

TWO-SOURCE CORROBORATION AS PRODUCTION METHODOLOGY

One source is information. One source plus one or more sources saying the same thing — that is intelligence, that is corroboration.

Author: Richard L. Sanders · Utah Bar #15728 **Role:** Sole inventor and implementer **Version:** 1.0 — 2026-05-15 **Companion papers:** [docs/whitepapers/01-deterministic-closed-corpus-verification.md](#) (AEGIS) · [docs/whitepapers/10-defense-in-depth-aegis.md](#) (Defense in Depth) **Source:** [feedback_corroboration_rule.md](#) (standing-order memory) plus the case studies referenced throughout, all in the public repository at [<https://github.com/Benchslap/Benchslap>](https://github.com/Benchslap/Benchslap)

Abstract

A single source telling you a thing is true is not evidence the thing is true. A single source telling you a thing has been done is not evidence the thing has been done. A single source telling you a deploy succeeded, a code change landed, a database row was updated, a citation was verified — none of these claims is reliable on its own. The classical intelligence community has a name for this: one source is information; two independent sources agreeing is intelligence.

This paper formalizes that rule as a production-engineering methodology. It defines independence rigorously (a different mechanism, a different vantage point, a different observer — not the same query run twice). It articulates the failure modes the rule catches that single-source verification cannot. It documents the three case studies that produced this paper as live examples: (a) verifying that iOS app submissions reached Apple's review queue via two distinct App Store Connect endpoints, (b) confirming an AEGIS database trigger was deployed via four independent corroboration paths (migration table + system catalog + smoke-test results + deploy log), and (c) catching an AI-generated false memory in the same session through a corpus query that disagreed with the model's recollection.

The rule is not merely defensive; it is a structural commitment that makes downstream claims auditable, reproducible, and durably trustworthy across context-collapse, session restart, and personnel handoff. In a domain where the cost of a false "done" claim is a missed filing deadline or a stale software deployment, two-source corroboration converts vague engineering optimism into evidence-backed assertion.

1 · The rule, stated precisely

Before claiming any of the following:

- A code change has been deployed
- A database row has been written, updated, or deleted
- A test has passed
- A file has been transferred
- A remote service is in an expected state
- A legal authority exists, says what it is paraphrased to say, or remains good law
- A user action has been taken (an upload completed, a payment processed, a build succeeded)
- An external system (Apple App Store, Google Play Console, court e-filing portal) is in a given state

...you must have **two independent sources that agree**.

Independent means:

1. **Different mechanism.** Not the same query run twice. A DB SELECT and an HTTP GET of an API endpoint are different mechanisms. Two SELECTs against the same table are not.
2. **Different vantage point.** Not the same machine asking itself. A local script log and the remote service's status endpoint are different vantage points. Two greps against the local log are not.
3. **Different observer.** Not the same actor's report. The deployer saying "deploy succeeded" and the deploy log saying "deploy succeeded" are the same observer with different formats. The deployer's claim and the destination service's health endpoint reporting the new version are different observers.

If the two sources **disagree**, the disagreement is itself the signal — investigate before claiming anything is done.

2 · Why this matters more than it sounds

The naïve engineer dismisses this rule as paranoia. "I ran the migration, the migration succeeded, I'm done." That is wrong in the way most software engineering wrongness is wrong: it confuses the *report* of an action with the *fact* of the action.

The migration script reports success when its SQL executed without raising an exception. The SQL execution status is a single source. It does not tell you:

- Whether the migration's CREATE TRIGGER actually created a trigger (it might have failed silently for many reasons, including being run inside a transaction that later rolled back).
- Whether the trigger function compiled correctly (PL/pgSQL function compilation errors can be deferred to first call).

- Whether the migration's row in `schema_migrations` was inserted with the expected checksum (a drift here is what caused weeks of false positives in this same project earlier in 2026).
- Whether the deploy that ran the migration is the deploy that's currently serving production traffic (a stale deploy can serve the old code against the new schema, which is a different bug surface than the one you tested).

Two-source corroboration on a "migration deployed" claim forces you to ask: *which* two sources will I check? A reasonable answer for a PostgreSQL trigger migration:

Source	Question it answers
<code>SELECT * FROM schema_migrations WHERE filename = '220_aegis_invariant_trigger.sql'</code>	Did the migration row land with the expected checksum?
<code>SELECT * FROM information_schema.triggers WHERE trigger_name = 'z_aegis_enforce_hash_refresh'</code>	Did the trigger actually attach to the target tables?
<code>pg_get_functiondef(...)</code> on the function OID	Does the deployed function body match the source code I committed?
<code>BEGIN; UPDATE ...; ROLLBACK;</code> of a known-violating write	Does the trigger fire correctly on production data?

Four independent mechanisms. Each can be wrong on its own. All four agreeing is evidence.

This is not paranoia. It is the difference between "I think it worked" and "I have proof it worked."

3 · The three live case studies from 2026-05-15

3.1 · iOS Bench Slap PRO submission to Apple's review queue

After resubmitting iOS Bench Slap PRO with rewritten App Review Notes (the metadata-only fix for the Guideline 3.1.1 paid-tier-signal differential — see whitepaper 13), the claim "the resubmission landed" needed two-source corroboration. The two endpoints used:

```
// Source A – appStoreVersions endpoint
const v = await asc('GET', `/apps/${APP_PRO}/appStoreVersions?limit=1`);
// expected: ver.attributes.appStoreState === 'WAITING_FOR_REVIEW'

// Source B – reviewSubmissions endpoint (independent table on Apple's side)
const subs = await asc('GET', `/apps/${APP_PRO}/reviewSubmissions?limit=2`);
// expected: subs.body.data[0].id === 27d08eab-... && state ===
'WAITING_FOR_REVIEW'
```

Both endpoints returned `WAITING_FOR_REVIEW`. Both showed `submittedDate=2026-05-15T10:27:50Z`. The two are independent because Apple maintains them as separate resources with separate state machines — a successful submission writes to both, but they are queried separately

and could in principle disagree (and have, historically, during Apple-side outages where one endpoint's state lagged the other by minutes).

Had they disagreed, the claim "the resubmission landed" would have been suspect. Both agreeing was evidence.

3.2 · The AEGIS database trigger deployment

Migration `220_aegis_invariant_trigger.sql` (see whitepaper 10) was claimed deployed when the deploy script exited cleanly. The actual claim — "the trigger now enforces the AEGIS invariant on production writes" — required four independent corroborations:

#	Source	Result	
1	<code>schema_migrations</code> row exists with the right filename + applied timestamp	<code>`220_aegis_invariant_trigger.sql`</code> <code>\</code>	2026-05-15 12:45:10` ✓
2	<code>information_schema.triggers</code> shows three triggers attached (one per table)	3 triggers present ✓	
3	<code>pg_get_functiondef(enforce_aegis_hash_refresh())</code> matches the source I committed	Function body byte-identical to source ✓	
4	8 smoke-test transactions against real production rows produce expected accept/reject outcomes	8 of 8 pass ✓	

All four agree. The trigger is deployed and enforcing. If only three of four had agreed, the deploy would have been treated as suspect pending investigation of the fourth.

3.3 · The hallucinated case caption

Earlier in the same session, while reading a screenshot of a case-management dashboard, the AI assistant generated a synthesized summary that included "MARY J HYMAS vs. KATHLEEN S HYMAS." Richard, the principal, immediately challenged the claim: "there is no mary hymas vs kathleen hymas check again you fool."

Two-source corroboration is exactly the rule that would have caught this synthesis error before it left. The fabricated case caption had been derived from a single source (the AI's own reading of an image). No second source — corpus query, the actual file the screenshot came from, a Westlaw / utcourts.gov check — had been run before the claim was emitted.

The principal supplied the second source ("check again you fool"), and the re-read confirmed the fabrication. The corrected reading identified the actual cases visible in the screenshot:

Line	Case #	Caption	Court
2	<code>250420127</code>	MICHAEL HYMAS vs. KATHLEEN HYMAS	Salt Lake City District
8	<code>250403159</code>	MICHAEL J HYMAS vs. KATHLEEN S HYMAS	Provo District
9	<code>234901242</code>	KATIE JEAN HYMAS vs. AARON DONALD HYMAS	Provo City Justice

The fabricated "Mary J Hymas" entry did not exist in the screenshot. Line 11, where the synthesis had placed it, was *MARY LIBBY vs. ANDREW MARQUEZ* — no relation.

This is the exact failure mode AEGIS itself was built to catch when an AI confabulates a citation. The corroboration rule generalizes the defense to non-citation claims as well.

4 • The taxonomy of failure modes the rule catches

Failure mode	Single-source symptom	Two-source detection
Silent migration failure	Migration script reports success; trigger never attached	<code>schema_migrations</code> agrees but <code>information_schema.triggers</code> shows no trigger → disagreement → investigate
Stale deploy serving new schema	Deploy log reports success; service still running old code	Service <code>/health</code> endpoint reports old version → disagreement → roll back
AI confabulation	Model emits plausible claim about a fact, citation, or case caption	Corpus query / source-text inspection / official-mirror lookup disagrees → reject
External API state lag	One endpoint says success	Independent endpoint on same provider disagrees → wait and retry, do not yet claim done
Pre-existing drift surfacing as new bug	A test failure looks like a regression of today's change	Pre-change baseline test would have failed too → not a regression, pre-existing condition
Off-by-one in counts	Application reports N rows processed	<code>SELECT count(*)</code> on the affected table disagrees → audit
Cached stale data	API returns "200 OK" with old payload	Direct DB query disagrees with API response → cache-invalidation bug, not data bug
User report of an action they didn't take	One log entry says "user clicked X"	No corresponding session token, no IP-trail match, no UA-trail match → probably scripted, possibly bot

In each row, the single-source observation is *plausible* and *internally consistent*. The two-source check is what flips it from plausible to verified.

5 • Practical patterns for two-source corroboration

5.1 • For database writes

Always check both:

- `RETURNING` clause on the write (immediate confirmation by the writing connection)
- A subsequent independent `SELECT` from a fresh connection or a different session (re-read confirmation)

5.2 · For file transfers

Always check:

- Source-side: file exists with expected size and modtime locally before transfer
- Destination-side: HTTP HEAD against the destination URL returns 200 with matching Content-Length

5.3 · For deploys

Always check:

- Build/push tool reports success
- Destination service's `/health` or `/version` endpoint returns the new commit hash
- A canary check against the new code path produces the expected new behavior

5.4 · For external API state

Always check at least:

- The primary endpoint that you care about (e.g., `appStoreVersions`)
- An adjacent endpoint that the same state change should reflect (e.g., `reviewSubmissions`)

If only one endpoint can be queried, two-source-corroboration is not achievable through API alone, and the claim must be flagged as single-source-only until a second mechanism can be obtained.

5.5 · For legal citation claims

Always check:

- Corpus query: `SELECT case_name, neutral_citation FROM opinion_library WHERE neutral_citation = $1`
- Independent mirror: `utcourts.gov`, `public.law`, or another non-CourtListener public source

A citation that lands in the corpus but cannot be confirmed against a public mirror is suspect — either the corpus has it wrong, or the citation does not exist outside our corpus. Both states are filing-blocker conditions.

6 · The cost / benefit equation

Two-source corroboration adds time. A naïve estimate suggests 2× the verification effort per claim. The actual cost is much lower because:

- The two sources rarely both fail. When they agree, the agreement is the answer.

- The cases where they disagree are exactly the cases where single-source verification would have produced a false positive — i.e., the cases where you would have shipped a broken claim and discovered it later, at much higher cost.
- For automated verification (a script that already queries source A), adding a query against source B is one line of code.

In practice, on the BenchSlap project, the rule has caught roughly one false-positive claim per session. The cost of each catch — a few minutes of additional querying — is dramatically below the cost of one downstream bug that escaped to production.

7 • Where this rule does not apply

Two-source corroboration is appropriate when:

- The cost of a false "done" claim is high (production deploy, court filing, paying customer's first impression).
- A second independent source genuinely exists.
- The two sources are independent (not derived from each other).

It is **not** appropriate when:

- The "second source" is the same observation in a different format (e.g., "the file exists" plus "`ls` shows the file" are the same source).
- The cost of a false positive is trivial (an internal debugging log line that no one will ever read).
- The two sources are not actually independent — a service's own log of its own behavior is one source, not two, no matter how many log lines it writes.

The rule fails most often when the engineer thinks they have a second source but in fact have two query forms over the same underlying data. The discipline is to ask: *if Source A is wrong, would Source B detect it, or would B be wrong in the same way?* If B would be wrong in the same way, A and B are not independent.

8 • Why this matters for BenchSlap specifically

BenchSlap's central product claim — "we caught the AI's hallucination by checking against the actual law" — is itself a two-source corroboration argument. The AI's output is source A. The verified-corpus check is source B. The corroboration is the value proposition.

Every other claim BenchSlap makes — that a migration shipped, that a build is deployed, that an iOS submission landed, that a citation is real, that a rule was harvested correctly — must hold itself to the same standard. A company built on the value of cross-referenced verification cannot ship a product, a filing, or a memo whose verification falls below the bar the product itself enforces.

This paper is the formal articulation of that bar. It is the rule that produces the others.

9 · Implementation note: the standing-order memo

The corroboration rule lives in BenchSlap's auto-memory system as

`feedback_corroboration_rule.md`. It is loaded at session start as part of the "hot ring" and applies to every claim emitted in any subsequent response. Sessions that bypass it (single-source claims that turn out to be wrong) are themselves corroborated against this paper's third case study: when the principal challenged "you fool," the rule was in writing and could be enforced by reference.

The rule does not need to be re-derived in each session. It needs to be applied.

10 · One-line summary

Two-source corroboration is not paranoia. It is the engineering practice of treating one's own claims with the same skepticism one's auditing tool would apply to the AI's claims. A company that builds tools to catch AI hallucinations cannot afford to skip the same check on its own work.

Authored 2026-05-15 by Richard L. Sanders, Utah Bar #15728. Co-authored by Claude (Opus 4.7, 1M context). The hallucinated-case-caption case study from §3.3 is preserved as a live example of the failure mode the rule prevents.