Arc Science
Technical report

Arc Harness

Arc Harness contributors

Abstract. Arc Harness is a scientific harness that never grades its own work. Its unit is the claim. A claim is a typed record that carries its evidence, an evidence level, traced numbers and a rung on a ladder of formality that runs from a typed record through script-checked obligations and seeded property tests to a Lean kernel proof. A claim graph links claims, evidence, sources and experiments. Verdicts come from a checker that did not build the work, and code claims are checked by Lean where a statement exists. Statistical parts score branches, sources and routing guesses, and every step enters a hash-chained ledger. A shared core, Arc Kit, serves the same rules to a desktop application and to other agent hosts. A small local rater is designed, and a public benchmark against published agents is planned.

In short

Arc is a scientific harness whose unit is the claim: each claim carries its evidence, its evidence level, its traced numbers and its rung on a ladder of formality. The agent that builds a claim never certifies it; a checker from another model family, a deterministic gate or the Lean kernel does.

1. Claim and evidence

Language models in the harness read sources, run tools and write evidence in a typed form. Small models and plain mathematics then score the claims and choose the next step. The claim graph holds four kinds of node: claims, evidence, sources and experiments. Three kinds of link join them: supports, contradicts and depends on. Each tool output lands as an evidence record with its source, its date and its evidence level, and only the nodes downstream of a change are worked out again.

Evidence levels are ordered from target through architecture, simulated and published to measured. A derived claim takes the lowest level among its own evidence and everything it depends on, so a claim never stands above its weakest support. A dependency cycle fails, and so does a missing dependency or a level that rises along a link. Each supporter moves the log-odds of a claim by its trust times a log likelihood ratio, and inputs that share data count once. The resulting score is a heuristic. An evidence engine with number and source tracing, a retraction check and a null model is planned.

2. Ladder of formality

Every claim names the strength of the check behind it. The ladder has four rungs. On the first rung the claim is parsed into a typed record. On the second a script checks stated obligations, for example that the units on both sides of a sum agree or that the numbers in the prose match the record. On the third a seeded property test exercises the claim. On the fourth the Lean kernel accepts a proof of a stated theorem. The level machine-checked is set only by a passing Lean gate row that carries the hash of the approved statement.

The design splits trust in the same way on every rung. A person approves a specification, which is pinned by its SHA-256 digest. A model writes the artifact, which is untrusted. A deterministic elaborator turns the artifact into a typed form or rejects it, and a deterministic checker compares that form with the specification. Each specification needs three controls: one known-good case that passes, one failing mutant for each obligation, and one past defect that the check would have caught. A check that did not run is recorded as blocked and never as a pass.

3. Independent checker

The builder of a claim never issues its verdict. In the agent cycle the builder records a proposal with a goal, a hypothesis, a boundary, checks and probes, and then freezes the candidate files by their SHA-256 digests. A fresh checker that did not build the work runs the checks and calls the accept gate. The gate compares the agent identifiers that the runtime reports. When the runtime reports none, the gate cannot prove independence and returns unmet. The roadmap adds a rule that the checker must run on a model of another family than the builder, and it records that rule as built after release 0.5.0.

The gate is deterministic. It accepts only when the files still match the freeze, each proposed check comes back word for word with exit code 0 and an output excerpt, every probe passes and no blocking or major finding is open. Missing evidence counts as unmet. From the fourth cycle the gate blocks and the agent asks the operator. On hosts without a hook system the shared tool server enforces the same two rules in code: only a checker other than the builder records an accepted verdict, and every tool call with a side effect needs a consent token from the user.

4. Lean gate

Code claims are checked by Lean wherever a statement exists, and the roadmap names this the top priority. The method follows AlphaProof Code (Google DeepMind, November 2026): a person approves a small trusted statement, a team of agents writes the proof, and a fixed script accepts or rejects it. No agent declares a goal finished. The gate script distrusts the exit code of the build. It scans for unfinished-proof tokens and added axioms outside comments, builds with warnings as errors, reads the axioms of every declaration, compares the statement with a held-out copy that no agent can write, and refuses a file that the build skipped.

The first targets are small: the guard and grant logic of the Rust validators and the harness's own checkers, with invariants such as tamper detection on the ledger chain and the ordering of evidence levels. The roadmap records the gate script and these targets as built and passing after release 0.5.0. A claim in prose that cites Lean must name its theorem, a status tag and what the statement does not cover, because a theorem can pass while it says less than the surrounding text. Machine-code proofs are outside the plan.

5. Statistical parts

The hypothesis tree keeps explicit state for each branch and replays from its recorded events to the same tree. A branch starts from a prior held between 0.35 and 0.65, and each observed check moves the log-odds by a likelihood ratio taken from the registered sensitivity and specificity of that check. The next check is chosen by value of information per unit cost. Branch budgets follow sequential halving: the budget is split over about log n stages, the better branches stay, and each survivor receives a larger share. The published guarantee assumes one fixed reward distribution per branch, so here it serves as a heuristic.

Source trust comes from the two-class Dawid-Skene method, which estimates a sensitivity and a specificity for each source. Gate verdicts enter as fixed known labels and anchor the meaning of yes and no. The method assumes that sources err independently, so sources that copy each other look better than they are. On simulated votes it recovers known reliabilities. Real votes will come from the first recorded missions. A held-out replay gate decides whether a patch to the harness's own policy may go to the operator, by a paired bootstrap lower bound on a fresh slice of frozen cases with a trap set of known false passes.

The routing rule is built. It takes a probability and an uncertainty and returns one of five outcomes: answer now, run one cheap check, open the hypothesis tree, ask the operator, or abstain. Each guess is logged beside the later outcome and scored by a Brier score, and the rule never grants a permission. The small rater that would supply these probabilities is designed and awaits a model choice and a licence check. It would answer typed questions, for example whether a piece of evidence supports a claim. Gate verdicts are its labels, and a fine-tune is planned once a few hundred exist.

6. Hash-chained ledger

Each proposal, freeze, gate verdict and decision is appended to a local ledger in which every line holds the hash of the line before it. A verification call walks the chain and names the first bad entry when the chain breaks. The Trace tab of the desktop application shows the ledger newest first and checks the chain again every 5 seconds while it is open. Each verdict also goes to shared memory after secret-shaped text is redacted. The ledger is tamper-evident for partial edits. It carries no key, so a party that rewrites the whole ledger and its reports defeats it.

7. Arc Kit

Arc Kit is the shared core. It joins the policy, the memory client, the Shelf, the Workshop toolpacks, the gate with its ledger, and the claim graph. The Shelf is a local reference corpus with pinned versions that is searched before a task. Workshop toolpacks load software and tool bundles when a task signal calls for them, under a consent token. The same core reaches a host at four depths: rule files, then agent skills and a tool server, then hooks and a plugin bundle, then a headless command and an agent protocol.

The desktop application is one surface of this system. It is a patched build of an MIT-licensed upstream agent application for Windows, with six plugins that carry the gate, the console, the themes and the figure studio. Other agent hosts such as command-line coding agents and editors receive the same rules through the rule files, the hooks and the tool server. The roadmap records the rule files, the hooks for one coding agent and the tool server as running, and it lists shared skills and a headless task mode as the next step.

8. Memory and gateway

Shared memory is Cammut, a separate server that the user runs. The agent receives a brief of at most six notes of 220 characters, refreshed every 20 minutes, framed as untrusted data and filtered for secret-shaped text. Notes never grant permission. Code allows five labels for a retained note: decision, correction, artifact, verified fact and lesson. It refuses notes that match key or token patterns before any network call and retires stale near-twin notes. One local program on the loopback address 127.0.0.1, port 7424, stands between the hosts, the model providers and the memory server, and it keeps provider keys outside workspace files. When memory does not answer, a circuit breaker opens and the gate still returns its verdict.

9. Evaluation plan

A startup baseline script and a headless benchmark runner are built. The planned public benchmark compares Arc Harness with other agents. It tests whether claims with evidence levels, an independent checker and a hash-chained ledger improve scientific agent work. The baseline script times the runtime and the repository. It does not start the application, so the window start time, the memory use and the interface lag fall outside what it records.

A valid benchmark needs four things. The harness runs headless on one published task set under the same budget as the published agents, and two candidate suites are under consideration. The score is compared with those agents on the same tasks. The calibration of claim scores and routing guesses is reported as a Brier score against outcomes that the gate later decided. The rater is evaluated on held-out labelled claims before and after fine-tuning. The run also needs a provider key and a spending limit approved by the operator. The roadmap states that the word better is used only after that run.

10. Compute and limits

The built parts run on the user's machine. The gate, the claim graph, the tree, the reliability estimate and the routing rule are deterministic code with seeded resampling, and they train no model. Model inference runs at the providers that the user configures. The only training compute in the plan is the fine-tune of the small rater on one cloud L4 graphics card inside 20 to 40 pounds a month, and it needs approval for paid compute. The choice of the rater model comes next, and its parameter count is stated when the choice is made.

The accept gate checks the evidence it receives, so a caller can still report a false exit code, and the excerpts need a reader. The claim and branch scores are heuristics until calibration is measured. Reliability rests on simulated votes. A Lean proof covers only its statement, and the first targets are small validators and checkers. Release 0.5.0 predates the parts recorded as built after it. Automated live checks run on Windows, and the first runs on macOS and Linux come next. The evidence engine, the small rater, the upstream source fork and the public benchmark run, which compares Arc Harness with other agents, are planned.

System design

Sources and tool runs become typed evidence records in a claim graph; an independent checker and the Lean gate set the rung of each claim, and every verdict enters a hash-chained ledger.

Implemented components

- Sources and tool runs.
- Arc Kit surfaces.
- Cammut memory.
- Typed evidence record.
- Local gateway.
- Claim graph.
- Hypothesis tree and branch budgets.
- Dawid-Skene source reliability.
- Independent checker.
- Lean gate.
- Accept gate.
- Ladder rung.
- Hash-chained ledger.

Specified components

- Small rater.
- Evidence engine.
- Public benchmark.

Connections

- Sources and tool runs → Typed evidence record.
- Arc Kit surfaces → Typed evidence record.
- Arc Kit surfaces → Local gateway.
- Local gateway → Cammut memory.
- Cammut memory → Arc Kit surfaces.
- Typed evidence record → Claim graph.
- Hypothesis tree and branch budgets → Claim graph.
- Dawid-Skene source reliability → Claim graph.
- Small rater → Claim graph.
- Evidence engine → Claim graph.
- Claim graph → Independent checker.
- Claim graph → Lean gate.
- Independent checker → Accept gate.
- Lean gate → Ladder rung.
- Accept gate → Ladder rung.
- Accept gate → Dawid-Skene source reliability.
- Accept gate → Small rater.
- Accept gate → Cammut memory.
- Accept gate → Hash-chained ledger.
- Ladder rung → Hash-chained ledger.
- Arc Kit surfaces → Public benchmark.

Source

This report is the printed form of the page https://arcscience.app/arc-harness/. The page is the source of record for every statement in it.