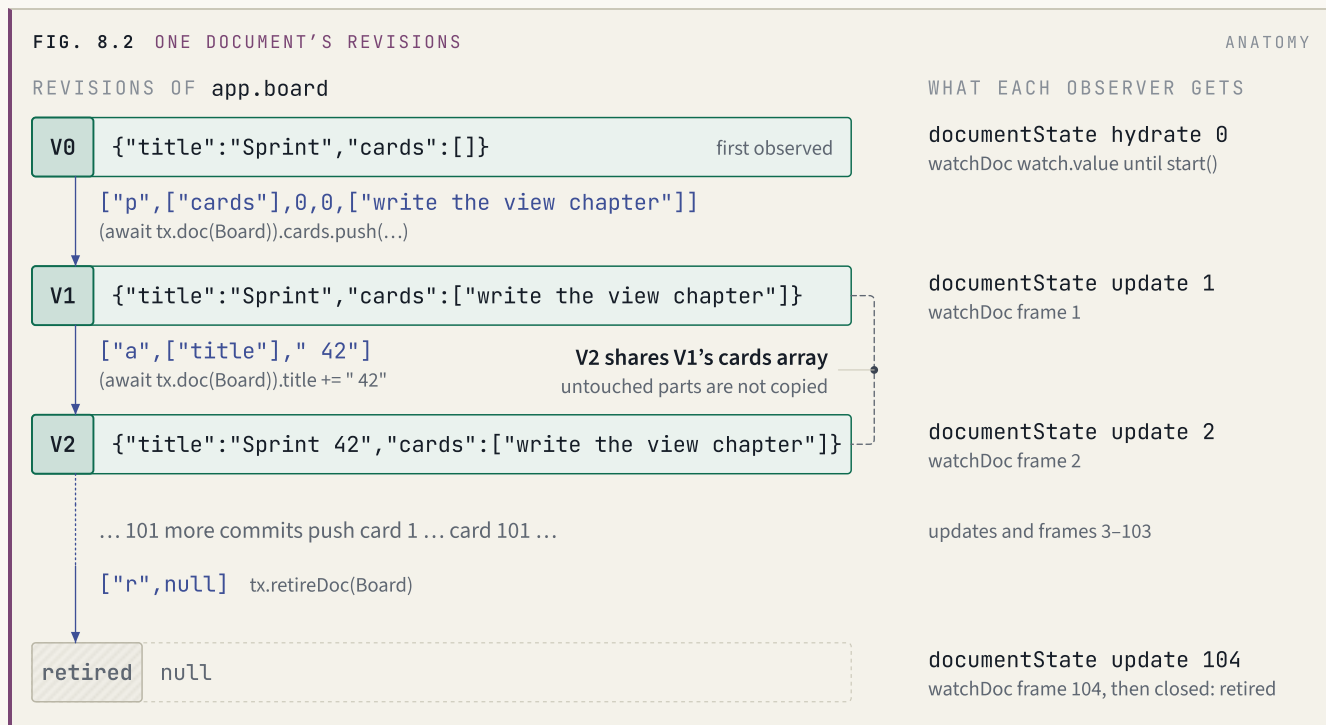
```
$ node --conditions=source --experimental-strip-types test/examples/04-chord-state.ts
Chord notes: hydrate 0 { text: 'first' }
Chord notes: update 1 { text: 'published through Chord' }
```

The `Notes` definition, the creating commit, the `undefined` check and the final `stopNotes()` and `dispose()` are elided. `subscribe` delivers the current value at once as `hydrate`, then one `update` per commit. Output from run 04.

`delivery.sequence` counts this state's own deliveries from 0. It is not a commit number, and nothing stores it [spec §9.1](#). Two states on one document are independent. Disposing one stops only that observer, never the document.

Revisions are immutable and shared

Each value you receive is a revision the Session will never change. Parts a commit did not touch are shared with the previous revision, not copied. In the capture behind the next figure, changing the title produced a new root, but `V2.cards` is the same array object as `V1.cards`.



Each commit yields a new immutable revision and the exact operations that produced it. A session-scoped `app.board` observed through `documentState()` and `watchDoc()` at once; 101 further commits come before the retirement. Values, operations, sequences and the sharing check are from `research/capture/extra-p8/doc-observation.txt`.

Sharing makes publication cheap, but nothing is frozen. If your code mutates a value it received, it silently corrupts the Session's copy and every other observer's. Copy first when you need to edit `spec §12`.

To keep a copy in another process, start from the first value and apply each batch of operations in order with Chord's `applyImmutable(value, ops)`. The test suite checks that this rebuilds every delivered value `test/session-watches.test.ts`.

Retirement, versions, close and crashes

Retired — the last frame is `[["r", null]]`. A state then reads `null`, never stale data. A watch delivers that frame and ends with reason `retired`. To see a recreated document, attach again.

New version — when a commit stores the document under a different definition version, the observer receives the whole new value as `["r", value]` instead of operations on a shape it does not have `src/session/session.ts`.

Session close — every state and watch ends. A state keeps its last value `spec §2.2`.

Crash — states and watches live in memory and die with the process. After reopening, attach again; the new state starts from what was committed `spec §9.1`.

A UI never loses anything it displayed in a crash, because it only ever displayed committed data. Progress since the last successful commit can vanish. The default 100 ms minimum interval does not bound that loss (the live document (p. 88)).

WHAT THIS MEANS FOR YOU

- Drive a UI from `documentState()`. It shows only committed data, so it is correct after a crash without extra work.
- Never mutate a value you received. Copy it first.
- Treat `["r", ...]` as “replace everything”. It arrives on retirement, version change and overflow.
- Attach again after a retirement or a reopen; observers never move to a new incarnation. To serve remote clients through a Chord service, follow `docs/pico-v5-chord-usage.md`.

Sources: `spec §1, §2.2, §9.1, §12`; `src/session/observation.ts`; `src/session/session.ts` (`documentState`, `watchDoc`, `observedOperations`); `src/types.ts` (`DocumentState`); `docs/pico-v5-chord-usage.md`; `research/pi/packages/chord/src/delta/README.md`; `test/session-states.test.ts`; `test/session-watches.test.ts`; `ex 04`; `run 04`; `research/capture/extra-p8/doc-observation.txt` (script `test/capture-p8-observe.ts`)

8.2

The conversation view and watch()

One value holds a conversation's transcript and what is running now; it changes only when a commit lands, and you can read it as a state or as a stream of frames.

A chat UI needs two things: the transcript, and what is happening right now, such as a model streaming text or a tool printing output. The conversation *view* puts both in one read-only value that changes only when a commit lands. After this section you can render it, keep a remote copy in step, and predict what a slow or late client sees.

What the view holds

The view is a plain bundle of stored data: nothing is computed or filtered. This is the view of the capture's conversation before its run began.

```
watch.value at attachment research/capture/watch-ops.txt JSON
{
  "conversation": { "id": 1 },
  "entries": [],
  "docs": {
    "pi.agent": { "model": { "provider": "faux", "modelId": "faux-1" } },
    "pi.live": {},
    "pi.inbox": { "items": [] },
    "pi.provider": { "sessionId": "01a10c8d-c25d-72c3-83b8-8dc6cf418d86" },
    "pi.usage": { "models": {}, "tools": {} }
  }
}
```

Each document is folded onto one line. `pi.live` is `{}` because nothing runs yet.

conversation — the conversation record, with its `parent` and `owner` when it has them ([records and IDs \(p. 25\)](#)).

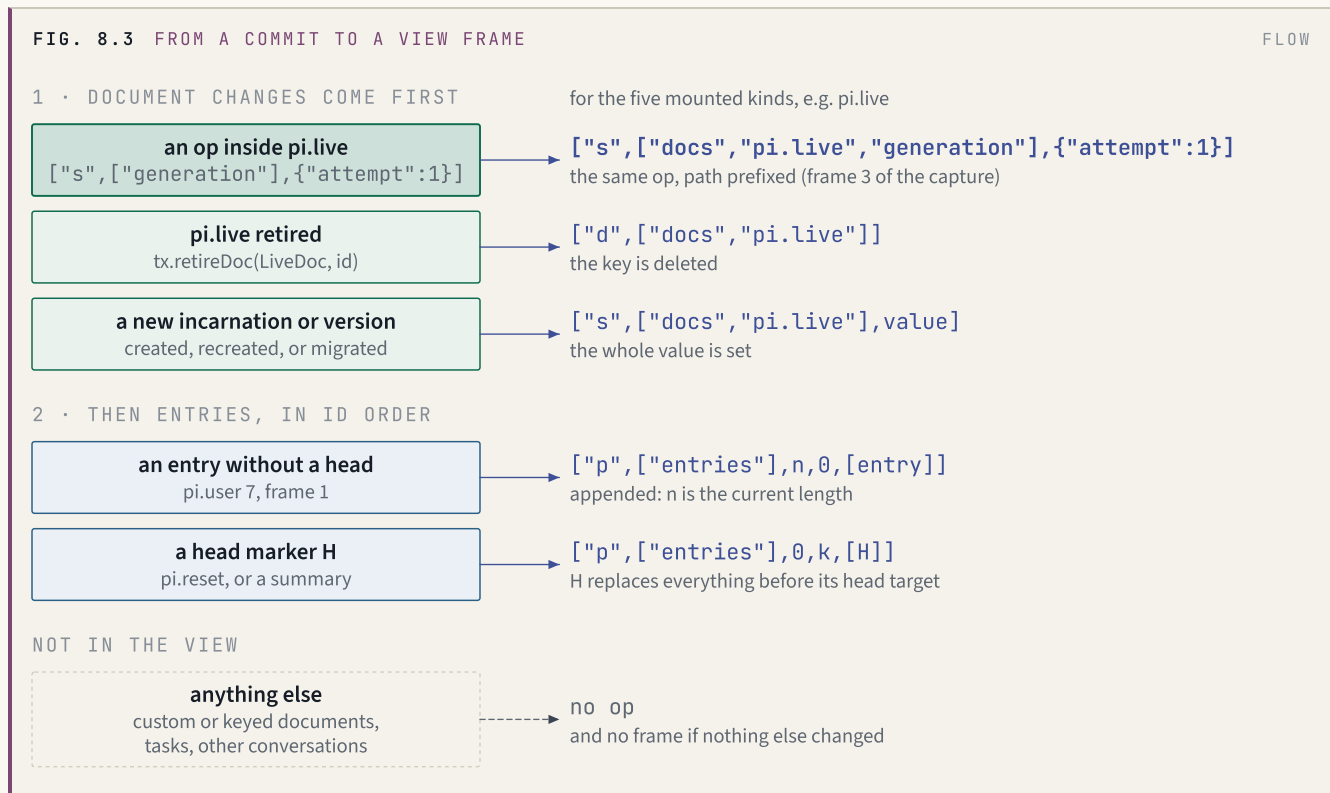
entries — the active transcript as stored: the newest head marker, then the non-head entries from its head target through the current tail. These are records, not the model's context ([from transcript to model context \(p. 40\)](#)).

docs — the five built-in documents, keyed by kind ([the built-in documents \(p. 60\)](#)). An absent document is an absent key.

These kinds and their fields are public protocol. Your own document kinds are not in the view; observe them with `documentState()` [spec §9.3](#).

How the view changes

The Harness keeps one shared view per conversation. It is built from storage on first use and dropped when its last observer leaves `src/harness/view.ts`. Every commit that touches the conversation produces one *frame*: a batch of operations that turns the old view into the new one.



A view frame is the commit's document changes, then its new entries, rewritten as paths into one value. From `advance()` in `src/harness/view.ts`. The first row is frame 3 of `research/capture/watch-ops.txt`; frame 1 shows the order, a `pi.live` set before the splice of entry 7.

Here is the frame that settles a tool. Five operations land together: the slot's status and entry, two deletions of its progress, and the result entry spliced in.

```

frame 14: one commit settles a tool  research/capture/watch-ops.txt  JSON

[
  ["s", ["docs", "pi.live", "tools", 0, "status"], "done"],
  ["s", ["docs", "pi.live", "tools", 0, "entry"], 13],
  ["d", ["docs", "pi.live", "tools", 0, "output"]],
  ["d", ["docs", "pi.live", "tools", 0, "details"]],
  ["p", ["entries"], 3, 0, [{ "kind": "pi.tool-result", "id": 13, ... }]]
]

```

The entry is elided. The slot points at entry 13 in the same frame that inserts it, so a renderer never sees a dangling reference.

Apply a frame's operations in order; they are not minimal. Parts of the view a frame does not touch stay the same objects. A frame that only appends an entry keeps the same `docs` object, so a renderer can compare references to skip work `test/harness-view.test.ts`.

Two ways to consume it

VIEWSTATE()

A Chord state, like `documentState()`. Use it in the same process when you only want the latest value, for example a terminal UI that re-renders on each update.

WATCH()

A stream of frames. Use it to keep a copy across a process boundary: send `watch.value` once, then each frame's operations in order. The other side applies them with `applyImmutable()`.

On reconnect, start a new watch and send its value again. Example 19 with `--ops` prints exactly this stream: the view first, then one line of operations per commit.

The watch contract

`watch()`, `watchDoc()` and `watchTaskGraph()` all return a `WatchHandle` with the same five rules.

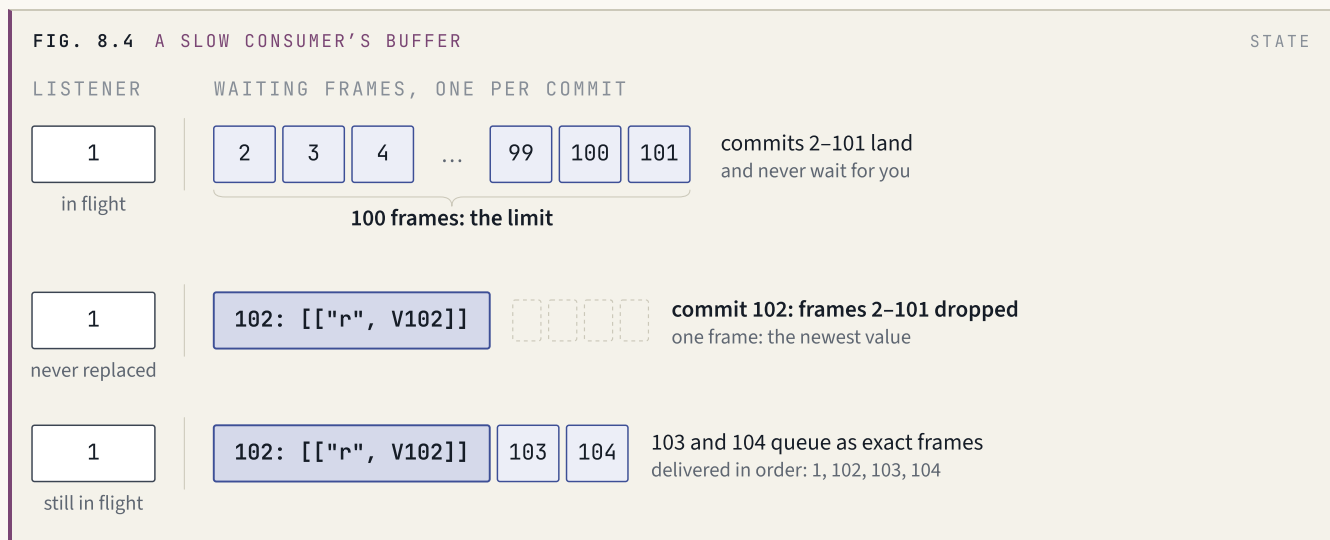
1. **Read `watch.value` first.** It holds the starting value. Frames committed before `start()` wait in a buffer.
2. **Call `start(listener)` once.** It never calls the listener synchronously. A second `start()`, or one after the watch ended, throws.
3. **One callback at a time.** Each receives (`value`, `ops`, `context`). The next starts after your promise settles. Commits never wait for you.
4. **The context is the commit's**, without its abort signal. Work your listener starts is yours to abort.
5. **`stop()` is safe to repeat.** It drops pending frames and returns how the watch ended. A callback already running is not aborted.

WATCHEND.REASON	CAUSE
stopped	<code>stop()</code> was called
cancelled	The Context passed when attaching was aborted
session_closed	The Session began closing
retired	The null frame of a retired document was delivered
listener_error	Your listener threw; carries <code>error</code> , and only this watch ends

`src/types.ts` (`WatchEnd`). The first reason wins. A watch opened inside a task or a tool also stops when that invocation ends (`src/harness/scheduler.ts`).

Slow and late consumers

A watch never makes commits wait. If your listener falls more than 100 frames behind, the waiting frames are thrown away and replaced by one frame holding the newest value.



A watch keeps 100 undelivered frames; the next commit replaces them all with the newest value. The frame being delivered is never replaced. Rule from spec §9.2 and `src/session/observation.ts`. A test commits 101 entries before `start()` and receives one frame, `[\"r\", view]`, holding all 101 entries.

The replacement is `[["r", newest]]`, so code that applies operations needs no special case. The cost is that some intermediate states are never delivered. A watch shows where things ended up, not every step. If you need every transition, record each one as an entry or in a document. For the outcome of one input, use `Submission.wait()` ([submissions and the inbox](#) (p. 84)).

A client that joins late, or reconnects, starts from the current view. Nothing is replayed. Example 21 attaches halfway through a tool that prints a line every 100 ms:

```
$ node --conditions=source --experimental-strip-types test/examples/21-late-join.ts
view entries: [ 'pi.user', 'pi.system', 'pi.assistant' ]
view tool slot: running "1\n2\n3\n4\n5\n"
view output now: "1\n2\n3\n4\n5\n"
snapshot tools: [ 'count running' ]
view output now: "1\n2\n3\n4\n5\n6\n"
event output now: "1\n2\n3\n4\n5\n6\n"
```

First lines of `research/runs/21-late-join.txt`. The view already holds the committed output; the event stream of [8.3](#) (p. 133) starts from the same state.

The late client sees five lines because each one was committed. Tool progress has a default minimum commit interval of 100 ms (`settings.progress`). A crash loses progress since the last successful commit; the gap can be longer ([run control and the live document](#) (p. 88)).

WHAT THIS MEANS FOR YOU

- Same process, latest value only: `viewState()`. Remote copy: `watch()` plus `applyImmutable()`.
- Handle `["r", value]` as a full replace at any point in the stream.
- Do not use a watch as an audit log. Use entries, a document, or `Submission.wait()`.
- On reconnect, attach again and resend `watch.value`.

Sources: spec §2.2, §9.2, §9.3, §12; README §Watching a Conversation; `src/harness/view.ts`; `src/session/observation.ts`; `src/types.ts` (`WatchHandle`, `WatchEnd`); `src/harness/scheduler.ts`; `test/harness-view.test.ts`; `test/session-watches.test.ts`; ex 19; ex 21; run 21; `research/capture/watch-ops.txt`; `research/capture/NOTES.md` (note 14)

Agent events

An experimental adapter turns each commit into one batch of coding-agent events: a snapshot is the state, and every batch is a change to it.

You may already have code that reads the coding agent’s events, such as `message_start` and `tool_execution_update`: a JSON log, a CLI, a UI. `watchEvents()` produces those events from Pi Durable’s commits. After this section you can consume the stream, rebuild streamed text from its deltas, and know which steps it may skip.

Attaching

`watchEvents(harness, conversationId, context)` is exported from the package root and marked experimental: its protocol may change. It returns a stream with a `snapshot` (the starting state), `start(listener)`, `stop()` and `closed`. It follows the watch rules of 8.2 (p. 129): one batch at a time, and commits never wait `src/harness/events.ts`.

The snapshot holds the conversation’s `entries`, running `tools`, `compactions`, `inbox`, `agent` and `usage`. While a run is in flight it also holds `run` and `generation`, the streaming message or a retry wait `research/capture/agent-events.txt`.

An event is emitted only after the commit that makes it true `spec §9.4`. The stream covers one conversation. A subagent’s work shows up only as the parent’s tool call; attach a second stream to the child conversation to see inside it (`subagents` (p. 121)).

One commit, one batch

Each commit produces at most one batch of events. The table follows one turn of the capture: the model thinks, calls `count {n: 3}`, then answers. A *generation* is one model call, run as a task.

COMMIT	WHAT CHANGED	EVENT BATCH
1	user message committed; the run starts	<code>message_start</code> , <code>message_end</code> , <code>submission</code> , <code>run_start</code> , <code>turn_start</code>
2	system prompt entry	<code>message_start</code> , <code>message_end</code>
3	attempt 1 of the generation begins	none
4	first streamed partial	<code>message_start</code>
5–7	thinking and text stream in	<code>message_update</code> each
8	assistant message committed; tool call queued	<code>message_end</code> , <code>usage_changed</code>
9	the tool starts running	<code>tool_execution_start</code>
10–13	tool output and details	<code>tool_execution_update</code> each
14	tool done; its result entry committed	<code>tool_execution_end</code> , <code>message_start</code> , <code>message_end</code>
15	the next generation replaces the first	<code>turn_end</code> , <code>turn_start</code>
16	attempt 1 begins	none
17–19	the answer streams in	<code>message_start</code> , <code>message_update</code> ×2
20	answer committed; the run ends	<code>message_end</code> , <code>turn_end</code> , <code>run_end</code> , <code>submission</code> , <code>usage_changed</code>

`research/capture/watch-ops.txt` (20 frames) and `agent-events.txt` (18 batches), one run of `test/capture/view-ops.ts`.

Commits 3 and 16 produce no batch: starting an attempt changes nothing an event describes. The tool’s start waits for commit 9, when the call actually runs. In commit 15 one generation ends and the next begins, so `turn_end` and `turn_start` share a batch.

Inside a batch the order is fixed and matches the coding agent. Progress comes first, then new entries, then ends, and beginnings come last. So a tool’s end comes right before its result message, and a finished run ends before the next one starts.



Inside one batch, progress comes first, then entries, then ends, and beginnings come last. Right: the batch a test expects from one commit that answers a run and starts the queued follow-up as the next run (test/harness-events.test.ts).

What each event means

EVENT	EMITTED WHEN THE COMMIT...
run_start / run_end	starts or ends a run. Steering input that joins a running run is not a new run
turn_start / turn_end	moves the run to a new generation / commits a generation’s outcome
message_start / message_end	commits an attempt’s first streamed partial, or a message entry that had none / appends an entry with model messages
message_update	changes the streamed partial; carries usage and changes
entry_appended	appends an entry with no model messages, such as a plain pi.reset
tool_execution_*	starts a tool call / changes its output, details or diagnostics / finishes it, or drops it with its run
submission	writes a submission record of this conversation
inbox_update, agent_changed, usage_changed	changes that document
auto_retry_start / _end, deferred_poll	starts or ends a retry wait; schedules or moves a deferred poll
task_failed	settles a task of the conversation faulted or orphaned
compaction_start / _end	starts or finishes a compaction

spec §9.4; src/harness/events.ts.

Every `message_start` gets a `message_end`, even when a run is aborted mid-stream. A partial is committed only once it has content, and every built-in path that clears it also appends the finished entry. The same holds across a crash: on reopen, generation turns a leftover partial into an aborted `pi.assistant` entry ([generation \(p. 91\)](#)), and the stream ends the message with it `spec §9.4; test/harness-events.test.ts`.

Deltas, not values

`message_update` carries only what changed in the streaming message. Appended text arrives as `text_delta` or `thinking_delta`, appended tool arguments as `toolcall_delta`, and a new content block as a `*_start`. Any other change resends the whole block or message. Ten commits of a partial cost ten small events, not ten copies `spec §9.4`.

the batch of commit 6 `research/capture/agent-events.txt`

JSON

```
{
  "type": "message_update",
  "usage": { "input": 56, "output": 19, ... },
  "changes": [
    { "type": "text_start", "contentIndex": 1, "block": { "type": "text", "text": "" } },
    { "type": "thinking_delta", "contentIndex": 0, "delta": "use the tool." }
  ]
}
```

Two view operations become two changes, in order. Usage fields are elided.

Tool output works the same way. `tool_execution_update.output` is `{ set }` when the output was replaced, or `{ trimStart?, append? }` when old text was trimmed from the front and new text appended. The capture shows `{"set": "1\n"}` for the first line and `{"append": "2\n"}` after. Example 21 rebuilds the output this way:

applying output updates `test/examples/21-late-join.ts`

TS

```
let output = stream.snapshot.tools[0]?.output ?? "";
stream.start(async (events) => {
  for (const event of events) {
    if (event.type === "tool_execution_update" && event.output !== undefined) {
      output = "set" in event.output
        ? event.output.set
        : output.slice(event.output.trimStart ?? 0) + (event.output.append ?? "");
    }
  }
});
```

Trimmed from the example. Removed `details` arrive as `null`, as when a replay-safe tool restarts ([6.4 \(p. 94\)](#)).

Falling behind

Like a watch, a stream holds at most 100 undelivered batches. On overflow they are replaced by one snapshot of the newest view, streaming partial included `test/harness-events.test.ts`.

WHAT THIS MEANS FOR YOU

- A **snapshot** can arrive at any time, not only first. Reset your state to it.
- Use events for animation and log lines; render the durable picture from the view `spec §9.4`. Example 19 prints a run with `--events`.
- Do not count on seeing every event; use `Submission.wait()` for outcomes.
- One stream covers one conversation. Attach to child conversations separately.

Sources: `spec §9.4`; `README §Agent Events (Experimental)`; `src/harness/events.ts`; `test/harness-events.test.ts`; `ex 19`; `ex 21`; `run 21`; `research/capture/watch-ops.txt`, `agent-events.txt`

8.4

The task graph and inspect()

The graph shows every live task as it is committed; inspect() adds what the scheduler would do with each one under the registry you have now.

When a run seems stuck, the cause is usually a task waiting on something the transcript does not show. Two read-only tools show you the tasks. `taskGraph()` is a live value for a task panel. `inspect()` is a one-time diagnosis that also says why a task cannot run. After this section you can go from “nothing happens” to the task and the reason.

A node

The graph is one value, `{ tasks }`, keyed by task ID. Each node is the stored task record minus its payloads: no input, checkpoint data, outcome data or memos.

a waiting task in the graph `research/capture/extra-p8/inspect-vs-graph.txt`

JSON

```
"8": {
  "id": 8,
  "kind": "app.join",
  "conversationId": 1,
  "background": false,
  "abortRequested": false,
  "state": { "status": "waiting", "phase": "done", "on": [7], "policy": "allSettled" },
  "conversations": []
}
```

No `owner` key: a conversation owns this task. A node owned by another task carries `"owner": <TaskId>`. `abortRequested` turns true in the commit that aborts the task.

STATE.STATUS	OTHER FIELDS	MEANING
pending	phase	Committed and not yet picked up; also every task that was running when the process died, until it runs again
running	phase	Picked up by the scheduler in this process
waiting	phase, on, policy	The stored wait, exactly as committed
completing	outcome	Finished, holding its outcome until its owned work ends

`src/harness/task-graph.ts`; spec §9.5. Statuses are defined in [tasks as state machines](#) (p. 63).

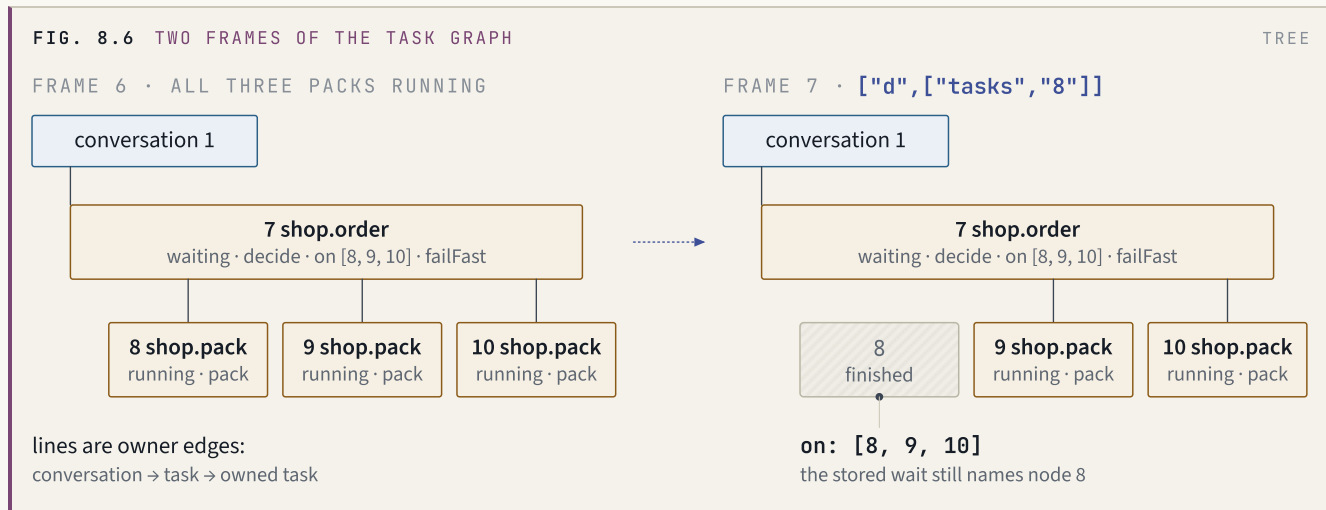
How the graph moves

A node appears in the commit that creates its task and disappears in the commit that finishes it. A commit that changes no node, such as a memo write, produces no frame. The graph shows committed facts, so read it with three rules in mind; the figure after them shows the first in a real frame.

on is stored — it lists every task the wait named, including finished ones. `inspect()` tells you which are still live.

running means this process — reopening a Harness turns surviving `running` tasks back into `pending` (the scheduler (p. 73)). They show as `pending` until they are picked up again.

Live tasks only — once a background subagent’s task finishes, no node lists its conversation. Place later tasks there with the conversation’s **owner**, which the conversation view also carries ([subagents \(p. 121\)](#)).



The task graph is the live ownership tree; a finished task leaves it in the commit that ends it. Frames 6 and 7 of scenario A in `research/capture/task-graph.txt`: an order waits `failFast` on three packs, and the first pack to finish is deleted while the order’s stored `on` still names it. The conversation box is drawn from `conversationId`; the graph holds task nodes only.

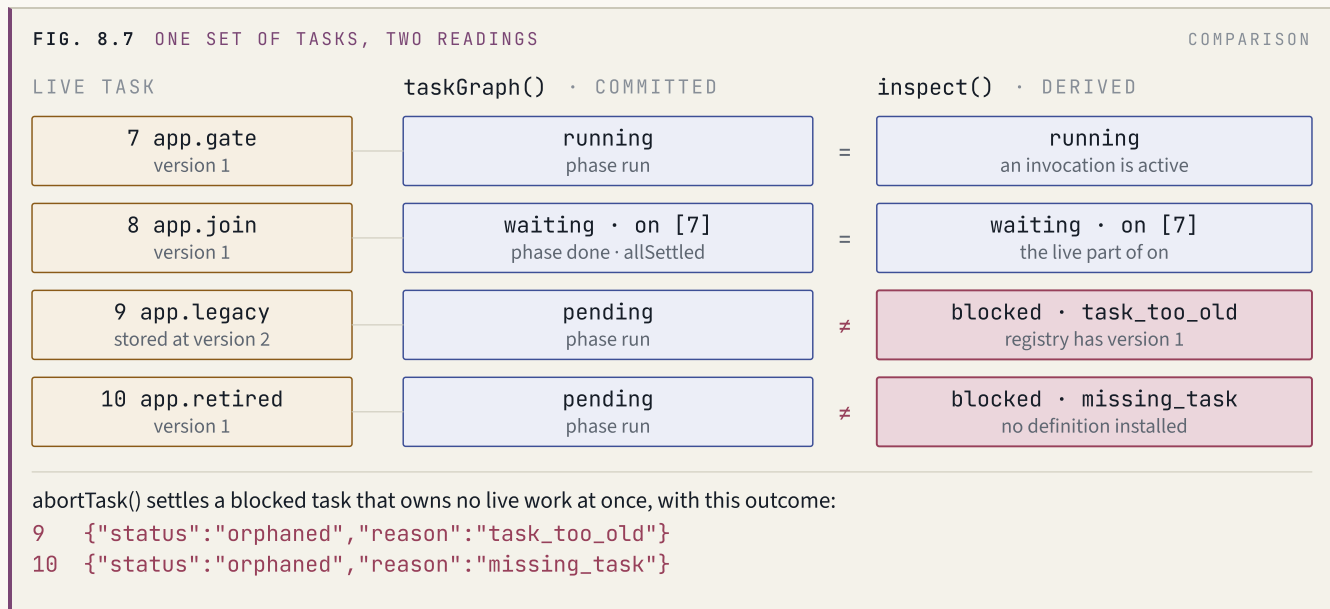
`taskGraph()` returns a Chord state; `watchTaskGraph()` returns a watch with the rules of [8.2 \(p. 129\)](#). Neither starts scheduling, so a viewer can attach to a paused Harness without starting any work [spec §9.5](#).

inspect(): what the scheduler would do

`harness.inspect(context)` writes nothing and runs no task code. It returns `scheduling` (paused before the first `resume()`, `running`, or `closing`), every live task with its full record and a derived `state`, and the queued and placed submissions.

STATE.KIND	WHEN
running	Its code is running in this process
completing	Its outcome is held until its owned work ends
waiting, on	The live tasks it still waits for
blocked, reason	No installed definition can run it: <code>missing_task</code> , <code>task_too_old</code> , or <code>migration_failed</code> (with error)
ready, migrates	The next scheduling pass picks it up; <code>migrates</code> when a newer definition will migrate it first

`src/harness/scheduler.ts`. `migration_failed` appears only after the scheduler tried; inspection never runs a migration.



The graph shows what is committed; inspect() adds what the scheduler would do under this registry. Four tasks created in one commit, read while the gate runs. app.legacy was stored by a version 2 definition and only version 1 is installed; app.retired is not installed. From research/capture/extra-p8/inspect-vs-graph.txt.

The graph cannot show “blocked”, because nothing stores it. A blocked task’s record is an ordinary pending or waiting record, and it never finishes on its own. Install a definition that can run it, or abort it: in the capture, abortTask() settles both blocked tasks at once as orphaned, with the blocked reason as the outcome’s reason.

From “nothing happens” to a reason

When a conversation stays busy or a waitForTask() never resolves, call inspect() and check in this order.

1. **scheduling is paused.** Nothing has called resume() or a call that starts it, such as submit() or waitForIdle(). Read-only viewers never start it.
2. **A task is blocked.** The registry is missing its kind or holds an older version. Fix the install, or abort the task to settle it as orphaned (the scheduler (p. 73)).
3. **A task is waiting.** Follow its on to the live tasks and repeat. An aborted task first waits for its owned work (abort, fault and orphan (p. 80)).
4. **A task is completing.** Its own work is done and its owned work is not; find the owned nodes in the graph (ownership and structured concurrency (p. 76)).
5. **A task is running.** Its code is active. For generations and tools, pi.live in the conversation view shows the attempt, retry wait or tool output (the live document (p. 88)).

A queued submission waits for the run ahead of it; a placed one belongs to a run that has not answered (6.1 (p. 84)). Finished tasks are not in inspect(); read them with getTask().

Sources: spec §2.2 (inspect()), §5.4, §9.5; README §Task Graph; src/harness/task-graph.ts; src/harness/harness.ts (inspect, taskGraph, watchTaskGraph); src/harness/types.ts (TaskInspection, HarnessInspection); src/harness/scheduler.ts; test/harness-task-graph.test.ts; test/harness-inspect.test.ts; research/capture/task-graph.txt; research/capture/NOTES.md (note 9); research/capture/extra-p8/inspect-vs-graph.txt (script test/capture-p8-observe.ts)

Storage and Environment

A small storage contract with three backends, and the execution environment where tools touch the world.

9.1	The storage contract	141
9.2	SQLite	144
9.3	JSONL	148
9.4	Memory, conformance and portability	152
9.5	The execution environment	155

09

The storage contract

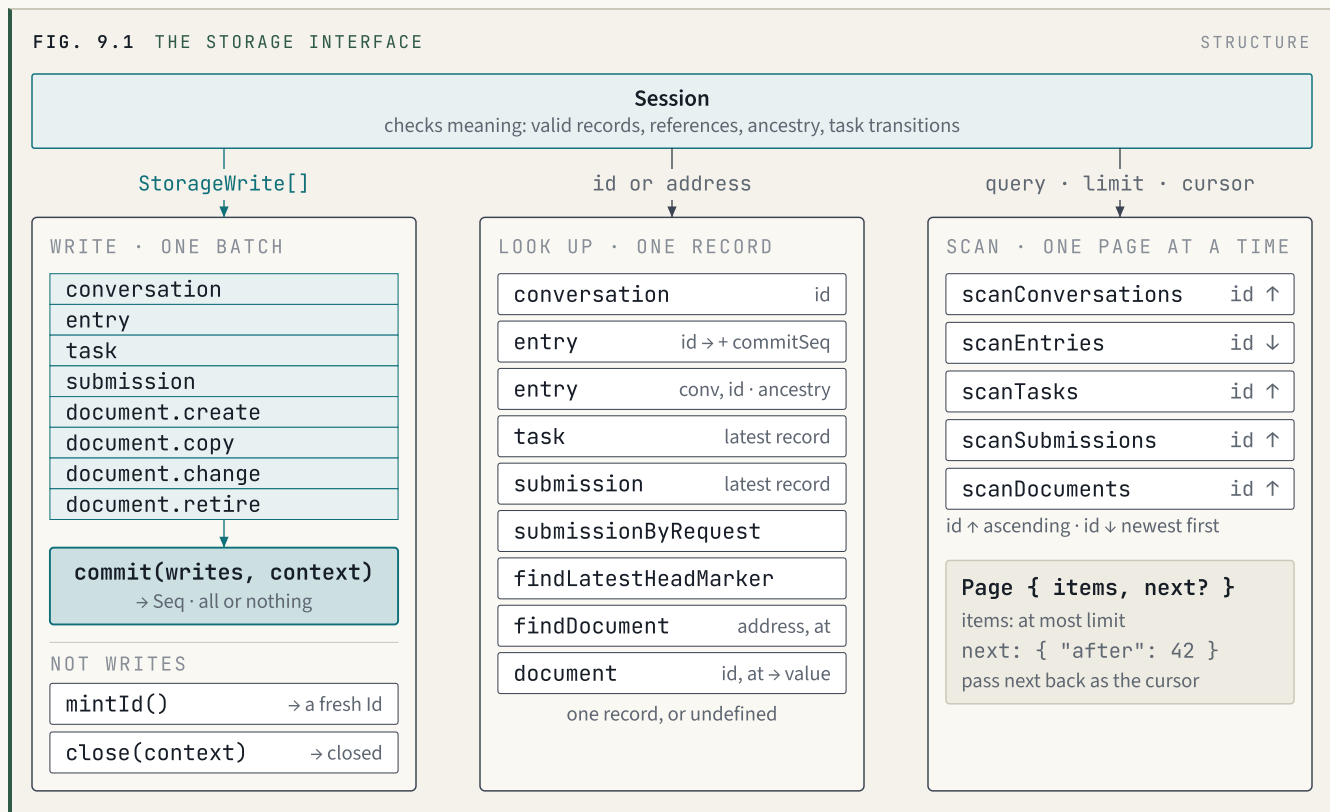
Storage stores each commit all or nothing and answers lookups and paged scans; every other rule belongs to the Session.

Everything a Session commits ends up in storage: a SQLite file, a directory of text files, or memory. A *commit* is an atomic save: everything in it is stored together, or nothing is. Storage must keep each commit that way and answer exact lookups and paged scans. That is nearly all Pi Durable asks of it. After this section you can say what storage promises, which rules belong to the Session instead, and what to do when a commit fails.

A small interface with a narrow job

A backend is any object that implements the Storage interface from `src/types.ts`. Three ship with the package: `SqliteStorage`, `JsonlStorage` and `MemoryStorage` ([SQLite](#) (p. 144), [JSONL](#) (p. 148), [memory](#) (p. 152)). You pick one and hand it to the Harness. From then on only the Session calls it, and every write goes through a commit ([the commit and the mutation line](#) (p. 29)).

The work is split on purpose. The Session checks meaning: valid records, references, ancestry and task transitions. Storage checks only what it can check alone: that a batch is atomic, that IDs are unique, that immutable records stay immutable, and that it hands back copies `src/types.ts`. Storage never sees a document definition; it stores document edits as given and replays them only to answer a read. The Session already runs commits one at a time, so a backend needs no lock of its own [spec §10](#).



Everything the Session stores goes through one call; everything it reads is a keyed lookup or a cursor scan. The eight write types and the methods of Storage in `src/types.ts`. The cursor shown is the shape all three shipped backends use.

One write: `commit`

`commit(writes, context)` stores a batch of writes atomically and returns its `Seq`, the commit’s sequence number. Sequences “strictly increase but may have gaps”, and once `commit()` resolves, every later read sees the batch [spec §10](#). Storage stamps the sequence onto what it keeps. That is how a document can later be read as it was when a given entry was written ([bases, deltas and migrations](#) (p. 56)).

WRITE TYPE	CARRIES	STORAGE KEEPS
<code>conversation</code>	<code>value: ConversationRecord</code>	a new immutable record
<code>entry</code>	<code>value: EntryRecord</code>	a new immutable record and its commit sequence
<code>task</code>	<code>value: TaskRecord</code>	the complete record, replacing the previous one
<code>submission</code>	<code>value: SubmissionRecord</code>	the complete record, replacing the previous one
<code>document.create</code>	<code>record, content (a base)</code>	a new incarnation, stamped <code>createdAt</code>
<code>document.copy</code>	<code>record, source: { id, at }</code>	a new incarnation that starts from the source’s committed value
<code>document.change</code>	<code>id, content (base or delta)</code>	one more revision
<code>document.retire</code>	<code>id</code>	the incarnation, stamped <code>retiredAt</code>

The `StorageWrite` union in `src/types.ts`. A *base* is a document’s complete value; a *delta* is a batch of Chord operations on top of the previous revision.

When a commit fails

Only one kind of failure is safe. `StorageRejected` “means a batch was rejected before any durable effect and is guaranteed not to have committed”. The Session rolls back and carries on. Any other error from `commit()` *poisons* the Session: it can no longer know whether the batch was stored, so it refuses further commits [spec §10](#). What poison means for your host is in [when a commit fails](#) (p. 33).

Reads: keyed lookups and paged scans

A keyed lookup returns one record, or `undefined`. A scan returns a page, `{ items, next? }`; pass `next` back to get the following page. The cursor is opaque: pass it back only to the same scan on the same storage [spec §10](#). The Harness reads 256 items per page, and every read it makes has an index behind it [spec §10](#).

METHOD	RETURNS	NOTES
<code>conversation(id)</code>	<code>record</code>	exact ID
<code>entry(id)</code>	<code>{ entry, commitSeq }</code>	any entry
<code>entry(conversationId, id)</code>	<code>{ entry, commitSeq }</code>	only if that conversation can see it
<code>findLatestHeadMarker(conversationId, at?)</code>	<code>entry</code>	the newest head marker at or before <code>at</code>
<code>task(id), submission(id)</code>	<code>record</code>	the latest complete record
<code>submissionByRequest(conversationId, requestId)</code>	<code>record</code>	host deduplication (submissions (p. 84))
<code>findDocument(address, at)</code>	<code>DocumentRecord</code>	which incarnation lives at an address
<code>document(id, at)</code>	<code>StoredDocument</code>	the value, its version, deltas replayed
<code>scanConversations(query)</code>	<code>page</code>	by owner; ascending ID
<code>scanEntries(query)</code>	<code>page</code>	an ID range, newest first, through fork ancestors
<code>scanTasks(query)</code>	<code>page</code>	by conversation, kind, status, abort mark, background
<code>scanSubmissions(query)</code>	<code>page</code>	by conversation, status; ascending ID
<code>scanDocuments(query)</code>	<code>page</code>	one scope, optional kind; ascending ID
<code>close()</code>	—	releases resources; every later call fails

Every method also takes a final `context`. [src/types.ts](#) and [spec §10](#).

`document(id, at)` is the one read that computes. It finds the newest base at or before the point, applies the deltas after it in order, and returns the value. At "current" a retired incarnation returns `undefined`. Asking for an old point of a document that keeps no history fails rather than guessing ([defining a document \(p. 50\)](#)).

WHAT THIS MEANS FOR YOU

- You rarely call storage yourself. Choose a backend, open it, and pass it to the Harness.
- A commit error other than `StorageRejected` poisons the Session. Close it and reopen the storage; the reopened state is the truth.
- Writing a backend? Pass the conformance suite ([9.4 \(p. 152\)](#)).

Details for backend authors

All records share one ID space from `mintId()`; ID 1 is the root conversation. A batch holds at most one create or change per document incarnation, plus an optional retirement, and a version change must be a base [spec §10](#). Keep nothing from the arguments after `commit()` settles, and return detached values [spec §4](#).

Sources: [spec §4](#), [§10](#); [src/types.ts](#) (`Storage`, `StorageWrite`); [src/session/session.ts](#); [src/harness/context.ts](#); [src/storage/memory.ts](#); [src/storage/sqlite/storage.ts](#); [src/testing/storage-conformance.ts](#); [docs/pico-v5-handoff.md §1-2](#)

SQLite

One SQL transaction is one commit; every record is kept whole as JSON beside the few columns its queries need.

SQLite keeps a whole Session in one file. It is the backend for anything that must survive a restart. After this section you can open a database, find any record with an ordinary SQLite client, predict what a crash or a power cut costs, and run the same backend outside Node.

Opening a file

`openNodeSqliteStorage(path, options)` from `@earendil-works/pi-durable/storage/sqlite/node` creates the parent directory and opens the file with Node's built-in `node:sqlite`. It turns on write-ahead logging (`journal_mode = WAL`, `synchronous = NORMAL`) and applies any pending schema migrations `src/storage/sqlite/node.ts`.

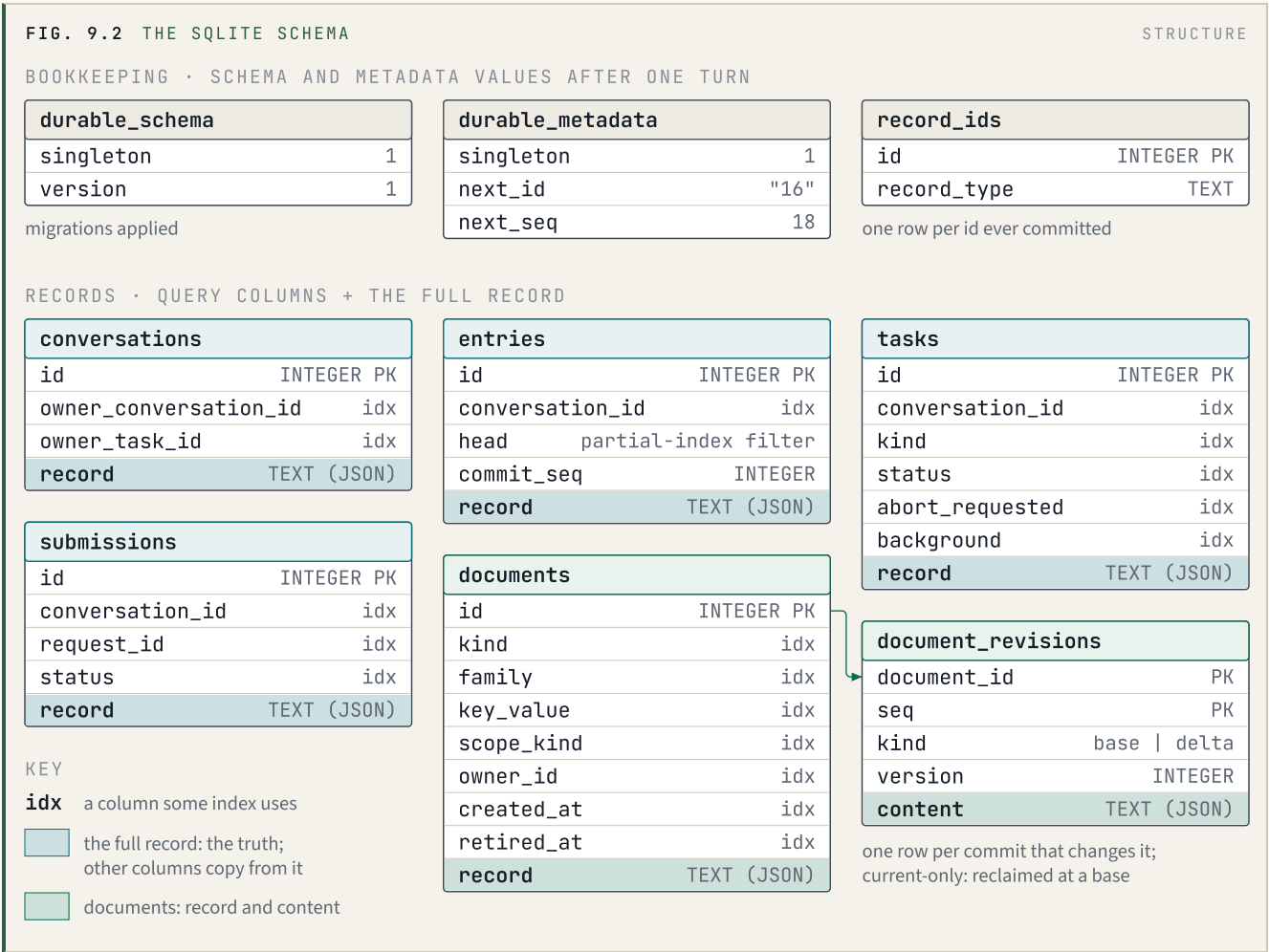
OPTION	DEFAULT	EFFECT
<code>walAutoCheckpointPages</code>	1,000	WAL auto-checkpoint threshold in pages; 0 disables it
<code>busyTimeoutMs</code>	5,000	how long to wait for another connection's lock (SQLite's own default is 0)

`NodeSqliteStorageOptions`, `src/storage/sqlite/node.ts`.

Those settings decide what survives. “WAL mode with `synchronous = NORMAL`: commits survive process crashes; the newest may be lost on power or host failure” [README §Storage](#). One SQL transaction is one commit, so a process that dies mid-commit leaves all of it or none of it [spec §11.2](#). `close()` folds the write-ahead log back into the file. After a close, the capture found a single `session.sqlite` and no WAL file [research/capture/extra-p9/sqlite-revisions.txt](#).

Nine tables

Every record table has the same shape: a few columns that queries filter on, and a `record` column holding the complete record as JSON. Reads select only `record` and parse it. The other columns are copies written from the record in the same statement, so the JSON is always the truth.



Each record table lifts its query columns out of a JSON record; document content lives apart, one row per commit that changes it. All nine tables of research/capture/sqlite-schema.sql; “idx” marks a column used by an index. Bookkeeping values are from research/capture/sqlite-rows.txt after one turn.

the tasks table and its indexes

research/capture/sqlite-schema.sql

SQL

```
CREATE TABLE tasks (  
  id INTEGER PRIMARY KEY,  
  conversation_id INTEGER NOT NULL,  
  kind TEXT NOT NULL,  
  status TEXT NOT NULL CHECK (status IN ('pending', 'running', 'waiting',  
                                         'completing', 'terminal')),  
  abort_requested INTEGER NOT NULL CHECK (abort_requested IN (0, 1)),  
  background INTEGER NOT NULL CHECK (background IN (0, 1)),  
  record TEXT NOT NULL CHECK (json_valid(record))  
) STRICT;  
  
CREATE INDEX tasks_by_status ON tasks (status, id);  
CREATE INDEX tasks_by_conversation ON tasks (conversation_id, id);  
...
```

Three more indexes, on kind, abort_requested and background, are elided. Each index ends in id, so a scan pages in ID order.

Conversations and entries never change, so they are plain inserts and a second write of the same ID fails. Tasks and submissions are replaced in place: “Live task transitions replace one row. Terminal tasks remain as small records” spec §11.2. Mid-run, the

capture shows generation 9 waiting on tool 12, which is running. After the run, both rows are terminal and hold only an outcome `research/capture/sqlite-rows-midrun.txt`, `sqlite-rows.txt`.

Reading the file by hand

- Indexed strings are stored JSON-encoded: a task's `kind` column reads `"\pi.generation\"`. This keeps odd Unicode lossless across SQLite bindings `src/storage/sqlite/storage.ts`.
- `durable_metadata.next_id` is TEXT ("16"), because `node:sqlite` rejects integers beyond JavaScript's safe range.
- `record_ids` holds one row per committed ID with its record type. It is what makes the ID space global.

One commit, one transaction

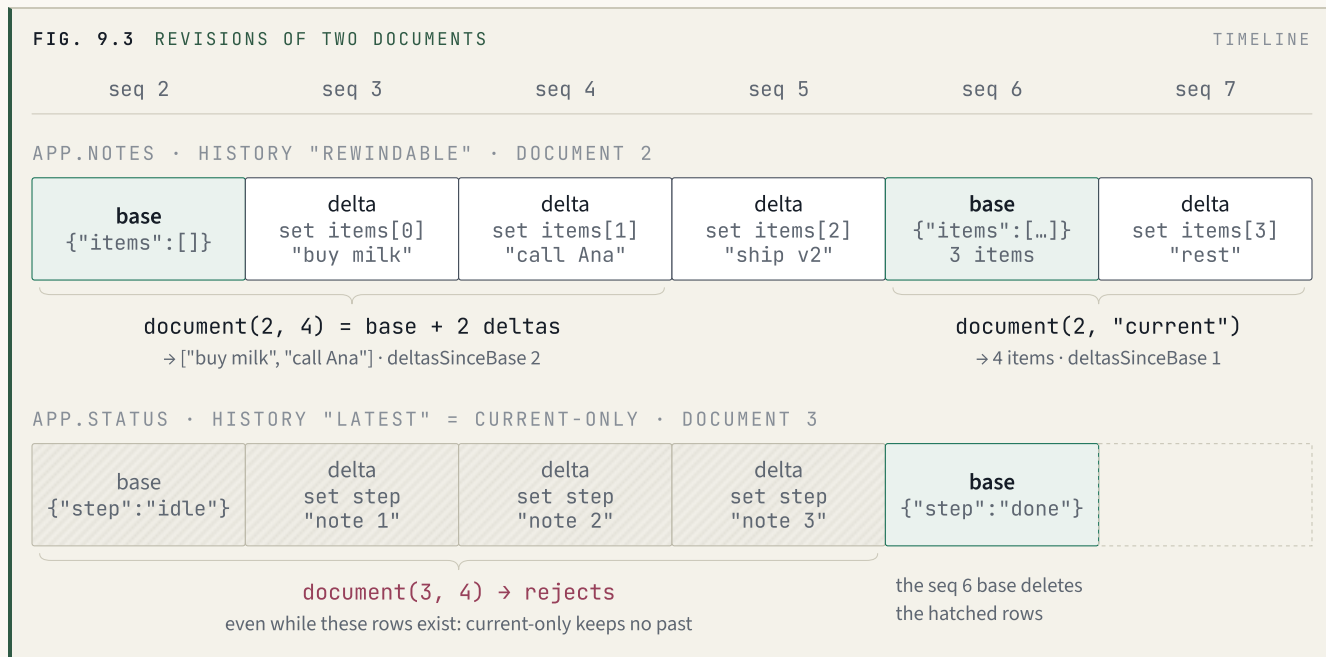
`SQLiteStorage.commit()` checks document commands in memory, then runs one SQL transaction: it reads the next sequence number, checks each new ID and document command, writes the rows, and advances the counters. If anything throws, the transaction rolls back and no sequence number is used `src/storage/sqlite/storage.ts`.

`mintId()` is an in-memory counter, not a write. An ID minted by a transaction that later failed may be handed out again after a reopen. That is safe: only IDs in a committed write must never be reused ([invariant 5](#) (p. 16)).

Document revisions

Document content lives in `document_revisions`, one row per document per commit that changed it. A row is either a *base*, holding the complete value, or a *delta*, holding the Chord operations. To read a document, SQLite finds the newest base at or before the point and applies the deltas after it, inside one transaction so a concurrent commit cannot swap the base

`src/storage/sqlite/storage.ts`.



A rewindable document keeps every revision; a current-only document drops its older rows when a base commits. Rows and reads from `research/capture/extra-p9/sqlite-revisions.txt`: two app documents changed in the same commits, seq 2–7.

Whether history is kept depends on the document. A rewindable document keeps every row, so `document(2, 4)` rebuilds the value as of seq 4 from the base at 2 and two deltas. Every other document is current-only: when a new base commits, the same transaction deletes its older rows. That is why each built-in document of the one-turn run ends as a single base row

research/capture/sqlite-rows.txt. Every one of these reads is an indexed search; the capture’s query plans show no table scans
 research/capture/extra-p9/sqlite-revisions.txt.

Beyond Node

The SQLite core needs no Node API. `SQLiteStorage.open(db)` from `@earendil-works/pi-durable/storage/sqlite` accepts any object that implements `SQLiteDatabase`. The same backend then runs wherever someone writes a small adapter, “for example on Bun or in Cloudflare Durable Objects” `README §Storage`.

the facade an adapter implements `src/storage/sqlite/database.ts`

TS

```
export interface SQLiteExecutor {
  exec(sql: string): Promise<void>;
  run(sql: string, ...params: SQLiteValue[]): Promise<void>;
  get<T extends object>(sql: string, ...params: SQLiteValue[]): Promise<T | undefined>;
  all<T extends object>(sql: string, ...params: SQLiteValue[]): Promise<T[]>;
}

export interface SQLiteDatabase extends SQLiteExecutor {
  transaction<T>(callback: (transaction: SQLiteExecutor) => Promise<T>): Promise<T>;
  close(): Promise<void>;
}
```

Doc comments and the `SQLiteValue` type (null, number, bigint, string or bytes) are elided.

The adapter’s rules are short but strict. Inside a transaction, use only the handle the callback receives; it is invalid once the callback settles. Other work waits until the transaction finishes, so calling the database itself from inside a callback never settles. If the callback rejects, roll back and reject with the same error. If the rollback fails too, reject with a different error, such as an `AggregateError`, “so callers cannot mistake the callback error for a guaranteed rollback” `src/storage/sqlite/database.ts`.

Schema changes are versioned migrations, applied in one transaction when the file opens. There is one so far, version 1. A file written by a newer package is refused (“Durable SQLite schema version 2 is newer than supported version 1”), and a failed migration rolls back with the data intact `src/storage/sqlite/migrations.ts; test/sqlite-migrations.test.ts`.

ONE PROCESS

“One process owns a storage at a time; there is no cross-process locking” `README §Storage`. Each `SQLiteStorage` mints IDs from its own counter, so two processes on one file can mint the same ID. The second commit then fails and poisons that Session, or silently replaces the other process’s task or submission. Running a single owner is the host’s job ([limits and non-goals](#) (p. 168)).

WHAT THIS MEANS FOR YOU

- Use SQLite for anything long-lived. A process crash loses nothing committed; a power cut may lose the newest commits.
- To debug, open the file with any SQLite client and read the `record` column.
- On another runtime, write a `SQLiteDatabase` adapter and run the conformance suite against it.

Sources: spec §4, §11.2; `README §Storage`; `src/storage/sqlite/{storage,node,database,migrations}.ts`; `test/sqlite-storage.test.ts`; `test/sqlite-facade.test.ts`; `test/sqlite-migrations.test.ts`; `research/capture/sqlite-schema.sql`, `sqlite-rows.txt`, `sqlite-rows-midrun.txt`; `research/capture/extra-p9/sqlite-revisions.txt`

JSONL

Sidecar records first, one marker in main.jsonl second: a commit exists exactly when its marker does.

JSONL storage is a directory of plain text files you can read with `cat`. It suits development and debugging, where seeing every record matters more than speed. After this section you can read the files, say what a crash can lose, and decide whether to turn on `fsync`.

The directory

`openNodeJsonlStorage(directory, context, options)` from `/storage/jsonl/node` opens or creates the directory. On another runtime, `JsonlStorage.open(directory, fs, context, options)` from `/storage/jsonl` does the same over any `FileSystem` (the execution environment (p. 155)). Every line is one JSON object, in one of three kinds of file spec §11.3:

`main.jsonl` — one *marker* line per commit. It only grows.

`doc-<id>.jsonl` — the content of one document: its bases and deltas.

`task-<id>.jsonl` — the successive records of one live task. It is deleted when the task finishes.

The document and task files are *sidecars*: they hold the bulky, changing data. A marker carries small records in full, such as entries and a task's final record. For the rest it carries a pointer: a file plus an `ordinal` that numbers the commit's sidecar lines from 0 `src/storage/jsonl/storage.ts`.

FIG. 9.4 ONE COMMIT, ONE MARKER, FOUR SIDECARS

ANATOMY

MAIN.JSONL · LINES 1-6 OF 10

<code>{"format":1,"type":"commit","seq":1,...}</code>
<code>{"format":1,"type":"commit","seq":2,...}</code>
<code>{"format":1,"type":"commit","seq":3,...}</code>
<code>{"format":1,"type":"commit","seq":4,...}</code>
<code>{"format":1,"type":"commit","seq":5,...}</code>
<code>{"format":1,"type":"commit","seq":6,...}</code>

each sidecar write names a record
by its seq and ordinal

LINE 6 · ITS "WRITES"

entry 11	inline · pi.assistant
task.sidecar 9	ordinal 0
task.sidecar 12	ordinal 1
document.change 2	ordinal 2
document.change 4	ordinal 3

task-9.jsonl

seq 2 · ord 0	pending · prepare
seq 3 · ord 0	running · prepare
seq 4 · ord 0	running · request
seq 6 · ord 0	waiting · tools

task-12.jsonl

seq 6 · ord 1	pending · call
seq 7 · ord 0	running · call
seq 8 · ord 0	running · execute

doc-2.jsonl

seq 1, 2, 5 · reclaimed by the seq 6 base	
seq 6 · ord 2	base
seq 8 · ord 1	delta
seq 9 · ord 0	delta
seq 10 · ord 0	delta

doc-4.jsonl

seq 6 · ord 3	base
---------------	------

A JSONL commit is one marker in main.jsonl that names its sidecar records by sequence and ordinal. Commit 6 of the one-turn run, copied while the count tool was running: `research/capture/jsonl-dir-midrun/`. The assistant entry rides in the marker; the two tasks and two documents live in sidecars.

the marker of commit 6 `research/capture/jsonl-dir-midrun/main.jsonl, line 6`

JSON

```
{
  "format": 1, "type": "commit", "seq": 6,
  "writes": [
    { "type": "entry", "value": { "model": [ ... ], "kind": "pi.assistant", "id": 11,
      "conversationId": 1, "byTaskId": 9 } },
    { "type": "task.sidecar", "id": 9, "ordinal": 0 },
    { "type": "task.sidecar", "id": 12, "ordinal": 1 },
    { "type": "document.change", "id": 2, "ordinal": 2 },
    { "type": "document.change", "id": 4, "ordinal": 3 }
  ]
}
```

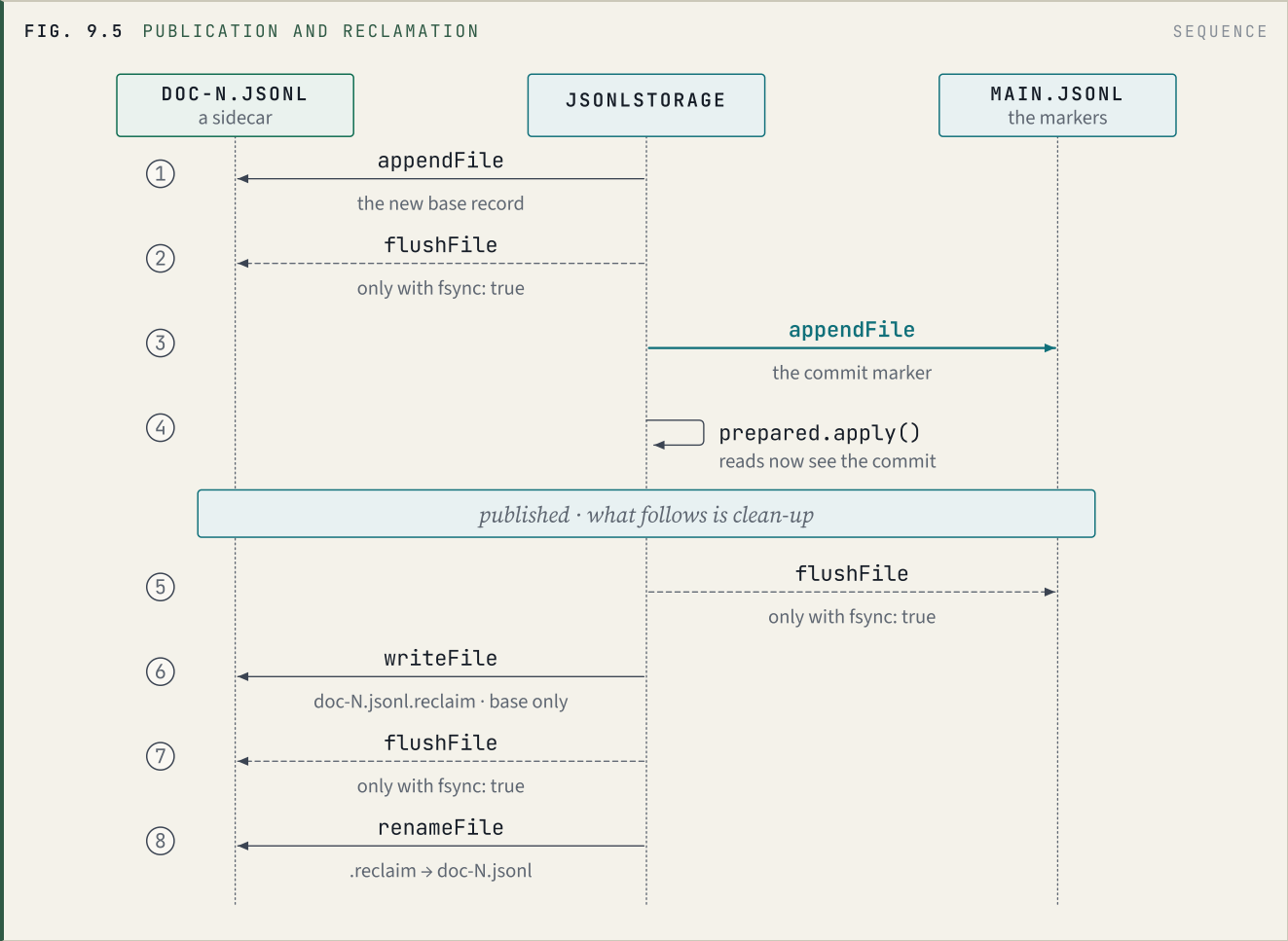
Pretty-printed; on disk it is one line. The entry's model message is elided.

After the turn, both task files were gone and doc-2.jsonl was a single base `research/capture/jsonl-files.txt`. Reclamation, below, did both.

Publication

Every commit follows the same three steps `spec §11.3`:

1. append the commit's records to every affected sidecar;
2. append one complete marker to `main.jsonl` listing those records;
3. only then make the commit visible to reads.



Sidecar records land first, the marker makes them a commit, and reclamation runs only after publication. A commit that writes a new base for a current-only document, in the file order asserted by `test/jsonl-storage.test.ts`; steps 2, 5 and 7 happen only with `fsync: true`.

A batch that fails validation touches no file. Once writing has begun, any failed append or flush poisons the backend: every call rejects with `JsonlStoragePoisonedError` until you reopen, and the `Session` is poisoned too (2.3 (p. 33)).

The `fsync` option decides what survives, and it defaults to `false`. Without it, JSONL survives a process crash, but not a power, kernel or disk failure. With `fsync: true`, each sidecar is flushed to disk before its marker is written. The newest commits can still vanish in a power cut, “but a marker that survives should not overtake its sidecar data” `spec §11.3`.

COMMIT	FSYNC: FALSE	FSYNC: TRUE
two new documents	append ×2, append main	append ×2, flush ×2, append main
task finishes	append main, remove sidecar	append main, flush main, remove sidecar

File operations per commit, from `test/jsonl-storage.test.ts`.

Recovery

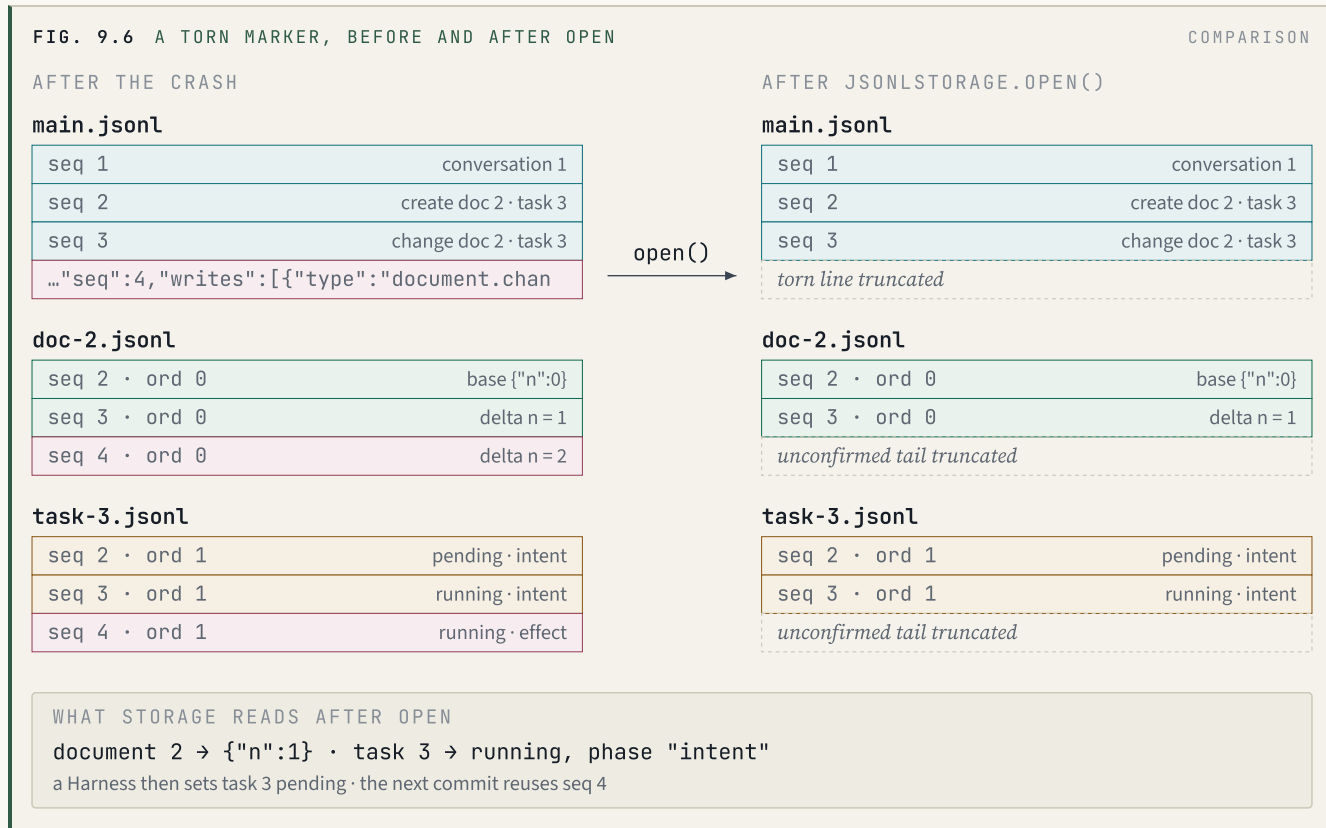
Opening a directory is a recovery pass, and its rules are short `spec §11.3`; `src/storage/jsonl/storage.ts`:

Torn lines — a file that does not end in a newline is cut back to its last newline.

Unconfirmed records — a sidecar line counts only if a marker names it. Unconfirmed lines at the end are cut.

Corruption — a confirmed line that is missing, out of order, or after an unconfirmed one fails the open with `JsonlCorruptionError` (unless a later base made it unnecessary). Nothing repairs it [spec §13](#).

Leftovers — files from an interrupted reclamation are deleted, and any reclamation still owed is done.



Opening a JSONL directory cuts the torn marker and every sidecar record no marker confirms. [research/capture/extra-p9/jsonl-crash.txt](#).

In the capture, commit 4 was meant to set a counter to 2 and move a task into phase `effect`. Its sidecar lines were written, but the marker was torn. Reopening cut all three: the counter reads `{"n": 1}`, the task is `running` in phase `intent`, and the next commit gets seq 4 again [research/capture/extra-p9/jsonl-crash.txt](#). Commit 4 never happened, so a reopened Harness runs the intent phase again ([5.3 \(p. 70\)](#)).

Reclamation

Reclamation deletes sidecar lines that a newer base, a retirement or a finished task made unnecessary, only after that commit [spec §11.3](#). Rewindable documents are never reclaimed. If reclamation fails, the commit stays published and the next open finishes the job [src/storage/jsonl/storage.ts](#).

WHAT THIS MEANS FOR YOU

- A process crash never leaves half a commit. A commit exists exactly when its marker line is complete.
- Turn on `fsync: true` if the files must survive a power cut in a consistent state. Expect slow opens on large stores ([9.4 \(p. 152\)](#)).

Sources: [spec §11.3](#), [§12](#), [§13](#); [src/storage/jsonl/storage.ts](#); [test/jsonl-storage.test.ts](#); [research/capture/jsonl-dir-midrun/](#), [jsonl-files.txt](#); [research/capture/extra-p9/jsonl-crash.txt](#)

Memory, conformance and portability

MemoryStorage defines what every backend means, one shared test suite checks it, and the storage cores run wherever an adapter can.

The three backends are interchangeable because they are held to one reference and one test suite. After this section you can choose a backend from measured costs, test your own backend with the shared suite, and run storage on a runtime other than Node.

MemoryStorage is the reference

Memory storage defines what the others must do. It copies every record it keeps and every record it returns: “This deliberately simulates the ownership boundary naturally created by SQLite encoding/decoding and JSONL serialization” [spec §11.1](#). So a caller that mutates a record it read cannot change storage, on memory or on disk. Memory storage also keeps or drops document history exactly as the durable backends do. A test that reads an old point of a current-only document fails on memory just as it would on SQLite.

Memory storage persists nothing. Use it for tests, examples and short-lived sessions. Most examples in `test/examples` run on it; the recovery and restart examples (ex 13, 23, 24, 31) use SQLite, and ex 17 uses JSONL. JSONL also uses memory storage internally to answer every read `src/storage/jsonl/storage.ts`.

The conformance suite

A custom backend proves itself by passing the suite the shipped ones pass. It is exported from `@earendil-works/pi-durable/testing` and runs under any Vitest- or Jest-compatible runner:

running the storage suite [README](#) [§Storage](#)

TS

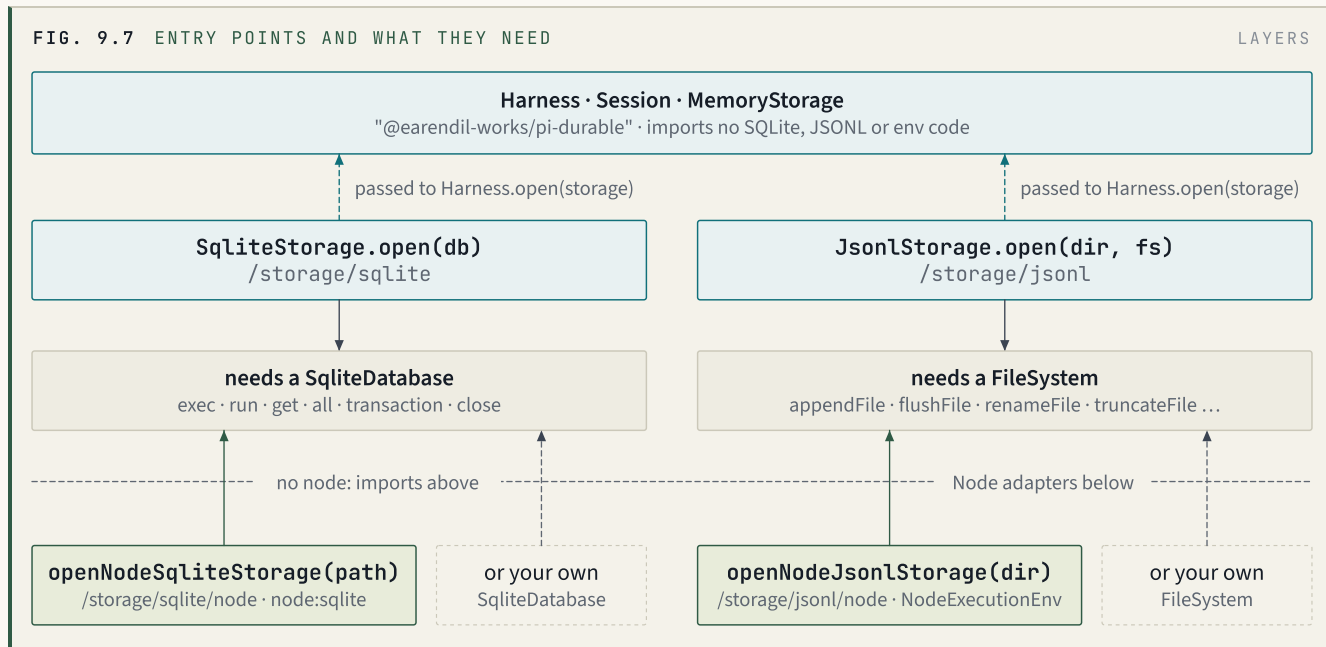
```
import { registerStorageConformance } from "@earendil-works/pi-durable/testing";
import { describe, expect, it } from "vitest";

registerStorageConformance({ describe, expect, it }, "My Storage", async (use) => {
  const storage = await openMyStorage();
  try {
    await use(storage);
  } finally {
    await closeMyStorage(storage);
  }
});
```

Call `use` exactly once per case, with a fresh, empty storage.

The suite has 23 cases. They cover the ID space, atomic batches and detached values, entries and forks, conversation and task scans, submissions, document history and addresses, and lossless string keys `src/testing/storage-conformance.ts`. The package runs it five times: on memory, SQLite and JSONL, and on SQLite and JSONL closed and reopened after every commit. The suite pins meaning, not mechanics. Crash recovery, flush order and query plans are tested per backend, and a custom backend needs its own tests for them.

Portable cores, Node adapters

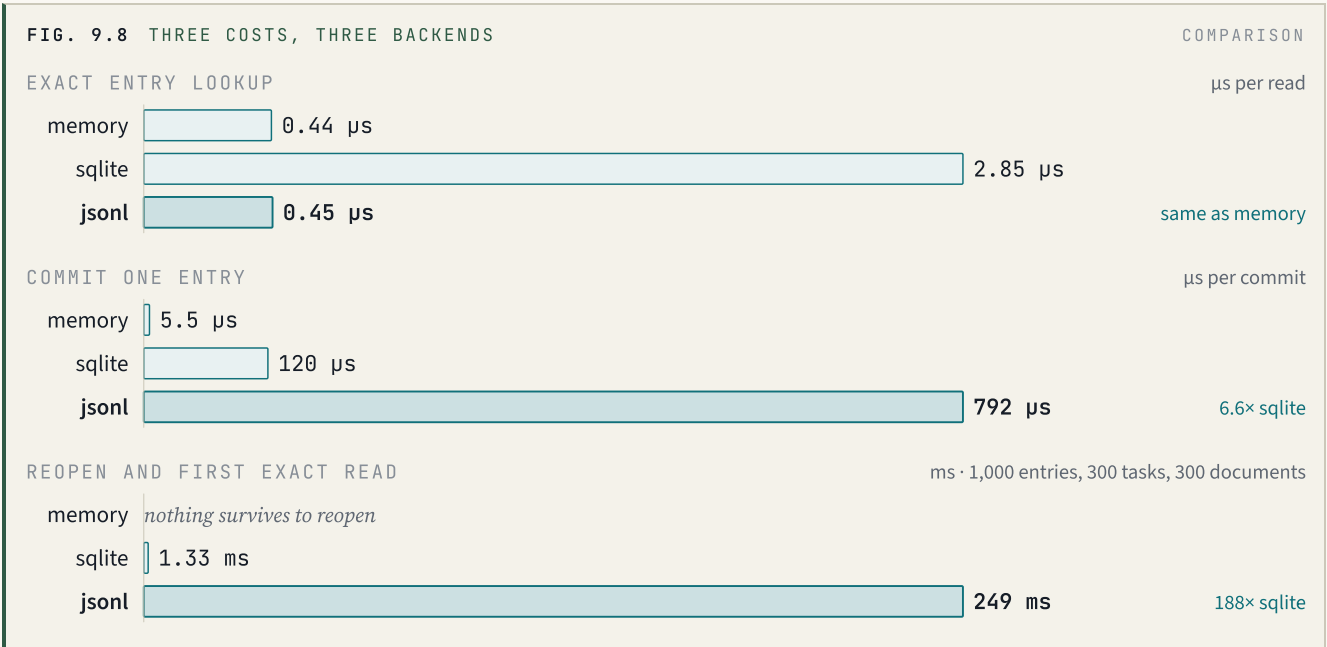


The SQLite and JSONL cores are portable; only their Node adapters touch Node APIs. Subpath exports of package.json; either core also accepts an adapter you write. The boundary is enforced by test/storage-runtime-boundary.test.ts, which walks each entry point's imports.

The split is enforced by a test. /storage/sqlite, /storage/jsonl and /env may not import any node: module, even indirectly. Only /storage/sqlite/node and /storage/jsonl/node are Node-specific. To run on another runtime, give SQLiteStorage.open() a SqliteDatabase (SQLite (p. 144)), or give JsonlStorage.open() a FileSystem (the execution environment (p. 155)).

Measured costs

The package ships a benchmark that runs the same reads and writes on all three backends: `npm run bench:storage`. Its dataset holds 1,000 entries, 300 tasks and 300 documents. The figure shows three of its rows from one run on an Apple M3 Max with Node 24.13.0.



JSONL reads as fast as memory because it is memory; it pays at every commit and at open. Means from `research/capture/extra-p9/bench-storage.txt` (Apple M3 Max, macOS 26.2, Node 24.13.0, 2026-10-05). Commit and reopen rows have 20 samples each, with margins of $\pm 8\text{--}66\%$. Each panel has its own scale.

The numbers follow from the designs. JSONL answers reads from memory, so an exact entry lookup costs what it does on memory: 0.44 μs , against 2.85 μs for SQLite. A JSONL commit appends to several files, 792 μs against SQLite’s 120 μs . Opening a JSONL directory replays every marker, 249 ms for this dataset, while SQLite opened and served its first read in 1.33 ms. At ten times the size, SQLite held 0.22 MiB of heap; JSONL and memory each held over 7 MiB `research/capture/extra-p9/bench-storage-memory.txt`.

BACKEND	SURVIVES	COSTS	USE IT FOR
memory	nothing	heap grows with the store	tests, examples, short sessions
SQLite	process crashes	slower reads; small heap; fast open	anything long-lived
JSONL	process crashes; power loss stays consistent with <code>fsync</code>	slow commits; whole store in memory; replays at open	debugging, readable files

Summary of this section and the two before it.

Timings from different machines are not comparable. To compare your own backend, use the scenarios exported from `/testing(seedStorageBenchmark()` and the read and write benchmark lists) and run every candidate in one process, as the package’s `test/storage.bench.ts` does.

WHAT THIS MEANS FOR YOU

- Default to SQLite; use memory in tests, and JSONL when you want to read the files.
- Whatever you choose, run one process per storage. No backend locks against a second.
- A custom backend must pass the 23-case suite, plus its own crash tests.

Sources: `spec §11.1`; `README $Storage`, `$Examples`; `src/storage/memory.ts`; `src/storage/jsonl/storage.ts`; `src/testing/{storage-conformance, runner, storage-benchmark}.ts`; `test/storage.bench.ts`; `test/memory-storage.test.ts`; `test/sqlite-storage.test.ts`; `test/jsonl-storage.test.ts`; `test/storage-runtime-boundary.test.ts`; `package.json`; `research/capture/extra-p9/bench-storage.txt`, `bench-storage-memory.txt`

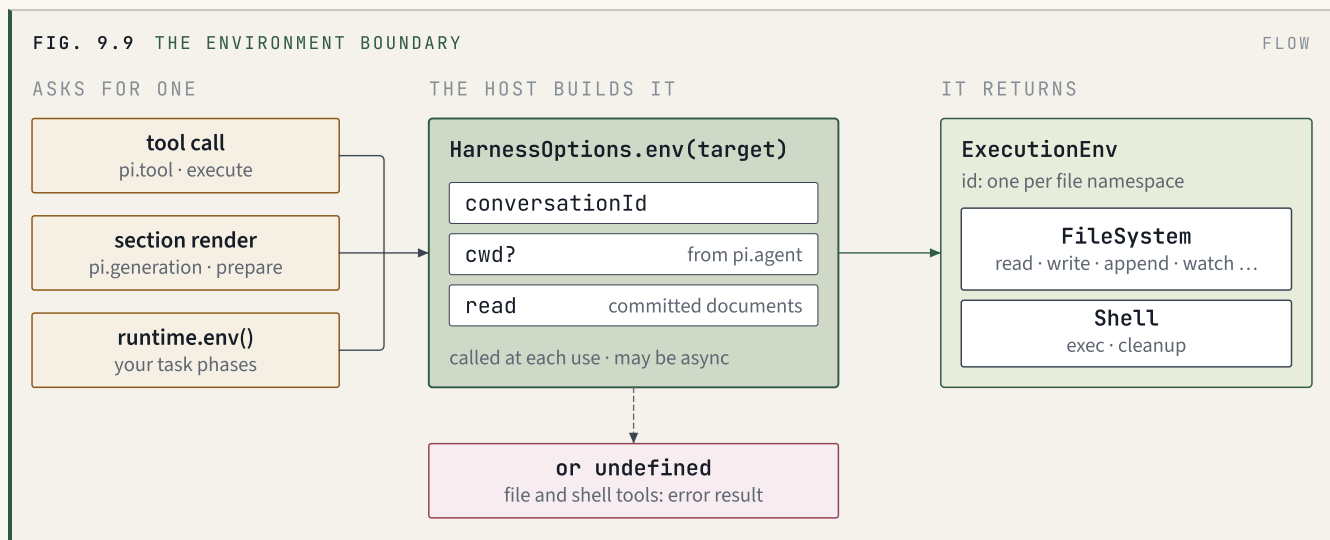
The execution environment

Tools touch files and processes only through an `ExecutionEnv` the host builds for each use, so where a conversation runs is the host's decision, not the tool's.

Your agent's tools read files and run commands. The execution environment decides where: your own disk, a sandbox directory per user, or a remote container. Tools never touch the machine directly; they go through an `ExecutionEnv` that the host builds each time one is needed. After this section you can give each conversation its own sandbox, implement the interface for a remote host, and know what the built-in tools expect.

Built at each use

The host supplies one function, `HarnessOptions.env`. The Harness calls it whenever work needs an environment, never on the Session line, and it may be async `src/harness/types.ts`. It receives the `conversationId`, the conversation's working directory `cwd` if one is set in its `pi.agent` document (the per-conversation agent (p. 106)), and `read`, which reads committed documents.



Tools, prompt sections and task phases reach files and processes only through an environment the host builds per use from committed state. Callers from `README $Environment` and `src/harness/{tool,generation,scheduler}.ts`; the target from `src/harness/types.ts`.

Three kinds of work ask for it. A tool call receives it as `api.env` (defining tools (p. 112)). A system-prompt section receives it as `input.env` (the system prompt (p. 115)). Any task phase can ask with `runtime.env()`. Returning `undefined` is allowed: the built-in tools then fail with an error result, “No execution environment is configured”. A throw from `env` also becomes the tool call's error result `README $Environment`; `src/tools/env.ts`.

Because the environment is built per use, a tool call rerun after a crash gets a fresh one, “with the conversation's current `cwd`, which may differ from the first attempt's” `spec §12`. Build it only from committed state, as example 29 does to give each conversation its own sandbox:

a sandbox per conversation

test/examples/29-sandbox-per-conversation.ts

TS

```
const harness = await Harness.open(
  new MemoryStorage(),
  {
    models,
    registry,
    // Committed reads only; a conversation without a sandbox gets no environment,
    // so its tools fail cleanly.
    env: async ({ conversationId, read }, envContext) => {
      const sandbox = await read.snapshot(Sandbox, conversationId, envContext);
      return sandbox?.path === undefined
        ? undefined
        : new NodeExecutionEnv({ cwd: sandbox.path });
    },
  },
  context,
);
```

```
$ node --conditions=source --experimental-strip-types test/examples/29-sandbox-per-conversation.ts
alice's sandbox: from alice
bob's sandbox: from bob
```

Sandbox is an app document set when each conversation is created. Both conversations ran the same `write` tool, and each note landed in its own directory (run 29). The ternary is rewritten to fit the page.

The interface

`ExecutionEnv` is a `FileSystem` plus a `Shell`, from `@earendil-works/pi-durable/env`. Every operation returns a result instead of throwing: `{ ok: true, value }` or `{ ok: false, error }`, where the error carries a code such as `not_found` or `timeout` `src/env/index.ts`.

GROUP	FILESYSTEM MEMBERS
identity	id, cwd
paths	absolutePath, joinPath, canonicalPath, exists
read	readTextFile, readTextLines, openTextLineReader, readBinaryFile, openBinaryReader
write	writeFile, appendFile, truncateFile, flushFile, renameFile
directories	fileInfo, listDir, openDirReader, createDir, remove
temporary	createTempDir, createTempFile, cleanup
changes	watch

`src/env/index.ts`. `truncateFile` and `flushFile` exist for JSONL storage.

`id` names the file namespace: “equal ids see the same files at the same paths, whatever their `cwd`” `src/env/index.ts`. Every local `NodeExecutionEnv` uses `"node:local"`; each container or remote host needs its own. The `id` lets the host build a new environment object per call. The built-in `edit` and `write` tools still take turns on one file, because they queue by `id` plus path. That queue is per process and “not a lock against bash or other processes” `src/tools/file-mutation-queue.ts`.

Running commands

`exec(command, options, context)` takes a string or an array. A string runs through a shell; `NodeExecutionEnv` uses `bash`, falling back to `sh -c`. An array runs `command[0]` directly with the rest as its arguments, so they reach the program exactly as

given `src/env/index.ts`; `src/env/node.ts`. Prefer the array form when arguments come from the model or a user.

In `NodeExecutionEnv`, `timeout` is in seconds, and a nonzero exit is a successful result with its `exitCode`. `onOutput` receives each chunk, tagged `stdout` or `stderr`. Aborting the context kills that command; `cleanup()` kills everything left, at shutdown.

Output windows

A tool that keeps only the tail of a command's output can say so, and a remote environment can then skip sending the rest. The caller passes a `window` (`maxBytes`, `maxLines` and pacing limits). The environment may drop output outside that tail, reporting what it dropped on the next chunk as `info.skipped` `src/env/index.ts`; `CHANGELOG 1.0.3`.



A windowed exec may drop output the caller would drop anyway, and counts exactly what it dropped. The rule of `info.skipped` (`src/env/index.ts`) and the conformance case that prints 2,000 lines through a 200-byte, 5-line window.

Skipping is safe because of one rule: the chunk after a skip always holds more than the window, so dropped text can never belong in the kept tail. The built-in `bash` tool uses windows `src/harness/tool.ts`; `src/tools/bash.ts`. `NodeExecutionEnv` ignores `window` and delivers everything; windows pay off when output crosses a network. A tool wrapper that transforms output must turn the window off, or skipped text would bypass its transform ([defining tools \(p. 112\)](#)).

WHAT THIS MEANS FOR YOU

- To isolate conversations, return a different environment per conversation from `HarnessOptions.env`. Return `undefined` to switch file and shell tools off.
- Build the environment only from committed state: a rerun after a crash builds it again.
- For a custom environment, implement `FileSystem` and `Shell`, return failures as results, give each file namespace its own `id`, and run `registerEnvConformance` from `/testing`, which has 21 cases.

Details: capabilities added in 1.0.3

Version 1.0.3 made four capabilities mandatory, so custom environments must add them `CHANGELOG 1.0.3`.

`openBinaryReader(path, { noFollow? })` reads one opened file by position, and its `scanLines()` finds a span of lines in one pass; the `read` tool uses both. `openDirReader(path)` pages through a directory. `watch(targets, onChange)` reports changed paths, an overflow (rescan everything) or an error. And `exec` accepts the array form.

Sources: `spec §12` (environment on recovery); `README §Environment`; `CHANGELOG 1.0.3`; `src/env/{index,node,node-watch}.ts`; `src/harness/{types,harness,tool}.ts`; `src/tools/{env,bash,file-mutation-queue}.ts`; `src/testing/env-conformance.ts`; ex 29, run 29

In Practice

Building real agents on the harness, the contracts that bite, and what it deliberately does not do.

10.1	Building a coding agent	159
10.2	Contracts that bite	163
10.3	Limits and non-goals	168

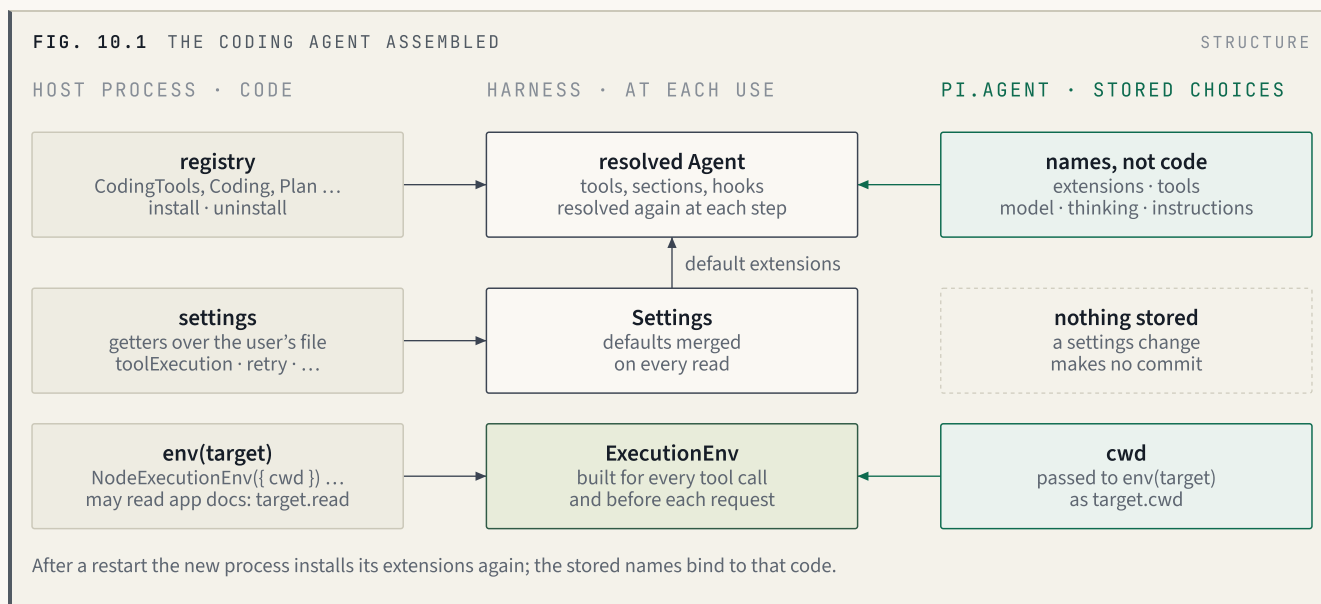
10

10.1

Building a coding agent

A coding agent is a registry, a settings object and an environment function; everything a user chooses per conversation is a name in `pi.agent`.

Picture a coding assistant. A user switches it to a read-only plan mode, points it at another folder, then closes the laptop. Tomorrow it must come back exactly as they left it. That works only if each choice lives in the right place. Examples 26 to 31 show where. After this section you can sort any product feature into code, policy or a stored choice, and say when a change takes effect.



Code and policy live in the process; a conversation stores only its choices. The Harness combines the two each time it needs them. Assembled from ex 26–29 and `src/harness/agent.ts`.

Three host inputs, one stored document

Your host program gives `Harness.open()` three things besides storage and models. None of them is stored, and changing one never makes a *commit*: an atomic save, where everything in it is stored together or nothing is.

Registry — the code: extensions installed by name. Installing a second extension with the same name replaces the first (extensions and the registry (p. 103)).

Settings — run policy shared by every conversation, such as retries and parallel tools. The Harness reads them each time it needs them.

Environment function — `env(target)` builds the file system and shell that a tool sees for one conversation.

What a user picks for one conversation is stored in that conversation's `pi.agent` document: model, thinking level, extension and tool names, instructions and working directory (`cwd`). It stores names, never code (the per-conversation agent (p. 106)). Before each step the Harness looks the names up in the registry and gets the actual tools and prompt sections.

Live settings and a folder that moves

Example 26 is the base agent. It installs `CodingTools` (the `read`, `write`, `edit` and `bash` tools) plus a small prompt extension. Nothing installs `CodingTools` for you `src/tools/index.ts`. The settings are getters over an object the user edits, and the environment follows the conversation's `cwd`.

settings that follow a file, an env that follows cwd `test/examples/26-coding-agent.ts`

TS

```
const userSettings = { parallelTools: true, maxRetries: 3 };
const settings: HarnessSettings = {
  get toolExecution() {
    return userSettings.parallelTools ? "parallel" : "sequential";
  },
  get retry() {
    return { maxRetries: userSettings.maxRetries };
  },
};
const env = ({ cwd = workspace }: { readonly cwd?: string }) =>
  new NodeExecutionEnv({ cwd });
""
await root.configure({ cwd: join(workspace, "app") }, context);
userSettings.parallelTools = false;
```

The `Harness.open()` call and the first turn are elided (...). The first `pwd` prints the workspace, the second `workspace/app` (run 26). Neither change restarted anything.

Both changes apply from the next use. Settings are re-read for every decision; a missing field takes its default `src/harness/agent.ts resolveSettings`. The `cwd` change is a commit to `pi.agent`, and the next tool call's `env()` sees it. One consequence: the environment is built per call, so a tool rerun after a crash runs in the directory the conversation has *then*, which may not be the one its first attempt used `src/harness/tool.ts:253`.

Plan mode is a change to the agent

Example 27 makes a conversation read-only without touching the registry. A `plan` extension brings a `submit_plan` tool and a `plan_mode` prompt section. It keeps the plan in its own document, `app.plan`, which is rewindable, so a fork keeps the plan it had at the fork point (defining a document (p. 50)). Entering and leaving plan mode are two `configure()` calls.

entering and leaving plan mode `test/examples/27-plan-mode.ts`

TS

```
const read = createReadTool();
const enterPlanMode = { extensions: { add: [Plan] }, tools: [read, submitPlan] };
const leavePlanMode = { extensions: null, tools: null };

await root.configure(enterPlanMode, context);
const input = { type: "input", content: "Make the port configurable." } as const;
await (await root.submit(input, context)).wait(context);
await root.configure(leavePlanMode, context);
```

Lines 60–62 and 93–98 of the example; the submission is pulled into `input` to fit the page.

```
$ node --conditions=source --experimental-strip-types test/examples/27-plan-mode.ts
tools: [ 'read', 'write', 'edit', 'bash' ]
plan mode tools: [ 'read', 'submit_plan' ]
plan: [ 'Read PORT from the environment', 'Default to 3000' ]
tools again: [ 'read', 'write', 'edit', 'bash' ]
system: { sections: { plan_mode: '<plan_mode>\n...' },
  added: [ 'read', 'submit_plan' ], ... }
system: { sections: { plan_mode: null },
  added: [ 'write', 'edit', 'bash' ], removed: [ 'submit_plan' ] }
```

Run 27, first `system` line shortened. The model saw each switch as one small system-prompt change, not a new prompt (the system prompt (p. 115)).

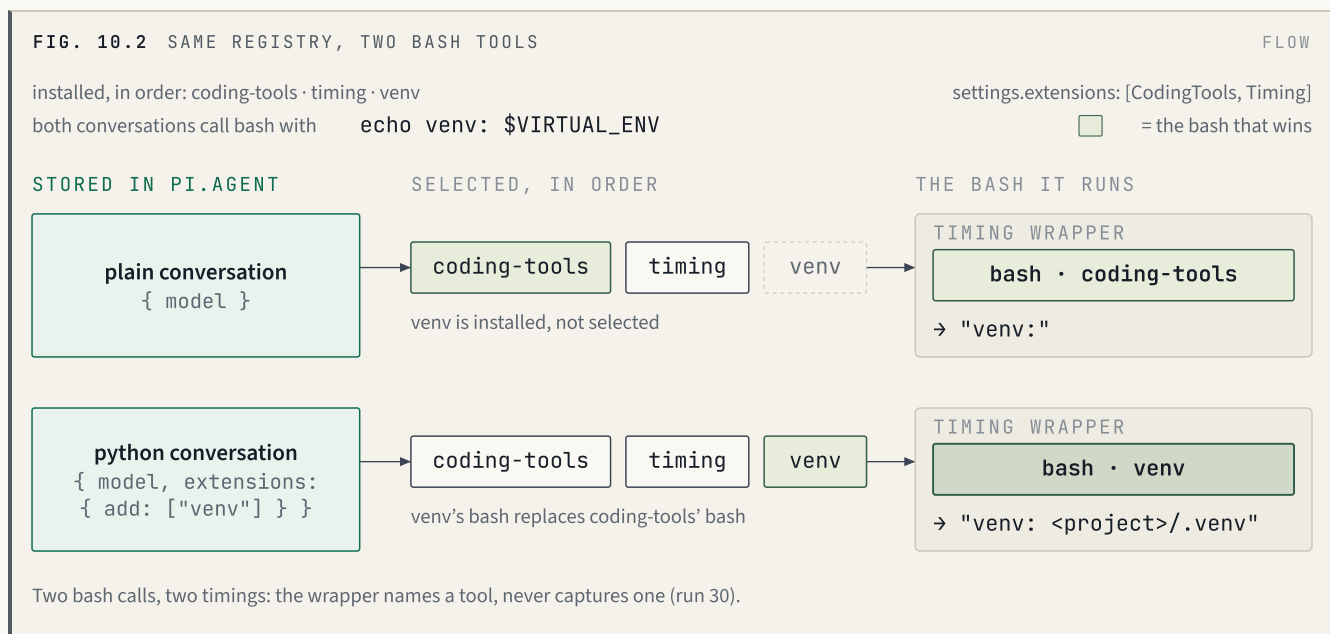
Three details make it work. `{ add: [Plan] }` adds to the host's default extensions, and the `tools` array offers exactly those tools, in that order. Setting both to `null` drops the override, so the conversation follows the host defaults again. And `submit_plan` returns `control: { terminate: true }`: when every tool result in a round asks for that, the run ends without another model request `README §Tools`.

A reviewer and a sandbox: the conversation is the unit

Example 28 adds a reviewer as a second conversation with its own agent: a cheaper model, only the `read` tool, and its own checkout as `cwd`. Its `reviewer` extension has an `onYield` hook. While an answer lacks “No further findings.”, the hook returns `{ continue: ... }`, which adds a user message and keeps the run going (hooks (p. 109)). All of this is stored in `pi.agent`, so after a restart the reviewer is still a reviewer.

Example 29 gives each user's conversation its own sandbox folder. An `app.sandbox` document holds the path, written when the conversation is created. The `env` function reads it and builds the environment from it. The document's fork policy is `initial`, so a fork starts with no path; `env` then returns `undefined`, and the built-in tools fail with “No execution environment is configured” `src/tools/env.ts`. A subagent does not inherit the document either: its creator must write one if the child should share the sandbox `spec §2.2`.

Replacing and decorating a tool



Selection order decides which same-name tool wins, and a wrapper decorates the winner. Both conversations share one registry. Values from ex 30 and run 30; the project path is shortened to `<project>`.

Example 30 needs a `bash` that activates a Python `virtualenv`, but only in some conversations. The `venv` extension carries its own tool named `bash`. Tools are collected by name in selection order, so a later extension's `bash` replaces an earlier one. Wrappers apply after that `src/harness/agent.ts resolveAgent`, so the `timing` wrapper times whichever `bash` won. A wrapper refers to a tool by name, so reloading either `bash` keeps it wrapped.

When a change arrives

Each input has its own moment. The table answers the first question a product team asks: if I change this now, what sees it?

CHANGE	READ WHEN	TAKES EFFECT
a settings getter	every use	at the next decision; nothing is written
<code>configure()</code> model, tools, sections	before each model request	at the next request; the current one keeps its prompt
<code>configure({ cwd })</code>	<code>env()</code> at each use	at the next tool call, even one the model already made
installing or replacing an extension	each phase start, request and call	at the next phase; a running call finishes on the old code
a process restart	after you reinstall	stored names bind to the newly installed code

From spec §2.2, README §Per-Conversation Agent and §Reload, and run 31: a call running during a reload prints `v1`, the next prints `v2`, and a restarted process offers no tools until it installs the extension (reload while running (p. 118)).

WHAT THIS MEANS FOR YOU

- Put code in the registry, shared policy in settings, and per-conversation choices in `pi.agent`.
- Anything else a conversation must remember across restarts goes in your own document.
- Install every extension before reopening; stored names are useless without their code.

Sources: ex 26–31; run 26–31; `src/harness/agent.ts` (`resolveSettings`, `resolveAgent`, `selectExtensions`); `src/harness/registry.ts`; `src/harness/tool.ts`:253; `src/tools/index.ts`; `src/tools/bash.ts`; `src/tools/env.ts`; spec §2.2, §7.1; README §Tools, §Per-Conversation Agent, §Settings, §Environment, §Reload

Contracts that bite

Pi Durable does not police its own rules; break one and nothing throws, until load, concurrency or a crash makes the failure show.

Suppose an extension awaits a network call inside a commit. Nothing throws. Every other write in the process, including the Harness's own, quietly waits behind it. The spec lists 38 such rules as “API footguns” and says: “These are contracts, not invitations to add defensive machinery”. This section groups them by where they bite, with the symptom and the fix. Use it as a review checklist for extension and host code.

Most rules follow from two facts. All writes in a process go through one queue, the *mutation line*, one commit at a time. And other code sees only committed state (the core rule (p. 16)). Code that holds the queue too long, writes around the Harness, or edits a value it was only meant to read breaks one of them.

Transactions and storage

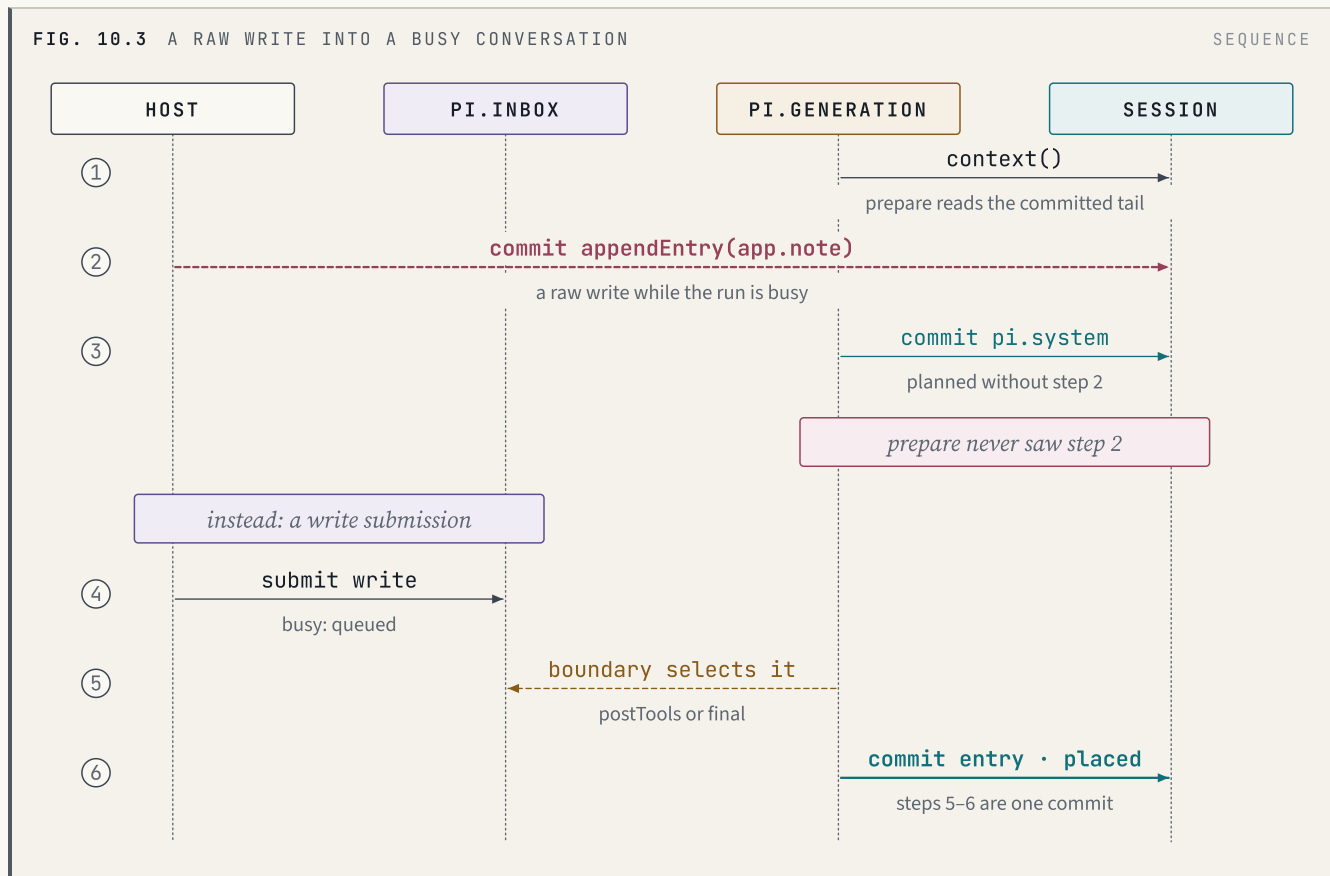
Treat a commit callback as a quick edit. It may be async, but every `await` inside it keeps the queue blocked, and every other commit waits.

RULE	BREAK IT AND...	DO INSTEAD
Keep commits short (2.2 (p. 29))	awaiting a model, tool, process, network call, person or another commit inside the callback stalls or deadlocks every write	use only the Tx you were given; do slow work between commits
No work left running after the callback (2.2 (p. 29))	once the callback returns, its drafts and Tx reject; un-awaited work before that is unsupported	await everything the callback starts
Read before you write (2.2 (p. 29))	reading a table after the first table write throws <code>ReadAfterWrite</code>	read every row first; document drafts stay usable
Create documents explicitly (4.2 (p. 53))	reading a document that was never created returns <code>undefined</code>	create it with <code>tx.doc()</code>
Family seeds apply once (4.2 (p. 53))	a seed passed for an existing family member is ignored	change the member with a draft instead
Storage failures are fatal (2.3 (p. 33))	after a failure where the write may or may not have landed, every later commit throws	do not catch and continue; reopen
JSONL needs <code>fsync</code> to survive power loss (9.3 (p. 148))	a confirmed commit may be lost if the machine goes down	pass <code>{ fsync: true }</code> where that matters

The fatal case deserves a sentence. If storage fails in a way that leaves it unknown whether the write landed, the in-memory state could drift from what a reopen will read. So the Session stops: it rejects every later commit and must be reopened, and the reopen reads whatever storage actually holds (when a commit fails (p. 33)). A `StorageRejected` error is the harmless case: storage refused the write before touching anything, the commit rolls back, and the Session carries on.

Runs, the inbox and progress

Only the Harness may add entries to a conversation while a run is active. The figure shows why. Before each model request, the Harness reads the end of the transcript, works out which system-prompt changes the model needs, and then commits them `src/harness/generation.ts prepare`. An entry you add in between is not part of that plan, but the changes land after it. A write submission avoids this: it waits for a *boundary*, a pause point after a round of tool calls or at the final answer.



A raw entry can land between the read and the commit of request preparation; a write submission waits for a boundary instead. Steps 1–3 follow the prepare phase of `src/harness/generation.ts`; steps 4–6 follow spec §6.

the same note, raw and as a submission README §Busy Conversations

TS

```

// Unsafe while a run may be active: lands wherever the queue puts it.
await root.commit((tx) =>
  tx.appendEntry(root.id, { kind: "app.note", data: "user opened a file" }), context);

// Safe: added at once when idle, otherwise at the next boundary.
await root.submit(
  { type: "write", entry: { kind: "app.note", data: "user opened a file" } }, context);
  
```

The second form is the README's own example. A write submission never starts a model turn.

RULE	BREAK IT AND...	DO INSTEAD
Don't write into a busy conversation (6.1 (p. 84))	raw entries can land among the run's system-prompt entries	use a write submission
Don't move the head into the past by hand (3.4 (p. 46))	the model's context changes, but an open view keeps showing old entries until rebuilt	use a write submission, which rejects it as stale
A failed run leaves its queue (6.1 (p. 84))	queued follow-ups stay in <code>pi.inbox</code> and their <code>wait()</code> never settles	submit again, or withdraw them
Progress is committed in steps (6.2 (p. 88))	a crash loses streamed text since the last successful progress commit; 100 ms is the default minimum interval, not a maximum loss window	tune <code>settings.progress</code> ; never show uncommitted text
Keep task results small (5.1 (p. 63))	finished task records stay in storage, so big results stay too	store the data in entries or documents; return their IDs
The transcript is not the model's context (3.2 (p. 40))	raw entries miss edits, display-only entries and filtering	use <code>context()</code> for what the model sees

Compaction

Compaction replaces old turns with a summary ([compaction \(p. 97\)](#)). The summary is written some time after it picks its range of entries. Anything that changes that range in between is at risk.

RULE	BREAK IT AND...	DO INSTEAD
Compaction costs a model call	every attempt pays for one summary request, even when the summary arrives too late to use	budget for it; it shows in <code>pi.usage</code>
Edits during compaction	an edit placed while the summary is written is lost if the summary covers its target; a head placed meanwhile is undone by the cut	make such writes before compacting
Summary timestamps	a queued summary carries the time its compaction finished, not when it was placed	judge freshness by entry order, as the Harness does

Ownership, hooks and close

Owned work forms a tree. A task cannot finish while work it owns is still running, and an abort reaches the children before any handler runs ([ownership and structured concurrency \(p. 76\)](#)).

One ownership rule is easy to break, because the failing commit looks reasonable: a task that creates a child task in the same commit that finishes it faults. In the capture, an order task charges a card, packs, and then creates a receipt task while finishing.

a child created in the finishing commit `research/capture/task-transitions.txt`

TEXT

```
seq= 25 task 13 shop.pack owner=12 terminal outcome={"status":"completed",...}
seq= 26 task 12 shop.order owner=conv running checkpoint={"phase":"decide",...}
seq= 27 task 12 shop.order owner=conv terminal outcome={"status":"faulted",
    "error":{"message":"Task owner 12 is completing"}}
effects in order: ["charged charge-d"]
```

Scenario D, timestamps and memos cut (...). The receipt task was never stored, and the charge was never refunded, because a fault runs no abort handler. Create children in an earlier commit; the finishing task then waits as `completing` (spec §5.5).

RULE	BREAK IT AND...	DO INSTEAD
Owned work holds its owner (5.5 (p. 76))	the owner stays <code>completing</code> ; a subagent's work holds up the tool call and the run	create work that must not hold anything as background
Work started by hooks (7.3 (p. 109))	it holds up the task that ran the hook; code reacting after the result cannot extend it	create it before the task records its result
No children from abort handlers (5.6 (p. 80))	an aborting task cannot create owned children	undo effects where they happened, or with background tasks
Hooks share memo names (7.3 (p. 109))	hooks and their task use one memo namespace, so names collide	prefix memo names
Count subagent spend once (6.6 (p. 100))	a tool that reports its child's usage is counted twice in totals	let the child's <code>pi.usage</code> count it
Honour abort signals (2.3 (p. 33))	code that ignores its signal keeps <code>close()</code> waiting and storage open	pass <code>context.abortSignal</code> to every await

Documents and watches

Document kinds and fork policies are a stored format: they outlive the code that defined them. Values you read are shared and must not be changed, and a watch buffers only so many updates (the conversation view (p. 129)).

RULE	BREAK IT AND...	DO INSTEAD
Store full copies sometimes (4.3 (p. 56))	if <code>checkpointWhen()</code> never returns true, storage grows without bound	return true now and then
Pick fork policies on purpose (4.1 (p. 50))	<code>current</code> , <code>initial</code> and <code>asOf</code> change what a fork sees	choose by what the product means
Kinds are permanent (4.3 (p. 56))	a migration cannot rename a kind; two definitions claiming one kind go unnoticed	to rename, copy to a new kind and retire the old
<code>pi.</code> names are reserved (3.1 (p. 37))	nothing enforces it; reusing one collides with the Harness	use your own prefix, such as <code>app.</code>
Read values are immutable (8.1 (p. 125))	changing a snapshot or watch value corrupts shared state	copy before changing
Start watches after reading (8.2 (p. 129))	before <code>start()</code> , <code>watch.value</code> is the value at the time you acquired it	initialize from it, then start
Watches can skip updates (8.2 (p. 129))	past 100 waiting updates, one full value replaces them; steps in between are lost	count events from entries or a log
<code>stop()</code> does not wait (8.2 (p. 129))	it neither aborts nor waits for a callback in progress	coordinate with your callback yourself
Close clients before the Harness (8.1 (p. 125))	a state ended by close keeps its last value forever	detach clients and services first

The watch rules matter most for remote clients. A client that mirrors the latest value stays correct when updates collapse, because the replacement carries the whole newest value. A client that counts transitions, such as “notify on every tool start”, does not: under load it sees the end state without the steps. Anything that must be counted belongs in an entry or a log the consumer reads.

Extensions and the agent

An extension is selected whole or not at all, so a guard runs only where it is selected. A child conversation copies its selection once, at creation (the per-conversation agent (p. 106)).

RULE	BREAK IT AND...	DO INSTEAD
Include guards in every list	an explicit extension list that leaves out a permissions extension runs without it	include guards in every list
Edits replace copied lists	<code>{ add, remove }</code> on a child that copied a list starts from the host default instead	build the full list from the resolved agent
Copies happen once	later changes to the parent never reach a task-owned child	configure the child directly
Reruns use today's directory (9.5 (p. 155))	a rerun after a crash builds its environment from the current <code>cwd</code>	do not assume the first attempt's directory
Default changes cost cache (7.5 (p. 115))	changing settings or installs adds <code>pi.system</code> entries and misses the prompt cache	expect one system change and one cache miss per change
Keep prompt text stable (7.5 (p. 115))	a section that embeds the time changes every request and defeats caching	render only real changes
Dispose late (7.6 (p. 118))	freeing resources at uninstall breaks calls still running	dispose after running calls end

Rows paraphrase spec §12 in its order within each group; “Do instead” is the spec’s own fix where it gives one.

NO SAFETY NET

Almost none of this is checked at runtime. The registry only checks section keys and that tool names are unique within one extension `src/harness/registry.ts`. Reserved names, kind collisions and immutability are up to your tests and review.

WHAT THIS MEANS FOR YOU

Three questions catch most of these in code review. A yes to any is a bug, even if every test passes, because it shows only under load, concurrency or a crash.

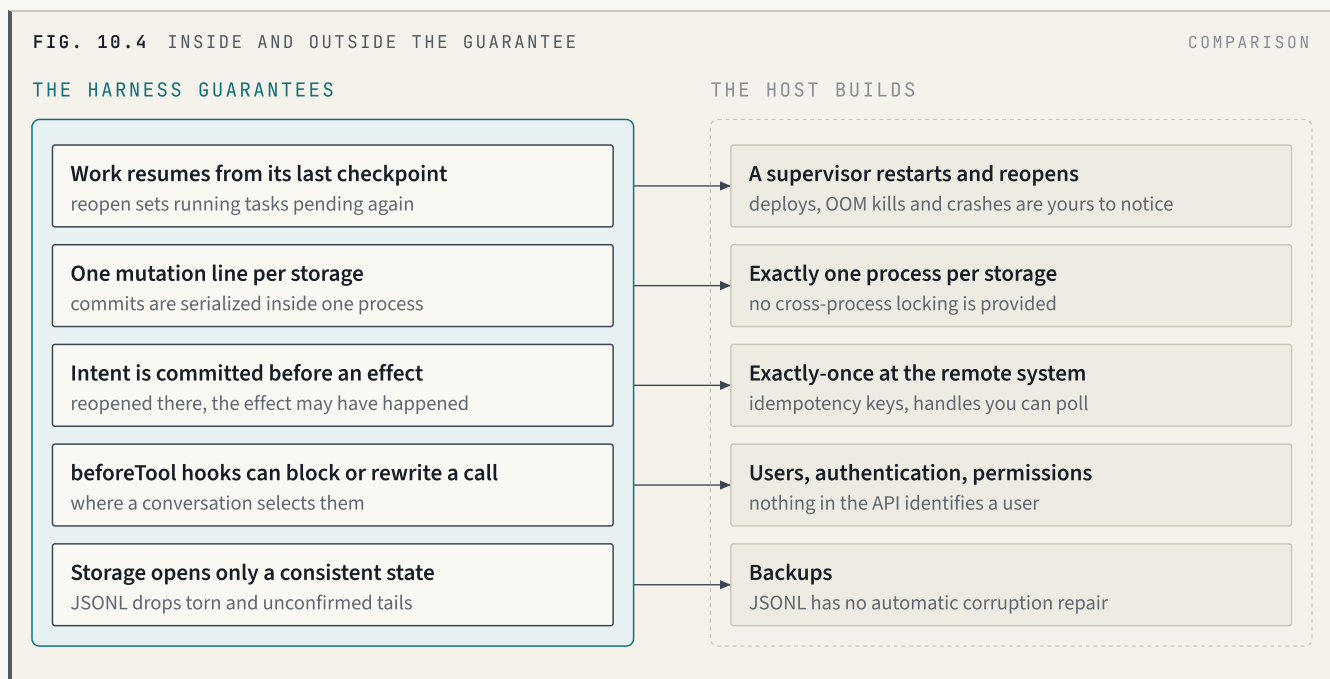
- Does any `await` inside a commit callback leave the process?
- Does any code write to a conversation whose run it does not own?
- Does any consumer change a value it read, or depend on seeing every update?

Sources: spec §12, §6, §4; `src/harness/generation.ts` (`prepare`); `src/harness/registry.ts` (`validateExtension`); `src/session/observation.ts` (`MAX_PENDING_WATCH_FRAMES`); README §Busy Conversations; `test/harness-inbox.test.ts`; `test/harness-view.test.ts`; `research/capture/task-transitions.txt` and `NOTES.md` §6

Limits and non-goals

Pi Durable guarantees what happens inside one process and one storage; supervision, identity, backups and remote exactly-once are the host's to build.

Pi Durable preserves committed internal state across process crashes. External effects can still run twice; tool replay policy and the remote system's safeguards decide that. It does not restart your crashed process, stop two processes from opening the same file, or know who your users are. This section draws that line. After it you can list what a production deployment must add around a Harness, and explain to a Pi coding-agent user what changes when sessions move to Pi Durable.



The Harness guarantees what is committed in its storage; the host owns everything around it. Each row pairs a guarantee with the work it leaves. From spec §2.2, §5.2, §11.3, §13, README §Storage, and src/harness/types.ts.

What the spec rules out

The spec ends with eleven things the system “initially has no”: scope decisions, not gaps to work around. The table lists them all. Two shape everyday code:

- *No visible-undurable publication.* A UI never shows text that storage does not hold, so streaming is as smooth as the progress commits, 100 ms apart by default (6.2 (p. 88)).
- *No forced termination.* A tool that ignores its abort signal cannot be stopped in-process; `c\lose()` waits for it. Pass the call's context to every await.

NON - GOAL	WHAT IT MEANS IN PRACTICE
whole-Session DOM	no single observable tree of everything; observe a conversation view, a document or the task graph (8.2 (p. 129))
visible-undurable publication	no fast path for uncommitted progress; partials appear at commit cadence (6.2 (p. 88))
Session-kernel semantic event journal	agent events are derived from commits, not stored (8.3 (p. 133))
session-scoped rewindable documents	rewindable exists only for conversation scope (4.1 (p. 50))
automatic checkpoint heuristic	a document stores a base only when <code>checkpointWhen()</code> says so (4.3 (p. 56))
automatic third-party view mounting	the view mounts the built-in documents only; read yours with <code>snapshot()</code> , <code>documentState()</code> or <code>watchDoc()</code>
CRDT/offline multi-writer merge	one writer per storage; no merge of concurrent histories
SQL translation of Chord operations	SQLite stores operations as records, not as SQL updates (9.2 (p. 144))
JSONL global compaction or repair	<code>main.jsonl</code> only grows; corruption fails the open (9.3 (p. 148))
compatibility with removed Pico prototypes	no migration path from earlier internal prototypes
forced termination of extension code	a handler that ignores its signal cannot be killed in-process; close waits for it

One storage, one owner

Only one process may use a storage at a time. “One process owns a storage at a time; there is no cross-process locking” [README §Storage](#). Nothing stops a second process from opening the same file, and each would then believe it is the only writer. The SQLite option `busyTimeoutMs` (default 5,000 ms) only makes SQLite wait for a file lock `src/storage/sqlite/node.ts`; it does not make two Harnesses safe on one file.

So crash recovery is your supervisor’s job. Something must notice the process died, start a new one, and reopen the storage. Reopening does the rest: tasks that were running go back to `pending`, and the next `resume()`, submission or wait starts them again (the scheduler (p. 73)). The same reopen handles a planned handover: once `close()` resolves, “a new Harness may open the same Storage” [spec §2.2](#).

Opening is itself a write. `Harness.open()` commits the reset of running tasks even if nothing resumes afterwards [research/capture/crash-reopen-tasks.txt](#), so a dashboard must not “just open” a storage another process owns. For SQLite, a separate read-only connection can read the live database safely; the capture kit read its mid-run rows that way [research/capture/sqlite-rows-midrun.txt](#).

Install every extension before you reopen. A task whose code is missing is not failed; it waits, blocked, until the registry can run it (5.4 (p. 73)).

Effects belong to the systems that perform them

The Harness can commit the intent to charge a card. It cannot make the charge happen exactly once. A tool call commits its intent, then runs, then commits its result. If the process dies in between, “the effect may have happened” [spec §5.2 \(the effect sandwich \(p. 70\)\)](#). Only the remote system can settle that: with an idempotency key it honours, or a handle you can ask about. A tool without `replay`: “safe” is not rerun at all; the model gets an `interrupted` error result instead (tool calls and [replay \(p. 94\)](#)). Nothing is rolled back automatically. Undoing an effect is code you write, in the abort handler of the task that did it.

Original Pi sessions and Pi Durable

Pi Durable “does not replace the Pi coding agent. It is a framework for building any agentic application, coding agents included” earendil.com/posts/pi-durable. The coding agent keeps its own session files. The table compares the two record models for a reader who knows the first (what Pi Durable is for [\(p. 10\)](#)).

ASPECT	PI CODING AGENT SESSION	PI DURABLE
Storage	one JSONL file per session under <code>~/ .pi/agent/sessions/</code>	Memory, one SQLite file, or a JSONL directory with sidecars
History shape	a tree in one file; the leaf is the current position	conversations; a fork is a new conversation capped at <code>parent.at</code>
Branching	<code>/tree</code> in place; <code>/fork</code> , <code>/clone</code> write new files	<code>fork(entryId)</code> in the same storage, with document fork policies
Write unit	one appended line per entry, file first written once a user or assistant message exists	one atomic commit across entries, tasks, submissions and documents
System prompt	system messages patching sections, <code>toolsAdded/toolsRemoved</code>	<code>pi.system</code> entries with the same positional contract
Compaction	<code>compaction</code> entry: <code>summary</code> , <code>firstKeptEntryId</code> , system checkpoint	<code>pi.compaction</code> entry whose head is the first kept entry
Context edits	<code>context_edit</code> entry with <code>targetId</code> and <code>replacement</code>	edits on an entry: <code>omit</code> or <code>replace</code>
Model and thinking	<code>model_change</code> , <code>thinking_level_change</code> entries	fields of the rewindable <code>pi.agent</code> document
Extension state	<code>custom</code> entries with <code>customType</code> and <code>data</code>	typed documents with scope, history and fork policy
Queued messages	steering and follow-ups in memory; abort returns them to the editor	durable submissions in <code>pi.inbox</code> , deduplicated by <code>requestId</code>
Work in flight	no entry type records a running response or tool call	<code>pi.generation</code> and <code>pi.tool</code> tasks with checkpoints

Left: coding-agent docs and `src/core/session-manager.ts`. Right: `src/types.ts`, `src/entries.ts` and spec §2–§8.

The two designs share the transcript ideas: append-only entries, positional system deltas, summaries that keep the original entries. They differ in what a write means. A Pi session line records that something happened. A Pi Durable commit records state that is allowed to be seen, including work that has not finished yet.

What to build yourself

Supervision — restart on crash, deploy and OOM; guarantee one process per storage, for example one storage per process slot or per tenant.

Identity and permissions — the API has no user, account or credential type. Guards are extensions with `beforeTool` hooks, effective only where a conversation selects them ([hooks \(p. 109\)](#)).

Backups — a reopened JSONL store drops half-written lines at its end, but “missing required confirmed data is corruption and opening fails” [spec §11.3](#). SQLite may lose the newest commit on power or host failure [README §Storage](#). Nothing repairs a damaged store.

API churn — the package is experimental: “The API changes without notice between releases” [README](#). Version 1.0.3 already broke custom `FileSystem` and `Shell` implementations [CHANGELOG 1.0.3](#). Pin the version and run the conformance suites after upgrades ([9.4 \(p. 152\)](#)).

Sources: [spec §2.2](#), [§5.2](#), [§11.3](#), [§13](#); [README §Storage](#); [CHANGELOG 1.0.3](#); [research/capture/crash-reopen-tasks.txt](#); [coding-agent docs/session-format.md](#), [src/core/session-manager.ts](#); earendil.com/posts/pi-durable

Reference

The API surface, every built-in kind, settings and defaults, errors, a glossary, the sources, and every figure in the book.

R.1	Harness and Conversation API	172
R.2	Built-in kinds	179
R.3	Settings and defaults	180
R.4	Errors and reasons	183
R.5	Glossary	185
R.6	Sources	187
R.7	Index of figures	189



R . 1

Harness and Conversation API

Every public entry point of the package root and its subpaths, with what it returns and where the book explains it.

Use these tables to find a method and the section that explains it. Signatures are shortened: `ctx` is the cancellation `Context` that every async method takes, and type parameters are dropped. The source of truth is `src/index.ts` and the files it re-exports.

Package root: functions and classes

EXPORT	SIGNATURE	DOES	SEE
<code>Harness.open</code>	<code>(storage, options, ctx)</code>	open a Harness on a storage; throws if the registry lacks the built-in tasks	1.2 (p. 13)
<code>createRegistry</code>	<code>()</code>	a new registry; it already holds the built-in tasks	7.1 (p. 103)
<code>createSession</code>	<code>(storage)</code>	a bare Session: storage and commits, no agent machinery	2.2 (p. 29)
<code>MemoryStorage</code>	<code>new MemoryStorage()</code>	in-memory storage, lost on exit; good for tests	9.4 (p. 152)
<code>defineDoc</code>	<code>(definition)</code>	declare a document kind (one per owner)	4.1 (p. 50)
<code>defineDocFamily</code>	<code>(definition)</code>	declare a document kind with many keyed members	4.2 (p. 53)
<code>defineEntry</code>	<code>(kind)</code>	declare an entry kind, with an <code>is()</code> type check	3.1 (p. 37)
<code>defineTask</code>	<code>(definition)</code>	declare a durable task; register it through an extension	5.1 (p. 63)
<code>defineExtension</code>	<code>(extension)</code>	declare an extension (adds types only)	7.1 (p. 103)
<code>defineTool</code>	<code>(tool)</code>	declare a tool; args are typed from parameters	7.4 (p. 112)
<code>section</code>	<code>(key, render, { tag }?)</code>	declare a system-prompt section; wrapped in <code><key></code> tags unless <code>tag: false</code>	7.5 (p. 115)
<code>hook</code>	<code>(task, handlers)</code>	attach handlers to a built-in or custom task, matched by name	7.3 (p. 109)
<code>wrapTool</code>	<code>(tool, wrapper)</code>	wrap the tool with that name wherever the extension is selected	7.4 (p. 112)
<code>wrapSection</code>	<code>(key, wrapper)</code>	wrap the section with that key	7.5 (p. 115)
<code>configure</code>	<code>(tx, conversationId, change)</code>	change a conversation's agent inside a commit you already have	7.2 (p. 106)
<code>watchEvents</code>	<code>(harness, conversationId, ctx)</code>	stream of agent events (runs, turns, messages, tool calls); experimental	8.3 (p. 133)
<code>ReadAfterWrite, StorageRejected, ConversationBusy</code>	error classes	see errors and reasons (p. 183)	2.3 (p. 33)

Package root: tokens and constants

EXPORT	KIND	VALUE	SEE
UserEntry, AssistantEntry, SystemEntry	entry tokens	<code>pi.user</code> , <code>pi.assistant</code> , <code>pi.system</code>	3.1 (p. 37)
ToolResultEntry, ResetEntry, CompactionEntry	entry tokens	<code>pi.tool-result</code> , <code>pi.reset</code> , <code>pi.compaction</code>	3.1 (p. 37)
AgentDoc, ProviderDoc, LiveDoc	document tokens	<code>pi.agent</code> , <code>pi.provider</code> , <code>pi.live</code>	4.4 (p. 60)
InboxDoc, UsageDoc	document tokens	<code>pi.inbox</code> , <code>pi.usage</code>	4.4 (p. 60)
GenerationTask, ToolTask, CompactionTask	task tokens	<code>pi.generation</code> , <code>pi.tool</code> , <code>pi.compaction</code>	R.2 (p. 179)
ROOT_CONVERSATION_ID	constant	1	2.1 (p. 25)
DEFAULT_RETRY_POLICY	constant	see settings and defaults (p. 180)	6.3 (p. 91)
DEFAULT_COMPACTIION_POLICY	constant	see settings and defaults (p. 180)	6.5 (p. 97)
DEFAULT_PROGRESS_POLICY	constant	see settings and defaults (p. 180)	6.2 (p. 88)

Subpath exports

IMPORT PATH	MAIN EXPORTS	SEE
.../env	ExecutionEnv, FileSystem, Shell, FileError, ExecutionError, line-scan helpers	9.5 (p. 155)
.../env/node	NodeExecutionEnv	9.5 (p. 155)
.../tools	CodingTools, createReadTool, createWriteTool, createEditTool, createBashTool	7.4 (p. 112)
.../storage/memory	MemoryStorage	9.4 (p. 152)
.../storage/sqlite	SqliteStorage, SqliteDatabase, applySqliteMigrations, SQLITE_MIGRATIONS	9.2 (p. 144)
.../storage/sqlite/node	openNodeSqliteStorage, openNodeSqliteDatabase	9.2 (p. 144)
.../storage/jsonl	JsonlStorage, JsonlCorruptionError, JsonlStoragePoisonedError	9.3 (p. 148)
.../storage/jsonl/node	openNodeJsonlStorage(directory, ctx, { fsync }?)	9.3 (p. 148)
.../testing	registerStorageConformance, registerEnvConformance, createStorageConformance, createEnvConformance	9.4 (p. 152)

... stands for [@earendil-works/pi-durable](#). From `package.json exports` and each subpath's index file.

Harness

METHOD	SIGNATURE	RETURNS	SEE
resume	()	start running tasks; safe to call twice; throws after close	5.4 (p. 73)
root	(ctx, { agent, init }?)	the root conversation, created on first use	3.3 (p. 43)
conversation	(id, ctx)	Conversation undefined	3.3 (p. 43)
createConversation	({ ownership, agent?, init? }, ctx)	Conversation, in one commit	3.3 (p. 43)
getTask	(id, ctx)	TaskRecord undefined	5.1 (p. 63)
inspect	(ctx)	HarnessInspection: unfinished tasks and submissions; writes nothing	8.4 (p. 137)
submission	(id, ctx)	Submission undefined; works after a reopen	6.1 (p. 84)
abortSubmission	(id, ctx, conversationId?)	"aborted" "already_placed" "settled" "not_found"	6.1 (p. 84)
abortTask	(id, ctx)	"marked" "terminal"	5.6 (p. 80)
waitForTask	(id, ctx)	SettledTask, once the task has finished	5.1 (p. 63)
waitForIdle	(ctx)	resolves when no foreground work is left anywhere	5.5 (p. 76)
usage	(ctx)	UsageState: token and cost totals of all conversations	6.6 (p. 100)
taskGraph	(ctx)	live, read-only tree of unfinished tasks; dispose when done	8.4 (p. 137)
watchTaskGraph	(ctx)	TaskGraphWatch	8.4 (p. 137)

Session methods (inherited by Harness)

METHOD	SIGNATURE	RETURNS	SEE
commit	(change: (tx) => T, ctx)	T, once the commit is stored	2.2 (p. 29)
close	(ctx)	stop new work, wait for running code, close storage	2.3 (p. 33)
subscribeCommits	(listener)	unsubscribe function; listener sees every commit, synchronously	2.2 (p. 29)
subscribeClose	(listener)	unsubscribe function; called when close begins	2.2 (p. 29)
snapshot	(token, owner?, key?, ctx)	current committed value, read-only, or undefined	4.2 (p. 53)
snapshotAsOf	(token, conversationId, key?, at, ctx)	value as of an entry; rewindable documents only	4.3 (p. 56)
documentState	(token, owner?, key?, ctx)	live read-only value that follows commits; dispose when done	8.1 (p. 125)
watchDoc	(token, owner?, key?, ctx)	DocumentWatch undefined: one callback per change	8.1 (p. 125)

The owner argument is a conversation ID or task ID by the token's scope, absent for Session scope; key applies to families.

Conversation

METHOD	SIGNATURE	RETURNS	SEE
<code>id</code>	property	<code>ConversationId</code> ; compare conversations by it	3.3 (p. 43)
<code>agent</code>	(ctx)	the <code>Agent</code> it would run with now; writes nothing	7.2 (p. 106)
<code>configure</code>	(change, ctx)	change model, tools, extensions or cwd, in its own commit	7.2 (p. 106)
<code>submit</code>	(draft, ctx)	<code>Submission</code> , once the input is committed	6.1 (p. 84)
<code>reset</code>	(handoff?, ctx)	start a fresh context, optionally with a handoff message	3.4 (p. 46)
<code>compact</code>	(instructions?, ctx)	<code>TaskId<CompactionResult></code> of a summary task	6.5 (p. 97)
<code>commit</code>	(change, ctx)	a commit where new tasks default to this conversation	2.2 (p. 29)
<code>context</code>	(ctx)	<code>ContextView</code> : active entries and the messages the model will see	3.2 (p. 40)
<code>entries</code>	(query, limit, cursor, ctx)	one page of entries, newest first, including inherited ones	3.1 (p. 37)
<code>fork</code>	(at, { ownership, agent?, init? }, ctx)	the new <code>Conversation</code>	3.3 (p. 43)
<code>abort</code>	(ctx, { background }?)	stop running work and withdraw queued inputs; resolves when idle	5.6 (p. 80)
<code>waitForIdle</code>	(ctx)	resolves when no foreground work is left	5.5 (p. 76)
<code>viewState</code>	(ctx)	live read-only <code>ConversationView</code> for a UI	8.2 (p. 129)
<code>watch</code>	(ctx)	<code>ConversationWatch</code> : one callback per change	8.2 (p. 129)

Submission and ConversationHandle

MEMBER	SIGNATURE	RETURNS	SEE
<code>Submission.id</code>	property	<code>SubmissionId</code> ; keep it to find the submission after a restart	6.1 (p. 84)
<code>Submission.status</code>	(ctx)	the stored record, now	6.1 (p. 84)
<code>Submission.wait</code>	(ctx)	the record, once done or unanswered	6.1 (p. 84)
<code>Submission.abort</code>	(ctx)	"aborted" "already_placed" "settled"	6.1 (p. 84)
<code>ConversationHandle.submit</code>	(inputDraft, ctx)	submit user input from inside a task; no writes	7.7 (p. 121)
<code>ConversationHandle.abort</code>	(ctx, { background }?)	as <code>Conversation.abort</code>	7.7 (p. 121)
<code>ConversationHandle.waitForIdle</code>	(ctx)	as <code>Conversation.waitForIdle</code>	7.7 (p. 121)

A `ConversationHandle` comes from `runtime.conversation()` or `api.conversation()` and rejects after the invocation ends.

Tx, the commit callback's surface

METHOD	SIGNATURE	RETURNS	SEE
conversation, entry, task	(id);entry(token, id)	the record or undefined (a table read)	2.2 (p. 29)
scanConversations	(query, limit, cursor?)	one page of conversations, filtered by owner	7.7 (p. 121)
scanEntries	(query, limit, cursor?)	one page of entries, newest first	3.1 (p. 37)
latestHeadMarker	(conversationId)	the entry that set the current context start	3.4 (p. 46)
scanTasks	(query, limit, cursor?)	one page of task records	5.1 (p. 63)
submissionByRequest	(conversationId, requestId)	record or undefined	6.1 (p. 84)
createConversation	({ ownership })	ConversationRecord; in a Harness, also creates the built-in documents	3.3 (p. 43)
forkConversation	(parentId, at, { ownership })	ConversationRecord	3.3 (p. 43)
appendEntry	(conversationId, draft) or (token, ...)	EntryRecord (a table write)	3.1 (p. 37)
createTask	(task, input, options)	TaskId (a table write)	5.1 (p. 63)
createSubmission	(create)	raw record, skipping the busy and queue rules	6.1 (p. 84)
settleSubmission	(id, settlement)	mark it done or unanswered	6.1 (p. 84)
placeSubmission	(id, entry)	record where it landed: input placed, write done	6.1 (p. 84)
doc	(token, owner?, key?, seed?)	an editable draft; creates the document if absent	4.2 (p. 53)
retireDoc	(token, owner?, key?)	end the document; a later doc() creates a new one	4.2 (p. 53)

TaskRuntime, one task invocation

MEMBER	SIGNATURE	RETURNS	SEE
taskId, conversationId	properties	identity of the task and its conversation	5.1 (p. 63)
signal	property	AbortSignal: fires on abort, close, or when this run of the handler ends	5.6 (p. 80)
registry	property	the registry as it was when this phase started	7.6 (p. 118)
settings, models	properties	settings re-read on each access; pi-ai Models	7.2 (p. 106)
agent	(ctx)	the conversation's Agent, fixed for this phase	7.2 (p. 106)
env	(ctx)	the conversation's environment, or undefined	9.5 (p. 155)

hooks	.each(name, invoke)	call each selected extension's handler, in order	7.3 (p. 109)
commit	((tx, current) => next?, ctx)	commit; return a new state to move the task on	5.1 (p. 63)
memo	(name, ctx), (name, candidate, ctx)	a small stored value; the first write wins	5.2 (p. 67)
getTask, waitForTask	(id, ctx)	the record; the record once finished	5.5 (p. 76)
outcomes	(ids, ctx)	results of finished tasks; rejects if one is unfinished	5.5 (p. 76)
conversation	(id, ctx)	ConversationHandle undefined	7.7 (p. 121)
entry	(id, ctx), (token, id, ctx)	a committed entry this conversation can see	3.1 (p. 37)
context	(conversationId, ctx, at?)	what the model would see, optionally as of entry at	3.2 (p. 40)
now, sleep	(); (until, ctx)	the Harness clock; wait until a time	5.2 (p. 67)
report	(error)	pass a non-fatal error to onReport	7.3 (p. 109)
snapshot, snapshotAsOf, watchDoc	as on Session	committed values; watches stop when the handler ends	8.1 (p. 125)

ToolExecutionApi, one tool call

MEMBER	SIGNATURE	RETURNS	SEE
taskId, conversationId, callId	properties	the tool task, its conversation, and the model's call ID	7.4 (p. 112)
registry	property	the registry as of this call	7.6 (p. 118)
agent	(ctx)	the calling conversation's agent	7.2 (p. 106)
env	property	built for this call, or undefined	9.5 (p. 155)
output	(chunk, skipped?)	stream output to the UI while running	7.4 (p. 112)
outputWindow	property	how much output tail is kept and how often it is committed, or undefined	9.5 (p. 155)
diagnostic	(diagnostic)	add a note the model sees with the result	7.4 (p. 112)
details	(value, ctx)	replace the structured details shown while running	7.4 (p. 112)
commit	(change, ctx)	a commit made on behalf of this tool call	7.4 (p. 112)
memo	as on TaskRuntime	a small stored value; the first write wins	5.2 (p. 67)
createTask	(task, input, options, ctx)	TaskId of a child task	5.5 (p. 76)
getTask, waitForTask	(id, ctx)	the record; the record once finished	5.5 (p. 76)
conversation	(id, ctx)	ConversationHandle undefined	7.7 (p. 121)
snapshot, snapshotAsOf, watchDoc	as on Session	committed values	8.1 (p. 125)

Registry

MEMBER	SIGNATURE	RETURNS	SEE
install	(extension)	add an extension, or replace one with the same name; takes effect at once	7.1 (p. 103)
uninstall	(extension)	remove the extension with that name	7.1 (p. 103)
snapshot	()	a frozen view of what is installed now	7.1 (p. 103)
subscribe	(listener)	unsubscribe function; called after every change	7.6 (p. 118)
snapshot installed, extension	(); (name)	extensions in install order; one by name	7.1 (p. 103)
snapshot tools, sections	()	every installed tool or section with its extension	7.1 (p. 103)
snapshot tasks, task	(); (name)	built-in and installed task definitions	5.4 (p. 73)

Sources: src/index.ts; src/types.ts (Tx, Session, TaskRuntime, Storage); src/harness/types.ts (Conversation, Harness, Submission, ConversationHandle, ToolExecutionApi, Registry); src/harness/harness.ts (Harness.open); src/harness/define.ts; src/harness/registry.ts; package.json exports; spec §2.2

R . 2

Built-in kinds

Six entry kinds, five document kinds and three task kinds carry every run; all are named with the reserved `pi.` prefix.

A kind is the name stored with each record, so it must never change. The `pi.` prefix is reserved by convention only; reusing it collides with the Harness ([contracts that bite](#) (p. 163)). The tokens that name each kind in code are listed in [R.1](#) (p. 172).

Entry kinds

KIND	MODEL	WRITTEN BY	SEE
<code>pi.user</code>	<code>[UserMessage]</code>	submissions, <code>onYield</code> continuations	3.1 (p. 37)
<code>pi.assistant</code>	<code>[AssistantMessage]</code> , any stop reason	generation	6.3 (p. 91)
<code>pi.system</code>	<code>[SystemMessage]</code> , content: ""	generation preparation	7.5 (p. 115)
<code>pi.tool-result</code>	<code>[ToolResultMessage]</code> ; data: { diagnostics }	tool tasks; generation for calls not offered	6.4 (p. 94)
<code>pi.reset</code>	absent, or <code>[UserMessage]</code> handoff; head: "self"	<code>reset()</code> , handoff control	3.4 (p. 46)
<code>pi.compaction</code>	<code>[UserMessage]</code> summary; head = first kept; data: { reason }	compaction tasks	6.5 (p. 97)

Document kinds

KIND	HISTORY · FORK	FULL COPY STORED	HOLDS	SEE
<code>pi.agent</code>	<code>rewindable · asOf</code>	every change	model, thinking level, extension and tool names, instructions, cwd	7.2 (p. 106)
<code>pi.provider</code>	<code>latest · initial</code>	every change	<code>sessionId</code> , a UUIDv7	6.3 (p. 91)
<code>pi.live</code>	<code>latest · initial</code>	when nothing runs	run, generation, tools, compactions	6.2 (p. 88)
<code>pi.inbox</code>	<code>latest · initial</code>	when empty	items: queued steers, follow-ups, writes	6.1 (p. 84)
<code>pi.usage</code>	<code>latest · initial</code>	every change	models by provider/model, tools by name	6.6 (p. 100)

All five belong to one conversation, are created with it (or its fork), and appear in its view as `docs["pi..."]`. “Full copy stored” paraphrases each `checkpointWhen`; between copies only changes are stored.

Task kinds

KIND	PHASES	RESPONSIBILITY	SEE
<code>pi.generation</code>	<code>prepare, request, retry, poll, tools</code>	build the prompt, call the model, retry, start the tool calls	6.3 (p. 91)
<code>pi.tool</code>	<code>call, execute</code>	check arguments, run hooks, commit intent, run the tool, commit the result	6.4 (p. 94)
<code>pi.compaction</code>	<code>select, summarize, retry</code>	pick old entries, summarize them, add the summary	6.5 (p. 97)

Every registry holds these three tasks; they cannot be removed or replaced.

Sources: `src/entries.ts`; `src/harness/agent.ts`, `provider.ts`, `live.ts`, `inbox.ts`, `usage.ts` (document definitions); `src/harness/generation.ts`, `tool.ts`, `compaction.ts` (phases); `src/harness/registry.ts` (BUILTIN_TASKS); spec §2.2, §8, §8.1

R . 3

Settings and defaults

Every option the host passes to `Harness.open()`, every settings key with its built-in default, and the other defaults the source hard-codes.

Settings are read each time they are needed and never stored. Leave a field out to get its default; in an object such as `retry`, each missing key gets its own default. Getters make settings live ([building a coding agent \(p. 159\)](#)).

HarnessOptions

FIELD	TYPE	DEFAULT	SEE
<code>models</code>	<code>pi-ai Models</code>	<code>required</code>	6.3 (p. 91)
<code>registry</code>	<code>RegistryReader</code>	<code>required; must hold the built-in tasks</code>	7.1 (p. 103)
<code>settings</code>	<code>HarnessSettings</code>	<code>every field at its default</code>	7.2 (p. 106)
<code>env</code>	<code>(target, ctx) => ExecutionEnv undefined</code>	<code>none: no environment</code>	9.5 (p. 155)
<code>conversationCreated</code>	<code>(tx, record) => void</code>	<code>none</code>	4.2 (p. 53)
<code>now</code>	<code>() => number</code>	<code>Date.now</code>	5.2 (p. 67)
<code>onReport</code>	<code>(error) => void</code>	<code>none; must not throw</code>	7.3 (p. 109)

HarnessSettings

KEY	DEFAULT	EFFECT	SEE
extensions	every installed extension, install order	extensions a conversation uses unless it picks its own	7.2 (p. 106)
stream	{}	pi-ai request options, below	6.3 (p. 91)
retry.enabled	true	retry failed model requests	6.3 (p. 91)
retry.maxRetries	3	retries after the first failure	6.3 (p. 91)
retry.baseDelayMs	2000	backoff base	6.3 (p. 91)
retry.maxAgentDelayMs	60000	backoff cap	6.3 (p. 91)
compaction.enabled	true	compact automatically when context fills	6.5 (p. 97)
compaction.reserveTokens	16384	compact before the request once context exceeds contextWindow - reserveTokens	6.5 (p. 97)
compaction.keepRecentTokens	20000	recent tokens kept word for word	6.5 (p. 97)
compaction.backgroundTokens	32768	start compacting in the background this many tokens earlier; 0 disables	6.5 (p. 97)
progress.partialIntervalMs	100	minimum gap between commits of streamed text	6.2 (p. 88)
progress.outputIntervalMs	100	minimum gap between commits of tool output	6.2 (p. 88)
toolExecution	"parallel"	or "sequential"	6.4 (p. 94)
steeringMode	"one-at-a-time"	or "all", per boundary	6.1 (p. 84)
followUpMode	"one-at-a-time"	or "all", per boundary	6.1 (p. 84)

Defaults from DEFAULT_RETRY_POLICY, DEFAULT_COMPACTION_POLICY, DEFAULT_PROGRESS_POLICY and resolveSettings in src/harness/agent.ts. The spec’s HarnessSettings listing omits progress; the source and CHANGELOG 1.0.3 include it.

stream options

KEY	MEANING
transport	pi-ai transport
timeoutMs	request timeout
maxRetries,maxRetryDelayMs	provider/SDK retries inside one attempt, unlike retry
headers,metadata	forwarded request headers and metadata
cacheRetention	pi-ai prompt-cache retention
deferred	boolean or { window: "15m" "1h" "24h" }

Absent fields use pi-ai defaults (ConversationStreamOptions, src/harness/types.ts).

Other defaults

WHERE	DEFAULT	SOURCE	SEE
tool replay	"unsafe"	src/harness/tool.ts	6.4 (p. 94)
tool executionMode	the settings' toolExecution	src/harness/types.ts	6.4 (p. 94)
tool outputLimits	50 KB, 2000 lines, retain: "head"	src/harness/tool.ts, src/truncate.ts	7.4 (p. 112)
bash output	retain: "tail"	src/tools/bash.ts	7.4 (p. 112)
section tag	true: <key>...</key>	src/harness/types.ts	7.5 (p. 115)
agent thinkingLevel	"off"	src/harness/agent.ts	7.2 (p. 106)
input whenBusy	"followUp"	spec §2.2	6.1 (p. 84)
watch pending frames	100, then one full-value frame	src/session/observation.ts	8.2 (p. 129)
JSONL fsync	false	src/storage/jsonl/storage.ts	9.3 (p. 148)
SQLite walAutoCheckpointPages	1000	src/storage/sqlite/node.ts	9.2 (p. 144)
SQLite busyTimeoutMs	5000	src/storage/sqlite/node.ts	9.2 (p. 144)

Sources: src/harness/types.ts (HarnessOptions, HarnessSettings, Settings, ConversationStreamOptions, ToolRegistration, PromptSection); src/harness/agent.ts; src/harness/harness.ts:180; src/harness/tool.ts:88, :176; src/truncate.ts; src/tools/bash.ts; src/session/observation.ts:15; src/storage/jsonl/storage.ts; src/storage/sqlite/node.ts; spec §2.2; CHANGELOG 1.0.3

R . 4

Errors and reasons

Thrown errors, and the strings the Harness stores when work ends badly: submission reasons, task outcomes, blocked reasons, diagnostics and watch ends.

When something goes wrong you see it one of two ways. An error class is thrown, and your code catches it. A reason is a string stored on a record, so it survives a restart and you can query it later.

Error classes

CLASS	THROWN WHEN	RECOVERY	SEE
<code>ReadAfterWrite</code>	a Tx table read after the first table write	read first; fix the code	2.2 (p. 29)
<code>StorageRejected</code>	storage refused a batch before any durable effect	the commit rolls back; the Session continues	2.3 (p. 33)
<code>ConversationBusy</code>	<code>whenBusy</code> : "reject" on a busy conversation; carries <code>conversationId</code>	nothing was written	6.1 (p. 84)
<code>JsonLCorruptionError</code>	opening found committed data missing or damaged	restore from backup	9.3 (p. 148)
<code>JsonLStoragePoisonedError</code>	a write failed and may or may not have landed; "must be reopened"	reopen	9.3 (p. 148)
<code>FileError</code>	environment file operations; code such as <code>not_found</code>	per code	9.5 (p. 155)
<code>ExecutionError</code>	environment exec; code such as <code>timeout</code> , <code>aborted</code>	per code	9.5 (p. 155)
<code>plain Error</code>	"Session is poisoned by a failed commit after storage admission; reopen it"	reopen	2.3 (p. 33)
<code>plain Error</code>	"Session is closed"	use a new Harness	2.3 (p. 33)

The first three are exported from the package root, the JSONL errors from `.../storage/jsonl`, the environment errors from `.../env`.

Submission unanswered reasons

REASON	SET WHEN	SEE
<code>aborted</code>	a queued item is withdrawn, or a run's task aborts	6.1 (p. 84)
<code>stale</code>	a head write targets an entry before the active range	6.1 (p. 84)
<code>reset</code>	a reset placed at <code>postTools</code> ends the run before an answer	3.4 (p. 46)
<code>no_model</code>	the agent has no model, or it does not resolve	6.3 (p. 91)
<code>model_error</code>	the provider failed after retries; <code>detail</code> is its text	6.3 (p. 91)
<code>faulted</code>	the run task faulted; <code>detail</code> is the error message	5.6 (p. 80)
<code>missing_task</code> , <code>task_too_old</code> , <code>migration_failed</code>	a blocked run task was aborted and orphaned	5.4 (p. 73)

Task outcomes and blocked reasons

VALUE	FIELD	MEANING	SEE
completed	outcome status	committed by the task with a result	5.1 (p. 63)
failed	outcome status	expected failure committed by the task, with error	5.1 (p. 63)
aborted	outcome status	the task's abort handler ran and chose this end	5.6 (p. 80)
orphaned	outcome status	aborted while no code could run it; reason says why	5.6 (p. 80)
faulted	outcome status	the task broke a rule: it threw, made no progress, or its commit was rejected	5.2 (p. 67)
missing_task	inspection blocked.reason	no registered definition with the task's kind	5.4 (p. 73)
task_too_old	inspection blocked.reason	registered version is older than the record's	5.4 (p. 73)
migration_failed	inspection blocked.reason	migrate threw or is missing	5.4 (p. 73)
model_error, no_model	failed error.detail.reason	generation or compaction could not get an answer	6.3 (p. 91)

Blocked reasons are derived at inspection, never stored, until an abort turns one into an **orphaned** reason.

Tool diagnostics and watch ends

CODE OR REASON	WHERE	MEANING	SEE
tool_unavailable	diagnostic	the tool was not offered or no longer resolves	6.4 (p. 94)
invalid_arguments	diagnostic	arguments failed TypeBox validation	7.4 (p. 112)
blocked	diagnostic	a beforeTool hook blocked the call	7.3 (p. 109)
interrupted	diagnostic	an unsafe tool was cut by a crash or close	6.4 (p. 94)
truncated, full_output	diagnostic	output was cut; full output spilled to a file	7.4 (p. 112)
stopped, cancelled	WatchEnd.reason	stop() called; the observed cancellation signal aborted	8.1 (p. 125)
session_closed, retired	WatchEnd.reason	the Session closed; the document was retired	8.1 (p. 125)
listener_error	WatchEnd.reason	the listener threw; carries error	8.1 (p. 125)

Sources: src/errors.ts; src/storage/jsonl/storage.ts:81-94; src/env/index.ts:35-75; src/session/session.ts:539-546; src/session/observation.ts; src/harness/submissions.ts; src/harness/inbox.ts; src/harness/generation.ts; src/harness/live.ts:171; src/harness/scheduler.ts:769-813; src/harness/tool.ts; src/tools/read.ts; src/tools/bash.ts; src/types.ts (TaskOutcome, WatchEnd); spec §5.4, §6, §8.3; research/capture/crash-after.txt; research/capture/inbox-steps.txt

R . 5

Glossary

Each term as the book uses it, with the section that defines it.

TERM	MEANING	SEE
Agent	what a conversation runs with: model, thinking level, extensions, tools, instructions, cwd	7.2 (p. 106)
Background	a conversation-owned task excluded from idle waits, conversation aborts and cascades	5.5 (p. 76)
Base	a complete stored document value; later changes are deltas on top of it	4.3 (p. 56)
Blocked	a live task no registered definition can take; derived, never stored	5.4 (p. 73)
Boundary	a point between turns where the inbox places items: <code>postTools</code> or <code>final</code>	6.1 (p. 84)
Checkpoint	the complete durable state a task resumes from; its <code>phase</code> names the handler	5.1 (p. 63)
Chord	the replicated-state library that represents documents as values and operations	8.1 (p. 125)
Commit	one atomic write of records and documents on the mutation line	2.2 (p. 29)
Completing	a task whose outcome is decided but held while owned work is live	5.5 (p. 76)
Context	the messages derived from a transcript for the next model request	3.2 (p. 40)
Conversation	a transcript scope; may fork another and be owned by a task	3.3 (p. 43)
Definition	a typed token describing a document, document family, entry kind or task	4.1 (p. 50)
Document	typed mutable JSON state stored as Chord operations and bases	4.1 (p. 50)
Effect sandwich	commit intent, perform the effect, commit the outcome	5.3 (p. 70)
Entry	an immutable transcript record with a kind, model messages and data	3.1 (p. 37)
Extension	a named bundle of tools, sections, hooks, wraps and tasks	7.1 (p. 103)
Fork	a conversation that inherits a parent's history up to <code>parent.at</code>	3.3 (p. 43)
Generation	one model request and its classification, run by <code>pi.generation</code>	6.3 (p. 91)
Harness	one open storage plus the machinery that runs agents on it	1.2 (p. 13)
Head	an entry field naming the first entry of the active context	3.4 (p. 46)
TERM	MEANING	SEE
Hook	an extension handler called by a built-in task, such as <code>beforeTool</code>	7.3 (p. 109)
Inbox	the <code>pi.inbox</code> document of queued submissions	6.1 (p. 84)
Invocation	one in-memory run of a task's phase or abort handler	5.4 (p. 73)
Memo	a small first-writer-wins value stored on a live task	5.2 (p. 67)
Mutation line	the Session's single serialized path for commits	2.2 (p. 29)
Orphaned	a blocked task settled by an abort without running any task code	5.6 (p. 80)
Outcome	the terminal result of a task: completed, failed, aborted, orphaned, faulted	5.1 (p. 63)
Owner, owned work	the conversation or task that created a task or conversation; owned work holds and aborts with it	5.5 (p. 76)
Phase	a named step of a task, one handler per phase	5.2 (p. 67)
Registry	the process-local, unstored set of installed extensions	7.1 (p. 103)
Replay	a tool's declaration whether an interrupted call may rerun: <code>safe</code> or <code>unsafe</code>	6.4 (p. 94)
Run	the turns from an admitted input to its final answer; the conversation is busy meanwhile	6.2 (p. 88)
Section	a named part of the system prompt, rendered before each request	7.5 (p. 115)

Session	the owner of one mutation line, its records and its documents	1.3 (p. 16)
Settings	Harness-wide run policy, read at every use and never stored	7.2 (p. 106)
Storage	the backend that persists commits atomically: Memory, SQLite, JSONL	9.1 (p. 141)
Submission	an admitted user input or entry write the host can wait for	6.1 (p. 84)
Task	a durable state machine attached to one conversation	5.1 (p. 63)
Tool call	one model request to run a tool, executed by a <code>pi.tool</code> task	6.4 (p. 94)
Turn	one assistant response and the tool calls it makes	1.3 (p. 16)

TERM	MEANING	SEE
View	a conversation's structural mount: record, active entries, built-in documents	8.2 (p. 129)
Watch	a serialized exact-frame observation with at most 100 pending frames	8.2 (p. 129)
Wrap	an extension's decorator of a tool or section, targeted by name	7.4 (p. 112)

Sources: `spec §1` (terms); `README §Concepts`; `src/types.ts`; `src/harness/types.ts`; `AUTHORING.md` canonical terms

Sources

Everything this book cites, at the commit it was read.

The repository is `github.com/earendil-works/pi`, package `packages/durable`, @earendil-works/pi-durable 1.0.3, commit `5b6c792b424e73edefbfa558b901bcd64788dad2` (2026-10-05). Paths are relative to `packages/durable/` unless they start with `research/`.

Documents and source

PATH	CITED AS	CONTENTS
<code>docs/spec.md</code>	<code>spec \$N</code>	the normative specification (internal name Pico5)
<code>README.md</code>	<code>README \$X</code>	package guide
<code>CHANGELOG.md</code>	<code>CHANGELOG 1.0.N</code>	releases 1.0.0 to 1.0.3
<code>docs/pico-v5-handoff.md</code>	<code>handoff</code>	implementation plan
<code>docs/pico-v5-chord-usage.md</code>	<code>chord usage</code>	how the package uses Chord
<code>docs/chord-delta-findings.md</code>	<code>delta findings</code>	Chord delta notes
<code>src/index.ts</code> , <code>src/types.ts</code> , <code>src/errors.ts</code>	<code>src/...</code>	public surface, record and storage types, errors
<code>src/session/</code>	<code>src/session/...</code>	Session, transactions, forks, observation
<code>src/harness/</code>	<code>src/harness/...</code>	Harness, scheduler, built-in tasks and documents, views, events
<code>src/storage/</code>	<code>src/storage/...</code>	Memory, SQLite and JSONL backends
<code>src/env/</code> , <code>src/tools/</code>	<code>src/env/...</code> , <code>src/tools/...</code>	execution environment; built-in tools
<code>src/testing/</code>	<code>src/testing/...</code>	storage and env conformance suites
<code>test/*.test.ts</code>	<code>test/...</code>	behaviour tests
<code>research/pi/packages/coding-agent/docs/</code>	<code>session-format.md</code> , <code>sessions.md</code> , <code>how-pi-works.md</code>	original Pi session format

Examples and runs

EXAMPLE	SHOWS
ex 00–05	Session layer: conversations, documents, forks, owned conversations, Chord state, watches
ex 06–11	Harness, configuration, conversations, context, registry reload, extension state
ex 12–13	a durable task; close, reopen and continue
ex 14–19	chat, system prompt, real model, coding tools, print mode, JSON mode
ex 20–21	the inbox while busy; late join
ex 22–24	foreground and background subagents; child tasks
ex 25	compaction: background, manual, overflow
ex 26–31	coding agent, plan mode, reviewer, sandbox per conversation, tool override, reload and restart

Examples live in `test/examples/NN-*.ts`; their captured output is `research/runs/NN-*.txt` (cited run NN). Ex 16 needs a real model and has no run file. All others use pi-ai’s faux provider.

Capture kit

FILE	CONTENTS
README.md, NOTES.md	how each artifact was made; observed behaviours vs the docs
sqlite-schema.sql, sqlite-rows*.txt, sqlite-commits.txt	SQLite schema and every row after one turn, mid-run rows, commit log
jsonl-dir/, jsonl-dir-midrun/, jsonl-*.txt	JSONL directory after and during one turn, listings, commit log
crash-README.md, crash-*.txt	a SIGKILL during two tool calls, reopen and resume
task-transitions.txt, task-graph.txt	every task-record transition; task graph frames
watch-ops.txt, agent-events.txt, view-final.json	view watch frames, agent-event batches, final view
inbox-steps.txt, context.txt	inbox and submissions while busy; raw entries vs model context
scripts/	the capture scripts, copies of test/capture/*.ts

All under `research/capture/`, produced on Node v24.13.0 against the same commit. The `extra-p*` directories hold further captures made for individual parts.

Web

SOURCE	USED FOR
Earendil Engineering, "Pi Durable", 1 October 2026, https://earendil.com/posts/pi-durable/	motivation only; never API facts

Sources: `package.json`; `README $Examples`, `$Design Documents`; `research/capture/README.md`; `research/runs/`

Index of figures

Every diagram in the book, by the claim it makes.

0.1	reading a sequence figure	8
1.1	a crash in the middle of a tool call	11
1.2	the layers of a harness	13
1.3	the commit as the only door	16
1.4	one answered input, commits 1–8	20
1.5	one answered input, commits 9–17	22
2.1	Five record types share one ID space, and their references are typed IDs.	25
2.2	Every record but the root draws its ID from one counter; commits are numbered separately.	27
2.3	The line is held from admission until the publication is enqueued; callbacks run after.	30
2.4	Failures before Storage admission roll back; an uncertain Storage result, or an adoption that fails after Storage committed, poisons the Session.	33
3.1	anatomy of an entry	37
3.2	one transcript, two readings	41
3.3	forks, caps and an owned conversation	44
3.4	the active range moving with heads	47
4.1	Scope fixes owner and lifetime; only conversation documents choose history and fork.	51
4.2	Creation is a write: the first tx.doc() of a missing document stores its whole value in that commit.	53
4.3	An incarnation lives from its creating commit up to its retiring commit; scope decides what retires it.	54
4.4	A read starts at the newest base and replays the deltas after it; only rewindable documents keep older records.	57
4.5	A fork copies each conversation document by its own policy, into a new incarnation with one base.	58
5.1	A task has five states; your code commits checkpoints, waits and outcomes, and the Harness does the rest.	64
5.2	Before each phase the step applies the first matching rule; no progress is a fault.	67
5.3	Commit the intent, perform the effect, commit the outcome; a crash in the middle reruns the effect phase.	70
5.4	The scheduler looks up a task's code by kind; a task no code can take stays pending, and only an abort ends it, as orphaned.	73
5.5	Tasks and conversations form one ownership tree, and a background task cuts off the subtree below it.	76
5.6	One failed payment aborts its live siblings in the next commit; the checkout itself is never abort-marked.	78
5.7	An abort mark flows down in its own commit, and abort handlers run from the leaves up.	81
6.1	the submission lifecycle	85
6.2	two boundaries, one inbox	86
6.3	pi.live over one run	89
6.4	the phases of pi.generation	91
6.5	a crash in the middle of two tool calls	95
6.6	budgets on one token axis	98
6.7	two compactions, one winner	99
6.8	where spend lands	101
7.1	Code lives in the process registry; a conversation stores only names.	104
7.2	A same-name install replaces in place; uninstalling and installing again moves the extension to the end.	105
7.3	Resolution collects tools in extension order, replaces same names in place, wraps the winner, then filters.	107
7.4	Every hook is a question a built-in task asks before the commit it informs.	110

7.5	A tool-result entry stores exactly what the model saw, plus the structured diagnostics.	113
7.6	Sections render in extension order, wrappers apply, deselected extensions drop out, and instructions come last.	115
7.7	A later request appends only the section that changed, at its place in the transcript.	116
7.8	Running work keeps the code it took; the next request or call uses the replacement.	119
7.9	A replay-safe subagent call finds its child by owner and its submission by request ID.	122
7.10	A foreground child belongs to the call; a background child hangs off a background anchor.	123
8.1	from commit to chord state	126
8.2	one document's revisions	127
8.3	from a commit to a view frame	130
8.4	a slow consumer's buffer	131
8.5	the order of one batch	134
8.6	two frames of the task graph	138
8.7	one set of tasks, two readings	139
9.1	Everything the Session stores goes through one call; everything it reads is a keyed lookup or a cursor scan.	141
9.2	Each record table lifts its query columns out of a JSON record; document content lives apart, one row per commit that changes it.	145
9.3	A rewindable document keeps every revision; a current-only document drops its older rows when a base commits.	146
9.4	A JSONL commit is one marker in main.jsonl that names its sidecar records by sequence and ordinal.	149
9.5	Sidecar records land first, the marker makes them a commit, and reclamation runs only after publication.	150
9.6	Opening a JSONL directory cuts the torn marker and every sidecar record no marker confirms.	151
9.7	The SQLite and JSONL cores are portable; only their Node adapters touch Node APIs.	153
9.8	JSONL reads as fast as memory because it is memory; it pays at every commit and at open.	154
9.9	Tools, prompt sections and task phases reach files and processes only through an environment the host builds per use from committed state.	155
9.10	A windowed exec may drop output the caller would drop anyway, and counts exactly what it dropped.	157
10.1	the coding agent assembled	159
10.2	same registry, two bash tools	161
10.3	a raw write into a busy conversation	164
10.4	inside and outside the guarantee	168