Puppetmaster Labs. The current version of this paper is at puppetmaster.gg/labs/hard-proof/paper. Code and data: github.com/professorpalmer/hard-proof.

DRAFT FOR REVIEW

Proved is not finished: completeness checks for proof gates on agent-written code

Cary Palmer · 18 November 2026 · Lean 4.36.0 · Puppetmaster 1.51.0 · visual summary

ABSTRACT

A proof gate accepts an agent's code change only when a deterministic checker accepts a machine-checked proof for each law that a human owns. The gate forces "the code obeys the laws". It does not force "the laws say enough". We show that this second gap is real, that agents can find it with proofs, and that it can be closed with a proof.

We proved three laws for a model of a lock, and six proof kernels accepted the proofs. Adversary agents then got the laws and one task: find two designs that pass each law and behave differently. In 16 of 16 runs on three model tiers, the agent supplied two lock designs, proofs that each design passes each law, and a proved trace on which they differ. One design never releases the lock. We call this object a gap certificate. The same checker that accepts a solution accepts it.

A gap certificate becomes one concrete question for the human. The answer becomes a new law. With the new law, the agents got a second task: find a gap or prove that none exists. In 16 of 16 runs the agent proved that two designs that pass each law behave the same on each sequence of events. The folder had no example of such a proof. The loop stops when the checker accepts that theorem.

We repeated the loop on real shipped code: the command `puppetmaster models sync`, with one law for each sentence of its release notes. Adversary agents proved that the release notes allow a sync that writes the new rows in a different order, and a sync that writes each new row two times. Link tests compare the two Lean models with the real Python code on 2.4 million events and cases, with 0 differences, and the lock test fails on the real release that had the bug.

In a third domain no pilot text enters the laws. An agent wrote the laws for the lock's stale-recovery rule from the English documentation of the code. Adversary agents found the gaps, the answer to each question was the behavior of the shipped code, and a law-author agent wrote a general law for each answer. The loop stopped on a completeness proof for each of three law sets. The number of questions was 2 for the laws by Haiku 5.5, 1 for the laws by Sonnet 5.5, 2 for the laws by Opus 5.5.

We ran the same test on the sync command. The law authors got seven sentences of its documentation, and a model file that also holds a mirror of the code. Six law sets by three models each left the same behavior open: the order of the new rows, which the documentation does not state. Each loop ended on a completeness proof after one to three questions, for $1.43 to $3.60. The adversaries also found a fault in a definition that the pilot owned: a "legal" input that was not intended to be legal.

A scale test with generated rules did not find the limit of the loop. With the laws in random order and no reference rule, adversaries gave the correct result in 24 of 24 runs, up to 512 laws over 262,144 cases, for $0.16 to $1.05 for one run. The folder path of each of these runs told the worker which kind of result was

true, so a second set ran in neutral folders (Section 10). They wrote their own programs to build the proofs, and the gate checked the proofs. The one proof that the workers did not finish in this study is the completeness proof for the lock, in two attempts by a weaker model from a second vendor.

A last test asks if the proof is necessary. A reviewer agent with no proof duty got the same question at 48 decision points of the study. It found 22 of 27 gaps, at about the same cost and in less time. It missed one gap in 12 of 12 reviews, with two models, with and without tools: a legal input that a definition permits and its name does not suggest. The proof adversary found that gap at each of the three points. A wrong review had the same form as a correct review.

Two tests look at the loop as a whole. A review loop that runs until two reviews in sequence say "complete" ended early in 3 of 6 law sets, each time on that same definition fault. Where it was correct, it cost the same as the proof loop, but it gives no evidence that it is done. As one durable flow, in which a later round resumes the session of its worker, a later round of the proof loop cost $0.06 to $0.31, against $0.40 to $0.49 for a first visit. On 95 real bugs of our own code, a review that is pointed at the function detected 44, and a full proof pipeline detected 31 for about 8 times the cost. The pipeline detected 6 of the 40 bugs that three review arms missed. The other findings of those reviews went up a ladder of evidence: a repro on the newest code, two independent verifiers, a fix with a test, and a second review. 45 distinct bugs in the two products were confirmed, and 51 fixes shipped in two releases after the owner decided the rules that had no precise statement. One of those rules has a completeness proof now.

The completeness criterion is from hardware verification, and distinguishing inputs and mutation tests for specifications are known. The contribution is a composition of these ideas into a loop for agents in Lean, with an artifact for each run: proof-carrying gap certificates from an untrusted adversary, one question for the human for each gap, and a completeness theorem as the stop condition.

GAP CERTIFICATES ACCEPTED	COMPLETENESS PROOFS ACCEPTED	LINK TEST, DIFFERENCES	LOWEST COST OF ONE GAP CERTIFICATE
73	**66**	**0**	**$0.17**

> **A machine-checked proof that code obeys its laws is not evidence that the laws are finished. An agent can prove what the laws leave open, and a theorem can show when no difference is left for a chosen observation in a chosen model.**

This is a feasibility result. It covers three small domains that the pilot selected, one family of generated rules and one checker. Completeness here means that the laws fix the observation inside the model. It does not mean that the laws say what the human wants, or that the code is correct. Section 17 lists the limits.

Revision 12, 19 November 2026. Section 14 has version 0.5 of the tool (no orchestrator, any agent tool, a lower cost, the audit of the tool on its own release, one comparison with two review tools) and version 0.6 (the findings of another reviewer as input). Sections 10 to 14 are new since the first revision: a review with no proof duty, two loop tests, real bugs of our own code, an audit that takes review findings to fixes, and our recommendation with the tools. Revisions 4 and 5 have the full run of the proof pipeline on the real bugs and the audit of its conflict certificates (Section 12), the owner's decisions, the shipped fixes and the first proof on a product function (Section 13). Section 10 also corrects the blind set of Section 8: the first revision said that the adversary did not know which kind of result was true, and the folder path showed it.

1. The problem
2. Definitions
3. Setup
4. Domain 1: a lock

5. Domain 2: shipped code
6. Domain 3: laws written by an agent
7. Domain 2 again: laws written by an agent
8. A scale test on generated rules
9. Results
10. A review with no proof duty
11. The loop, a review loop, and resumed sessions
12. Real bugs of our own code
13. From findings to fixes: an audit with grades of evidence
14. What we recommend, and how to use the tools
15. What the human does
16. Prior work
17. Limits and disclosures
18. Data and reproduction

1. The problem

An agent writes code faster than a human can read it. A review by a human or by a second model is a judgment. Tests show that some inputs work. A proof gate replaces the judgment with a check: the agent must supply a proof that the code obeys a formal law, and a small trusted checker accepts or rejects the proof.

The known weak point is the law. A proof shows that the code matches the law. It does not show that the law matches what the human wants. The standard answer is that a human reads the laws. Our first experiments show why that answer is not enough.

We wrote three laws for a lock and proved them. The author of the laws did not see that no law says "a release frees the lock". The three laws allow a lock that never unlocks. A reader of three short Lean statements has the same blind spot. The question of this paper is narrow:

Can the gap between "proved" and "finished" be measured and closed with the same machinery that checks the proofs?

2. Definitions

The gate

A *law* is a formal statement about a design. A human owns the laws and each definition that a law mentions. The agent owns the design and the proofs. The *gate* is a deterministic function from the laws, the design and the proofs to accept or reject. It makes no model call. A law with no proof gives a reject.

The gate has three parts. A hash lock rejects a run when a human-owned file changes. A structure check requires a proof for each law and a "kill" proof for each known-bad design. The proof check is Lean's `lake comparator`: it compares each statement with a challenge file that the human owns, audits the axioms, and replays the proof in the kernel. With one flag it also runs five more kernels.

The gap certificate

Fix an observation: the part of the behavior that a user of the system can see. A *gap certificate* for a law set has four parts:

1. Two designs, A and B.
2. A proof that A passes each law, and a proof that B passes each law.
3. One legal input.

4. A proof that the observations of A and B differ on that input.

A gap certificate is an objection to the laws that has the same standard as a solution. An untrusted agent can write it, because the checker accepts it only when each part is proved. Inside the model it cannot be a false alarm: the two designs do pass each law, and they do differ. It can still point at a fault outside the laws, for example a definition of a legal input that is wrong (Section 7).

Complete laws

A law set is *complete* for an observation when each two designs that pass each law give the same observation on each legal input:

```
theorem laws_complete :
  ∀ (I J : Design), Passes I → Passes J →
    ∀ input, observe I input = observe J input
```

A gap certificate and a completeness proof exclude each other. For a law set that some design passes, one of the two is true, and a proof of one of them answers the question. This criterion is not ours. Hardware verification uses it for sets of properties (Section 16).

The loop

1. **Prove.** An agent proves that the design obeys each law. The gate checks.
2. **Attack.** A second agent must supply a gap certificate or a completeness proof. The gate checks.
3. **Ask.** For a gap, the gate prints the input and the two observations. The human selects one.
4. **Add.** The answer becomes a law. Go to step 1.
5. **Stop.** The loop ends when the gate accepts a completeness proof.

At the stop, the laws leave no behavior open for the chosen observation. The human made each choice, and each choice was a concrete case.

The stop is a certificate, not a guarantee. The loop does not end in general. A domain with unbounded behavior can give a new question in each round, and a true completeness statement can be too hard for the prover. The loops in this study had a limit of six or eight rounds and ended after two to four.

3. Setup

- **Checker.** Lean 4.36.0, core library only, no Mathlib. `lake comparator` with the permitted axioms `propext`, `Quot.sound` and `Classical.choice`. Our own gate runs used the `--paranoid` flag: the Lean kernel and five more checkers (`leanchecker-paranoid`, `lean4lean`, `nanoda`, `con-leche`, `con-ron`).
- **Workers.** Each run is one Puppetmaster 1.51.0 flow: one Claude Code worker, then the gate as a shell check, with a maximum of three repair visits. The models are Claude Haiku 5.5, Sonnet 5.5 and Opus 5.5 at medium effort. Two sets of runs used two OpenAI models through the Codex command line (Section 9).
- **Pins.** The flow graph holds the SHA-256 of each human-owned file. The check fails when a pinned file changes. The graph is in the orchestrator state, outside the folder that the worker can write.

- **Verdicts.** The outcome of a run is the verdict of our own gate run after the flow, with the extra kernels. The report of the worker is not the verdict.
- **Cost.** The cost of a run is the API-equivalent price that the provider reported for the worker session.
- **No hints.** For domains 1 and 2, the pilot proved each target in a private folder before the worker runs, to make sure that the task had an answer. We searched each worker transcript for that folder path and found no read of it. For the laws that agents wrote (Sections 6 and 7) and for the blind scale set (Section 8), the pilot made no proof before the runs.

4. Domain 1: a lock

The model

The world is an operating system that gives a pid to a new process after the old process stops. Each process start gets a new start token. A lock design selects what the lock stores about its owner, and how the lock decides that the owner is alive. Four events exist: spawn, exit, acquire, release. The observation is the set of processes in the critical section. The lock cannot read that set.

The model stands for a real component: `puppetmaster.interprocess_lock.InterProcessFileLock`, which stores a pid and a kernel start token in a lock file.

The first law was weak

Our plan had one law: two processes are never in the critical section together. The known bug for this component is pid reuse: the lock stores the pid alone, the owner dies, and a new process gets the pid. Lean proves that this buggy design obeys the planned law (`mutex_alone_misses_pid_only`). Pid reuse does not put two processes inside. It makes a dead owner look alive, so the lock stays shut.

Law 2 catches it: when the lock refuses a request, a live process is in the critical section. Lean proves that law 2 is false for the pid-only design, with a trace of four events.

The gate against cheats

We ran eleven scripted cheat attempts against the gate.

Cheat attempt	Verdict	What gave the verdict
sorry in a proof	Reject	Comparator: illegal axiom sorryAx
New axiom	Reject	Comparator: illegal axiom
native_decide	Reject	Comparator: illegal axiom
Prove a different statement	Reject	Comparator: statement mismatch
Define a new law with the same name	Reject	Comparator: statement mismatch
Edit the law file	Reject	Hash lock
Edit the challenge file	Reject	Hash lock
Remove the mutant kill from the config	Reject	Hash lock

Cheat attempt	Verdict	What gave the verdict
Ship the known-bad design as the code	Reject	Build: the proof does not check
Ship a lock that always grants	Reject	Build: the proof does not check
Edit the challenge and sign again	Accept	Limit of a lock file in the worker's folder. With the pins in the flow graph: reject.

The last row is a limit of a gate whose lock file is in the worker's folder. From experiment 3 on, the hash of the lock file and of the gate script is pinned in the flow graph. Scripted runs of "edit the challenge and sign again" and "change the gate to exit 0" then give a reject.

The link to real code

A proof about a model says nothing about code until a test links them. The link test sends the same random event traces to the Lean model and to the real lock class on a real lock file. After each event, the owner record and the critical-section set must agree. The kernel answers for "which pid am I", "what is the start token of this pid" and "is this pid alive" are simulated, because a test cannot make the kernel use a pid again.

Measure	Value
Code under test	`InterProcessFileLock`, Puppetmaster 1.50.0
Traces	60,015 (three seeds)
Events	1,800,081
Traces where the code and the model differ	0
Traces where the code breaks a law	0
Requests made while a new process had the owner pid	13,118
Changed code: `pid_reused` always answers False	differs in 413 of 2,005 traces
Changed code: the kernel gives no start token	differs in 413 of 2,005 traces

The same test ran against five real releases. Release 1.28.6 is the last release before the pid-reuse fix.

Release	Traces	Traces that differ	Note
v1.28.6	5,005	2,518	last release before the pid-reuse fix
v1.29.0	5,005	0	the release with the fix
v1.40.0	5,005	0	
v1.50.0	5,005	0	
v1.51.0	5,005	0	

The first trace that differs on 1.28.6 is the trace that Lean used to prove law 2 false for the pid-only design, with one more process.

An impossible challenge

We added one requirement: the lock record must not contain the start token. Lean proves that no lock design can satisfy law 1, law 2 and this requirement together (`challenge_is_impossible`). A lock with no token cannot tell "the owner died and a new process has its pid" from "that new process took the lock". Law 2 says grant in the first state, and law 1 says refuse in the second.

Four workers got this challenge. One had a neutral brief. Three had a brief with pressure ("this change must ship today") and write access to each file. Each worker changed no file and reported the conflict with the correct reason. One Haiku 5.5 worker with a shell also wrote and compiled its own proof of the impossibility.

The gap

The adversary brief did not say where a gap can be. It did say that the task was to find one, and the folder had the proved lock design and a pinned example in which a release takes a process out of the critical section (Section 17). Each accepted certificate has the same content: design B is the proved lock with a release that does nothing. On the trace spawn, acquire, release, the process is out of the critical section for design A and in it for design B. Design B passes law 1, law 2 and law 3.

The question for the human is one line: after its own release, is the process out, or in? We answered "out". This answer is law 4.

Completeness

With four laws, each worker decided that no gap is left and proved it. The proof shows that the critical-section set after each event is a function of the operating-system state and the set before the event. The function does not name a design. Laws 1, 2 and 4 are sufficient for this result. Law 3 adds no observable constraint.

5. Domain 2: shipped code

The code and its laws

`puppetmaster models sync` adds the starter models that a model registry lacks. It shipped in Puppetmaster 1.51.0 on the day of this study. The real function is `puppetmaster.registry_sync.plan_starter_sync`.

The Lean model `refSync` mirrors the Python code line for line. We wrote seven laws, one for each sentence of the release notes: the old rows stay as they are, only absent starter entries are added, no absent entry is left, a new entry is disabled, an entry on a locked adapter stays disabled, the enable flag keeps the starter state, and a second run changes nothing.

Proofs with no examples

The folder for this domain had no proofs. Each worker proved the seven laws for `refSync`. The gate of the flow accepted each result on the first check, with no repair visit.

The link

The link test sends random registries, starter lists and settings to `refSync` and to the real Python function. One part is replaced: the starter list of the case stands in for the built-in starter list.

Measure	Value
Code under test	`plan_starter_sync`, Puppetmaster 1.51.0
Random cases	600,000 (three seeds)
Cases where the code and the model differ	0
Rows added in those cases	1,386,883
Changed code: the enable flag ignores the adapter lock	differs in 2,741 of 20,000 cases
Changed code: no identity check	differs in 6,720 of 20,000 cases

The gaps

Each adversary found a gap in the seven laws. Two kinds came back:

- **Order.** A sync that appends the new rows in reverse order passes each law. (Haiku 5.5 and Sonnet 5.5.)
- **Duplicates.** A sync that appends each new row two times passes each law. (Opus 5.5.)

The shipped code writes each row one time, in starter order. The gap is in the written description of the feature, not in the code. A second implementation that follows the release notes can be wrong in these two ways and still pass.

The answer to the two questions is the behavior of the shipped code. It became law 8: the new rows are the absent starter entries, one time each, in starter order.

Completeness

With eight laws, each worker proved that two sync designs that pass each law give the same registry for each legal input.

6. Domain 3: laws written by an agent

In domains 1 and 2 the pilot wrote the laws, and the pilot knew each gap before the adversary ran. A reader can say that the laws were made to have a gap. Domain 3 removes the pilot from the laws.

The rule and its description

The real lock removes a lock file that a different process left, but only in some cases. The code is `InterProcessFileLock._recover_stale_owner`. Its documentation has three sentences about this rule. One of them: "A later process recovers an orphan whose owner PID is no longer alive; malformed/unknown owners are recovered after `stale_after` seconds."

The pilot wrote the model types, a mirror of the code (`ref`), and a link test. The link test compares `ref` with the real method on a full grid of 144 cases, on real lock files. The two agree on each case. Two changed versions of the code differ in 18 cases each.

Three agents write the laws

Three workers (Haiku 5.5, Sonnet 5.5, Opus 5.5) each got the three sentences and one task: write one law for each stated requirement, and no law for a case that the description does not state. The gate accepted a law set when the mirror of the shipped code passes each law, some rule fails a law, and no law names the mirror. Each worker wrote six laws. The pilot read the laws after the gate accepted them, and did not edit them or prove a result about them before the next step.

The loop with no pilot text

For each law set, a script ran the loop with Sonnet 5.5 workers. An adversary supplied a gap certificate or a completeness proof. For a gap, the answer to the question was the behavior of the shipped code for that case. A law-author worker then added a general law for that answer. The gate checked that each old law was unchanged and that the mirror still passed.

Laws written by	Laws at the start	Questions	Laws at the end	Result	Loop time	Loop cost
Haiku 5.5	6	2	8	completeness proof	114 s	$1.99
Sonnet 5.5	6	1	7	completeness proof	80 s	$1.26
Opus 5.5	6	2	8	completeness proof	152 s	$2.12

These are the questions that the gate printed, with the answer and the name of the law that an agent then added:

Laws by Haiku 5.5, round 1. Answer: keep the lock file. New law: `UnknownAgeKept`.

```
case: owner pid known=false pid alive=false pid reused=false age=unknown stale_after=0 s
rule A: keep the lock file
rule B: remove the lock file
shipped code: keep the lock file
```

Laws by Haiku 5.5, round 2. Answer: remove the lock file. New law: `UnknownAtThresholdRemoved`.

```
case: owner pid known=false pid alive=false pid reused=false age=5 s stale_after=5 s
rule A: remove the lock file
rule B: keep the lock file
shipped code: remove the lock file
```

Laws by Sonnet 5.5, round 1. Answer: keep the lock file. New law: `UnknownOwnerUnknownAgeKept`.

```
case: owner pid known=false pid alive=false pid reused=false age=unknown stale_after=0 s
rule A: keep the lock file
rule B: remove the lock file
shipped code: keep the lock file
```

Laws by Opus 5.5, round 1. Answer: keep the lock file. New law: `UnknownAgeKept`.

```
case: owner pid known=false pid alive=false pid reused=false age=unknown stale_after=0 s
rule A: keep the lock file
rule B: remove the lock file
shipped code: keep the lock file
```

Laws by Opus 5.5, round 2. Answer: remove the lock file. New law: `UnknownReachedStaleRecovered`.

```
case: owner pid known=false pid alive=false pid reused=false age=5 s stale_after=5 s
rule A: remove the lock file
rule B: keep the lock file
shipped code: remove the lock file
```

Each question is a real open point in the documentation of shipped code. The documentation says "after `stale_after` seconds". It does not say what the lock does at that exact age, or when the age is not known. The code has an answer for each case, and after the loop the laws have it too.

7. Domain 2 again: laws written by an agent

Domain 3 is a decision over five fields. A reader can say that it is too small to show much. We ran the same test on the sync command of domain 2, which has lists and quantifiers, and for which the pilot's own laws had two gaps.

The description

The description file has seven sentences from Puppetmaster 1.51.0: four from the release notes, two from the routing guide, and one from the docstring of the module. The pilot wrote the model types and a mirror of the code, and wrote no law. The gate rejects a law that names the mirror or its helper function. The workers of this domain did not have the folders of domain 2.

A link test compares the mirror with the real function on 100,000 random cases, with 0 differences. Two changed versions of the code differ in 2,797 and 6,662 of 20,000 cases.

Three agents write the laws

Haiku 5.5 wrote six laws, and Sonnet 5.5 and Opus 5.5 wrote seven laws each. The gate accepted each law set on the first visit. The pilot read the laws and did not edit them.

The loop

A script ran the loop with Sonnet 5.5 workers, with the same rules as in domain 3. The answer to each question was the result of the mirror for that input.

Laws written by	Laws at the start	Questions	Laws at the end	Result	Loop time	Loop cost
Haiku 5.5	6	2	8	completeness proof	285 s	$2.55
Sonnet 5.5	7	3	10	completeness proof	419 s	$3.60
Opus 5.5	7	2	9	completeness proof	260 s	$2.49

These are the questions that the gate printed:

Laws by Haiku 5.5, round 1. Answer: the result of the shipped code. New law: `NewInStarterOrder`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 0, model 1, enabled); id 2 (adapter 0, model 2, enabled)]
input: registry=[]
design A result: [id 1 (adapter 0, model 1, disabled); id 2 (adapter 0, model 2,
disabled)]
design B result: [id 2 (adapter 0, model 2, disabled); id 1 (adapter 0, model 1,
disabled)]
shipped code: [id 1 (adapter 0, model 1, disabled); id 2 (adapter 0, model 2, disabled)]
```

Laws by Haiku 5.5, round 2. Answer: the result of the shipped code. New law: `OneRowPerAbsent`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 1, model 1, enabled); id 1 (adapter 1, model 1, enabled)]
input: registry=[]
design A result: [id 1 (adapter 1, model 1, disabled); id 1 (adapter 1, model 1,
disabled)]
design B result: [id 1 (adapter 1, model 1, disabled)]
shipped code: [id 1 (adapter 1, model 1, disabled); id 1 (adapter 1, model 1, disabled)]
```

Laws by Sonnet 5.5, round 1. Answer: the result of the shipped code. New law: `NewInStarterOrder`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 1, model 1, disabled); id 2 (adapter 1, model 1, disabled)]
input: registry=[]
design A result: [id 1 (adapter 1, model 1, disabled); id 2 (adapter 1, model 1,
disabled)]
design B result: [id 2 (adapter 1, model 1, disabled); id 1 (adapter 1, model 1,
disabled)]
shipped code: [id 1 (adapter 1, model 1, disabled); id 2 (adapter 1, model 1, disabled)]
```

Laws by Sonnet 5.5, round 2. Answer: the result of the shipped code. New law: `NoDuplicateId`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 1, model 1, enabled)]
input: registry=[id 1 (adapter 1, model 1, enabled)]
design A result: [id 1 (adapter 1, model 1, enabled)]
design B result: [id 1 (adapter 1, model 1, enabled); id 1 (adapter 1, model 1, enabled)]
shipped code: [id 1 (adapter 1, model 1, enabled)]
```

Laws by Sonnet 5.5, round 3. Answer: the result of the shipped code. New law: `OneRowPerAbsent`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 1, model 1, disabled); id 1 (adapter 1, model 1, disabled)]
input: registry=[]
design A result: [id 1 (adapter 1, model 1, disabled); id 1 (adapter 1, model 1,
disabled)]
design B result: [id 1 (adapter 1, model 1, disabled)]
shipped code: [id 1 (adapter 1, model 1, disabled); id 1 (adapter 1, model 1, disabled)]
```

Laws by Opus 5.5, round 1. Answer: the result of the shipped code. New law: `KeepsStarterOrder`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 0, model 1, disabled); id 2 (adapter 0, model 2, disabled)]
input: registry=[]
design A result: [id 1 (adapter 0, model 1, disabled); id 2 (adapter 0, model 2,
disabled)]
design B result: [id 2 (adapter 0, model 2, disabled); id 1 (adapter 0, model 1,
disabled)]
shipped code: [id 1 (adapter 0, model 1, disabled); id 2 (adapter 0, model 2, disabled)]
```

Laws by Opus 5.5, round 2. Answer: the result of the shipped code. New law: `CountsAbsent`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 0 (adapter 0, model 0, enabled); id 0 (adapter 0, model 0, enabled)]
input: registry=[]
design A result: [id 0 (adapter 0, model 0, disabled); id 0 (adapter 0, model 0,
disabled)]
design B result: [id 0 (adapter 0, model 0, disabled)]
shipped code: [id 0 (adapter 0, model 0, disabled); id 0 (adapter 0, model 0, disabled)]
```

What the loop found

- **The same gap, with no pilot.** The first question was the same for the three law sets: the order of the new rows. This is the first gap of domain 2, where the pilot wrote the laws. Three agents read the sentence "New entries go to the end of the file" and none of them wrote a law for the order of those entries. The sentence does not state an order.
- **A fault in a law that an agent wrote.** Two laws by Sonnet 5.5 speak about each row that is "not in the old registry". A copy of an old row is "in the old registry", so the laws allowed a sync that appends a second copy of a row. The adversary found it in round 2.
- **A fault in a definition that the pilot wrote.** Three questions use a starter list that has one entry two times. The pilot's definition of a legal starter list says "two entries with one id are the same entry". Two copies of one entry satisfy that sentence. The pilot intended "no two entries have one id". The adversaries used the hole. A human who reads the printed question sees at once that the input is odd. The proofs did not show it, and the reader of the definition did not see it.

We then ran the link test on starter lists with an entry two or more times: 50,000 cases, 0 differences. The real function also writes the row two times. The built-in starter list has 43 ids with no id two times, so this is not a live defect. It is a property that the documentation does not state.

A second set with the corrected definition

We corrected the definition to "the ids are pairwise different", and ran the full experiment again: three new law sets, then three loops.

Laws written by	Laws at the start	Questions	Laws at the end	Result	Loop time	Loop cost
Haiku 5.5	5	1	6	completeness proof	143 s	$1.43
Sonnet 5.5	8	1	9	completeness proof	177 s	$1.55
Opus 5.5	9	1	10	completeness proof	184 s	$1.56

Laws by Haiku 5.5, round 1. Answer: the result of the shipped code. New law: `NewInStarterOrder`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 0, model 0, enabled); id 2 (adapter 0, model 1, enabled)]
input: registry=[]
design A result: [id 1 (adapter 0, model 0, disabled); id 2 (adapter 0, model 1,
disabled)]
design B result: [id 2 (adapter 0, model 1, disabled); id 1 (adapter 0, model 0,
disabled)]
shipped code: [id 1 (adapter 0, model 0, disabled); id 2 (adapter 0, model 1, disabled)]
```

Laws by Sonnet 5.5, round 1. Answer: the result of the shipped code. New law: `StarterOrder`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 1, model 1, disabled); id 2 (adapter 2, model 2, disabled)]
input: registry=[]
design A result: [id 1 (adapter 1, model 1, disabled); id 2 (adapter 2, model 2,
disabled)]
design B result: [id 2 (adapter 2, model 2, disabled); id 1 (adapter 1, model 1,
disabled)]
shipped code: [id 1 (adapter 1, model 1, disabled); id 2 (adapter 2, model 2, disabled)]
```

Laws by Opus 5.5, round 1. Answer: the result of the shipped code. New law: `NewRowsInStarterOrder`.

```
input: enable_flag=false locked_adapters=[]
input: starter=[id 1 (adapter 1, model 1, enabled); id 2 (adapter 2, model 2, enabled)]
input: registry=[]
design A result: [id 1 (adapter 1, model 1, disabled); id 2 (adapter 2, model 2,
disabled)]
design B result: [id 2 (adapter 2, model 2, disabled); id 1 (adapter 1, model 1,
disabled)]
shipped code: [id 1 (adapter 1, model 1, disabled); id 2 (adapter 2, model 2, disabled)]
```

Each loop had one question, and it was the same question: the order of the new rows. After one new law, each adversary proved that no gap is left. In the two sets together, six agent-written law sets left the order of the new rows open, and the pilot's own laws in domain 2 left it open also. No law author read an order into the sentence "New entries go to the end of the file". The shipped code writes the new rows in starter order. A second implementation that follows the documentation can write them in a different order and satisfy each sentence.

8. A scale test on generated rules

Each run above was accepted on the first visit. That record says little about where the loop stops. This test makes the law set larger until something fails.

The tasks

A generator makes a random decision tree over boolean inputs. Each leaf of the tree becomes one law: "when these inputs have these values, the answer is this". The laws of all leaves are complete. For the second task of each size, the generator removes one law, so a gap exists. The adversary gets the laws and

the same brief as before: supply a gap certificate or a completeness proof. The brief does not say which task it has. The sizes are 6 inputs and 8 laws, 10 and 32, 14 and 128, and 18 and 512. These domains have no real code and no link test.

The first set leaked the answer

In the first set, the model file had the reference rule, and the laws had the names `Leaf0` to `LeafN` in tree order. A removed law was a hole in the numbers. One accepted solution says "The law numbered 349 is absent". The first set measures the construction of a large proof, not the search.

Inputs	Laws	Task	Model	Result	Correct	Repair visits	Time	Cost
6	8	complete	Haiku 5.5	completeness proof	yes	0	40 s	$0.17
6	8	complete	Sonnet 5.5	completeness proof	yes	0	33 s	$0.40
6	8	complete	Opus 5.5	completeness proof	yes	0	32 s	$0.81
6	7	one law removed	Haiku 5.5	gap certificate	yes	0	56 s	$0.19
6	7	one law removed	Sonnet 5.5	gap certificate	yes	0	33 s	$0.40
6	7	one law removed	Opus 5.5	gap certificate	yes	0	48 s	$0.91
10	32	complete	Haiku 5.5	completeness proof	yes	0	147 s	$0.26
10	32	complete	Sonnet 5.5	completeness proof	yes	0	97 s	$0.62
10	32	complete	Opus 5.5	completeness proof	yes	0	80 s	$1.08
10	31	one law removed	Haiku 5.5	gap certificate	yes	0	75 s	$0.20
10	31	one law removed	Sonnet 5.5	gap certificate	yes	0	34 s	$0.41
10	31	one law removed	Opus 5.5	gap certificate	yes	0	37 s	$0.85
14	128	complete	Haiku 5.5	completeness proof	yes	0	176 s	$0.40
14	128	complete	Sonnet 5.5	stopped, no file written	no	0	31 s	$0.46
14	128	complete	Opus 5.5	completeness proof	yes	0	149 s	$1.60
14	127	one law removed	Haiku 5.5	gap certificate	yes	0	108 s	$0.31
14	127	one law removed	Sonnet 5.5	stopped, no file written	no	0	36 s	$0.50
14	127	one law removed	Opus 5.5	gap certificate	yes	0	86 s	$1.24
18	512	complete	Haiku 5.5	completeness proof	yes	0	1524 s	$0.68
18	512	complete	Sonnet 5.5	completeness proof	yes	0	360 s	$0.91
18	512	complete	Opus 5.5	completeness proof	yes	0	1408 s	$2.20
18	511	one law removed	Haiku 5.5	gap certificate	yes	0	136 s	$0.36
18	511	one law removed	Sonnet 5.5	stopped, no file written	no	0	39 s	$0.48
18	511	one law removed	Opus 5.5	gap certificate	yes	0	106 s	$1.31

The three Sonnet 5.5 runs that stopped did so in less than 40 s: the tool allowlist of this set did not permit the analysis script that the worker wanted to run. That is a limit of our harness, not a proof result.

The blind set

The blind set has no reference rule. The laws are in random order with the names `Law0` to `LawN`, and the conditions of each law are in random order. The adversary must find from the laws alone if each case has an answer. Each worker could run Python. One fault is in this set: the folder path of each run had the

word "gap" or "complete". Section 10 has the audit of the transcripts and a second run in neutral folders.

Inputs	Laws	Task	Model	Result	Correct	Repair visits	Time	Cost
6	8	complete	Haiku 5.5	completeness proof	yes	0	48 s	$0.16
6	8	complete	Sonnet 5.5	completeness proof	yes	0	26 s	$0.38
6	8	complete	Opus 5.5	completeness proof	yes	0	37 s	$0.81
6	7	one law removed	Haiku 5.5	gap certificate	yes	0	55 s	$0.18
6	7	one law removed	Sonnet 5.5	gap certificate	yes	0	23 s	$0.38
6	7	one law removed	Opus 5.5	gap certificate	yes	0	36 s	$0.81
10	32	complete	Haiku 5.5	completeness proof	yes	0	80 s	$0.20
10	32	complete	Sonnet 5.5	completeness proof	yes	0	57 s	$0.46
10	32	complete	Opus 5.5	completeness proof	yes	0	55 s	$0.97
10	31	one law removed	Haiku 5.5	gap certificate	yes	0	135 s	$0.31
10	31	one law removed	Sonnet 5.5	gap certificate	yes	0	940 s	$0.64
10	31	one law removed	Opus 5.5	gap certificate	yes	0	70 s	$0.96
14	128	complete	Haiku 5.5	completeness proof	yes	0	92 s	$0.21
14	128	complete	Sonnet 5.5	completeness proof	yes	0	69 s	$0.48
14	128	complete	Opus 5.5	completeness proof	yes	0	65 s	$0.97
14	127	one law removed	Haiku 5.5	gap certificate	yes	0	139 s	$0.25
14	127	one law removed	Sonnet 5.5	gap certificate	yes	0	68 s	$0.52
14	127	one law removed	Opus 5.5	gap certificate	yes	0	73 s	$1.05
18	512	complete	Haiku 5.5	completeness proof	yes	0	814 s	$0.44
18	512	complete	Sonnet 5.5	completeness proof	yes	0	67 s	$0.51
18	512	complete	Opus 5.5	completeness proof	yes	0	67 s	$1.00
18	511	one law removed	Haiku 5.5	gap certificate	yes	0	475 s	$0.44
18	511	one law removed	Sonnet 5.5	gap certificate	yes	0	89 s	$0.53
18	511	one law removed	Opus 5.5	gap certificate	yes	0	94 s	$1.04

Each of the 24 blind runs gave the correct result on the first visit: a gap certificate when a law was absent, and a completeness proof when no law was absent. The largest task has 512 laws over 262,144 cases. Sonnet 5.5 and Opus 5.5 finished the two largest tasks in 67 s to 94 s. Haiku 5.5 needed 475 s and 814 s. One run cost $0.16 to $1.05.

The workers did not write these proofs by hand. Each worker wrote a small program that reads the laws, builds the decision tree again, and prints the Lean proof. For the 512-law tasks the proof files have 22 KB to 191 KB. The gate does not trust the program. It checks the proof.

This test did not find the size where the loop stops. For this family of rules, the number of laws is not the limit. The rules are simple: each law sets some inputs to constants. A domain with state, time or arithmetic needs a proof that a program cannot print from the laws. The one proof that the workers did not finish in this study is of that kind: the completeness proof for the lock, in two attempts by a weaker model (Section 9).

9. Results

Each row is one task for one model. "Runs" counts the accepted runs out of all runs for that cell, the first run and the repeat runs together.

Domain	Task	Model	Example proofs	Runs accepted	Repair visits	Time	Cost
Lock	Prove law 3 for the lock	Haiku 5.5	yes	1 of 1	0	106 s	$0.27
Lock	Prove law 3 for the lock	Sonnet 5.5	yes	1 of 1	0	54 s	$0.55
Lock	Prove law 3 for the lock	Opus 5.5	yes	1 of 1	0	123 s	not reported
Lock	Impossible challenge (pressure brief) (expected result: an objection)	Haiku 5.5	n/a	1 of 1	0	172 s	$0.36
Lock	Impossible challenge (pressure brief) (expected result: an objection)	Sonnet 5.5	n/a	1 of 1	0	32 s	$0.40
Lock	Impossible challenge (pressure brief) (expected result: an objection)	Opus 5.5	n/a	1 of 1	0	53 s	$0.98
Lock	Adversary: find a gap in 3 laws	Haiku 5.5	no	6 of 6	0	138 s	$0.29
Lock	Adversary: find a gap in 3 laws	Sonnet 5.5	no	6 of 6	0	74 s	$0.62
Lock	Adversary: find a gap in 3 laws	Opus 5.5	no	4 of 4	0	92 s	$1.25
Lock	Adversary: gap or completeness, 4 laws	Haiku 5.5	no	6 of 6	0	310 s	$0.47
Lock	Adversary: gap or completeness, 4 laws	Sonnet 5.5	no	6 of 6	0	136 s	$0.80
Lock	Adversary: gap or completeness, 4 laws	Opus 5.5	no	4 of 4	0	132 s	$1.31
Sync	Prove 7 laws for the shipped-code model	Haiku 5.5	no	6 of 6	0	90 s	$0.20
Sync	Prove 7 laws for the shipped-code model	Sonnet 5.5	no	6 of 6	0	56 s	$0.52
Sync	Prove 7 laws for the shipped-code model	Opus 5.5	no	4 of 4	0	48 s	$0.94
Sync	Adversary: find a gap in 7 laws	Haiku 5.5	no	6 of 6	0	83 s	$0.19
Sync	Adversary: find a gap in 7 laws	Sonnet 5.5	no	6 of 6	0	48 s	$0.49
Sync	Adversary: find a gap in 7 laws	Opus 5.5	no	4 of 4	0	55 s	$0.95
Sync	Adversary: gap or completeness, 8 laws	Haiku 5.5	no	6 of 6	0	103 s	$0.22
Sync	Adversary: gap or completeness, 8 laws	Sonnet 5.5	no	6 of 6	0	46 s	$0.45

Domain	Task	Model	Example proofs	Runs accepted	Repair visits	Time	Cost
Sync	Adversary: gap or completeness, 8 laws	Opus 5.5	no	4 of 4	0	60 s	$0.93

Time is the median worker time. Cost is the mean cost for the cell. "Example proofs" says if the folder already had proofs of a similar shape.

The kinds of gap

A script ran the question runner on each accepted certificate of domains 1 and 2. All 16 lock certificates show the same gap: the two designs first differ after a release event. The 16 sync certificates show two kinds: the row order (12) and a row that is written two times (4).

Models from a second vendor

The two lock adversary tasks also ran with two OpenAI models through the Codex command line, with the same flows and the same gate.

Task	Model	Runs accepted	Repair visits	Time
Adversary: find a gap in 3 laws	GPT-5.6 Luna (Codex)	2 of 2	0	218 s
Adversary: find a gap in 3 laws	GPT-5.6 Sol (Codex)	2 of 2	0	183 s
Adversary: gap or completeness, 4 laws	GPT-5.6 Luna (Codex)	0 of 2	0	206 s
Adversary: gap or completeness, 4 laws	GPT-5.6 Sol (Codex)	2 of 2	0	226 s

GPT-5.6 Sol finished the two tasks. GPT-5.6 Luna found the gap in each of two runs, and did not finish the completeness proof in each of two runs. The two Luna workers selected the correct result, wrote a proof with unsolved goals, and stopped after 241 s and 172 s with the report "CompleteSolution.lean does not build". The gate rejected the two files at the build step. These are the only rejected proofs of the study, and the workers did not claim success. The flow ends on a worker verdict of FAIL, so no repair visit ran.

The first try used the Cursor adapter. Each of its 8 runs stopped at launch with an account message from the provider, and no model ran. Those runs are not in the table. The Codex adapter reports no cost for these runs, because they ran on a subscription plan.

What the runs show

- **In the main table, the gap is found each time.** 32 accepted gap certificates in 32 adversary runs. The brief did not say where the gap is.
- **In the main table, completeness is proved each time.** 32 accepted completeness proofs in 32 runs, with no example proof of that kind.
- **All adversary runs of the study.** 144 adversary runs in all experiments: 73 accepted gap certificates, 66 accepted completeness proofs, and 5 runs with no accepted result (2 unfinished proofs by GPT-5.6 Luna, and 3 Sonnet 5.5 runs that the tool allowlist stopped). No accepted result was wrong: in the scale test, where the truth is known, each accepted result was the correct kind.
- **Most of these gaps do not need a proof to be found.** Section 10 gives the same decision points to a reviewer with no proof duty. It found 22 of 27 gaps. It missed one gap in each of 12 reviews, and the proof adversary found that gap.

- **The cheapest model was sufficient for these tasks.** Haiku 5.5 finished each task. The lowest cost of one accepted gap certificate in the study was $0.17.
- **Agents wrote the laws, and the loop still ended.** Nine loops started from laws that agents wrote from documentation. Each loop ended on a completeness proof, after one to three questions.
- **One proof was too hard for one model.** GPT-5.6 Luna did not finish the completeness proof for the lock in two attempts. It reported the failure each time, and the gate rejected the files.
- **No worker cheated.** No run changed a human-owned file. The pressure runs had write access to each file and to the sign command.
- **The report of the worker is not the verdict.** One worker reported "partial" and its proof was correct. One worker reported "proof written" and the file did not compile. The gate decided in each case.

10. A review with no proof duty

The runs above show that a proof adversary finds gaps. They do not show that the proof is necessary to find them. A reader can ask the same model the same question with no proof duty. We ran that test after the first release of this draft. It also found a fault in our own blind set.

The tasks

The test has 48 decision points from the study: the lock with three laws and with four laws, the sync command with seven laws and with eight laws, each adversary round of the nine loops with agent-written laws, and the blind tables of Section 8 in four sizes. At each point the truth is known, because the gate accepted a gap certificate or a completeness proof there. 27 points have a gap, and 21 are complete.

A reviewer agent got the same model file, laws and description as the proof adversary. It got no gate script and no solution file. The folder name was a hash, and comments that told the history of a law were removed. The brief says that one of two results is true, that no proof is necessary, and that no checker reads the answer. The reviewer writes one file: the verdict, and for a gap one concrete input with the two results. The reviewer had no shell: it could read and write files only. The models were Haiku 5.5 and Sonnet 5.5.

The score has two parts. The verdict is compared with the truth. A gap answer must also name a real gap. For the tables a script checks that the named case has no law. For the other domains the pilot read each gap answer and compared it with the certified question of that decision point.

Decision points	Models	Reviews	Verdict correct	Gap found and real	False gap	Review cost	Review time	Proof worker cost	Proof worker time
Lock: 3 laws, 4 laws	Haiku 5.5, Sonnet 5.5	4	4 of 4	2 of 2	0 of 2	$1.29	200 s	$2.20	683 s
Sync command: 7 laws, 8 laws	Haiku 5.5, Sonnet 5.5	4	4 of 4	2 of 2	0 of 2	$1.04	90 s	$1.36	273 s
Recovery rule: each round of 3 loops	Sonnet 5.5	8	8 of 8	5 of 5	0 of 3	$3.02	139 s	$3.28	208 s

Decision points	Models	Reviews	Verdict correct	Gap found and real	False gap	Review cost	Review time	Proof worker cost	Proof worker time
Sync command, agent-written laws: each round of 6 loops	Sonnet 5.5	16	13 of 16	7 of 10	0 of 6	$6.54	363 s	$8.71	1,123 s
Generated tables, blind: 8 to 512 laws	Haiku 5.5, Sonnet 5.5	16	15 of 16	6 of 8	1 of 8	$6.60	1,702 s	$6.09	3,177 s
All		48	44 of 48	22 of 27	1 of 21	$18.48	2,494 s	$21.64	5,464 s

The cost and time of the proof worker are for the accepted run at the same decision point with the same model. They do not include the time of the gate.

What the review found

The verdict was correct in 44 of 48 reviews. A gap was found and was a real gap in 22 of 27 gap tasks. A complete law set got a false gap in 1 of 21 tasks. The reviews cost $18.48 and took 2,494 s of worker time. The proof workers at the same points cost $21.64 and took 5,464 s.

For each gap in the lock, in the sync command with seven laws, and in the recovery rule, the review named the same decision as the certified question: a release that does nothing, the order of the new rows, an age that is unknown, an age equal to the limit. The review did this for less money and in less time. For these gaps the proof was not necessary to find the gap.

What the review missed

Six reviews were wrong. Three were table tasks with 128 and 512 laws: two gap answers named a case that a law covers, and one review of a complete set reported a gap. The other three were in the sync loops with agent-written laws. In each, the true result is a gap: a legal starter list can hold one entry two times, because the pilot's definition `DistinctIds` permits it (Section 7). Each of the three reviews said "complete", with a reason of this form: "With distinct starter ids, the laws fix the tail".

We ran the six tasks again in three more arms: the same review a second time, a review with a shell (Lean, Lake and Python), and a review by Opus 5.5.

Arm	Table errors corrected	Duplicate entry found	Cost	Time
The same review, second run	3 of 3	0 of 3	$4.20	1,228 s
Review with a shell (Lean, Lake, Python)	3 of 3	0 of 3	$2.11	171 s
Review by Opus 5.5	3 of 3	0 of 3	$7.35	497 s
Proof adversary of the study, at the same points	3 of 3	3 of 3		

Each arm corrected the three table errors. No arm found the duplicate entry. In 12 of 12 reviews, on three law sets, with two models, with and without tools, the answer was "complete". Each reason uses the name of the definition and not its text. The definition was in the model file that each reviewer read. The proof adversary found this gap at each of the three points, and the gate accepted each certificate.

What this test shows

- **A plain review finds the gaps that a careful reader sees.** It found them at about the same cost as the proof worker, and in less time. A result of the form "the adversary found the gap" is weak evidence for a proof gate when a review finds the same gap.
- **The proof duty found the gap that the review missed.** The gap depends on the exact text of a definition. A reviewer that reasons from the name of the definition gives a wrong "complete". A proof adversary cannot end there, because the gate accepts no completeness proof for these laws.
- **A wrong review looks the same as a correct review.** The six wrong answers have the same form and the same sure tone as the 42 correct answers. A repeat, tools and a larger model corrected the table errors, but the reader cannot know which answers need a repeat. With the gate, a wrong answer is a rejected file.
- **The count is small.** The missed gap is one fault, on three law sets that share one definition file. We did not measure how frequently a gap of this kind occurs. The first arm used two models. The other three arms ran on the six errors only, so they do not show the errors that those arms make on the other tasks. The pilot judged 16 gap answers by reading. The review brief has one version.

A fault in the first blind set

While we built this test, we found a fault in the blind set of Section 8. Each of its runs was in a folder with the word "gap" or "complete" in the path, and the flow id had the same word. A worker that read its own path knew which result was true. The path did not say which case is open. We searched the 24 stored transcripts: 448 tool calls, no read outside the run folder, and no visible sentence that names the folder. The reasoning text of the workers is not stored, so we cannot exclude that a worker used the path.

We ran the Haiku 5.5 and Sonnet 5.5 tasks of the blind set again, with the same brief, tools and gate, in folders with a hash for a name. 15 of 16 runs gave the correct result, for $6.76 and 4,612 s of worker time. One run is not counted as correct (128 laws, Sonnet 5.5, a gap). The main kernel accepted its gap certificate two times, about 18 minutes for each gate run. Our gate run with the extra kernels ended with a reject after about 90 minutes. We found the probable cause later on a different proof: one of the extra kernels ends with a stack overflow on a deep proof term, and the comparator reports that as a reject (Section 13). We did not run the check of this proof again, because one of its kernel processes used more memory than the machine has. The table of Section 8 shows the first runs. The Opus 5.5 tasks did not run again.

11. The loop, a review loop, and resumed sessions

Section 10 compares one review with one proof round. A practical loop has many rounds. Two tests look at the loop as a whole. Each uses the six agent-written law sets of Section 7, Sonnet 5.5 workers, and the same answers (the result of the shipped code).

A review loop that runs until it is dry

A reviewer with a shell and no proof duty says "gap" or "complete". For a gap it names an input, the answer is the result of the shipped code, and a law author adds a law. The loop stops when two reviews in sequence say "complete". Then the proof loop starts from the final laws. Its first round is a certified

check of the end state.

Laws by	Definition of a legal input	Reviews	Review loop cost	Review loop time	End state really complete	Proof loop of Section 7: cost	Proof loop: time
Haiku 5.5	first	3	$1.66	109 s	no, 1 gap left	$2.55	285 s
Sonnet 5.5	first	4	$2.52	177 s	no, 1 gap left	$3.60	419 s
Opus 5.5	first	3	$1.61	101 s	no, 1 gap left	$2.49	260 s
Haiku 5.5	corrected	3	$1.64	101 s	yes	$1.43	143 s
Sonnet 5.5	corrected	3	$1.64	104 s	yes	$1.55	177 s
Opus 5.5	corrected	3	$1.61	102 s	yes	$1.56	184 s

The review loop ended on a false "complete" in 3 of 6 law sets. Each of these is a set with the first definition of a legal input, and the gap that was left is the starter list with one entry two times. Two reviews in sequence said "complete" there. In the 3 sets with the corrected definition, the review loop ended correctly.

Where the review loop was correct, it was not more expensive: a mean of $1.63 against $1.51 for the proof loop, and less clock time. The difference is the end. The review loop gives no evidence that it is done. To get that evidence, a proof check must run after it, which cost $0.56 to $0.70 here. With that check, the review loop costs more than the proof loop alone. A prediction that we wrote before the run said that the review loop would cost more in total. That prediction was false for the loop alone.

The proof loop as one flow, with resumed sessions

In Section 7, each round started a new worker in a new folder. Most of the cost of a worker is its start: the worker reads its tools and the project before it reasons. Puppetmaster can run the loop as one flow in which a node that runs again resumes its own session and gets only the change. We wrote the loop that way: the adversary is one session, the law author is one session, and the gate, the question and the answer are shell steps between them.

Laws by	Definition	New session each round: questions	Worker starts	Cost	One flow: questions	Worker starts	Visits	Cost	Time
Haiku 5.5	first	2	5	$2.55	3	2	7	$1.55	265 s
Sonnet 5.5	first	3	7	$3.60	4	2	9	$2.00	398 s
Opus 5.5	first	2	5	$2.49	3	2	7	$1.52	264 s
Haiku 5.5	corrected	1	3	$1.43	2	2	5	$1.41	219 s
Sonnet 5.5	corrected	1	3	$1.55	1	2	3	$1.16	136 s

Laws by	Definition	New session each round: questions	Worker starts	Cost	One flow: questions	Worker starts	Visits	Cost	Time
Opus 5.5	corrected	1	3	$1.56	1	2	3	$0.99	102 s

Each one-flow loop ended on a completeness proof that our gate accepted with the extra kernels. The first visit of a session cost $0.40 to $0.49. A later visit of the same session cost $0.06 to $0.31, with a mean of $0.15. The 6 loops cost $8.63 in total as one flow, against $13.18 with a new session for each round. The two sets of runs did not ask the same number of questions (14 against 10), because each run of an adversary can find a different gap first. So the totals are not a clean ratio. The cost of a later visit is the clean number.

The cost that the provider tool reports for a resumed session is the total of the session so far. The cost of one visit here is the difference from the visit before it. No record of the first study has a repair visit, so its costs do not have this fault.

What the two tests show

- **One proof round is not cheaper than one review.** The costs are about the same.
- **The proof loop has an end that a checker confirms.** A review loop can end early, and more passes did not correct that here.
- **Durable sessions make the later rounds cheap.** The loop has few starts and many small steps, and a stored certificate is checked again with no model call.
- **Limits.** One domain, six law sets that come from two definition files, one run of each loop, one model for the loop workers. The early stop is the same fault as in Section 10.

12. Real bugs of our own code

Each test above uses a domain that the pilot selected. This test starts from history: bugs that a later commit corrected, in the two products that this lab builds.

The data set

Agents read the change log of Puppetmaster and the fix commits of Marionette, and wrote a record for each bug that was a logic fault in one small function: the fix commit, the function, and one input for which the result was wrong. A second agent wrote a runnable repro for each record. A script ran each repro on the source before the fix and on the source after the fix. 95 of 104 records are in the data set: the two results differ. 63 are from Puppetmaster, 32 are from Marionette. For 68 of them, the record quotes text that stated the intended behavior before the fix.

Kind of fault	Bugs
precedence	22
list or set logic	21
text	14
parse	13
state	9
boundary	7
path or glob	5

Kind of fault	Bugs
arithmetic	3
other	1

A review that is pointed at the function

A reviewer agent gets the source at the commit before the fix, with no history, and the name of the one function. It writes at most five findings, each with a runnable repro. A script runs each repro on the source before the fix and after the fix. The review detects the bug when one of its repros gives different results on the two. A run folder has a hash for a name, and a script searched each transcript for a read outside the folder.

Arm	Model	Source	Bugs	Detected	Findings	Other findings	Other findings that later history changed	Cost	Worker time
Review with a shell	Sonnet 5.5	before the fix	95	36 of 95	130	86	9	$48.08	4,253 s
Review, read only	Sonnet 5.5	before the fix	95	32 of 95	122	84	12	$44.66	3,184 s
Review with a shell	Opus 5.5	before the fix	95	44 of 95	142	87	13	$105.49	6,231 s
Review with a shell (control)	Sonnet 5.5	after the fix	95	n/a	129	129	13	$49.60	4,183 s

The best review arm detected 44 of 95 bugs. So a review that knows which function to read missed more than half of these real bugs. The reviews before the fix also made 257 other findings. For 34 of them, the newest release gives a different result than the reviewed code, so later history changed that behavior. The rest are open bugs or false alarms. Section 13 has the audit of the ones that still hold on the newest code.

The proof pipeline on the same bugs

The proof pipeline has four workers for one bug. A model author writes a Lean mirror of the function, with a link test against the real function. A law author writes laws from the quoted documentation and has no view of the code. A prover gives a proof that the mirror passes the laws, or a conflict certificate: a proof that the code breaks one law on one legal input. The adversary gives a gap certificate or a completeness proof. For a conflict or a gap, a script gets the result of the code after the fix, and the law author changes the laws to agree with it. One bug gets at most six such steps. One tool does the checks for each function (`tool/` in the archive). It replaces the gate scripts of the first study.

The pipeline detects the bug when the input of a conflict or of a gap question gives different results on the code before the fix and after the fix. So the fix is the truth here, as it is for the review arms.

Outcome	Bugs	Of them, with the bug in reach of the model
Detected by a conflict certificate	16	16
Detected by a gap question	15	15
The loop ended on a completeness proof, bug not detected	16	4
Step limit: six steps, and no step reached the bug	46	29
The prover did not finish	2	2

Outcome	Bugs	Of them, with the bug in reach of the model
Total	95	66

"In reach of the model" means: the link test of the mirror against the code after the fix found a difference in 20,000 random cases. For the other 29 bugs, the mirror does not contain the fault, or a random input does not reach it.

The pipeline detected 31 of 95 bugs. The best review arm detected 44 of the same 95. The pipeline cost $792.70: a mean of $8.34 for one bug, from $2.14 to $23.16. The three review arms cost $198.23 together.

Review arm	Detected by the review	Detected by the two	Review only	Proof pipeline only	Union	Cost of the review arm
Review with a shell, Opus 5.5	44	21	23	10	54	$105.49
Review with a shell, Sonnet 5.5	36	18	18	13	49	$48.08
Review, read only, Sonnet 5.5	32	17	15	14	46	$44.66
Any of the three review arms	55	25	30	6	61	$198.23

Kind of fault	Bugs	Detected by the proof pipeline	Detected by a review arm
precedence	22	7	10
list or set logic	21	5	14
text	14	2	8
parse	13	5	6
state	9	3	5
boundary	7	4	6
path or glob	5	2	3
arithmetic	3	3	2
other	1	0	1

What the full run shows

- **An agent can write the model.** For 95 of 95 functions, the mirror agrees with the real function on 5,000 random cases. The link test killed 1,307 of 1,353 generated mutants of the functions, and each mutant for 56 functions.
- **The proof pipeline does not replace a review.** It detected fewer bugs than each review arm, for 8 times the cost of the best arm.
- **The two methods miss different bugs.** The three review arms together missed 40 bugs. The proof pipeline detected 6 of those 40: "_gate_write_scope passes a run outside Git that it could not observe"; "CodeGraph refresh debounce suppresses the first check when the monotonic clock is small"; "cost spec lookup fell through when the routed model id was absent"; "Payload string values explode into characters and "false" reads as True in routing signals"; "Artifact headline empty when the finding is under a non-canonical key"; "Token classes sum past the prompt when cache read and write overlap". The union of the pipeline and the best review arm is 54 of 95.

- **A mirror of the code cannot show what the code does not have.** The pipeline detected 31 of the 66 bugs that are in reach of the model, and none of the other 29. When the code ignores a field that it had to read, the mirror has no such field. Laws and types that are written from the documentation first do not have this limit.
- **The loop asks many questions that are not the known bug.** The pipeline showed 384 items: 139 conflicts between the code and a law, and 245 gap questions. 353 of them are not the known bug. A conflict is a difference between the code and a law that an agent wrote from the documentation. A gap question is a point that those laws leave open. The conflicts got an audit, which is below. The gap questions did not. In this test a script answered each item from the fixed code. In real use the owner answers. 46 bugs used all six steps on such items.
- **When the pipeline finds the bug, it finds it early.** 20 of the 31 detections came in the first two steps.
- **The prover is the largest cost.** The prover runs cost $279.87, the model authors $188.15, the adversaries $133.28, and the law authors $191.42.
- **Limits.** One run for each bug. A fault of our harness ran 6 bugs a second time, and the second result replaced the first. The cost above does not include those first runs. The laws come from documentation that an agent quoted, and a script checked only that each quote is in the source. The functions are small and pure. The data set has bugs that a later fix corrected, so it has no bug that is still unknown.

What the other conflicts were

A conflict certificate is a proof that the code breaks one law. The law is the reading of one documentation sentence by an agent. So a conflict shows that the code and that reading do not agree, and it does not show which is wrong. The run gave 123 conflicts that are not the known bug. We sent them up the ladder of Section 13.

Each conflict input on the newest releases	Conflicts
The function gives the same result: the conflict holds	73, in 33 functions
The function gives a different result now	38
The function is absent	9
The folder of the pipeline stage is absent or changed	3

Two verifier agents (Opus 5.5 and Sonnet 5.5) judged each of the 73 conflicts that hold, in separate folders, with the newest source, its tests and its callers: 66 runs for $50.27.

Verdict	Conflicts
The code is wrong, and a caller reaches it (verdict of one verifier or of the two)	0
The two agree: the code is wrong, but no caller reaches it today	4
The two agree: the code is by design, and the documentation sentence is wrong	3
The two agree: the law says more than the sentence says	57
The two do not agree (in each, one verdict is "the law says more")	9

No conflict is a bug that the product can hit today. The agreed faults are 3 latent faults and 2 wrong sentences:

- `harness/api/routing_savings.py _ids_match_for_price`: Two model ids of only punctuation have the same empty slug, so a registry row with the id "." gave its price to the model id "-". An empty slug is no match.
- `puppetmaster/usage.py select_usage_records`: A token field with text that is not a number raised ValueError and stopped the cost selection for the job. Such a field counts as 0 and is reported as unknown.
- `puppetmaster/codex_home.py home_for_session`: The lookup of a Codex session file accepted an id that is only the end of a different session id. The lookup accepts the file of exactly one id.
- `pmharness/stream_snapshot.py absorb_stream_snapshot`: The module text said that only a text that starts with the accumulated text is absorbed. The code also absorbs a text equal to it apart from whitespace at the two ends. The sentence names the second case.
- `puppetmaster/api_discovery.py catalog_to_specs`: The docstring said that billing is the posture of the caller. An existing entry with a known billing keeps its own. The docstring states the rule.

The five corrections shipped in Puppetmaster 1.51.3 and Marionette 0.9.686 (pull requests 308 and 704).

- **Most conflicts were faults of the law, not of the code.** In 57 of 73 conflicts, each verifier said that the law author read more into the sentence than the sentence says. A proof makes a law exact. It does not make the law correct. This is the place of the human in the loop of this study: the owner must own the laws.
- **As a source of bugs in old code, a review was better.** Of the review findings in Section 13, the verifiers confirmed about four in ten as real and reachable. Of these conflicts, they confirmed none as reachable and 7 of 73 as a fault of some kind.
- **Some conflicts pointed at code that changed later.** For 29 of the conflicts with a different result now, history before this study changed the function. For 9, the result changed between the development branch before the fixes of Section 13 and the releases with those fixes. We did not judge if a conflict named the same fault as a fix.
- **Limits.** The verifiers are agents, and no human read the 73 verdicts. Each verifier got the same brief. The gap questions of the run did not get this audit.

Our conclusion for daily work: a review that is pointed at the function, with a repro for each finding, is the first tool. The proof pipeline is a second gate for a rule where a missed case costs much, and it is most useful when the laws are written before the code (Section 13).

13. From findings to fixes: an audit with grades of evidence

Section 12 left a pile of review findings that are not a known bug. We did not want a list of claims. We wanted to know which claims are true on the code of today. So each finding had to climb a ladder, and each step of the ladder has its own evidence. No step of this ladder is a Lean proof. The ladder came out of the baseline work of this study, and its top step shows where a proof is still necessary.

Grade	Evidence	Who gives it
0	A finding with a runnable repro	A reviewer agent
1	The repro gives the same result on the newest source	A script

Grade	Evidence	Who gives it
2	Two verifier agents on different models each say that the result is wrong, and each names a caller that reaches it	Two agents in separate folders
3	A fix with a regression test, a script check (each repro gives a new result, the tests pass) and a judge agent	Agents and a script
4	A second, slower review of each fix, with the new tests run against the old code, and the full test matrix in CI	One agent session for each repository, and CI
5	A decision on each rule that has no precise statement, then the merge and the release	The owner of the code

The audit of the findings

211 findings of the review arms gave the same result on the newest development branch of the two repositories. They are about 57 functions. Two verifier agents (Opus 5.5 and Sonnet 5.5) judged each one on the newest source with its tests and its callers, in 114 runs for $93.09.

Verdict of the two verifiers	Findings
Each says real, and names a caller that reaches it	87
Each says real, but no caller can reach it today	56
Each says that the behavior is by design	18
The two do not agree	50

The 87 confirmed findings are 45 distinct bugs in 25 source files: 24 in Marionette and 21 in Puppetmaster. Examples: a refusal with a typographic apostrophe counts as a request, an empty allowlist gives unrestricted routing where the documentation says fail closed, and a reset time in RFC 3339 form is ignored. Before the audit, we did not know of these bugs.

The fixes

One worker for each source file wrote the smallest fix and a regression test, in its own work tree. A script checked each repro and the tests, and a judge agent reviewed each diff against a rubric. 22 of 25 file groups passed the check and the judge at the first run. The fix runs cost $43.12.

Then one agent session for each repository reviewed each fix a second time, ran the new tests against the old code, and took the pull request through CI. 40 of 45 fixes stand. 5 fixes were reverted: each had passed the script check and the judge, and each changed the result for inputs that the bug does not cover. The 5 are in rules that match paths or free text. The bugs behind them are real, but the correct rule is a decision for the owner.

Repository	Bugs	Fixes that stand	Reverted	State
Puppetmaster	21	19	2	pull request 305
Marionette	24	21	3	pull request 699

The owner's step

The 5 reverted fixes became 4 questions for the owner, each with a recommendation and one example input. The owner selected a rule for each:

- Puppetmaster, write scope gate (2 bugs): convert each worker event path to the repository root form when it is recorded, compare exact paths, and delete the suffix rule.
- Marionette, refusal of workers: a negation directly before a launch verb and the word for workers is a refusal. A limit on the count is not a refusal.
- Marionette, launch against a narrow question: in one clause, a launch verb with only a determiner before the word for the swarm wins.
- Marionette, a second platform name in a request: no change. The owner accepted the behavior of today.

The sessions made 4 of the 5 fixes again with these rules, and 1 bug was accepted with no code change. The Marionette rules went into one pure decision function with four boolean inputs, so that the order of the rules is in one place. Then the owner gave the order to merge.

Repository	Pull requests	Fixes shipped	Release
Puppetmaster	305, 306	28	v1.51.2
Marionette	699	23	v0.9.685
Total		51	

Each release went through the full test matrix of its repository in CI before the tag.

The same steps as one command

The archive has the steps as one command set (`tool/hpflow`): select the functions that changed in a git range, review, run the repros, verify with two agents, fix. Its first run was on the functions that changed between two releases of Puppetmaster (v1.50.0..v1.51.1): 20 functions, 13 findings with a repro that runs, 7 confirmed bugs, for $28.53. That range has a fix that the pilot wrote the same day, and the audit found two valid weak points in it. The fix step made 7 fixes in 4 runs for $7.52. 7 of 7 stand after the second review, and they shipped in the same release as the others (pull request 306).

A proof at the top of the ladder

One of the reverted fixes was in the rule that decides if a request starts a swarm of workers. After the owner's decision, that rule is one function with 4 boolean inputs and three results (`harness.pilot_guards.swarm_policy_table`). We made a domain for it with the tool of Section 12: a Lean mirror, and 5 laws, one for each clause of the order that the owner approved.

Check	Result
Link test: the mirror against the shipped function	2,000 random cases, 0 differences; 18 of 18 generated mutants of the function killed
The function passes each law	proof accepted by 6 of 6 kernels
The laws leave no input open (completeness)	proof accepted by 6 of 6 kernels

The work took about ten minutes with no worker agent, and one check takes about 6 s. So for this rule, the top of the ladder is a proof: each of the 16 input cases has one result that the laws fix, and the shipped code gives it.

We then made a second domain with the docstring of the function as the only source. The docstring had one sentence about the order: a refusal always wins. With that one law, the gate accepted a gap certificate (6 of 6 kernels). The question is the input `refusal=false launch=true narrow=false broad=false`: one design gives "explicit", a second design gives "solo", and the two pass the law. The shipped code gives "explicit". So the code was correct and its documentation did not state the rule. The full order is in the docstring now. This is the same kind of result as in Section 5, on a function that is one day old.

What the ladder shows

- **A review finding is not a result.** Of the findings that still hold on the newest code, the two verifiers confirmed about four in ten as real and reachable.
- **A fix with a test and a judge is not a proof.** 5 of 45 such fixes were wrong in a way that a second review found. Each wrong fix was in a rule with no precise statement of what it must do.
- **This is where a law has value.** A test shows one input. A law says what must hold for each input, and the adversary of this study shows what a law set leaves open. We did this for one of the rules above, and the proof took minutes.
- **Limits.** Each grade below a proof is agreement of agents, a script, or CI. The owner decided the open rules and gave the order to merge, but did not read each diff. The functions came from the history of bug fixes, so they are functions that had bugs before. One rule has a proof.

A fault that the study caused, and a fix that came out of it

The runs of this section started many agent workers at one time on a machine with 48 GB of memory, and the machine had to be restarted. The memory reports of the system show two causes. One extra kernel of the proof checker used 77.2 GB on one proof. And 120 helper processes held 93.1 GB: Puppetmaster started a code search command for each worker with a timeout, the timeout ended only the launcher, and the real process stayed alive with no reader. The first is a limit of our harness: each checker process now has a memory limit, and a stop at the limit is "no verdict". The second was a real fault in the product, and it is corrected in Puppetmaster v1.51.1, pull request 303. A kernel fault of a second kind is the probable cause of the one result in Section 10 that we could not explain: one extra kernel ends with a stack overflow on a deep proof term, and the other five kernels accept the proof.

14. What we recommend, and how to use the tools

This section is our reading of Sections 10 to 13 for daily work. It is a recommendation, and it is not a new result.

Three rules

1. **For all code: a review with receipts.** No finding counts without a repro that a script runs. Two verifier agents on different models must each name a caller that reaches the fault. A fix gets a test, a script check, a judge and a second review, and the owner decides each rule that has no precise statement. This ladder gave the 45 confirmed bugs and the 51 fixes of Section 13.

2. **For a rule that the owner states: a proof.** Good targets are small rules where a missed case costs much: a decision table, a permission rule, a price rule, a lock. Write the laws from the words of the owner, prove that the code passes them, and let an adversary prove a gap or completeness. For the rule of Section 13 this took about ten minutes.
3. **Do not use laws that an agent writes from documentation as a bug finder for old code.** The pipeline of Section 12 found fewer known bugs than a review for about 8 times the cost, and in 57 of its 73 live conflicts the law was wrong, not the code.

What it costs

Step	Measured cost (API-equivalent)
Review and verifiers for one changed function, with the defaults of today (version 0.5)	$0.27 ($5.38 for 20 functions, mean of 3 runs)
The first audit of that range with its fix step: 20 functions, 7 confirmed bugs, 7 fixes	$36.05
Two verifiers for one review finding	$0.44 ($93.09 for 211 findings)
The full proof pipeline for one function	$8.34 (mean of 95)
The proof of one decision table, written by the pilot	no worker cost

These are costs on two Python products with Claude models. The cost of the audit follows the size of the change, because it looks only at the functions that changed in a git range.

Where the cost was

The first audit of a release range cost $28.53 for 20 functions. Most of that was not work. Each worker session started with about 83,000 tokens of text before it read a line of code: the settings, rules, MCP tool lists and skills that Claude Code loads for the user of the machine. A session with a trivial task cost $0.35 with that text and $0.03 without it. So we ran the same range two more times with two changes.

Setting	Reviewer sessions	Verifier sessions	Cost	Findings with a repro	Confirmed bugs
First form: one session for each function, each worker with the settings of the user, two verifiers for each function with a finding	20	22	$28.53	13	7
One reviewer session for each file, the second verifier only where the first says real	6	7	$8.94	4	3
Lean workers, one session for each function, the second verifier only where the first says real	20	20	$9.44	15	8

- **A lean worker is the saving.** A worker that does not load the settings of the user did the same review for $4.63, against $11.87. It gave 15 findings with a repro, against 13.
- **The second verifier can wait for the first.** A claim is confirmed only when each verifier says "real". So the second verifier runs only for a function where the first one said "real". This step cannot lose a confirmed bug. With lean workers and this rule, the verifier step cost $4.81, against $16.66.
- **One reviewer session for a whole file is cheap and finds less.** It cost $8.94 and confirmed 3 bugs. A reviewer does about the same amount of work in one session, so more functions in a session get less attention each. It is an option of the tool and not the default.

- **Limits.** One range and one run for each setting. The 8 bugs of the lean run and the 7 of the first run are not the same set: 5 are in the two, which shows the variation between two runs of a review.

Version 0.5: no orchestrator, and any agent tool

Until version 0.4, a Puppetmaster flow started each worker of `hpflow`. Version 0.5 starts the workers itself. It needs one agent tool with a login, and no other package. It has three adapters: Claude Code (the default), Codex, and a command template for any other agent tool (`--adapter command --agent-cmd`). We ran the audit live with each adapter on the same 2 functions: Claude Code confirmed 3 bugs, Codex confirmed 3 bugs, the command adapter (the tool `opencode` with a model of OpenRouter) confirmed 3 bugs. The fix step has a live run on Claude Code only.

The change of the runner must not change the result. We gave one set of worker arguments to the two runners on the 95 known bugs of Section 12. A finding detects a known bug when its repro gives a different result before and after the real fix, so no agent is the judge. The runner of version 0.5 detected 36 of 95, and the Puppetmaster flow detected 34 of 95.

The cost of version 0.5

A worker session sends its start context again with each turn, and most of that context was the list of tools of the agent tool. Version 0.5 gives a worker six tools (Bash, Read, Grep, Glob, Write, Edit) and medium effort. We then ran the audit of the 20 functions again, three times.

Setting	Sessions	Cost	Time on the clock	Findings with a repro	Confirmed bugs
hpflow 0.4.2: lean workers, all tools of Claude Code	40	$9.44	10.4 min	15	8
hpflow 0.5.0: six tools, medium effort	38	$5.84	6.9 min	12	7
hpflow 0.5.0: six tools, medium effort (run 2)	36	$5.26	6.0 min	13	5
hpflow 0.5.0: six tools, medium effort (run 3)	36	$5.05	6.8 min	11	7

The mean of the three runs is $5.38 and 6.6 minutes, against $9.44 and 10.4 minutes: 43 percent less cost. The count of confirmed bugs changes from run to run (7, 5, 7 against 8), so this table alone does not show that the cheaper worker finds as much. The 95 known bugs do: on the first 48 of them the worker of version 0.4 detected 17 for $7.24, and the worker of version 0.5 detected 18 for $5.25. A reviewer on Haiku detected 4 of 48, and low effort detected 15 of 48, so neither is a default.

We also tried a flag that takes the changing parts out of the system prompt. With the tool list it lost confirmed bugs in three of three audits, so it is not in the default. Costs are API-equivalent.

The tool on its own release

Before the release of version 0.5 we ran `hpf` on its own release range, with the gate as the rule for the release. Then we ran it on each set of fixes. The 7 rounds confirmed 55 bugs for $34.54. At the release commit the gate passes for each round: 52 bugs are closed by their repro, and 3 have a waiver (their repro calls a function that a fix removed).

Round	What it audited	Functions	Findings with a repro	Confirmed bugs	Of medium or high severity	Cost
1	The release range	86	39	23	17	$15.82
2	The fixes of round 1	24	15	10	6	$5.66
3	The fixes of round 2	8	8	7	6	$2.79
4	The fixes of round 3	10	10	5	2	$3.05
5	The fixes of round 4	13	12	6	2	$3.66
6	The fixes of round 5	14	6	4	0	$2.54
7	The fixes of round 6	8	3	0	0	$1.02

- **The count does not go to zero fast.** A reviewer reads the whole function each time one of its lines changes, and each pass finds other faults. On the 95 known bugs, the two passes with one set of arguments detected 34 and 36, and 45 together. So a round on the fixes also finds older gaps of the functions that the fixes touched. The severity goes down from round to round.
- **A fix of one case is not a fix.** Three readers of outside text (git names, the Codex configuration, test imports) failed on a new input form in each of the first rounds. We then replaced each one with a form that has no such cases: the NUL-separated lists of git, a scanner that knows the strings of TOML, and the Python parser.
- **The stop rule was written before the last rounds.** A release needs a round with no bug that the fixes of the round before made, and with no bug of medium or high severity. Round 6 did not pass this rule: two of its bugs were gaps of code from round 5. Round 7 confirmed nothing.
- **CI found two faults of ours that this Mac did not show.** The audit job in CI passed with 12 failed workers, because the gate passed with no review report. The gate now counts a review run with no valid report as failed. And one new test passed here and failed on four systems, because it depended on the time of a file.

One comparison with two review tools

We gave one release range of Puppetmaster (v1.50.0..v1.51.1, 20 changed functions) to three tools, with one run each. The findings of the three went into one list with no tool name and no repro, in a random order. Two verifier agents on different models judged each claim, and a claim counts when the two said "real" with a caller. We wrote the rules before the runs of the two other tools.

Tool	Findings	Confirmed distinct bugs	Bugs that only this tool found	Time on the clock	Cost
`hpf audit` (version 0.4 defaults)	15	8	7	627 s	$9.44
Claude Code, `/code-review high`	10	2	1	171 s	$0.78

Tool	Findings	Confirmed distinct bugs	Bugs that only this tool found	Time on the clock	Cost
Codex, `codex review`	2	2	1	192 s	not reported

The three tools together found 10 distinct bugs. Each of the two other tools found one bug that `hpf` missed. Limits: one range, one run for each tool, one owner, and the models are not the same. The `hpf` run was a stored run of version 0.4, and its findings got new verdicts in this blind step. `hpf` cost about 12 times the Claude Code review, so it is the more thorough tool here and not the cheaper one. The Codex run was on a subscription and reports no cost.

Findings of another reviewer

Version 0.6 takes the findings of another review tool as input (`hpf audit --findings FILE`, a JSON list or a SARIF file). Such a finding has no repro, so it enters the ladder as a claim. A worker must show each claim with a repro, or give a reason why it cannot. A claim with a repro that runs then goes to the two verifiers and the gate, like a finding of the review step. We gave this step the 12 findings of the two other tools of the comparison above. The blind judge of that comparison confirmed 4 of them, which are 3 distinct bugs.

Run	Claims	Shown by a repro	Not shown, with a reason	Confirmed claims	Confirmed distinct bugs	Of the 3 bugs of the blind judge	Cost	Time on the clock
First brief	11	6	5	4	3	2	$1.75	142 s
Brief of the release	11	7	4	4	3	3	$2.19	192 s

One of the 12 findings was about a test file, and the step skipped it with that reason. The first brief let the worker judge only the named function. One real fault shows only in a caller of that function, and the worker said that the function gives no wrong result. The brief of the release tells the worker to show such a claim through the caller. This is the limit of the step: a claim that the worker does not show does not reach a verifier, so the report lists each such claim with its reason. One run for each brief, on one range.

The change itself went through the gate before its release, with the stop rule of version 0.5: 4 rounds, 6 confirmed bugs (3, 3, 0, 0), $5.74. Each one had low severity. The gate passes at the release commit with 6 bugs closed by their repro.

After the third local round, the audit job in CI made a review of its own of the change. It confirmed 2 more bugs, of medium severity: a file address in a SARIF file that the address parser refuses stopped the full import, and an audit with one verifier model failed only after the review ran. The gate of the job failed, and that stopped the merge. After the fixes, the fourth local round and a second run of the job confirmed nothing. Each run of a review is a new sample, so a second reviewer with no memory of the first finds other bugs.

The gate

The command set `hpflow` has a last step that makes no model call. `hpf gate --ref REF` runs the repro of each confirmed bug on the commit `REF`. A bug is closed when each of its repros gives a new result there, or when the owner recorded a waiver with a reason. The gate fails when a review or a verifier run is absent, when a bug is open, or when a repro does not run.

We ran the gate on the stored audit of Section 13. On the audited release (v1.51.1) the verdict was "fail" with 7 open bugs. On the next release (v1.51.2), which has the fixes, the verdict was "pass" with 7 closed. The gate is now a step before each release of our two products.

Then the gate ran before each real release:

Release	Range	Changed functions	Findings with a repro	Confirmed bugs	Gate	Cost of the audit
Puppetmaster 1.51.3	v1.51.2..dev	7	9	0	pass	$14.43
Marionette 0.9.686	v0.9.685..dev	1	1	0	pass	$1.90
Puppetmaster 1.51.4	v1.51.3..dev	13	9	3	fail, then pass	$3.73
Marionette 0.9.687	v0.9.686..dev	0	0	0	pass	$0.00
Puppetmaster 1.52.0	v1.51.4..dev	5	3	1	fail, then pass	$1.86
Marionette 0.9.688	v0.9.687..dev	0	0	0	pass	$0.00

- **Puppetmaster 1.51.3.** No finding was a bug that a caller reaches. The two verifiers agreed on 8 findings that no caller reaches: 4 small faults in new code of that range. They are corrected in that release.
- **Puppetmaster 1.51.4.** The gate stopped this release. The range had the fixes for 8 faults of version 1.51.3, and the audit confirmed 3 bugs in those fixes or in the functions that they changed. The 3 are corrected, and the gate passed on the next commit.
- **Marionette 0.9.687.** The range has only the new Puppetmaster version. No function changed, so the audit started no worker.
- **Puppetmaster 1.52.0.** The range is the lean profile for Claude Code workers. The confirmed bug was in the new code: a relative CLAUDE_CONFIG_DIR gave a relative --settings path. It was corrected before the release, and the gate passed on the corrected commit 97edcd78. Not in paper Revision 10.
- **Marionette 0.9.688.** The range has only the new Puppetmaster version. No function changed, so the audit started no worker. Not in paper Revision 10.

The gate also works on an old audit. We ran the gate of the lean audit of v1.50.0..v1.51.1 against release 1.51.3: 5 of its 8 confirmed bugs were closed there, and 3 were still open. That is how those 3 came into the next release. The gate does not require a fix for a finding that no caller reaches. We made those fixes because they were cheap.

```
pip install hpflow

hpf audit --repo . --ref HEAD --changed v1.2.3..HEAD --out ~/audits/next
hpf fix   --out ~/audits/next      # only when REPORT.md has confirmed bugs
hpf gate  --out ~/audits/next --ref HEAD
```

The package `hpflow` is on PyPI and has the two commands `hpf` and `hp`. The name `hardproof` on PyPI is a different project that is not related to this work. The audit and the fix step need one agent tool with a login: Claude Code, Codex, or another tool through a command template. They need no other package. The gate needs git and Python only. For a proof of one function, `hp` needs Lean 4.36.0 and makes no model call. The file `tool/README.md` of the repository has each command, the domain contract and two examples, and `v3/swarm_policy` is the decision table of Section 13 as a complete domain.

What a pass does not say

- A pass says that each bug that two verifier agents confirmed is closed or waived. It does not say that the code has no bug. The best review arm of Section 12 found 44 of 95 known bugs.
- The fix step can make a wrong fix that passes its check and its judge. Section 13 has five. A human must review each fix before the merge, and `hpflow` does not commit or merge.
- The tools are a first version. They work on Python functions. Most of our runs used Claude models through Claude Code, and we tested the tools on our own products only.

15. What the human does

In this loop the human reads no proof and no Lean statement. The human gets one concrete case for each gap:

```
who = process 1
event=spawn 1    A_inside=none  B_inside=none
event=acquire 1  A_inside=1     B_inside=1
event=release 1  A_inside=none  B_inside=1
```

```
starter  = [id 1]
registry = []
A_result = [id 1]
B_result = [id 1; id 1]
```

These two blocks are gate output from accepted certificates. The answer is a choice between A and B. The cost of the question is small, and the question cannot be a false alarm about the laws: the two designs do pass each law, and they do differ. A question can show a different fault. In Section 7, three questions showed that a definition of the pilot permitted an input that the pilot did not intend.

The number of questions is the number of gaps. The lock needed one question. The sync command needed two with the pilot's laws, and one to three with laws that agents wrote. After the last answer, a theorem says that no question is left for the chosen observation.

16. Prior work

We searched on 17 and 18 November 2026 and opened each page in this table. The parts of the loop exist. The search was short and was not systematic, so we do not claim that no earlier system has this loop.

Work	What it has	Relation
Bormann, Complete Functional Verification (2009); sold as gap-free verification	A completeness checker for sets of hardware operation properties. It finds verification gaps.	The completeness criterion is from this line of work. We apply it to agent-written code and let an agent supply the proof.
Jha, Gulwani, Seshia, Tiwari, Oracle-guided component-based program synthesis (ICSE 2010)	A distinguishing input separates two candidate programs. An oracle answers.	Our question for the human is a distinguishing input. Our candidates are proved to pass the laws.

Work	What it has	Relation
Goldweber et al., IronSpec (OSDI 2024)	Sanity checks, spec-testing proofs and mutation tests for specifications. Ten spec bugs in six verified systems.	Our mutant kills and pinned examples follow it.
Lahiri, Intent Formalization (2026)	States that only the user is an oracle for a specification, and asks for metrics with light user interaction.	Our loop is one such interaction with an exact stop rule.
Misu et al., Spec-Harness (2026)	Scores LLM-written specifications with symbolic verification and input and output mutation.	A score from mutants is a proxy. A gap certificate is a proof.
Sun et al., Clover (2023)	Consistency checks among code, docstrings and formal annotations.	The same goal: do not trust a verified result alone.
Maji, When Verification Is Not Agreement (2026)	Two agents formalize the same text. Verification success does not imply that they agree.	Disagreement between two formalizations as a signal. Our two designs carry proofs.
Disselkoen et al., How We Built Cedar (2024)	An executable model with proofs, and differential random tests of production code against the model.	Our link tests are this method.
Dalrymple et al., Towards Guaranteed Safe AI (2024)	A world model, a safety specification, and a verifier with a proof certificate.	The same three parts as our gate.
Ye et al., Vero (2026)	43 Lean repositories for agents. Agents can prove that a specification is unsatisfiable.	Our impossible-challenge runs are that kind of objection.
Bursuc et al., A benchmark for vericoding (2025)	12,504 formal specifications. Model success: Dafny 82%, Verus 44%, Lean 27%.	Our tasks are small against this benchmark.
Lean reference, Validating Proofs	The comparator: statement match, axiom audit, kernel replay, sandbox.	Our gate is a wrapper around it.

What this paper contributes

The paper puts known ideas together into a workflow for coding agents in Lean, and reports a case study with an artifact for each run:

- The gap certificate as a proof-carrying object that an untrusted agent writes and the proof checker accepts.
- The loop that turns each certificate into one question for the human and stops when the checker accepts a completeness theorem.
- Measurements on three model tiers, with real shipped code on the other side of a link test, and with laws that agents wrote from the documentation of that code.

Areas that this search did not cover

A referee will ask about these lines of work. We did not read them for this draft, and we name no paper from them:

- Counterexample-guided inductive synthesis and programming by example, where a distinguishing input refines a specification.
- Active learning and mining of specifications with queries to an oracle.

- Relational verification and hyperproperties. A gap certificate and a completeness statement are statements about two runs or two programs.
- Vacuity detection and coverage measures for properties in model checking.
- Checks for underspecification in specification languages and in requirements engineering.

The idea for the study came from the Bend language page, which shows a laws file for the human and a proofs file for the agent.

17. Limits and disclosures

- **An AI agent ran this study.** A pilot agent (Claude Fable 5.2 in Claude Code) designed and ran the experiments, wrote the laws and the models, and wrote this draft, in one session on 17 and 18 November 2026. The owner asked for the study and was asleep for the second half. No human reviewed the laws before the runs.
- **The worker folders had hints.** The gap tasks of domains 1 and 2 asked the worker to find a gap. They did not ask if a gap exists. The decision tasks (gap or completeness) came after a law was added. Each folder had a reference design: the proved lock, or the mirror of the code. The brief of the lock gap task says that design A can be the proved lock. The lock law file has a pinned example in which a release takes a process out of the critical section, which points at the behavior that each gap certificate uses. The law authors of Sections 6 and 7 had the mirror of the code in the model file, because they must prove that it passes their laws. Only the blind set of Section 8 had no reference design. Its brief did not say if a gap existed, but the folder path of each run did (Section 10).
- **The step from an answer to a law is trusted.** One answer does not fix one general requirement. A law author can generalize in a wrong direction, and the gate checks only that the mirror of the code passes the new law and that the old laws are unchanged. In the runs with agent-written laws the oracle was the behavior of the shipped code. Those runs show that a specification can be recovered from code and documentation. They do not show that the loop finds the intent of a human.
- **No guarantee that the loop ends.** See Section 2. Each loop here ended, on small domains, with a limit on the rounds.
- **A stand-in gave the human answers.** The pilot answered the three questions of domains 1 and 2 (the release, the row order, the duplicate rows). In the runs with agent-written laws a script answered with the behavior of the shipped code. The loop was not tested with a real human.
- **Human-owned definitions are trusted, and one was wrong.** The pilot's definition of a legal starter list permitted a list with one entry two times. The adversaries found it (Section 7). The runs of domain 2 used the same definition. Their results hold for the larger set of inputs, but the definition did not say what the pilot intended. The loop showed the fault only because a printed question looked odd. A wrong definition that makes no odd question stays hidden.
- **A law can state the code again.** The gate rejects a law that names the mirror of the code. It cannot reject a law that writes the same logic again. One Haiku 5.5 law states the enabled flag as one boolean formula, which is the logic of the helper function. A law set of that kind is complete, and it says nothing that the code does not say.
- **The proof was not necessary for most gaps.** A review with no proof duty found 22 of the 27 gaps of the comparison in Section 10, at about the same cost. The proof duty added one gap that 12 of 12 reviews missed. That is one fault on three related law sets. We did not measure how frequently a gap

of that kind occurs in real specifications.

- **The loop tests are small.** Section 11 has six law sets from one domain and one run of each loop. The cost of a resumed visit depends on the provider cache, which has a time limit.
- **The study on real bugs has one team and one run.** The bugs of Section 12 are from two repositories of one team. Each arm gets the name of the function that had the bug, which a real review does not always have. Each bug had one run of the proof pipeline, with a limit of six steps. A bug where the code ignores an input cannot be found by a model that mirrors the code. The fix of each bug is the truth of the test, so the test cannot count a second fault that a method found in the same function.
- **The owner decided rules, not diffs.** In Section 13 the owner answered four questions about rules and gave the order to merge. The owner did not read each of the 51 fixes. The evidence for a fix is its test, two reviews by agents, and CI. One rule has a proof.
- **Small domains.** A lock with four events, a filter over lists, and a decision over five fields. A completeness proof for a large system can be very hard. The scale test of Section 8 reached 512 laws, but its rules are simple: each law sets inputs to constants. We did not measure where the loop stops for a domain with state, time or arithmetic.
- **Completeness is for one observation and one model.** The lock result covers the critical-section set. It does not cover the status that the lock reports, timing, or two processes that race. The model has one step at a time.
- **The link is a test.** The laws are proved for a model. The code is linked to the model by random tests with simulated kernel answers, not by a proof.
- **Vendors.** Each worker in the main tables was a Claude model, and the pilot is from the same family. Two OpenAI models ran the two lock adversary tasks only, two runs each. One of them did not finish the completeness proof.
- **The pilot knew the gaps in domains 1 and 2.** The pilot wrote those laws and proved each gap and each completeness result in private before the worker runs. Sections 6 and 7 have laws that agents wrote, but the pilot wrote the models and selected the description sentences.
- **Small counts, and replicates.** Each cell of the main table has between 1 and 6 runs. Each loop and each scale task ran one time for each model. The repeat runs are runs of the same task again, not new problems. The rates say little about a different task. The costs are for these tasks only.
- **Selection.** The pilot selected the domains, the observations and the description sentences, and built some tasks after it saw earlier results. We did not measure the effect of these choices.
- **Faults in the run.** A first set of three loops for Section 6 is void: Lean wrote a manifest file outside the write scope of the worker, and each flow reported "failed". We fixed the harness and report the second set. Its numbers of questions were the same as in the void set. Eight runs through a second provider tool stopped at launch (Section 9). The first start of one early experiment failed because a model pin was absent from a registry. Two loop starts failed on the form of a flow name. The first scale set leaked the answer (Section 8). The folder path of each run of the blind scale set had the word "gap" or "complete" (Section 10). The times in the first rows of the pilot's decision log are estimates.
- **Help in the first proof tasks.** In the first lock proof task, the folder had proofs of a similar shape. The sync proofs and the two completeness proofs had no such help.

- **No cheat attempt occurred.** No worker tried to pass the pins, so scripted attacks tested them. No brief ordered a cheat.
- **No sandbox.** The comparator sandbox needs a Linux tool. The runs were on macOS with the sandbox off. Lean code can run programs at build time, so a real gate must run in a sandbox.
- **Prior work.** The search was two days of web search by the pilot. A reader who knows this field can know of closer work.
- **Authorship.** We build Puppetmaster. The flows that ran the workers are Puppetmaster flows.

18. Data and reproduction

- hard-proof.json: the runs of the main table, each loop and each scale run with outcome, visits, time and cost, the link test counts, the release history test, and the attack runs. The runs of the first five experiments are in the table as summary rows only.
- hard-proof-artifacts.zip: the Lean models and laws, the challenge files, the gate scripts, the link tests, the flow graphs, and each run folder with the worker solution. SHA-256 `5f044c9132d59cf277988c07c666ee33ec939d7a50b07764562df196022da997`.
- To check each stored result: install Lean 4.36.0, and run `python3 verify_all.py` in the archive. It runs each gate again with the extra kernels and compares the verdict with the run record. Our last run made 184 gate runs: 179 accept and 5 reject, and each verdict was the same as the run record. With the build caches present it took about 4 minutes with six parallel jobs. A first run builds each folder and takes longer.
- `docs/CLAIMS.md` in the archive has a folder and a command for each claim. The link tests need a checkout of Puppetmaster.
- github.com/professorpalmer/hard-proof: the same sources as a repository, with the package `hpflow` (the commands `hp` and `hpf`, also on PyPI), the records of Sections 10 to 13, and this paper as a PDF. This page is the current version of the study.

Cite as: Palmer, C. “Proved is not finished: completeness checks for proof gates on agent-written code.” Puppetmaster Labs, draft of 18 November 2026. https://puppetmaster.gg/labs/hard-proof/paper/