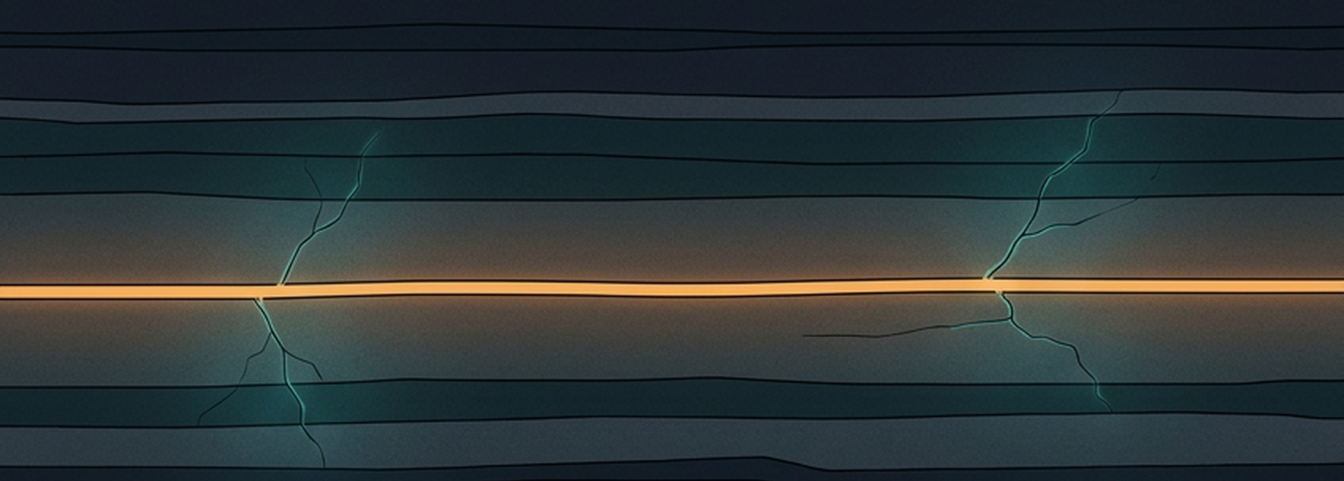


CRASH-PROOF AI AGENTS

Durable execution for LLM agents,
tested in the lab



DBOS · TEMPORAL · RESTATE · PREFECT
REAL CRASHES, REAL OUTPUT
CRASH TESTS FOR YOUR CI

GROUND TRUTH BOOKS

Crash-Proof AI Agents

Durable Execution for LLM Agents, Tested in the Lab

Ground Truth Books

This book is available at <https://leanpub.com/crash-proof-ai-agents>

This version was published on 2026-10-11



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Ground Truth Books

Contents

- About This Book** **i**
- Introduction** **1**
 - Why agents crash 1
 - The reference agent 1
 - What you will learn 3
 - How to use this book 5
- The Problem** **7**
- The reference agent and its baseline** **8**
 - Why agents die in the wild 8
 - Timeouts and stalled calls 8
 - Rate limits and 429s 9
 - Deploys and OOM kills 10
 - The failure modes in-lab 10
 - What a restart loses 11
 - The ticket agent flow 12
 - The stand-in model and ledger 13
 - Running the baseline lab 14
 - kill -9 after the refund 15
 - Reading ledger.log 16
- Durable execution fundamentals** **17**
 - Workflows, steps, and activities 17
 - Replay and determinism 17
 - Exactly-once versus at-least-once 17
 - What the engine journals 17
 - Why replay must not re-bill 17

Durable Execution with DBOS	18
DBOS Transact on Postgres	19
Postgres as the engine	19
Workflows and workflow IDs	19
The DBOSDurability capability	19
The 3.0 schema break	19
Running the DBOS lab	19
DBOS step tools for tool calls	20
Model calls versus step calls	20
Wrapping tools with DBOS.step	20
dbos-crash and dbos-step-tools compared	20
Reading operation_outputs	20
Why the refund runs once	20
Recovering interrupted DBOS runs	21
What a crash leaves behind	21
How workflow recovery works	21
Recovering pending workflows	21
Workflow IDs and idempotency	21
Upgrading code during recovery	21
Operating recovery in production	21
Durable Execution with Temporal	23
Temporal workers and task queues	24
The Temporal service and workers	24
Workflows versus activities	24
Polling task queues	24
Event history persistence	24
Starting a Python worker	24
Temporal tool calls as activities	25
The TemporalDurability capability	25
Tool calls as activities	25
The temporal-crash lab	25
The in-flight request retried	25
Reading the event history	25

Replay safety in Temporal	26
Determinism on replay	26
Non-determinism failures	26
Safe payload conversion	26
Replay-safe tracing	26
Replay testing your workflows	26
Durability Patterns	27
Idempotent tools and idempotency keys	28
Where double effects come from	28
Idempotency keys on tools	28
kill -9 during the refund	28
Treating calls as retries	28
The ledger after recovery	28
Retries, fallbacks, and replay billing	29
Configuring retries per step	29
Fallback models and providers	29
Honoring Retry-After	29
When replay must not re-bill	29
Coupling retries to durability	29
Human-in-the-loop waits	30
Why waits break plain agents	30
Durable sleep and signals	30
Signals in Temporal	30
Human approval across a restart	30
Approval across a kill -9	30
Versioning workflows in flight	31
When code changes mid-run	31
workflow.patched and deprecate_patch	31
Worker and task-queue versioning	31
Upgrading DBOS workflows	31
Versioning the reference agent	31
Observability across restarts	32
Tracing what an agent did	32

CONTENTS

Spans and events across recoveries	32
Replay-safe exporters	32
Correlating attempts by run id	32
Tracing the crash lab	32
Putting It to Work	33
Crash tests as a CI suite	34
Kill points as pytest tests	34
Recovering with a fresh process	37
Asserting the refund occurs once	37
Ledger counts after recovery	38
DBOS step-name checks	39
Temporal replay tests	41
Regressions the suite catches	43
A CI job	44
Retrofitting an existing agent, end to end	47
The retrofit roadmap	47
DBOSDurability capability first	47
Tools as DBOS steps	47
Adding the idempotency key	47
Adding tracing	47
The Temporal upgrade path	47
From double to single refund	48
Replaying both histories	48
Alternatives and Decisions	49
The Restate option	50
Restate's virtual objects	50
Durable steps in Python	50
Restate with pydantic-ai	50
Recovery and journaling model	50
Running the Restate lab	50
The Prefect option	51
Flows, tasks, and results	51
Result persistence and retries	51

CONTENTS

Resuming a crashed flow	51
The Prefect durability capability	51
Running the Prefect lab	51
Choosing your durability stack	52
What each engine journals	52
When files-plus-resume is enough	52
Comparing Restate, Inngest, Hatchet	52
Cost of complexity versus guarantees	52
Decision guide for your agent	52
Conclusion	53
What the lab proved	53
What you can now build	53
Where to go next	53
Appendix A: LangGraph checkpointers	54
What a checkpointer is	54
Backends	54
The saver interface	54
Durability modes	54
Recovery by thread_id	54
Interrupts and approval	54
What a checkpointer does not do	55
Between LangGraph and the engines	55
Appendix B: OpenAI Agents SDK sessions	56
What a session is	56
The Session protocol	56
Built-in backends	56
Retrieval and compaction	56
The durability boundary	56
Sessions and human approval	56
What to reach for instead	57
Appendix C: The lab code	58
The container	58
ticket.py	58
dbos_app.py	58
temporal_app.py	58

history.py	58
test_crashes.py	58
test_replay.py	59
provider_api.py	59
tracing.py	59
prefect_app.py	59
restate_app.py	59
Glossary	60
References	61

About This Book

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Introduction

Why agents crash

A request handler that answers in a second does not face this problem. It starts and returns, and the process forgets it [1]. An agent breaks that shape. It calls models and tools with real side effects, and it may wait for a human or run for minutes or days [1]. Each step can fail on its own, and the failure mode is not simply “the process crashed”. The harder question is what the run had already done [1]. Did the model call complete, so a retry bills you twice? Did the refund reach the provider before the kill? Is a human’s approval already sitting in a queue [1]?

Say your agent refunds a customer’s order at 3am, and the process dies before it emails the customer. A plain Python process keeps no record of that run, which lives only in the process’s memory [2]. The DBOS team describes what a crash leaves behind. The agent cannot tell which calls finished, so the only way forward is to start over, burning tokens and repeating completed work [3]. In the lab’s baseline, the second run starts from nothing, re-bills the model calls the first run paid for, and issues the refund a second time [4]. Rate limits and slow recoveries make it worse, and later chapters measure both.

A durable execution engine closes most of that gap. It records each model call and tool result as the run makes them, and a restarted process resumes from the last record instead of from the start [5, 1]. It does not close all of it. An engine journals only the units you declare to it, so a tool you did not wrap runs again on recovery. A side effect the engine does not own, such as a refund at a payment provider, still runs twice when the crash lands inside it, unless the provider recognizes an idempotency key [6]. And the model call in flight at the crash is billed a second time on every engine the lab tested [7, 8]. This book shows both halves in a lab. One reference agent takes the same `kill -9` on four engines, and the logs show which calls were saved and which ran again.

The reference agent

One agent runs through the whole book, so only the engine changes between chapters. It is a support agent for a broken-headphone ticket, one that looks up order A-1042, issues a refund of EUR 49, and emails the customer [9]. The model is a stand-in, a `pydantic-ai FunctionModel` that serves the same four replies each run, which keeps runs free and repeatable [9]. Each call the stand-in serves appends a line to `ledger.log`, the way a real provider would bill. The refund appends to `refunds.log`, the email to `outbox.log` [9]. `ledger.log` is a count, not a bill, and none of the stand-in's text stands in for what a real model would say [9].

Four scripted replies drive the loop, and with no tools run yet the model asks to look up the order. With `lookup_order` done, it asks to refund it. With the refund done, it pauses, then asks to send the email. With the email sent, it answers with "Refunded EUR 49 for A-1042 and told the customer." [9]. Environment variables hold the run open at the moments a crash matters. `SLOW_MODEL_S` sleeps inside each model call so a kill can land in any of them, and `SLOW_AFTER_REFUND_S` sleeps inside the model call that follows the refund, so a kill can land after the money moved and before the email [9]. That is why model call 3 is the call in flight in every capture that kills the agent right after the refund.

The no-durability baseline is `plain.py`. Here is the whole file [2]:

```

1  """No durability: the agent run lives only in this process's memory."""
2
3  import asyncio
4
5  from ticket import build_agent
6
7  agent = build_agent()
8
9  if __name__ == "__main__":
10     result = asyncio.run(agent.run("Customer Dana says order A-1042 arrived
        ↳ broken. Sort it out.))
11     print(result.output)

```

The baseline capture kills it and shows the bill. Run `plain.py`, run `kill -9` on the process when `refunds.log` holds one line, then run `plain.py` again [4]. Before the crash the ledger holds three model calls and one REFUND line,

and the outbox is empty. After the rerun the ledger holds seven model calls, `refunds.log` holds two REFUND lines, and the email finally goes out [4]. The second run re-billed the model calls the first run paid for, and the customer was refunded twice [4].

The seed of the fix is already in `ticket.py`. `payments_refund` accepts a key, and a key it has seen returns the first result instead of moving money again, the way Stripe's Idempotency-Key works [9]. Durable execution supplies the recording. For a side effect the engine does not own, the key turns the second execution into a no-op [6]. The chapter *Idempotent tools and idempotency keys* shows the provider using the key to drop the second refund.

What you will learn

Durable execution is the mechanism this book builds around, and its claim is narrow: one run survives a `kill -9`. The engine checkpoints state transitions, such as model responses and tool results, to durable storage, then resumes from the last checkpoint [5, 1]. Recovery requires replay determinism, because a recovered run re-executes the workflow, which must produce the same steps in the same order each time [10, 11]. And the guarantee has a sharp edge. Replay and retry can execute an operation twice, so under at-least-once delivery an operation must tolerate a second run [1].



Durable execution does not make side effects exactly-once. A refund killed in the middle of its step runs again on recovery, unless the provider recognizes an idempotency key [6, 1].

`pydantic-ai 2.55` exposes durability as a capability, so the reference agent is rebuilt with the same model and tools under the same name, plus `capabilities=[DBOSDurability()]` or `capabilities=[TemporalDurability()]` [12, 13]. Each engine wraps model requests as its own durable units, and you attach one engine per agent [5]. Two engines carry the book, and two more end it.

DBOS Transact turns the Postgres you already run into the engine [3]. A workflow is an ordinary Python function. Each model request runs as a step, and a tool call becomes a step only when you wrap it with `@DBOS.step` [14]. Step outputs land in tables named `dbos.operation_outputs` and

`dbos.workflow_status`, which you can query with `psql` [14]. Recovery is a fresh process that launches DBOS and replays interrupted workflows from the recorded steps [7, 14]. In this lab each workflow row carries the `application_version` of the process that started it, so bringing up new code while runs are in flight is a versioning problem [15].

Temporal separates the workflow definition from its execution [16]. The service persists an event history while workers poll task queues, and any worker can resume a workflow by replaying that history [16]. Model requests and tool calls run as activities, with policy-driven retries and signals for human approvals [16, 17]. Because replay runs the workflow body against the recorded history, that code must be deterministic, and the book shows the traps that break replay and how `pydantic-ai`'s integration handles them [10, 18].

The disciplines that apply to both engines get a chapter each:

- Idempotent tools and keys. When the refund step is killed mid-run, DBOS and Temporal both run that step again. A key built from the workflow ID, one the provider recognizes, moves the money once [6, 19].
- Retries and fallbacks. When the provider answers 429, Temporal's default retry policy tries the model request again after 1 second, then 2 [20]. DBOS's default does not retry at all and ends the run `ERROR`, and it waits the same 1 and 2 seconds only once you enable step retries [21]. A tenacity transport that honors `Retry-After` waits what the header asks [21]. A rejected request is never a billed call, so a retried 429 costs one call.
- Human waits. A wait that lasts days is durable only when the engine persists it, as a `DBOS.recv_async` step or a Temporal `approve` signal with `wait_condition` [22, 17].
- Versioning in flight. DBOS recovers a paused run only with the version that started it, whereas Temporal's `workflow.patched` markers let old history replay against new code [15, 18].
- Tracing across restarts. `pydantic-ai` derives `gen_ai.agent.call.id` from the workflow run id, and that attribute ties the attempt's spans together [23, 24].
- Crash tests in CI. `test_crashes.py` kills the DBOS agent at five points and counts refunds, emails, and billed calls. `test_replay.py` replays exported Temporal histories against current code in under a second, with no server [25, 26].

The same reference agent then runs on Restate and Prefect. Restate's server keeps the invocation and retries it against a fresh process, which replays the

journal. By default only the model calls are in that journal [27]. Prefect caches task results, so retrying the same flow run after a crash skips the tasks that completed [28]. *Retrofitting an existing agent, end to end* takes the plain agent onto DBOS and Temporal one change at a time, and on DBOS the crash suite goes from 4 failed tests to 6 passed [29].

How to use this book

The program output in this book comes from a live capture, a full session recorded in a container with the agent code and the commands that produced it [4, 7]. The captures pin the stack: Python 3.13, pydantic-ai 2.55.0, dbos 3.2.0, temporalio 1.34.0, Temporal CLI 1.9.1, and Postgres 17 [4]. The Prefect capture adds Prefect 3.8.8 [28], and the Restate capture `restate_sdk` 1.0.5 and `restate-server` 1.7.13 [27]. The crash-test captures add `pytest` 8.4.2 [25, 29]. Each capture header names its versions, because a durability claim is only as good as the code it was tested against [7].

Appendix C, *The lab code*, prints the container recipe and the agent files in full, so you can rebuild the lab and rerun any capture. Inside the container the files live in `/lab/agent/`, and each capture starts Postgres and, where it needs one, the Temporal dev server before it runs them. `ticket.py` defines the reference agent [9]. `plain.py` is the baseline [2]. `dbos_app.py` and `temporal_app.py` wrap the agent for the two engines [12, 13], and `test_crashes.py` and `test_replay.py` encode the crash suite [30, 26]. Where a capture shows the stand-in “asking” for a tool, the script wrote that request, so the text does not demonstrate LLM behavior [9]. What the captures do prove is mechanical. They show which calls were journaled and which steps ran again [4].

The chapters run in a deliberate order, in six parts. *The Problem* establishes the baseline and the fundamentals, in *The reference agent and its baseline* and *Durable execution fundamentals*. *Durable Execution with DBOS* rebuilds the reference agent on DBOS: *DBOS Transact on Postgres*, *DBOS step tools for tool calls*, and *Recovering interrupted DBOS runs*. *Durable Execution with Temporal* does the same on Temporal: *Temporal workers and task queues*, *Temporal tool calls as activities*, and *Replay safety in Temporal*. *Durability Patterns* applies across engines, with chapters on idempotency keys, retries and billing, human-in-the-loop waits, versioning, and observability. *Putting It to Work* turns the crash scenarios into a CI suite and retrofits an existing agent end to end.

Alternatives and Decisions runs the same agent on Restate and Prefect, and *Choosing your durability stack* lays out the choice.

A few topics have no capture: Temporal Worker Versioning deployments, fallback models, and Logfire. There the book quotes the pinned API and the exact commands, and it says what to look for in your own run, without inventing output. Printed output is what a lab run produced; where no capture exists, the book shows the code and the commands instead [4, 29].

The next chapter starts where a crash leaves you, with the baseline capture and the bill for the rerun [4].

The Problem

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The reference agent and its baseline

Why agents die in the wild

The agent that dies at 3 a.m. has usually just spent money. Ours has looked up order A-1042, decided to refund EUR 49, and watched the payment provider move the money. It is one model call away from emailing the customer when the container restarts, for any of the reasons below. The process is gone, and with it the only copy of the conversation that said the refund is done. Whatever you rerun in the morning starts from the customer's first message.

Four conditions actually kill long-lived agents, and none of them is exotic. Network timeouts, API rate limits, pod restarts, and deployment rollouts terminate in-flight agent runs silently [31]. A deployment kills running processes, the LLM provider rate-limits your requests during a traffic spike, and memory pressure restarts the container [32]. Microsoft lists restarts, deployments, scale-in events and transient failures as ordinary causes [33]. They are the operating conditions of a service that runs around the clock.

All four share one mechanism. The agent loop (input, model call, tool call, model call) runs synchronously in one process, and the conversation lives inside that process until the reply goes out [33]. When the process disappears halfway through, the searches already completed, the tool results already gathered, and the intermediate reasoning all vanish with it [32].

The bill is the point. The reference capture that anchors this chapter shows the arithmetic after one crash: three model calls and one refund before the kill, then seven model calls, two refunds, and one email after the rerun [4]. The provider billed the first three calls, and the process lost them anyway.

This chapter walks the four conditions one by one, shows how one lab scenario reproduces them, then runs the plain agent and reads the ledger the crash leaves behind.

Timeouts and stalled calls

A model call can simply stop answering. The pydantic-ai HTTP client keeps a request open for 600 seconds before it gives up, with a 5-second connect timeout [34]. Ten minutes of waiting is the default tolerance of the client layer. During those ten minutes the run has no next decision to make and no result to record.

When the timeout finally fires, the agent still does not know what happened on the provider's side. A backend engineer on the community forum described the hole [35]. A request times out or the connection drops before the response arrives, and the retry sends the same request twice, but the first one may already have been processed and billed [35]. A 429 almost always means the request was rejected before processing, so retrying it is generally safe [36], whereas a timeout leaves that question open [37]. The unknown is the dangerous case for a refund or any tool with a real effect. The step may have succeeded before it was retried, and the retry repeats the action unless the receiving system dedups [33].

The layering makes it worse. Provider SDKs and pydantic-ai's own retry transports each re-issue failed requests, so attempts multiply across the layers. A modest configuration can put eighteen requests on the wire for one model response, each billed once it reaches the model, each with its own per-attempt deadline [38]. Timeouts are not an aberration you can engineer away. They are a regular part of the model workload, like the 429s of the next section.

Rate limits and 429s

A 429 Too Many Requests means the client sent too many requests in a given amount of time, and the server is asking it to slow down [39, 40]. The response may carry a Retry-After header telling the client when to try again, either as seconds or as an HTTP date [41, 39]. The model provider's guidance is consistent. Treat Retry-After as the minimum wait, add a small random delay so a fleet of clients does not retry in lockstep, and fall back to capped exponential backoff with jitter when the header is missing [42, 43].

A 429 refuses only the timing of the request, so the same request usually succeeds once you wait [37]. When Retry-After is present, the wait is the server's to give, and the header is authoritative, beating any backoff you

computed yourself [37]. The OpenAI community asks the API to keep honoring the header per the spec so stock HTTP libraries can use it [44], and the openai-node issue confirms the library already respects it when the delay is at most 60 seconds [45].

The subtlety for an agent is that you do not control each layer that reacts to a 429. The official SDKs automatically retry eligible 429 and 503 responses subject to their settings [42], so a rate limit can be absorbed silently or can surface as the run's failure, depending on the client. With pydantic-ai's FallbackModel, the fallback does not activate until the provider SDK's 60-second retry budget is exhausted [46]. In the lab, the fake provider sends a genuine 429 with a Retry-After header of 3 seconds, and the stand-in model surfaces it as a ModelHTTPError carrying the status code and the response headers [47, 9]. An agent with no layer that retries fails there, and the rest of the run is lost.

Deploys and OOM kills

Deploys and OOM kills need no network at all. Deployments and scaling cause process termination as a normal part of operations [32]. New code versions require restarting processes, and scaling down kills instances [32]. Pods killed during scale-down delete the agent's state wholesale [31]. Memory pressure restarting the container sits on the same list [32]. In a world of daily deploys and cgroup limits, these are the ordinary events of operating a service [33].

The mechanism is the bluntest of the four. A process crash from memory exhaustion or infrastructure issues terminates execution immediately [32]. The agent gets no final word. The baseline capture's only trace of the process after the kill is the silence between the kill line and the next command [4].

The refund scenario makes the cost concrete. The deploy that lands between the money moving and the email being sent is exactly the ordinary failure this book is built around [4]. At the kill the refund line is on disk, the email was never sent, and kill -9 gives the process no chance to finish either. That window is ordinary, which is why long-lived agents need durability before they ship.

The failure modes in-lab

One scenario reproduces all four conditions, and it needs no real LLM. The lab runs the plain agent against a stand-in model, a pydantic-ai FunctionModel whose four replies are scripted, so runs are free and deterministic [9]. Three knobs turn the four conditions into crashes you can land on command.

The hardest crash, the deploy and the OOM kill, is plain `kill -9`. To the process, that is exactly what an OOM kill or a terminated pod looks like [4]. No signal handler runs, and no cleanup follows. To land the kill inside a model call, the stand-in sleeps `SLOW_MODEL_S` seconds on each call, so the kill can land inside any of the four calls [9]. To land it at the crash point this chapter studies (after the refund, before the email), the stand-in sleeps `SLOW_AFTER_REFUND_S` seconds, default 4, inside the model call that decides to email [9].

For 429s, the lab runs `provider_api.py`, a fake provider HTTP API on 127.0.0.1:8429 that answers 429 Too Many Requests with a `Retry-After` header of 3 seconds to the requests whose numbers appear in `RATE_LIMITED`, and 200 to the rest [47]. Each stand-in model call posts to it first when `PROVIDER_API` is set [9], so a rate limit reaches the agent as a real HTTP exchange with a real header.

Condition	In the lab	Knob
timeout / stalled call	the model call sleeps	<code>SLOW_MODEL_S</code> [9]
deploy / OOM kill	kill -9, no cleanup runs	none [4]
429 rate limit	fake provider on :8429	<code>RATE_LIMITED</code> [47]
crash window after refund	sleep before the email call	<code>SLOW_AFTER_REFUND_S</code> [9]

The baseline capture that follows uses the second and fourth rows together. The kill lands in the window after the refund, inside a model call, and the rerun bills every model call again.

What a restart loses

What survives a kill -9 is only what the logs recorded. `plain.py`'s docstring states the same limit. The agent run lives only in this process's memory [2].

Each line the tools append is flushed and fsynced before the append returns [9], so `refunds.log` really does hold the first refund after the crash [4]. But the rerun does not consult that file. It is a fresh `agent.run()` that re-derives its decisions from the model's scripted replies [2, 9].

The restart loses more than progress. It loses the memory of the side effect, and that is what doubles the refund. The durable-task guidance describes the failure. Without recovery, the agent starts over and pays again for model calls it already made, and an agent that crashed after issuing a refund, before recording that it issued it, may issue the refund again on restart [33].

The ticket agent flow

The reference agent is small enough to hold in your head, and `plain.py` is its entire entry point [2]:

```
1  """No durability: the agent run lives only in this process's memory."""
2
3  import asyncio
4
5  from ticket import build_agent
6
7  agent = build_agent()
8
9  if __name__ == "__main__":
10     result = asyncio.run(agent.run("Customer Dana says order A-1042 arrived
        ↳ broken. Sort it out.))
11     print(result.output)
```

The agent is a support agent for a broken-headphone ticket. Order A-1042 cost EUR 49 and arrived broken. Dana wants it sorted. `build_agent()` constructs the agent with three tools: `lookup_order` reads an order, `issue_refund` moves money by appending a REFUND line to `refunds.log`, and `send_email` appends an EMAIL line to `outbox.log` [9]. In a real deployment the last two would be a payment API and an email provider. In the lab they are two append-only files with the same failure semantics.

The model drives the flow. The stand-in model reads the names of the tool results already in the history and matches on them [9]:

```

1 match done:
2     case []:
3         return ModelResponse(parts=[ToolCallPart("lookup_order", {"order_id":
4             ↪ "A-1042"})]))
5     case ["lookup_order"]:
6         return ModelResponse(parts=[ToolCallPart("issue_refund", {"order_id":
7             ↪ "A-1042", "amount_eur": 49})]))
8     case ["lookup_order", "issue_refund"]:
9         await asyncio.sleep(SLOW_AFTER_REFUND_S)
10        return ModelResponse(parts=[ToolCallPart("send_email", {"to":
11            ↪ "dana@example.com", "body": "Refund of EUR 49 issued for
12            ↪ A-1042."})]))
13    case _:
14        return ModelResponse(parts=[TextPart("Refunded EUR 49 for A-1042 and
15            ↪ told the customer.")]))

```

A clean run costs four model calls. The first returns the `lookup_order` call, the second the `issue_refund` call, the third the `send_email` call, and the fourth the final text [9]. A clean run's ledger is four lines, `refunds.log` holds one REFUND line, `outbox.log` holds one EMAIL line, and the printed output is "Refunded EUR 49 for A-1042 and told the customer" [4].

The stand-in model and ledger

The lab's measurements all ride on one decision. The model is a stand-in, so each call it serves appends one line to `ledger.log`, the way a real provider would bill it [9]. The line records the call number and the tool results already in the conversation, for example "model call 3 (after ['lookup_order', 'issue_refund'])" [4]. The suffix comes from `tool_results`, which collects the names of the `ToolReturnParts` in the message history [9], so each line states what the model had seen when it made the decision. `ledger.log` counts the calls a real provider would have billed. It is not a bill. It carries no prices and no tokens, only the count and the context of each call, which is enough to count the calls a rerun bills again.

The append helper makes the ledger trustworthy across a crash. It opens the file, writes the line, flushes, and fsyncs before returning [9]. A line that reached the file is on disk, which is why the post-crash capture shows the first three calls and the first refund line intact [4].

The same helper writes `refunds.log` and `outbox.log`, so the three files tell the whole story of a run: what the model was asked (`ledger.log`), what money moved (`refunds.log`), and what the customer was told (`outbox.log`). With `PROVIDER_API` set, each stand-in call first posts to the fake provider API, and with `HONOR_RETRY_AFTER=1` the HTTP transport retries a 429 inside the call, waiting as long as the `Retry-After` header says via `pydantic-ai`'s `wait_retry_after` [9, 38]. Those two knobs matter in the rate-limit chapters, but the baseline does not use them.

Running the baseline lab

The baseline capture is one terminal session, quoted here verbatim [4]. The model is the stand-in, so `ledger.log` counts calls a real provider would have billed.

```
1 $ python /lab/agent/plain.py &
2 $ kill -9 3      # plain, right after refunds.log reached 1 line(s)
3 --- plain output before the crash ---
4
5 $ cat ledger.log 2>/dev/null || echo '(empty)'
6 model call 1 (after nothing)
7 model call 2 (after ['lookup_order'])
8 model call 3 (after ['lookup_order', 'issue_refund'])
9
10 $ cat refunds.log 2>/dev/null || echo '(empty)'
11 REFUND A-1042 EUR 49
12
13 $ cat outbox.log 2>/dev/null || echo '(empty)'
14 (empty)
15
16 $ python /lab/agent/plain.py
17 Refunded EUR 49 for A-1042 and told the customer.
18
19 $ cat ledger.log 2>/dev/null || echo '(empty)'
20 model call 1 (after nothing)
21 model call 2 (after ['lookup_order'])
22 model call 3 (after ['lookup_order', 'issue_refund'])
23 model call 4 (after nothing)
24 model call 5 (after ['lookup_order'])
25 model call 6 (after ['lookup_order', 'issue_refund'])
26 model call 7 (after ['lookup_order', 'issue_refund', 'send_email'])
27
28 $ cat refunds.log 2>/dev/null || echo '(empty)'
29 REFUND A-1042 EUR 49
```

```
30 REFUND A-1042 EUR 49
31
32 $ cat outbox.log 2>/dev/null || echo '(empty)'
33 EMAIL to dana@example.com: Refund of EUR 49 issued for A-1042.
```

Two runs, same prompt, same code. The first run appended three ledger lines and one refund line before the kill. The second appended four ledger lines, a second refund, and the email. The second process had no access to the first run's memory of the conversation, so it re-derived the refund decision from scratch [33]. The phrase “right after refunds.log reached 1 line(s)” is what makes the capture reproducible. That file reaching one line is the signal, and the default four-second SLOW_AFTER_REFUND_S window is the time the kill has to land before the email goes out [4, 9].

kill -9 after the refund

Where exactly did the kill land? After `issue_refund` returned, the run made its third model call. That call appended ledger line 3, then slept inside SLOW_AFTER_REFUND_S before returning the `send_email` tool call [9]. At that moment the refund was on disk, the money had moved, the email had not been sent, and the model response that would trigger the email was in flight [4]. `kill -9` ended the process there, and the capture shows no output after the crash [4].

The rerun starts from the customer's first message and makes the same decisions again. Its first model call returns `lookup_order`, its second returns `issue_refund`, and `payments_refund` appends a second REFUND line because it was called without an idempotency key [9]. The plain agent installs no key. `plain.py` does not set `ticket.idempotency_key`, so `payments_refund` sees `key=None` and writes the line unconditionally [9]. That is why the refund is issued twice instead of once. The rerun is faithful. It re-decides, from scratch, a decision the dead process had already carried out, and that is exactly what “starts over” means [33].

Note the asymmetry in the three files. `refunds.log` holds two lines, so the money moved twice. `outbox.log` holds one line, so the customer was told once, at the end of the rerun. The email did not double, because the first run did not reach it. The refund doubled, because it was the last side effect the first run completed. At-least-once delivery, the edge the introduction named, is this

capture's shape: two REFUND lines against seven billed calls, read from three text files [4].

Reading ledger.log

Read the ledger by the suffix. Each “(after ...)” names the tool results the model had seen when that call was made [9]. Lines 1 through 3 are the first attempt's bill. Line 3, made after the refund with the email still in flight, is the last thing the first process paid for. Lines 4 through 7 are the rerun's bill. Line 4 repeats line 1's decision, line 5 repeats line 2's, line 6 repeats line 3's, and line 6's response triggers send_email. Line 7 is the call made after the email was sent, the one the first attempt never made, its suffix the proof the tool ran [4].

Finishing the task needs a final call after the email is sent, so the rerun's bill is four calls. Add the first attempt's three and you get seven billed calls for one ticket, on a run that should have cost four [4]. The two runs share the ledger file, so a naive line count gives you 7. The suffix is what separates the attempts. The same arithmetic holds for refunds.log, which holds two REFUND lines, one from each attempt, because the second run had no key and no memory [9].

Cross-check the three files and the shape of the failure is complete. Ledger: 7 lines, 3 wasted. Refunds: 2 lines, one a duplicate that pays the customer twice. Outbox: 1 line, the email sent once, by the rerun. The process-level cause was a single kill -9, and the customer-visible result is a doubled refund [4]. Each engine in this book is measured against this capture. The next chapter lays out the checkpoint an engine must record to turn 7 billed calls into 5 (the 4 a clean run bills, plus the call in flight at the kill) and two refunds into one, and the rule that a replayed run must not re-execute what already happened.

Durable execution fundamentals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Workflows, steps, and activities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Replay and determinism

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Exactly-once versus at-least-once

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What the engine journals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Why replay must not re-bill

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Durable Execution with DBOS

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

DBOS Transact on Postgres

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Postgres as the engine

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Workflows and workflow IDs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The DBOSDurability capability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The 3.0 schema break

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Running the DBOS lab

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

DBOS step tools for tool calls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Model calls versus step calls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Wrapping tools with DBOS.step

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

dbos-crash and dbos-step-tools compared

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Reading operation_outputs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Why the refund runs once

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Recovering interrupted DBOS runs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What a crash leaves behind

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

How workflow recovery works

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Recovering pending workflows

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Workflow IDs and idempotency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Upgrading code during recovery

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Operating recovery in production

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Durable Execution with Temporal

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Temporal workers and task queues

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The Temporal service and workers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Workflows versus activities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Polling task queues

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Event history persistence

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Starting a Python worker

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Temporal tool calls as activities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The TemporalDurability capability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Tool calls as activities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The temporal-crash lab

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The in-flight request retried

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Reading the event history

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Replay safety in Temporal

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Determinism on replay

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Non-determinism failures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Safe payload conversion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Replay-safe tracing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Replay testing your workflows

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Durability Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Idempotent tools and idempotency keys

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Where double effects come from

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Idempotency keys on tools

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

kill -9 during the refund

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Treating calls as retries

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The ledger after recovery

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Retries, fallbacks, and replay billing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Configuring retries per step

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Fallback models and providers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Honoring Retry-After

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

When replay must not re-bill

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Coupling retries to durability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Human-in-the-loop waits

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Why waits break plain agents

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Durable sleep and signals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Signals in Temporal

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Human approval across a restart

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Approval across a kill -9

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Versioning workflows in flight

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

When code changes mid-run

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

workflow.patched and deprecate_patch

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Worker and task-queue versioning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Upgrading DBOS workflows

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Versioning the reference agent

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Observability across restarts

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Tracing what an agent did

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Spans and events across recoveries

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Replay-safe exporters

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Correlating attempts by run id

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Tracing the crash lab

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Putting It to Work

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Crash tests as a CI suite

Kill points as pytest tests

The earlier chapters ran the reference agent through a `kill -9` and read the logs by hand, one capture at a time. A session like that is research, and research does not rerun when someone edits the agent. This chapter turns the same kill points into a pytest suite that runs on each commit. `test_crashes.py` starts the DBOS agent in a subprocess, kills it with `kill -9` at five points, recovers it in a fresh process, and then asserts that the refund and the email each happened once and that the ledger holds the expected count [30]. A companion test reads the workflow's step list from Postgres and asserts that the side-effecting tools are journaled. `test_replay.py` replays exported Temporal histories against the current workflow code without a server [26, 25]. The suite passes on the finished agents. It fails on the two classic regressions, a missing DBOS step and an unpatched Temporal deploy, in CI rather than in production [25].

Both files read their configuration from the environment, the same variables the lab's container sets [30, 9]:

- `DBOS_SYSTEM_DATABASE_URL` points DBOS at Postgres, the `ticket_agent` database the DBOS chapters used. DBOS 3.2 creates that database on its first launch if the Postgres user may create databases.
- `LAB_DIR` is the directory where `ticket.py` writes `ledger.log`, `refunds.log` and `outbox.log`. The suite sets it to each test's own `tmp_path`.
- `STEP_TOOLS=1` wraps the side-effecting tools in `DBOS.step()`, and `IDEMPOTENT=1` sends the payment provider a key derived from the workflow ID. You set these two. They stand for the decorator and the key in your own code.
- `SLOW_MODEL_S` and `SLOW_IN_REFUND_S` hold each model call and the refund open so a kill lands inside them. The suite sets both to 1 itself.

The full `test_crashes.py`, including the helpers `lines`, `env_for`, `app` and `start_and_kill` that the excerpts below call, is printed in Appendix C, along with `dbos_app.py`, `temporal_app.py` and the container recipe.

The suite lives in `test_crashes.py`, which pulls the agent in only through `APP = Path(__file__).with_name("dbos_app.py")`. The same crash tests therefore run against whichever agent the environment configures. With `STEP_TOOLS=1` and `IDEMPOTENT=1` they pass, and with `STEP_TOOLS=0` they fail [30, 25]. Each kill point is a row in a table that pairs a name with a log file and a line count, quoted from `test_crashes.py` [30]:

```

1  ...
2  # (file, lines): kill as soon as <file> has <lines> lines. Every model call and
   ↳ the refund are slowed
3  # down by 1 s, so the kill lands inside that call; everything before it has
   ↳ finished.
4  KILL_POINTS = {
5      "in model call 1": ("ledger.log", 1),
6      "in model call 2, after lookup_order": ("ledger.log", 2),
7      "in model call 3, after issue_refund": ("ledger.log", 3),
8      "in model call 4, after send_email": ("ledger.log", 4),
9      "inside issue_refund": ("refunds.log", 1),
10 }
11 # the call in flight at the kill is billed again; a kill inside the refund
   ↳ leaves no model call in flight
12 CALLS_BILLED = {name: 4 if file == "refunds.log" else 5 for name, (file, _) in
   ↳ KILL_POINTS.items()}
13 ...

```

The kill lands inside the right call because the suite slows the run down. `env_for` sets `SLOW_MODEL_S=1` so each model call takes a second, and `SLOW_IN_REFUND_S=1` so the refund takes a second. `SLOW_AFTER_REFUND_S` is off, because the suite aims each kill at a file's line count and does not need the extra window the manual captures used [30]. When `ledger.log` reaches line 1 the process is inside model call 1, still sleeping. When `refunds.log` reaches line 1 the refund's money has moved and the step has not yet returned [30]. In `ticket.py` the same `SLOW_MODEL_S` delay applies inside each model call, which is what lets a kill land inside any of them [9].

The five rows also name what has already happened when the kill lands. Model call 1 asks for the order, model call 2 asks to issue the refund, and model call 3 asks for the email. Model call 4 returns the final answer, so the label `after issue_refund` on the third row records that the refund tool had already run.

The refund itself gets its own kill point, the row that watches refunds .log [30, 9].

`start_and_kill` drives the crash. It starts the agent with `subprocess.Popen` and polls the target file on a 20 ms interval. It asserts the process is still alive and that the 30-second deadline has not passed before calling `proc.kill()` and waiting [30]. `proc.kill()` delivers the same `kill -9` the lab sends by hand [29], so between the last `fsync` and the signal no further byte is written and no cleanup runs. Each test runs in its own `tmp_path` directory and under a workflow id built from a uuid, so the log files and the DBOS workflow rows stay apart across the five runs [30]. `pytest.mark.parametrize` over the table turns one test body into five named instances [30]. The deadlines are guard rails, a 30-second cap on waiting for the kill and a 60-second cap on recovery, so a hung agent fails an assertion with a message instead of hanging the CI job [30].

The model is the book's stand-in, a `pydantic-ai FunctionModel` that scripts the same four replies, so each run is free and repeatable and each ledger line stands for a call a real provider would have billed [9]. The `ci-crash-tests.md` capture shows the green run [25]. Its first line is printed the way the capture harness echoed it, without quotes. In a real shell write `export PYTEST_ADDOPTS='-p no:cacheprovider' STEP_TOOLS=1 IDEMPOTENT=1`, or the shell takes `no:cacheprovider` as a second variable name and fails. The option only stops pytest from writing a `.pytest_cache` directory, which the read-only lab mount could not hold.

```

1  $ export PYTEST_ADDOPTS=-p no:cacheprovider STEP_TOOLS=1 IDEMPOTENT=1
2
3  $ pytest -v /lab/agent/test_crashes.py
4  ===== test session starts
   ↳ =====
5  ...
6  ../lab/agent/test_crashes.py::test_crash_and_recover[in model call 1] PASSED [
   ↳ 16%]
7  ../lab/agent/test_crashes.py::test_crash_and_recover[in model call 2, after
   ↳ lookup_order] PASSED [ 33%]
8  ...
9  ../lab/agent/test_crashes.py::test_crash_and_recover[inside issue_refund]
   ↳ PASSED [ 83%]
10 ../lab/agent/test_crashes.py::test_side_effects_are_steps PASSED          [100%]
11
12 ===== 6 passed in 48.48s
   ↳ =====

```

Run it with `pytest -v test_crashes.py`. A green run takes about 49 seconds.

Recovering with a fresh process

Recovery uses a second subprocess launched by the test. `app("resume", workflow_id, work)` runs `dbos_app.py resume` with the workflow id through `subprocess.run` with a 60-second timeout [30]. The usage line in `dbos_app.py` describes the mode as a fresh process that recovers the workflow and waits [12]. The agent has to run in its own process, because a `kill -9` has to be able to land on it without taking `pytest` down with it.

The new process carries no memory of the run. When an application process restarts, DBOS recovers every workflow that was executing before the restart, scanning the system database for all `PENDING` workflows and resuming each from its last completed step [93]. A deploy or an OOM restart hits that same cold-start path, which is why the suite tests it rather than an in-process recovery helper [93]. In resume mode `dbos_app.py` retrieves the workflow handle with `DBOS.retrieve_workflow_async(workflow_id)` and prints `handle.get_result()`, the same call a production orchestrator would use to await a recovered run [12].

The two processes share the log files because `env_for` points `LAB_DIR` at the same `tmp_path` in both [30]. Once resume returns, the test asserts that the process exited cleanly and printed `Refunded EUR 49 for A-1042 and told the customer.`, then reads the shared `refunds.log`, `outbox.log` and `ledger.log` to ask whether the recovered run completed against a world that changed the right number of times [30].

Asserting the refund occurs once

`refunds.log` records the money movement, `outbox.log` the email, and `ledger.log` the calls a real provider would bill [9]. After each kill point the suite asserts the refund and the email each happened once. The assertions are the tail of `test_crash_and_recover`, quoted from `test_crashes.py` [30]:

```

1     ...
2     recovered = app("resume", workflow_id, work)
3
4     assert recovered.returncode == 0, recovered.stderr
5     assert ANSWER in recovered.stdout
6     assert len(lines(work / "refunds.log")) == 1, lines(work / "refunds.log")
7     assert len(lines(work / "outbox.log")) == 1, lines(work / "outbox.log")
8     assert len(lines(work / "ledger.log")) == CALLS_BILLED[kill_point],
    ↪ lines(work / "ledger.log")

```

A second REFUND line means the engine let the tool run twice, and a second EMAIL line means the same for the email [30].

These are the assertions the non-durable baseline fails. The retrofit capture opens with `plain.py` killed right after the refund and re-run, and its logs show 7 ledger lines and 2 REFUND lines, with the email sent once [29]. In CI the same rerun now fails the build on the second REFUND line.

The email assertion matters as much as the money one. The outbox has no provider in front of it, so a duplicate EMAIL line is a pure engine failure: the payment provider's idempotency key can absorb a repeated refund, but nothing sits in front of the outbox to absorb a repeated email [25].

At each kill point the refund assertion holds on the finished DBOS agent, including the kill inside the refund step, the hardest case. The provider has moved the money and the engine has not recorded the result, so recovery re-runs the step. The provider sees the same key twice, so it moves the money once and returns the first result [6]. The key is built from the run's DBOS workflow id, so the line the suite counts reads REFUND A-1042 EUR 49 key=ci-`<id>`:refund:A-1042. The manual retrofit run shows the same line with its own id, REFUND A-1042 EUR 49 key=ticket-42:refund:A-1042 [12, 29].

That assertion stays silent about the journal. It proves the refund happened once, not that the refund was journaled, and the step check in the next section closes that gap [25].

Ledger counts after recovery

The third assertion is the ledger, and the expected count depends on where the kill landed. `CALLS_BILLED` assigns 5 billed calls to a kill inside a model call and 4 to the kill inside the refund [30]. The comment in the source carries the mechanism. The call in flight at the kill is billed again, because its result did

not reach Postgres and recovery re-runs it. A kill inside the refund leaves no model call in flight [30]. At-least-once delivery means the provider serves that in-flight call twice, and the ledger records both.

In the ledger, a kill in model call 1 ends with model calls 1 through 5, the first call counted twice. A kill inside the refund bills nothing a second time, and the recovered run still finishes with its four calls [30]. The expected count matches the finished run in the `dbos-kill-in-refund` capture, 4 model calls in both runs [6].

The labels on the ledger lines make the counts legible. `stand_in_model` numbers each call from the lines already written, so a line like `model call 3` (after `['lookup_order', 'issue_refund']`) means the run has paid for both the lookup and the refund decision and is about to ask for the email [9]. When the count fails, the assertion prints the whole ledger in its message, so the mis-count shows exactly which call the code path skipped or repeated [30].

The five expected counts also name the regression when a test fails. A failing case names its kill point in the test id, so re-run it alone with `pytest -k`, and the printed ledger shows which call repeated. The assertions run in order, so a test that fails on `refunds.log` or `outbox.log` never reaches the ledger count [30].

DBOS step-name checks

The behavioral checks share one blind spot. All of them read side effects, and none reads the journal, so the idempotency key can make a missing step invisible at the provider. The suite adds a structural check of what DBOS recorded [25]. `test_side_effects_are_steps` runs a complete start and connects a `DBOSClient` to the system database. Then it reads the workflow's steps from `test_crashes.py` [30]:

```

1 def test_side_effects_are_steps(tmp_path: Path) -> None:
2     work = tmp_path
3     workflow_id = f"ci-{uuid.uuid4().hex[:8]}"
4     assert app("start", workflow_id, work).returncode == 0
5
6     client =
7         ↪ DBOSClient(system_database_url=os.environ["DBOS_SYSTEM_DATABASE_URL"])
8     steps = [step["function_name"] for step in
9         ↪ client.list_workflow_steps(workflow_id, load_output=False)]
10
11     assert "issue_refund" in steps, steps
12     assert "send_email" in steps, steps

```

`list_workflow_steps` is one of DBOS's programmatic workflow-management calls, also reachable as `DBOS.list_workflow_steps` or from the CLI as `dbos workflow steps` [92]. You can run the same check by hand with `dbos workflow steps` and the workflow id against a running workflow [92]. The test passes `load_output=False` because it needs only the recorded function names [30]. The names it expects are the bare tool names, `issue_refund` and `send_email` [30]. With `STEP_T00LS=1` both tools are wrapped in `DBOS.step()` and appear under those names, while the four model requests appear as `support__model.request` rows [12, 25]. Flip `STEP_T00LS` to 0 while leaving the key on, and the suite fails 2 of 6 [25]. The kill in model call 4 sends the email twice, because the killed run's `send_email` was never journaled, so recovery runs it again and nothing sits between the agent and the outbox to absorb the repeat. The step list holds only the four model rows:

```

1 $ STEP_T00LS=0 pytest --tb=short /lab/agent/test_crashes.py
2 ===== test session starts
3 ↪ =====
4 ...
5 ../lab/agent/test_crashes.py ...F.F [100%]
6
7 ===== FAILURES
8 ↪ =====
9 _____ test_crash_and_recover[in model call 4, after send_email]
10 ↪ _____
11 /lab/agent/test_crashes.py:74: in test_crash_and_recover
12     assert len(lines(work / "outbox.log")) == 1, lines(work / "outbox.log")
13 E   AssertionError: ['EMAIL to dana@example.com: Refund of EUR 49 issued for
14 ↪ A-1042.', 'EMAIL to dana@example.com: Refund of EUR 49 issued for A-1042.']
15 E   assert 2 == 1
16 E   + where 2 = len(['EMAIL to dana@example.com: Refund of EUR 49 issued for
17 ↪ A-1042.', 'EMAIL to dana@example.com: Refund of EUR 49 issued for
18 ↪ A-1042.'])

```

```

13 E     +     where ['EMAIL to dana@example.com: Refund of EUR 49 issued for
    ↳ A-1042.', 'EMAIL to dana@example.com: Refund of EUR 49 issued for A-1042.']
    ↳ = lines((PosixPath('/tmp/pytest-of-root/pytest-1/test_crash_and_recover_i
    ↳ n_mode3') /
    ↳ 'outbox.log'))
14 _____ test_side_effects_are_steps
    ↳ _____
15 /lab/agent/test_crashes.py:86: in test_side_effects_are_steps
16     assert "issue_refund" in steps, steps
17 E   AssertionError: ['support__model.request', 'support__model.request',
    ↳ 'support__model.request', 'support__model.request']
18 E   assert 'issue_refund' in ['support__model.request',
    ↳ 'support__model.request', 'support__model.request',
    ↳ 'support__model.request']
19 ===== short test summary info
    ↳ =====
20 FAILED ../lab/agent/test_crashes.py::test_crash_and_recover[in model call 4,
    ↳ after send_email]
21 FAILED ../lab/agent/test_crashes.py::test_side_effects_are_steps -
    ↳ AssertionError...
22 ===== 2 failed, 4 passed in 45.84s
    ↳ =====

```

Temporal replay tests

Temporal needs a different check. Tool calls are already activities, so there is no step list to inspect. The danger is code that no longer replays the histories of runs still in flight [62, 25]. The CI guard is `test_replay.py`, which replays each exported history against the current workflow code, and the whole file is short enough to quote [26]:

```

1 """Replay tests for the Temporal ticket workflow: every recorded history must
    ↳ replay against this code.
2
3 The histories are JSON exported with `temporal workflow show -o json`
    ↳ (HISTORY_DIR, default
4   ./histories). A code change that would break a run already in flight fails
    ↳ here, not in production.
5 """
6
7 import asyncio
8 import os
9 from pathlib import Path
10
11 import pytest

```

```

12 from temporalio.client import WorkflowHistory
13 from temporalio.worker import Replayer
14
15 from pydantic_ai.durable_exec.temporal import PydanticAIPlugin
16 from temporal_app import TicketWorkflow
17
18 HISTORIES = sorted(Path(os.environ.get("HISTORY_DIR",
    ↪ "histories")).glob("*.json"))
19
20
21 @pytest.mark.parametrize("path", HISTORIES, ids=[p.stem for p in HISTORIES])
22 def test_history_replays(path: Path) -> None:
23     history = WorkflowHistory.from_json(path.stem, path.read_text())
24     replayer = Replayer(workflows=[TicketWorkflow],
    ↪ plugins=[PydanticAIPlugin()])
25
26     asyncio.run(replayer.replay_workflow(history))

```

The histories are JSON exported with the Temporal CLI, `temporal workflow show --workflow-id ticket-42 -o json > histories/ticket-42.json`, the documented way to save an execution's event history to a file [25, 163]. One human-in-the-loop run, with the approval signal and the refund, is an 83 KB file [25]. The capture records the export and the passing replay [25]:

```

1 $ mkdir histories && temporal workflow show --workflow-id ticket-42 -o json >
  ↪ histories/ticket-42.json && ls -l histories
2 total 84
3 -rw-r--r--. 1 root root 82971 Oct 10 17:02 ticket-42.json
4
5 $ pytest -v /lab/agent/test_replay.py
6 ===== test session starts
  ↪ =====
7 ...
8 ../lab/agent/test_replay.py::test_history_replays[ticket-42] PASSED [100%]
9
10 ===== 1 passed in 0.81s
  ↪ =====

```

The Replayer runs in-process, re-executes the workflow code from the top against the recorded history, and feeds back each activity's recorded result instead of calling the activity again, so a command the history does not expect throws [164, 107]. The PydanticAIPlugin must travel with the Replayer, because the plugin installs the data converter and workflow runner that pydantic-ai needs, and it reads `__pydantic_ai_agents__` off TicketWorkflow to register the agent's activities [110, 113].

Two histories replay in 0.88 seconds [29]. That speed is what makes the suite a deploy gate. `temporal_app.py` also offers an in-process replay mode that fetches the history from the server and prints replayed cleanly or the failure, but CI wants the offline form [13]. Replay testing is the only way to prove a code change stays compatible with histories already in flight, and the failures it catches match the ways workflows break determinism, a changed activity type, a reordered activity, or a new activity in the past [164, 62].

The 83 KB file holds the activity events that recorded each model request, the approve signal, and the refund activity, so replaying it exercises the same path a resuming deployment would take [25, 164].

Regressions the suite catches

Two regressions, one per engine, are what the suite is for [25]. The first is a missing step decorator, the `STEP_TOOLS=0` run in the step-check section. Its refund assertions stay green because the key dedups the re-run refund at the provider, so only the email assertion and the step check catch it.

The retrofit capture shows the same suite tracking a refactor one change at a time [29]. Killed after the refund and re-run, the unmodified `plain.py` writes 7 ledger lines and 2 REFUND lines. With `DBOSDurability` alone, 4 of 6 tests fail, and the kills in model calls 1 and 2 pass because nothing with a side effect had run yet. Adding `STEP_TOOLS=1` leaves only the kill inside `issue_refund` failing. Adding `IDEMPOTENT=1` makes all six pass. Each test maps to one missing property, so the failure list tells you exactly which stage the agent is at.

The second regression is an unpatched version-2 Temporal deploy. `FRAUD_CHECK=unpatched` adds a `check_fraud` activity before `ask_approver` with no `workflow.patched`, and replaying the recorded ticket-42 history against that code fails the workflow task [13, 25]:

```

1 $ FRAUD_CHECK=unpatched pytest --tb=short --show-capture=no
  ↳ /lab/agent/test_replay.py
2 ===== test session starts
  ↳ =====
3 ...
4 ../lab/agent/test_replay.py F [100%]
5
6 ===== FAILURES
  ↳ =====
7 _____ test_history_replays[ticket-42]
  ↳ _____
8 /lab/agent/test_replay.py:26: in test_history_replays
9     asyncio.run(replayer.replay_workflow(history))
10 ...
11 E   temporalio.workflow._exceptions.NondeterminismError: Workflow activation
  ↳ completion failed: Failure { failure: Some(Failure { message: "[TMPRL1100]
  ↳ Nondeterminism error: Activity type of scheduled event 'ask_approver' does
  ↳ not match activity type of activity command 'check_fraud'", source: "",
  ↳ stack_trace: "", encoded_attributes: None, cause: None, failure_info:
  ↳ Some(ApplicationFailureInfo(ApplicationFailureInfo { r#type: "",
  ↳ non_retryable: false, details: None, next_retry_delay: None, category:
  ↳ Unspecified })) }), force_cause: NonDeterministicError }
12 ===== short test summary info
  ↳ =====
13 FAILED ../lab/agent/test_replay.py::test_history_replays[ticket-42] -
  ↳ tempora...
14 ===== 1 failed in 0.88s
  ↳ =====

```

Under `FRAUD_CHECK=patched`, `workflow.patched("fraud-check")` makes pre-patch runs skip the new activity, and the same history replays cleanly in 0.90 seconds [25, 13]. The modes live in `fraud_check_enabled()` in `temporal_app.py`. Unset keeps the version-1 code, unpatched ships the broken deploy, and patched runs the new activity only behind a marker. A fourth mode, `deprecated`, calls `workflow.deprecate_patch("fraud-check")`, which replays the patched run cleanly and fails on any pre-patch history [13].



The exported histories are fixtures, and they stay representative only if you refresh them. Export fresh ones on each release, and keep one history per shape of code you have shipped until each pre-patch run has finished [62, 29].

A CI job

`test_crashes.py` needs Postgres at `DBOS_SYSTEM_DATABASE_URL` and about 49 seconds for six tests [25, 30]. `test_replay.py` needs only the histories directory and the agent's directory on the import path, because it imports `TicketWorkflow` from `temporal_app.py`, and it runs in under a second [26]. The workflow below is an example. It is not part of the lab, and the lab never ran it. It is a GitHub Actions job that gives both files what they need, assuming the repository keeps the lab files in `agent/` and the exported histories in `histories/`. The pins are the lab's versions from Appendix C. `opentelemetry-sdk` is listed because `temporal_app.py` imports it at module level.

```

1  name: crash-tests
2  on: [push, pull_request]
3  jobs:
4    crash-tests:
5      runs-on: ubuntu-latest
6      services:
7        postgres:
8          image: postgres:17
9          env:
10             POSTGRES_PASSWORD: lab
11             POSTGRES_DB: ticket_agent
12          ports: ["5432:5432"]
13          options: >-
14             --health-cmd pg_isready --health-interval 5s
15             --health-timeout 5s --health-retries 10
16      env:
17        DBOS_SYSTEM_DATABASE_URL:
18          ↪ postgresql://postgres:lab@127.0.0.1:5432/ticket_agent
19        STEP_TOOLS: "1"
20        IDEMPOTENT: "1"
21        PYTHONPATH: agent
22        HISTORY_DIR: histories
23      steps:
24        - uses: actions/checkout@v4
25        - uses: actions/setup-python@v5
26          with:
27            python-version: "3.13"
28        - run: >-
29            pip install "pydantic-ai-slim[dbos,temporal,retries]==2.55.0"
30            "dbos==3.2.0" "temporalio==1.34.0" "psycpg[binary]==3.3.6"
31            "opentelemetry-sdk==1.45.1" "pytest==8.4.2"
32        - run: pytest -v agent/test_crashes.py
33        - run: pytest -v agent/test_replay.py

```

```
33     env:  
34     HITL: "1"
```

The postgres service creates `ticket_agent` at start-up, and DBOS would create it on first launch anyway. The replay step sets `HITL=1` because the capture recorded ticket-42 after `export HITL=1`, and a replay must run the configuration that recorded the history: with `HITL` unset, `temporal_app.py` builds a workflow without the approval activity and the replay fails. Keep histories recorded under another configuration in their own directory, with their own replay step. Commit a fresh export to `histories/` on each release, as the warning above says, and the replay step guards every deploy against the runs still in flight.

Retrofitting an existing agent, end to end

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The retrofit roadmap

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

DBOSDurability capability first

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Tools as DBOS steps

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Adding the idempotency key

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Adding tracing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The Temporal upgrade path

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

From double to single refund

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Replaying both histories

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Alternatives and Decisions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The Restate option

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Restate's virtual objects

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Durable steps in Python

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Restate with pydantic-ai

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Recovery and journaling model

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Running the Restate lab

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The Prefect option

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Flows, tasks, and results

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Result persistence and retries

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Resuming a crashed flow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The Prefect durability capability

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Running the Prefect lab

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Choosing your durability stack

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What each engine journals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

When files-plus-resume is enough

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Comparing Restate, Inngest, Hatchet

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Cost of complexity versus guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Decision guide for your agent

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What the lab proved

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What you can now build

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Where to go next

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Appendix A: LangGraph checkpoints

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What a checkpoint is

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Backends

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The saver interface

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Durability modes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Recovery by thread_id

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Interrupts and approval

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What a checkpointer does not do

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Between LangGraph and the engines

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Appendix B: OpenAI Agents SDK sessions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What a session is

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The Session protocol

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Built-in backends

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Retrieval and compaction

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The durability boundary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Sessions and human approval

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

What to reach for instead

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Appendix C: The lab code

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

The container

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

ticket.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

dbos_app.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

temporal_app.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

history.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

test_crashes.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

test_replay.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

provider_api.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

tracing.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

prefect_app.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

restate_app.py

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/crash-proof-ai-agents>.