THE WORKSHOP

Blameless Post-Mortems

A facilitator playbook for the meeting that turns an incident into a structural improvement. Five phases, three actions, no scapegoats. Pin the summary on the wall, keep the guide open on your tablet, and run the session.

2026-12-04
barkingiguana.com/writing/the-workshop-blameless-post-mortems/

Contents

The meeting that runs after an incident, structured so it produces a structural fix instead of a quieter team. Worked example: Two Squads, One Direction.

About the post-mortem

A "post-mortem" is the meeting where a team reviews an incident and decides what to do differently. "Blameless" means the meeting is structured to find the systemic conditions that allowed the failure rather than the people who happened to be holding the failing piece when it broke.

The distinction matters because of how engineering teams behave under blame. If the meeting feels like an interrogation, the next near-miss won't get reported. The meeting after that will be about a worse incident: one that started as the unreported near-miss and grew. Blame creates silence. Silence creates repeat incidents.

Two pieces of writing back this up. Sidney Dekker's *The Field Guide to Understanding Human Error* makes the case from a safety-science angle: human error isn't the cause of failure, it's the consequence of conditions that made a reasonable action lead to a bad outcome. John Allspaw, who built post-mortem culture at Etsy, makes the same case from an engineering angle: psychological safety is a reliability strategy, not a kindness. Engineers who feel safe to report mistakes produce better incident data, which produces better systemic fixes, which produces fewer incidents.

The post-mortem is where that culture gets enacted, one incident at a time.

At a glance

- *Who, for how long:* five to eight people, including the on-call engineer who diagnosed the incident, the developer who shipped the change, a representative from any other team affected, and a facilitator who isn't the person most affected. Around ninety minutes.
- *What you walk out with:* a clear timeline, one structural root cause, and two or three concrete action items, each with a named owner and a date.
- *When to reach for it:* an incident that reached customers, a near-miss that could have, or a recurring pattern worth closing. Not for every minor bug fix, and not as a substitute for the immediate incident debrief.

Intent

A blameless post-mortem produces three things in roughly ninety minutes:

- a clear, agreed timeline of what happened
- a root cause that's structural, not personal
- two or three concrete action items that reduce the chance of recurrence

The aim isn't a comprehensive analysis of every contributing factor; it's a *useful* analysis, one that produces work the team will actually do.

When to use it

Run a post-mortem when:

- an incident reached customers, even briefly
- a deployment caused unexpected downtime or behaviour
- an internal system failed in a way that risked customer impact
- a near-miss surfaced that would have been a serious incident under different timing
- the same kind of issue has appeared more than once

Don't run one for every minor bug fix. Post-mortem fatigue is real. Reserve the format for incidents where structural learning is genuinely available.

Schedule within forty-eight hours of the incident, while memory is fresh. Do not run it during the incident itself; that's what the incident channel is for.

Participants

Five to eight people. The room needs:

- the people directly involved in the incident response
- the on-call engineer who diagnosed it
- the developer who shipped the change (if a change triggered the incident)
- a representative from any other team affected
- a facilitator (see the facilitator note below)

Optional but useful: a manager or tech lead with the authority to commit to action items. Without them in the room, the actions don't get prioritised.

The facilitator should *not* be the person most affected by the incident. They need enough distance to keep asking *"why"* without emotional investment in the answers. A coach, a sibling-team lead, or a senior engineer from elsewhere in the org all work.

Materials and timing

Phase	Duration	Materials	Key question
Set the room	5 min	,	"Are we agreed on the Prime Directive?"
Timeline	15-25 min	Whiteboard + two colours of sticky	"What happened, in what order?"
Root cause (5 Whys)	20-30 min	Whiteboard	"Why did the system allow this?"
Contributing factors	10 min	Whiteboard	"What made it worse or harder to catch?"
Actions	15 min	Whiteboard, three columns	"What two or three things will we change?"
Total	~90 min		

Space: a room with a whiteboard and enough wall space for sticky notes. Remote-only is workable on a Miro board with the team on video; the room version is better when you can get it.

Pre-work: before the meeting, the on-call engineer drafts a rough timeline from logs, monitoring, and the incident channel. Five minutes of preparation saves twenty minutes of *"what time was that again?"* in the room.

Facilitator playbook

Five phases. Keep them in this order.

Phase 1. Set the room (5 min)

Read the Prime Directive (formalised by Norm Kerth in *Project Retrospectives*, 2001) aloud (the same wording retrospective practitioners use; post-mortem culture borrows it directly):

> *"Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand."*

This is the facilitator's tool, not a gate at the door. People will say *"yes, agreed"* and arrive with blame anyway; the Prime Directive is what you point back at when blame surfaces in Phases 2 or 3, not a contract you enforce up front.

Then state the meeting's three deliverables: timeline, root cause, two-or-three actions. Tell people that the facilitator's job is to keep the discussion on systemic factors. If anyone hears blame creeping in, they should call it out, including calling out the facilitator.

Phase 2. Timeline (15-25 min)

On the whiteboard, draw a horizontal line. Mark the start of the incident and the resolution time at either end.

Walk through what happened in chronological order. The on-call engineer leads. Stop and add timestamps to the line as events come up. Use one colour of sticky for events ("API change deployed", "first customer email", "fix shipped") and another for *signals* that were noticed or missed ("schema mismatch alert fired", "no alert fired").

No interpretation yet. Just facts. If someone wants to argue about why something happened, write the question on a sticky and put it off to the side. You'll come back to it.

The timeline ends when the incident is over, not when the meeting agrees on a fix. The point is to see the *gaps*: how long between the change and the symptom? Between the symptom and detection? Between detection and resolution? Each gap is a place a system either failed or worked.

Phase 3. Root cause (20–30 min)

A note on this phase before we start. The "New View" (Sidney Dekker's term for a stance that treats human error as a symptom of systemic causes rather than a cause itself) tradition the post draws on. Sidney Dekker, John Allspaw, is sceptical of the *"root cause"* framing. Complex failures don't have a single root; they have a constellation of conditions that conspire. 5 Whys produces a clean linear chain that often terminates at whatever the asker finds satisfying, which means the same incident, asked by a different facilitator, can yield a different *"root."* Treat 5 Whys as a *tactic* for surfacing one informative chain, not as the structural truth of the failure. If the chain feels too clean, run it twice from different starting symptoms and compare.

What we want from this phase isn't a single sentence; it's the *structural gap most worth closing.* The 5 Whys chain points at one. The contributing-factors phase below catches the others.

Use the 5 Whys as the chain technique. Start with the observable failure ("customers received the wrong product") and ask *why* until you reach something structural.

Write the chain on the whiteboard, vertically:

```
Why did customers receive the wrong product?
  -> Because the reconciliation produced bad data.
Why did the reconciliation produce bad data?
  -> Because the upstream API format changed.
Why didn't the reconciliation catch the format change?
  -> Because there's no schema validation on the consumer side.
Why is there no schema validation on the consumer side?
  -> Because no contract test exists between the two services.
Why is there no contract test?
  -> Because no mechanism exists for one squad to know what the other is changing.
```

The last answer is the root cause. It's structural; it would have been true regardless of who shipped the change.

"5 Whys" doesn't mean exactly five. Sometimes three, sometimes seven. The discipline of asking *why* one more time than feels comfortable is what matters. Stop when you reach something the team can change with engineering work, not with a memo asking people to be more careful.

If the chain produces *"because the developer didn't check"*, you've stopped too early. Ask one more *why*: why didn't the system make checking easy or automatic?

Phase 4. Contributing factors (10 min)

Root cause is the structural gap. Contributing factors are everything that made the incident worse, harder to detect, or harder to fix. List them on the whiteboard:

- a missing alert
- a deployment that landed late on a Friday
- documentation that was out of date
- a runbook that didn't match reality
- a developer's reasonable assumption that turned out to be wrong

A reasonable assumption is a contributing factor, not a root cause. *"I didn't think it'd affect you"* is honest. It's also evidence that the system gave the developer no way to know. Fix the system; the assumption stops being dangerous.

Phase 5. Actions (15 min)

This is where most post-mortems either succeed or collapse.

Write three columns on the whiteboard: action, owner, deadline. Brainstorm everything the team could do to reduce recurrence; write each one on a sticky. Then force-rank them.

Pick two or three. No more.

Three actions you'll actually complete are worth more than twenty you won't. Each action gets a named owner (a person, not a team) and a deadline (a date, not "next sprint"). If you can't name an owner or set a date, the action isn't ready; park it for later.

Good actions are concrete and verifiable:

- *"Add contract tests between the subscription and reconciliation services. Priya, by Friday next week."*
- *"Schema-mismatch alerts on the reconciliation pipeline. Tom, by end of sprint."*
- *"Cross-squad notification process for shared-API changes. Charlotte to draft, agreed at next squad-leads sync."*

Bad actions are vague and ownerless:

- *"Improve our communication process"*
- *"Be more careful with API changes"*
- *"The team will think about adding tests"*

Close the meeting by reading the actions and their owners aloud. Each owner says "yes" out loud. That's the commitment.

Worked example: an allergen incident

A subscription team ships an API change on Tuesday afternoon. The change is for a new "pause subscription" feature: it modifies the subscription record's response shape. Wednesday morning, an automated reconciliation job runs against the changed API. It produces malformed data without raising an error. Three customers receive boxes containing produce they're allergic to. The on-call engineer diagnoses and fixes the code in twenty minutes; the broader fix takes longer.

The post-mortem timeline:

Time	Event
Tue 4:47 pm	API change deployed
Wed 5:30 am	Reconciliation runs against new format, produces bad allocations silently
Wed 9:00 am	First customer emails received
Wed 9:15 am	Founder calls affected customers personally
Wed 9:35 am	Engineer identifies the format mismatch
Wed 11:14 am	Fix deployed

Fourteen hours between the change and the fix. Most of that gap is a *signal* that didn't fire: the reconciliation ran overnight, unmonitored, with no schema validation. That's where the systemic problem lives.

The 5 Whys chain reaches: *no mechanism exists for one squad to know what another is changing.* That's the root cause: structural, not personal.

The three actions:

1. Contract tests across bounded contexts (bounded contexts are vocabulary boundaries inside a system; same word can mean different things on different sides), one developer, by next Friday
2. Cross-squad notification process for shared-API changes, two developers, by next Friday
3. Reconciliation alerts on schema mismatches, one developer, by end of sprint

The bad version of this same meeting:

- Timeline gets compressed into "the change broke reconciliation"
- Root cause is recorded as *"developer didn't notify the other team"*
- Action item is *"check with the other squad before shipping API changes"*

Six weeks later, a different developer ships a different API change without checking with the other squad, because nobody told them to and the team has had three sprints of higher-priority work since then. The same incident happens again, with different names on the commits.

What can go wrong

The patterns that derail post-mortems, with their fixes.

The twenty-item action list. The team lists everything that could possibly be improved. Nobody prioritises. Nobody owns most of the items. Four weeks later, two items are done (the easy ones) and eighteen are forgotten. The post-mortem felt thorough but accomplished nothing.

Fix: force-rank actions by impact. Pick two or three. If you can't pick, ask: "Which of these would have most reduced the chance of this incident?" That question almost always produces a clear top three.

The post-mortem nobody attends. Scheduled for Friday at 4 pm. Half the team has "conflicts." The people who need to hear the analysis aren't there.

Fix: schedule within forty-eight hours of the incident, during core hours. If someone can't attend, they read the write-up and sign off on the action items by the next day.

The recurring root cause. The same issue appears in three post-mortems over six months. Either the action items aren't being completed, or the fixes aren't working.

Fix: at each post-mortem, review the action items from the last two. If they're not done, that's the first topic. If they're done and the problem persists, go deeper; the previous root-cause analysis stopped too early.

Blame creeping back in. Someone says *"this wouldn't have happened if you'd checked with the other team."* The room goes cold. The person named stops contributing.

Fix: re-read the Prime Directive aloud. If blame surfaces, redirect immediately: *"That's a human action. What's the systemic condition that made that action risky?"* Don't let it stand.

The status meeting in disguise. A manager wants to know what happened and who's fixing it. No root-cause analysis, no 5 Whys. Fifteen minutes, one action: "don't do that again." This is an incident debrief, not a post-mortem.

Fix: separate the two. Run the debrief immediately for situational awareness (what happened, what's the immediate fix). Run the post-mortem within a week (why did the system allow it, what structural changes prevent recurrence).

The post-mortem that becomes a blame avoidance exercise. The opposite failure: every contributing factor is "the system." No one will name the human action that triggered the chain, even when it's relevant context.

Fix: describe the human action without judgement. *"The developer shipped without notifying the consumer team"* is a fact. *"They should have known better"* is blame. The first belongs in the timeline. The second doesn't belong anywhere.

The post-mortem nobody reads. The write-up gets filed somewhere nobody looks. Six months later, the same root cause shows up again because the team had no idea this had happened before.

Fix: keep an incident log: a single spreadsheet with one row per incident: date, summary, root cause, action items, status. Pin the link in the team channel. Project it at quarterly planning.

Outputs

Two artefacts come out of every post-mortem.

The write-up. A one-page document, posted somewhere the team can find it (an `#incidents` Slack channel works well; a wiki page works too). The template:

- Title and date of the incident
- Summary: two sentences: what happened, what the impact was
- Timeline: the table from Phase 2
- Root cause: one sentence, structural
- Contributing factors: bullet list
- Actions: table of action / owner / deadline

- Follow-up date: when the team will check whether the actions are done

The incident log entry. A row in the running log: date, type (deployment/integration/external/process/other), description, root cause, status. The log is what produces *trends*: when a single incident is bad luck and three similar incidents in a quarter is a structural problem.

The log is the unsung hero of post-mortem culture. Individual incidents feel like noise. The log is where they become signal.

Where to go next

A post-mortem and a retrospective are different things, run for different reasons. The post-mortem asks *what happened and why?* The retrospective asks *how do we work differently going forward?* Run both. Post-mortem within days. Retrospective at the regular sprint cadence.

If multiple incidents in a row share a root cause, escalate the conversation. A trend across three incidents calls for a structural investigation: maybe a focused technical project, maybe a cross-team review, maybe a Wardley map of the area where things keep going wrong.

Worked examples in the wider writing:

- the allergen incident in Two Squads, One Direction
- a Friday-deployment outage in Threat Modelling

Further reading:

- Sidney Dekker, *The Field Guide to Understanding Human Error.* The definitive argument for "New View" thinking about failure.
- John Allspaw, *Blameless PostMortems and a Just Culture* (Etsy engineering blog).
- *Site Reliability Engineering*, Chapter 15: "Postmortem Culture: Learning from Failure" (Google, free online).

The principle

Every system fails eventually. The question is whether the failure makes the system stronger or just makes people quieter.

A blameless post-mortem turns an incident into a structural improvement. It asks *why did the system allow this?* instead of *who did this?* It produces concrete actions with owners and deadlines. It feeds an incident log that reveals patterns over time.

"I didn't think it'd affect you" is not negligence; it's a reasonable assumption that the system failed to catch. Fix the system. The person who made that assumption will be the first to tell you when the next one looks risky, but only if honesty isn't punished.

About this playbook

This playbook is part of *The Workshop*, a reference series of facilitator playbooks published at barkingiguana.com. The canonical, up-to-date version lives at barkingiguana.com/writing/the-workshop-blameless-post-mortems/.

These posts are LLM-aided. Backbone, original writing, and structure by Craig. Research and editing by Craig + LLM. Proof-reading by Craig.