CC Verifier Sidecar and Ready Gate — Observe Rollout (MC #105472)

CC Verifier Sidecar and Ready Gate — Observe Rollout (MC #105472)

Current state

The persistent CC verifier sidecar and exact-revision MC ready gate are installed in **observe-only** mode. Enforcement remains disabled; the pilot allowlist is empty and the enforcement watermark is unset. This page does not authorize `pilot-enforce` or `enforce-new`.

Accepted implementation

- WP1 verifier core: `044c0d30e7d7e973009991d91e42b8c2f77371ce`
- WP2 identity/handoff remediation: `ad28aa306350b44d568aa9c75f4502ec04634bfb`
- WP3 system source: `2cee8e17b1cbea58ba9320015e9c8cca793cb56d`
- Hook candidate: `c0871b8689da0063bd2d6cca26322f702a55c0d1`
- Live system commit containing accepted WP3 bytes: `18e8ef5d257e34ebfdcb91ff806e7a377010a874`
- Live hook commit: `6c52ef129dd68008551e355ef88c97b94fe2202b`

The live commits do not share normal ancestry with the isolated review branches, so acceptance is based on direct byte equivalence of candidate-owned files, not inferred ancestry.

Independent evidence

- WP1 review: `/Users/makinja/system/evidence/105474/proveo-independent-review.md`
- WP2 review: `/Users/makinja/system/evidence/105479/proveo-review-identity-remediation/REPORT.md`
- Correct live Stop-hook pilot: `/Users/makinja/system/evidence/105479/live-pilot-105501/REPORT.md`
- Final WP3 exact-SHA review: `/Users/makinja/system/evidence/105472/proveo-wp3-system-2cee8e17/REPORT.md`
- Updated post-install acceptance: `/Users/makinja/system/evidence/105472/WP3-HOOK-ACCEPTANCE-UPDATE.md`
- Independent post-install review: `/Users/makinja/system/evidence/105472/proveo-wp3-postinstall/REPORT.md`
- Trusted exact-revision verdict: `/Users/makinja/system/evidence/105505/cc-verifier/2cee8e17b1cbea58ba9320015e9c8cca793cb56d/verdict.json`

Final post-install validation passed 82/82 focused tests, 14/14 adjacent Stop-hook tests, and the shell integration smoke. The ready hook blocks only intentional verifier exit code 2 and fails open to the existing legacy gates on infrastructure errors.

Identity and trust boundaries

- Task identity must come from agreeing explicit trusted input, orchestrator-injected numeric `MC_TASK_ID`, or demonstrably session-owned metadata.
- Unsafe, conflicting, ambiguous, stale/global-marker-only, non-positive, or larger-than-`Number.MAX_SAFE_INTEGER` IDs are rejected before MC database or revision resolution.
- Verdicts are bound to an exact immutable revision. Any later commit invalidates PASS.
- Verification commands and evidence roots come from trusted policy; builder-provided commands and arbitrary evidence paths are never executed or trusted.
- Durable SQLite attestation detects accidental drift and one-artifact tampering, but is not a cryptographic boundary against a same-OS-user privileged process.

Promotion boundary

Promotion to an enforcing mode requires a separate explicit approval, a healthy attestation store, fresh attested PASS at the candidate revision, and review of observe telemetry. Historical `ready_for_review` backlog must never be enrolled. Rollback is configuration-only: set `gate_rollout_mode` to `observe` to remove blocking while preserving audit telemetry, or `off` to disable gate decisions.

---

title: CC Verifier Ready-Gate — Operator Runbook mc_task: 105505 parent_mc_task: 105472 status: observe (audit-only, no enforcement) project_path: /Users/makinja/system

CC Verifier Ready-Gate — Operator Runbook

Covers `tools/cc-verifier-gate.ts` (WP3 of MC #105472): the exact-revision check that runs before `mc.js ready`. This is the operator guide for the rollout lifecycle — activation, telemetry, invalidation, promotion, rollback, and fail-open behavior. It does not restate the WP3 design (see the header comment in `tools/cc-verifier-gate.ts` and `specs/cc-verifier-sidecar-105472-plan.md`).

1. Observe activation

The gate ships live with `config/cc-verifier-pilot.json`:

```
"gate_rollout_mode": "observe",
"gate_pilot_enforce_task_ids": [],
"gate_enforce_new_watermark_epoch_ms": null
```

In `observe` mode (`decide()` in `tools/cc-verifier-gate.ts:363-365`), every failing finding is recorded to the audit log but the decision is always `{ action: 'audit', enforced: false }` — it can never block `mc.js ready`. `observe` is the only mode this pilot should run in until a human explicitly promotes it (Section 4). No code change is required to activate observe — it is the config default and the live-safe fallback if the config file is missing or unparseable (`loadGateConfig()` falls back to `DEFAULT_GATE_CONFIG`, which is also `observe`/empty/null).

To confirm the live config is still in the safe state:

```
node -e "const c=require('./config/cc-verifier-pilot.json'); console.log(c.gate_rollout_mode,
c.gate_pilot_enforce_task_ids, c.gate_enforce_new_watermark_epoch_ms)"
# expected: observe [] null
```

2. Audit telemetry

Every `checkReadiness()` call appends one JSON line to the audit log (`~/system/state/cc-verifier-gate-audit.jsonl` in production, overridable via `CC_VERIFIER_GATE_AUDIT_LOG` for tests), regardless of mode or outcome:

```
{"ts":"...","component":"cc-verifier-
gate","task_id":"...","dry_run":false,"finding":{...},"decision":{...}}
```

`finding.reason` carries the machine-readable failure code (e.g. `revision_mismatch`, `digest_mismatch`, `verdict_field_tampered`, `stale_revision`, `no_verdict_found`) so audit volume can be triaged by failure class. Audit writes are best-effort and wrapped in try/catch (`appendAudit()`) — a logging failure never changes the gate decision.

To review recent audit activity:

```
tail -n 50 ~/system/state/cc-verifier-gate-audit.jsonl | node -e
"process.stdin.on('data',d=>d.toString().trim().split('\n').forEach(l=>{const
e=JSON.parse(l);console.log(e.task_id, e.decision.mode, e.decision.action,
e.finding.reason)}))"
```

Before promoting past `observe`, review the audit log for the candidate task id(s) and confirm the only findings are expected ones (e.g. a deliberately stale revision), not systemic issues like schema drift or a misconfigured `MC_DB_PATH`.

3. Exact-revision invalidation

A verdict is only valid for the exact revision it was computed against. `evaluate()` resolves the trusted current revision from `--repo/--commit`, explicit `--repo` + live `git rev-parse HEAD`, or (default) the MC task's `tasks.project_path` + live HEAD — never from the verdict itself (`resolveTrustedRevision()`, `tools/cc-verifier-gate.ts:162-206`). If the stored verdict's `revision` does not match the current HEAD, the finding is `revision_mismatch` and `ok:false`.

If no verdict exists at the current revision but one exists at a prior revision, the reason is `stale_revision` (as opposed to `no_verdict_found`, which means the task was never verified at all) — this is the expected signal after any new commit lands on a previously-passing task; a fresh `cc-verifier` run is required before the gate can pass again at the new revision.

Every evidence file hash and the top-level `verdict` field are also independently recomputed from the checks array and re-read evidence bytes — a hand-edited `verdict.json` (e.g. flipping `PARTIAL` to `PASS`, or editing a stdout/stderr file after the fact) is caught as `verdict_field_tampered` or `digest_mismatch` even when the stored `revision` field is untouched.

4. Promotion prerequisites (pilot-enforce / enforce-new)

Both enforcing modes are opt-in per config field and require a human edit to `config/cc-verifier-pilot.json` — there is no code path that auto-promotes.

`pilot-enforce` — blocks only task ids explicitly listed in `gate_pilot_enforce_task_ids`. Before adding a task id:

1. Confirm the task has at least one `verdict_status: "PASS"` audit entry at its current revision (Section 2).
2. Confirm no unexplained `digest_mismatch`/`verdict_field_tampered` entries exist for that task in the audit log.
3. Add the task id as a string to `gate_pilot_enforce_task_ids` (e.g. `["105505"]`). Tasks not in the list continue to audit-only (`decide()` line 371: `not in pilot allowlist` falls through to `action: 'audit'`).

`enforce-new` — blocks only tasks whose MC `created_at` is strictly after `gate_enforce_new_watermark_epoch_ms`. Before setting the watermark:

1. Run the pilot in `pilot-enforce` for a representative set of tasks first and confirm no false-positive blocks in the audit log.
2. Set `gate_enforce_new_watermark_epoch_ms` to an epoch-ms value at or after the promotion decision time — this must be a *second*, independent watermark from the WP2 event-ingestion `activation_watermark_epoch_ms`; do not reuse or overwrite that field.
3. Any task created at or before the watermark, or when the watermark is `null`/non-finite, is never eligible for blocking (`decide()` lines 374-381) — this is the live-safe default and must remain the fallback for any config read error.

Both modes still only ever block when `finding.ok === false` AND the task is enrolled/eligible; a passing finding always resolves to `action: 'allow'` regardless of mode (`decide()` line 359-361).

5. Rollback to off

To disable the gate entirely (including audit-only behavior), set:

```
"gate_rollout_mode": "off"
```

`decide()` line 355-357 short-circuits before evaluating the finding: `off` always returns `{ action: 'allow', enforced: false }`. Note `evaluate()` still runs and is still written to the audit log by `checkReadiness()` — `off` only changes the decision, not whether the check executes or is recorded. To roll back from `pilot-enforce` or `enforce-new` without fully disabling audit visibility, set `gate_rollout_mode` back to `observe` instead — this preserves telemetry while immediately removing all blocking.

Rollback is a config-only change; no deploy, restart, or code change is required, since `loadGateConfig()` reads the file fresh on every `checkReadiness()` call.

6. Fail-open incident handling

The gate is fail-open by design at every layer that is not the explicit enforcing-mode decision:

- **Missing/unparseable config file** — `loadGateConfig()` catches the read/parse error and returns `DEFAULT_GATE_CONFIG` (`observe`, empty allowlist, null watermark).
- **Unknown `gate_rollout_mode` value** — `loadGateConfig()` ignores it (keeps the default `observe`); if an unrecognized mode value somehow reaches `decide()` anyway, the final fallback branch returns `{ action: 'audit', enforced: false, reason: 'unknown_mode_fail_open_audit' }`.
- **MC db missing/locked** — `queryTaskRow()` catches the error and returns `null`; `resolveTrustedRevision()` then returns `no_trusted_project_path`, which is a normal `ok:false` finding (still subject to the mode's decision logic — in `observe` this only audits).
- **Audit log write failure** — `appendAudit()` swallows the error; it never affects `finding`/`decision`.
- **No commit/task/repo resolvable** — `evaluate()` returns `invalid_task_id` or `revision_undeterminable`; these are ordinary findings, not exceptions, and flow through the same mode-gated `decide()` path as any other failure.

If an incident is suspected (e.g. unexpected blocks reported by a builder), the immediate mitigation is:

```
node -e "const fs=require('fs');const p='config/cc-verifier-pilot.json';const
c=JSON.parse(fs.readFileSync(p));c.gate_rollout_mode='observe';fs.writeFileSync(p,
JSON.stringify(c,null,2))"
```

This reverts to audit-only without needing to identify root cause first. Then inspect the audit log (Section 2) for the affected task id(s) to determine whether the block was a correct enforcement (real tamper/stale revision) or a gate defect, before re-promoting.

7. Durable attestation store health

`tools/cc-verifier.js` (F1) writes one row per `(task_id, revision)` to a SQLite store at `~/system/databases/cc-verifier-attestations.db` (overridable via `CC_VERIFIER_ATTESTATION_DB` for tests), recording the SHA-256 of the exact `verdict.json` bytes it just persisted, the `verifier_identity`, and `written_at`. This happens inside `writeAtomicJSON()` immediately after the verdict file is written, so a verdict and its attestation are never observed out of order by a reader.

The gate (`tools/cc-verifier-gate.ts`) reads this store read-only (`cc.getAttestation()`) and requires all three to hold before treating a verdict as genuine:

- an attestation row exists for `(task_id, revision)` — otherwise `attestation_missing`;
- its `verdict_sha256` matches the SHA-256 of the on-disk `verdict.json` bytes — otherwise `attestation_mismatch`;
- its `verifier_identity` matches the verdict's own `verifier_identity` field — otherwise `attestation_identity_mismatch`.

`cc-verifier.js`'s own idempotency cache (`readAttestedVerdict()`) applies the same first two checks before it will reuse a cached verdict instead of re-running checks — a `verdict.json` with no matching attestation is treated as absent, not as a cache hit.

To check store health directly:

```
node -e "
const Database = require('better-sqlite3');
const db = new Database(process.env.CC_VERIFIER_ATTESTATION_DB
  || require('os').homedir() + '/system/databases/cc-verifier-attestations.db',
  { readonly: true, fileMustExist: true });
console.log(db.prepare('SELECT COUNT(*) AS n FROM attestations').get());
db.close();
"
```

If this fails (file missing, not a database, or table missing), the store is unhealthy — see Section 7.2. `writeAttestation()` itself never throws (all failures are caught and swallowed): a write failure does not crash `verify()`/`finalizeBlocked()`, it only means that verdict will show as `attestation_missing` to the gate until the store is repaired and a fresh verify run is performed. This is deliberate fail-closed-for-the-gate, fail-open-for-the-builder-pipeline behavior — verification work is never blocked by attestation-store trouble, but the gate will not silently trust an unattested verdict either.

7.1 Same-OS-user trust boundary

The attestation store raises the bar for forging a verdict; it does not change the trust boundary. Both `verdict.json` and `cc-verifier-attestations.db` are ordinary files writable by whichever OS

user runs `cc-verifier.js`/`cc-verifier-gate.ts` (no setuid, no separate service account, no OS-level ACL beyond normal file permissions). Any process running as that same user can, with `better-sqlite3` and the schema in Section 7, hand-craft an `attestations` row with a `verdict_sha256` that matches a hand-edited `verdict.json` — exactly as it could previously hand-edit `verdict.json` alone. The store's value is narrower than a same-user attacker: it converts a one-file edit into a two-artifact forgery (matching JSON bytes *and* a matching DB row), which is enough to catch accidental drift, partial edits, and non-adversarial tooling bugs, but it is **not** a defense against a same-user, same- privilege adversary with full filesystem access. Cross-user tampering (a different OS user, or a user without write access to `~/system/databases/`) is out of scope for this store and is not claimed to be mitigated by it.

7.2 Integrity-mismatch recovery and reverification

`attestation_missing`, `attestation_mismatch`, and `attestation_identity_mismatch` are ordinary `ok:false` findings — they flow through the same mode-gated `decide()` path as `revision_mismatch` or `digest_mismatch` (Section 3) and, in `observe` mode, only audit. They do not indicate gate malfunction by themselves; they indicate the on-disk verdict is not attested at the store's current state. Recovery is always the same: re-run `cc-verifier` for the task at the current revision. There is no repair path that edits or re-derives an attestation row in place — `getAttestation()` never invents or fixes a row, and there is no supported "resync" operation; a fresh, trusted verify pass is the only way to re-establish attestation for a given `(task_id, revision)`:

```
node tools/cc-verifier.js replay --task <id> --revision <sha>
```

If mismatches appear across many unrelated tasks at once (rather than one task after a targeted hand-edit), treat it as a store-health incident, not a per-task tamper event:

1. Confirm the store file itself is intact — re-run the health check in Section 7 read-only query. A `SqliteError`/`ENOENT` there (as opposed to individual row mismatches) points to filesystem/disk trouble, a moved or deleted database file, or a botched migration, not tampering.
2. Confirm `CC_VERIFIER_ATTESTATION_DB` is not accidentally pointed at a test/tmp path in the production environment (it is test-override-only; production should always resolve to the `~/system/databases/cc-verifier-attestations.db` default).
3. Once the store itself is confirmed healthy, mismatches are per-task and require the standard recovery above (fresh `cc-verifier` run) rather than any store-wide remediation.

7.3 Hard prerequisite for promotion

Sections 4's `pilot-enforce`/`enforce-new` promotion steps assume the attestation store is healthy. This is a hard, non-negotiable prerequisite, not an additional nice-to-have check: `evaluate()` treats `attestation_missing`/`attestation_mismatch`/`attestation_identity_mismatch` as ordinary `ok:false`

findings, and in an enforcing mode `ok:false` for an enrolled/eligible task blocks `mc.js ready` (Section 4, last paragraph). **If the attestation store is unhealthy (missing, corrupt, or unreachable) at the moment `pilot-enforce`/`enforce-new` is enabled, every verdict — including genuinely correct, untampered ones — will read as `attestation_missing` and be blocked.** Before adding any task id to `gate_pilot_enforce_task_ids` or setting `gate_enforce_new_watermark_epoch_ms`:

1. Run the Section 7 health check and confirm it returns a row count without error.
2. Confirm the candidate task's own audit log entries (Section 2) show `PASS` outcomes with no `attestation_*` reasons at the current revision — a healthy store but a stale/never-attested verdict for this specific task still means "re-run `cc-verifier` first," not "promote anyway."
3. If the store is unhealthy, fix it (Section 7.2) and obtain a fresh attested `PASS` before promoting — never promote to an enforcing mode "around" a known-unhealthy store on the theory that observe-mode telemetry looked fine; observe mode never blocks, so it cannot surface this failure mode the way an enforcing mode will.

Current-session hard-enforcement gap and required remediation

Finding

The verifier decision logic, durable handoff, attestation, and observe-mode ready-gate work as reviewed. However, changing `gate_rollout_mode` alone does **not** guarantee enforcement over every already-running or non-Claude session.

At the time of diagnosis, three native Claude Code processes were active and all had started before the latest `~/.claude/settings.json` installation. The active Bilko task `#105568` had no matching `cc.verification_requested` event. Hook presence in an already-running process is therefore not assumed without an explicit runtime handshake.

The current exact-revision ready gate is invoked by a Claude Code `PreToolUse` hook. It is not yet called authoritatively inside the central `mc.js ready` state transition. Consequently, a stale native session, Pi/non-Claude caller, direct MC invocation, or completion path that does not traverse Claude's Bash hook can bypass the session-local gate.

Local diagnosis artifact:

`/Users/makinja/system/evidence/105472/ENFORCEMENT-GAP-CURRENT-SESSIONS.md`

Required hard-enforcement architecture

1. Move the authoritative exact-revision verifier check into the shared central MC ready transition, before any status mutation.
2. Resolve task identity from the MC operation itself and repository/revision from trusted MC/project state, never from builder-authored payloads.
3. For enrolled tasks, require a matching durable attestation and exact-revision PASS. If absent or stale, reject `ready`, enqueue trusted verification, and require retry after PASS.
4. Apply the same invariant to `done`, merge, and deploy paths that can bypass `ready`.
5. Keep the Claude `PreToolUse` hook as early operator feedback, not as the trust boundary.
6. Register each spawned session with task ID, PID/session ID, repository, starting revision, and hook-version handshake so coverage is observable.
7. Treat existing pre-watermark sessions as untrusted until they register or encounter the central state-transition gate; restarting or killing them is not required.
8. Keep observe/non-enrolled flows fail-open. Once explicitly enrolled in `pilot-enforce` or eligible under `enforce-new`, central MC enforcement must fail closed if verifier identity, revision, verdict, or attestation cannot be established.

Corrected rollout statement

The installed system currently provides independently accepted **observe-only** verification and proven pilot-enforcement decision behavior. It must not be described as universal hard enforcement across all active sessions until the central MC transition remediation above is implemented and independently reviewed.

Revision #4
Created 2026-07-13 12:06:52 UTC by John
Updated 2026-08-10 07:37:38 UTC by John