RedScope

WHY MOST AGENTS DIE IN PRODUCTION — AND WHAT THE SURVIVORS GET RIGHT

Building Durable, Long-Running Autonomous Agents.

Parminder Singh
RedScope

TWO YEARS AGO — A CONTENT PLATFORM

One AI agent per keyword. 20 steps. 20 minutes. A few dollars per run.

- → Agents crashing mid-task — dollars lost, pipelines jammed
- → Customers filing refund requests, asking for credits
- → Team overwhelmed — not with building, but with firefighting

> *"This wasn't a bug. The agents weren't broken — the architecture was."*

"Most AI agents work in demos.
Few survive in production."

Three distinct problems. Three pillars.

PILLAR	THE PROBLEM	THE QUESTION
Durable Execution	Your agent *will* crash	Does it matter?
Durable Autonomy	Your agent *will* make confident wrong decisions	Does anyone catch it?
Durable Statefulness	Your agent *will* go off track	Can it find its way back?

PILLAR ONE

Durable Execution

Making failures irrelevant — not just less frequent

THE STAKES

At scale, failures cascade

20
pipeline steps per article

20min
of compute time per run

$X
in API costs per failure

ANY FAILURE MEANS

- Full restart — every step, every cost
- Other customers' jobs blocked
- Refunds, credits, churn

THE EVOLUTION

The first instinct: add retries

0 NO FAULT TOLERANCE
Any failure = full restart from scratch

1 RETRY HARNESS AROUND LLM CALLS
Transient LLM API failures handled

THREE GAPS THAT REMAIN AT LEVEL 1

Process crash → no survival

State lives in memory. Server restart, OOM kill, deployment — all state is lost. Full restart regardless.

Non-LLM failures → not wrapped

Web scraping, search APIs, database writes — large parts of the pipeline are completely unprotected.

No error classification

A 429 rate limit and a 400 bad request get identical treatment. Retrying the wrong errors wastes time or causes harm.

> *"Durability in execution is not about preventing failures — it's about making them irrelevant."*

PROPERTY 01

State Persistence

Workflow state must survive process crashes. Lives outside the process — in an external store. If the process dies, state is recoverable.

PROPERTY 02

Universal Fault Tolerance

Fault tolerance cannot be selective. Every operation that can fail must be wrapped — LLM calls, API calls, database writes, file ops.

PROPERTY 03

Intelligent Retry Policies

A 429 is worth retrying with backoff. A 400 is not. Classify errors as retryable or fatal — and apply the appropriate policy.

THE FIELD

Frameworks and architectural patterns

DURABLE EXECUTION ENGINES

Temporal
Workflow orchestration, deterministic replay. Most mature, heavier infra.

LangGraph
Explicit state machines, graph-based control flow. Lighter footprint.

Prefect / Dagster
Data pipeline roots; increasingly used for agents.

AWS Step Functions
Fully managed, no infra — JSON workflows can get unwieldy.

ARCHITECTURAL PATTERNS

Checkpointing
Persist state at key steps; resume from last checkpoint on failure.

Idempotency
Every step safely runs twice. Foundational to any retry strategy.

Event Sourcing
Every state change logged. Recovery = replay. What Temporal does internally.

Saga Pattern
Define compensating actions per step so failures can be rolled back.

THE COMPARISON

Temporal vs LangGraph — the philosophy

	TEMPORAL	LANGGRAPH
PHILOSOPHY	Make my code durable	Make my control flow explicit
DURABILITY MODEL	Engine handles it	Developer controls it
STATE	Implicit, automatic	Explicit, transparent
INFRASTRUCTURE	Heavier	Lighter

> *Temporal: "Write normal code. We make it durable."*

> *LangGraph: "Model your agent explicitly. Durability follows from structure."*

THE COMPARISON

Temporal vs LangGraph — failure and recovery

TEMPORAL

Durable event log

Every activity result recorded. On failure → deterministic replay. Expensive operations are not re-run — their saved results are replayed.

Recovery is automatic and invisible

The developer doesn't write recovery logic — the engine handles it.

⚠ Workflow orchestration code must be deterministic. Non-deterministic operations belong inside activities.

LANGGRAPH

Checkpoint at each node transition

State persisted to external store (Redis, Postgres) at every node. On failure → resume from last checkpoint. No replay.

Recovery is explicit and visible

Developer controls checkpoint configuration and external store reliability.

✓ No determinism constraint — resuming from snapshot, not replaying execution history.

THE COMPARISON

Temporal vs LangGraph — state management

TEMPORAL

- Implicit State
- Opaque
- Managed Responsibility

LANGGRAPH

- Explicit State
- Transparent
- Your Responsibility

DECISION FRAMEWORK

When to choose Temporal vs LangGraph

Choose Temporal when…

- Long-running, complex workflows with **high cost of failure**
- You want **durability as infrastructure** — not your problem
- **Strong execution guarantees** required — exactly-once, audit trail
- Control flow is **relatively linear and predictable**

Choose LangGraph when…

- **Complex branching logic**, dynamic routing, cycles
- **Transparency and debuggability** are priorities
- **Lighter infrastructure footprint** needed
- **Fine-grained control** over agent behavior and state

PILLAR 01 TAKEAWAY

> *"The goal is not to prevent failures. It's to make failures irrelevant."*

0	Full restart on any failure
1	Retry harness — handles transient failures, misses the rest
2	Durable execution — state persisted, failures irrelevant

YOUR CHECKLIST

- [ ] Is workflow state persisted **outside** the process?
- [ ] Are **all** operations wrapped — not just LLM calls?
- [ ] Do you distinguish **retryable** from fatal errors?
- [ ] Can your agent resume from the **failure point**, not from scratch?

PILLAR TWO

Durable Autonomy

Running autonomously — without sacrificing quality

02

THE FAILURE MODE

Two kinds of agent failure

LOUD FAILURES

WHAT HAPPENS
Crash, error, exception

VISIBILITY
Immediately obvious

EXAMPLE
Process crash at step 16

SOLVED BY
Durable Execution

SILENT FAILURES

WHAT HAPPENS
Agent completes successfully

VISIBILITY
Discovered downstream

EXAMPLE
Generic article that misses ICP

SOLVED BY
Durable Autonomy

> *"A crash is loud — you know something went wrong. A silent failure looks like success."*

FROM THE SEO PLATFORM

In all three cases — the agent didn't fail. It succeeded at the wrong thing.

SEARCH INTENT

Agent guesses what ranks for this keyword

Human knows what actually ranks

Content optimized for the wrong intent — confidently

POINT OF VIEW

Agent produces generic, well-structured content

Human brings a unique angle and brand voice

Indistinguishable from every competitor

BUSINESS RELEVANCE

Agent optimizes for traffic and search volume

Human ensures alignment with ICP and conversions

Ranks well — doesn't convert

The fundamental tradeoff

More autonomy	→ fewer checkpoints →	more silent failures
More durability	→ more human input →	less automation

> ***Run autonomously — without sacrificing durability.***

Not a binary choice. A spectrum to navigate — and a maturity journey to walk through deliberately.

THE MATURITY JOURNEY

Four stages of durable autonomy

1

Full Autonomy

Agent runs end-to-end, never pauses, never asks

HIGH SPEED

NO SAFETY NET

2

Human in the Loop

System decides when to pause — fixed policy checkpoints

PREDICTABLE

RIGID

3

Human as a Tool

Agent decides when to ask — `askHuman` is just another tool

DYNAMIC

NEEDS CALIBRATION

4

Escalation Matrix

Principled, dynamic scoring across confidence + novelty + history

HIGH AUTONOMY

LEARNS OVER TIME

STAGE 2

Human in the Loop — policy-gated

```
interrupt_on:
  send_email: true
  execute_sql: true
  delete_file: true
  publish: true
```

Fixed rules. Defined at build time. Agent has no say — the policy decides when to pause.

✓ **Safe, auditable, predictable**

Easy to explain. Easy to audit after the fact.

✕ **Rigid and doesn't scale**

Rules don't adapt to context. Humans become bottlenecks.

- Most production teams are here today

STAGE 3

Human as a Tool — agent-initiated

```
tools = [
  search_docs,
  send_email,
  ask_human ← just another tool
]
```

> *"I'm uncertain about the search intent. I should call askHuman before proceeding."*

THE HYBRID MODEL FOR PRODUCTION

LAYER 1 — POLICY-GATED

Hard safety boundaries — always

Irreversible actions, high-stakes ops — no exceptions

LAYER 2 — AGENT-INITIATED

When the agent detects uncertainty

Dynamic, context-aware — escalates what policy doesn't anticipate

STAGE 4

The Escalation Decision Matrix

UNCERTAINTY

How sure am I?

Low → proceed

High → escalate

NOVELTY

How familiar is this?

Low novelty → safer to proceed

High novelty → escalate

INTERVENTION BENEFIT

What happened before?

Low intervention benefit → proceed

Intervention helped → escalate

```
Escalation Score = f(High Uncertainty, High Novelty, High Intervention Benefit)

Low uncertainty + Low novelty + Low Intervention Benefit → Proceed

High Uncertainty OR High novelty OR High Intervention Benefit → Escalate
```

THE IMPLEMENTATION

This is implementable today

UNCERTAINTY SCORE

- Structured LLM output with explicit confidence field
- Run same prompt N times → measure output variance
- Detect hedging language in reasoning trace

NOVELTY SCORE

- Embed current input
- Cosine distance from library of past inputs
- High distance = high novelty

INTERVENTION BENEFIT SCORE

- Log of past escalations + outcomes
- Did human intervention change the output?
- Lookup by task type + context

```
// Weighted scoring function

Score = (W1 × Uncertainty)
      + (W2 × Novelty)
      + (W3 × Intervention Benefit)

if Score > Threshold:
  → escalate
else:
  → proceed
```

Start with equal weights. Adjust as data accumulates. The system improves over time.

PILLAR 02 TAKEAWAY

> *"Durable Autonomy is not about removing humans from the loop — it's about progressively earning the right to need them less."*

FROM FIXED RULES → DYNAMIC JUDGMENT

WHO DECIDES	System policy	Agent's judgment
WHEN DEFINED	Build time	Runtime
TRIGGER	Fixed checkpoints	Dynamic scoring
IMPROVES	No	Yes — over time

PILLAR THREE

Durable Statefulness

"Your agent didn't fail. It just lost track of the task."

03

THE DEFINITION

Two ideas working together

STATEFULNESS

The agent can assess its position in the task at any moment.

— What has been done
— What hasn't been done yet
— What decisions were made
— What artifacts exist

DURABILITY

That positional awareness survives:

— Context resets
— Process restarts
— Agent swaps

AN ANALOGY

A nurse walks in at the start of her shift.

She picks up the patient chart — medications administered, procedures done, vitals from 3am, what's scheduled next. She reads it. She continues care without missing a beat.

The outgoing nurse's clinical knowledge — how to read deteriorating vitals, how to handle a complex condition, years of training and experience — that doesn't transfer.

The chart does.

Memory is for reasoning.
State is for continuity.
At the shift handoff, only state transfers.

THREE DISTINCT CONCEPTS

Drawing the lines

CONCEPT	PURPOSE	LIVES	SURVIVES RESET?
State	Continuity	Outside the model	✓ Yes — by design
Memory	Reasoning	Outside the model, queried on demand	✓ Yes — in a store
Context	Active working surface	Inside the model's attention window	✕ No — ephemeral

> *"Context is not state. Context is not memory. It is what you have chosen to surface from both — right now, for this inference call."*

WHAT GOES WRONG — ARCHITECTURE

Two constraints that undermine statefulness

CONTEXT ROT

As context grows, the agent silently loses its grip on the task.

— No crash. No error. No warning.

— The model's ability to recall and reason over earlier context quietly degrades.

— You don't know it's happening.

CONTEXT WINDOW LIMITS

Eventually the context hits its hard ceiling and resets.

— The next session wakes up with no memory of what came before.

— Without externalized state, it has no idea where the task is.

— Statefulness breaks at that boundary.

Context rot — the slow silent erosion. Context limits — the hard sudden break.

WHAT GOES WRONG — BEHAVIOR

Two failures that result from poor statefulness

PREMATURE VICTORY

The agent looks around, sees progress, and declares the job done.

— No reliable external record of what's actually complete

— Judgment made from current context window only

— Gets it wrong — confidently

ONE-SHOTTING

The agent tries to do too much at once.

— Runs out of context mid-implementation

— Next session wakes up in a half-finished, undocumented state

— No external record of what was done, what wasn't, or why

> ***Both failures share the same root cause: no external authoritative record of where the task stood.***

THE FIX

Two foundational moves

01 Define markers of progress before execution begins

— Break the task into discrete, verifiable units

— What does done look like — for each unit, and for the whole task?

— Write it down before the agent starts

02 Externalize state into structured artifacts

— State that only lives in the context window doesn't exist

— Artifacts survive context resets, restarts, agent swaps

EXAMPLES

AGENT TYPE	STATE ARTIFACTS
Coding agent	Progress files, JSON feature lists, git commits
Investment research	Task completion logs, Excel files, research notes per company

> ***Define where you are. Write it down. Outside the model. Every time.***

THE ANTHROPIC GIT-COMMIT PATTERN · SETUP

The initializer agent creates four state artifacts

01 · FEATURE LIST JSON

Every feature the task requires — all initially marked failing.

The definition of done. Written down. External. Unambiguous.

02 · PROGRESS FILE

An empty log waiting to be written to.

Every future session reads this first to understand what happened before.

03 · GIT REPO

An initial commit showing what files exist.

Every future session reads the git log to understand task history.

04 · INIT.SH

Instructions for how to start the environment.

Every future session runs this first — no guessing, no setup time lost.

> ***The initializer agent doesn't do the work.*** ***It creates the conditions under which the work can survive.***

THE ANTHROPIC GIT-COMMIT PATTERN · EXECUTION

Every session follows the same four-step cycle

01 Wake Up

— Read the progress file

— Read the git log

— Run init.sh

— Run a basic end-to-end test

Never assume the environment is healthy — verify.

02 Orient

— Read the feature list

— Find the highest priority feature not yet passing

That is the next unit of work — one feature, not five.

03 Work

— Implement the feature

— Test end-to-end as a real user would

— Mark passing only after careful verification

Never declare victory without proof.

04 Write Back

— Commit progress to git with a descriptive message

— Update the progress file

— Leave the environment clean for the next session

> ***The agent doesn't just do the work. It maintains the state that makes the next session possible.***

PILLAR 03 TAKEAWAY

> *"State is not optional for long-running agents. The depth of your state architecture should match the duration of the work."*

For simple agents completing tasks in a single context window — none of this is necessary. This is for long-running autonomous agents specifically.

THE SYNTHESIS · SELF-ASSESSMENT

A stack, not a list — where are you on it?

Each pillar depends on the ones below it. Locate yourself on each row.

	L0	L1	L2
PILLAR 03 · DURABLE STATEFULNESS *Your agent endures.*	No external state	Progress artifacts	Self-maintaining state
PILLAR 02 · DURABLE AUTONOMY *Your agent decides well.*	No checkpoints	Static human-in-the-loop	Dynamic escalation
PILLAR 01 · DURABLE EXECUTION *Your agent survives.*	Full restart on failure	Retry harness	Durable execution engine

> ***Start at the bottom. The goal isn't L2 everywhere — it's knowing where you are and moving deliberately.***

ONE LAST THOUGHT

Production doesn't reward intelligence. It rewards endurance.

The smartest agent in the room is the one still running tomorrow.

— THANK YOU.