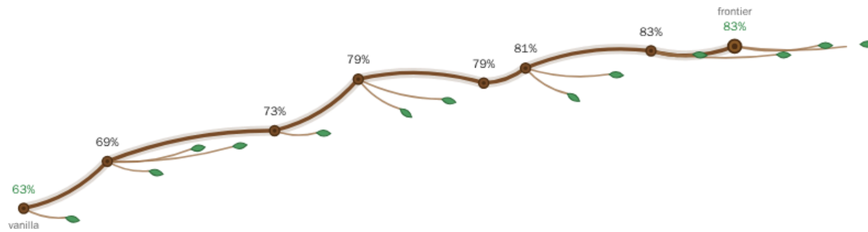
Don't Train the Model, Evolve the Harness



An automated loop rewrites the scaffold around a fixed open model and lifts it +16.7 points on Harvey's Legal Agent Benchmark

AUTHOR

Joel Niklaus

AFFILIATION

Hugging Face

PUBLISHED

Jun. 18, 2026

Table of Contents

1 Introduction
- 1.1 What we found
- 1.2 What's in this post

2 Setup
- 2.1 The benchmark (LAB)
- 2.2 The model and what a "harness" is
- 2.3 Building a held-out split
- 2.4 What an evaluation costs
- 2.5 The vanilla baseline

3 How the loop works
- 3.1 The loop at a glance
- 3.2 The scaffold: meta_harness.py
- 3.3 The protocol: SKILL.md
- 3.4 The promotion gate
- 3.5 Why three trials
- 3.6 Where this sits: program evolution, and what we add on top of Meta-Harness

4 Results
- 4.1 The climb
- 4.2 What the mechanisms are
- 4.3 The optimizer out-found us: matter fidelity
- 4.4 Does it transfer?
- 4.5 Open vs closed, in context
- 4.6 The ceiling

5 Lessons: making it trustworthy
- 5.1 A promotion-gate bug (the cost-λ boundary)
- 5.2 Provider reliability as a first-class variable
- 5.3 Hardening the LLM as an optimizer

6 Conclusion

6.1 What the loop reliably finds

6.2 What's next

Introduction

Open models are getting good at legal work, but they keep dropping the ball at the last step. On Harvey's Legal Agent Benchmark (LAB) (Harvey, 2025), an open model often reads the documents, spots the right issues, and writes a sound analysis, and then it saves the result to the wrong filename, or leaves it in a scratch folder, or never writes the file at all. The grader never sees the work, so a good answer scores close to zero.

That gap is interesting because it is not mainly about legal reasoning. It is about the code around the model: the loop that decides what the model reads, where it writes, and when it is allowed to stop. That wrapper is the harness (Hugging Face, 2025). The model is fixed, the harness is ours to change, and a lot of what looks like missing capability turns out to be missing plumbing.

So we asked ourselves a simple question: if we keep the model frozen and let an automated loop rewrite only the harness, how far can the score move?

What we found

We ran the Meta-Harness loop (Lee et al., 2026) on LAB with a fixed open model, `deepseek-ai/DeepSeek-V4-Pro`, served through Hugging Face Inference Providers. A Claude proposer (Opus 4.8, max effort) reads the run history, copies the current best harness, adds exactly one new mechanism, and an outer loop keeps the change only if it clears a noise margin on a small held-out gate. Over 22 iterations the harness climbed from a vanilla baseline of 63.1% to 83.3% pooled criterion pass rate on the 24-task dev set, a gain of +20.2 points, with whole-task success going from 0% to 4.2%.

The harness was tuned only on the dev set. Run untouched on a held-out 100-task test split that the loop never saw, it lifted the same model from 63.4% to 80.1% (+16.7 points), and whole-task success from 0% to 5.0% (!). The gain generalized, which suggests the loop found a transferable harness rather than overfitting on the dev set.

> The short version
>
> The single biggest lever was plumbing, not prompting. Most of the gain came from deterministic post-processing that puts the right deliverable in the right place, plus one self-check the loop discovered on its own. And most of the engineering went into making the loop trustworthy enough to believe its own numbers.

What's in this post

We start with the setup: what LAB measures, what a harness is, how we built a hard held-out split, and what a single evaluation costs. Then we open up the loop itself, the Python scaffold and the protocol the proposer follows, and place it next to related work on evolving code and prompts with language models. The results section walks through the climb, the mechanisms that drove it, the failure mode the optimizer found that we had not, how the harness transfers across models, and where it stops improving. We close with the part that took the most time and taught us the most: making an automated optimizer you can actually trust.

Setup

The benchmark (LAB)

Harvey's Legal Agent Benchmark is a set of realistic legal matters (Harvey, 2025). Each task hands the agent a closed workspace of documents (contracts, emails, spreadsheets, slide decks) and asks for a concrete deliverable: a diligence memo, an issue list, a redlined agreement, a drafted instrument. A task is graded by an LLM judge against a long rubric of specific criteria, and the judge only reads files the agent leaves at the top level of an `output/` folder, under the exact filename the task asked for.

We track two numbers. The pooled criterion pass rate is the fraction of all rubric criteria passed across the run, pooled over tasks. It is dense and moves smoothly, so it is the signal we optimize. The all-pass rate is the fraction of tasks where every criterion passes. It is LAB's headline number and the one that matters in practice, but it is sparse and noisy on a small set, so we treat it as a secondary signal. Even frontier models complete fewer than 10% of these tasks end to end (Harvey, 2025b), so all-pass starts near zero.

Now, finally, frontier labs have started reporting capabilities on legal work. When Anthropic launched Claude Fable 5 in June 2026 (Anthropic, 2026), it was, as far as we know, the first frontier model release to put a legal agent benchmark in its headline comparison. The result is striking: across the benchmarks Anthropic reported, the Legal Agent Benchmark was Fable 5's weakest by a wide margin, far below its coding and knowledge-work scores.

Claude Fable 5 across its launch benchmarks

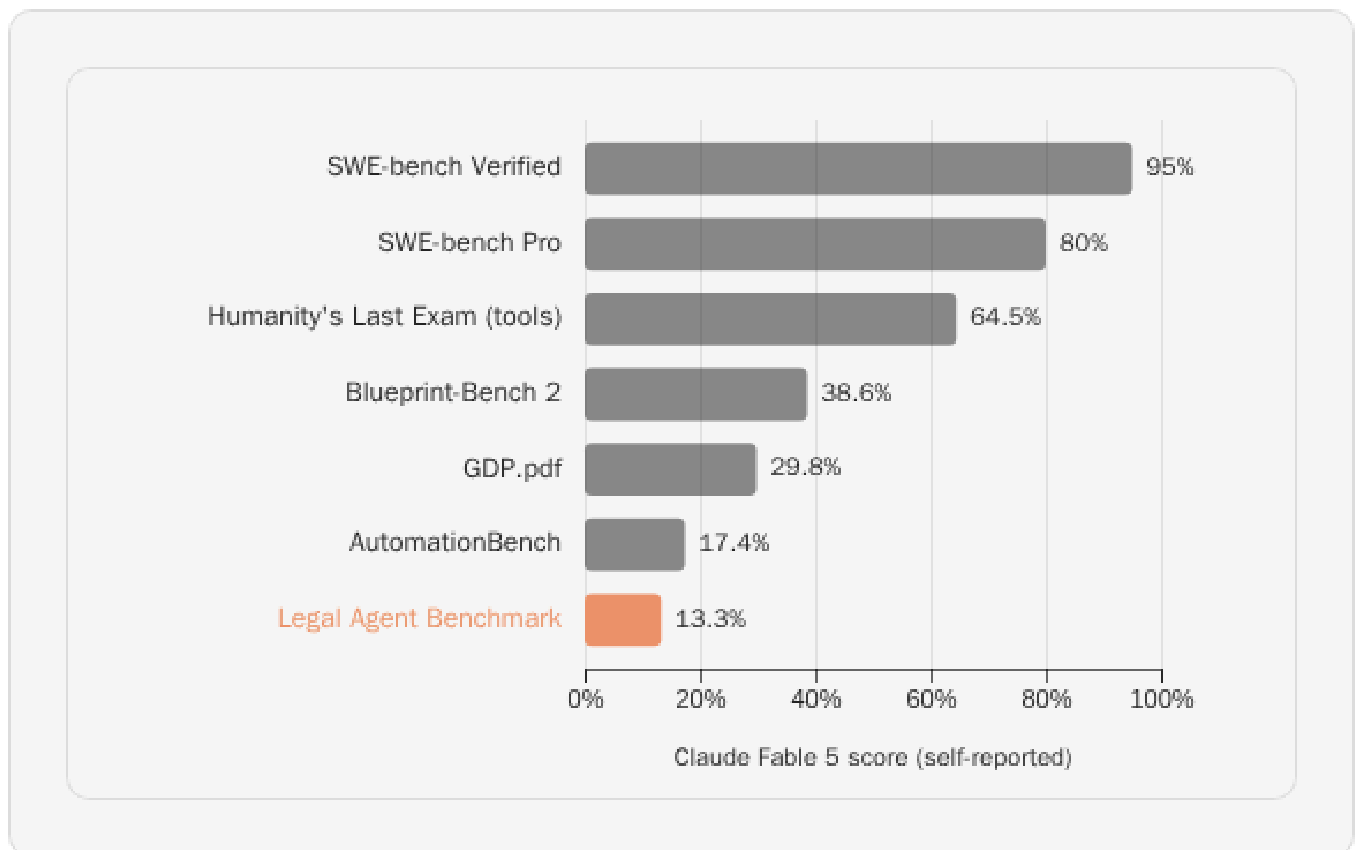


Figure 1 · Self-reported Claude Fable 5 scores from the June 2026 launch (Anthropic). The Legal Agent Benchmark (highlighted) is its lowest, well below software engineering and knowledge work.

And legal work is hard for everyone, not only open models. Here is where the field sits on LAB's all-pass rate today (Pereyra, 2026). Apart from Fable 5 (13.3%, not shown here, and pulled from general availability soon after launch), even the strongest model finishes only about 7% of tasks end to end, and most of the field sits near zero. That low ceiling is exactly why the gains from harness changes, rather than model swaps, are interesting.

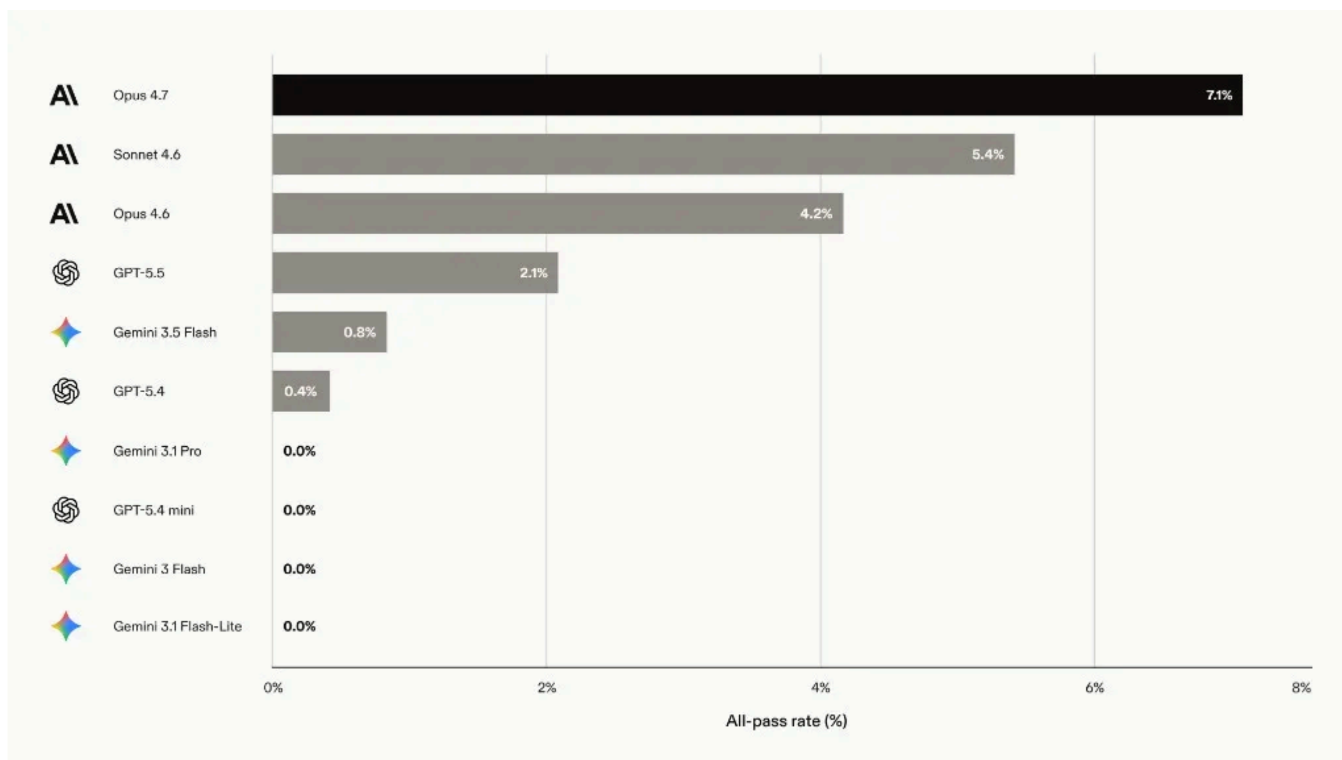


Figure 2 · Current state of the art on LAB (all-pass rate). Even the strongest models clear well under 10% of tasks end to end. Source: Gabe Pereyra.

The model and what a “harness” is

The model we optimize against is fixed: `deepseek-ai/DeepSeek-V4-Pro`, served through Hugging Face Inference Providers behind an OpenAI-compatible router. We never fine-tune it; the harness is the only thing that changes. (Later we take the resulting harness and run it untouched on other models too, to see how well it transfers.) The judge is a separate model, `claude-sonnet-4-6`.

Everything we are allowed to change lives in the harness: the scaffold of code around the model that decides which documents it can read, which tools it can call, what goes into its context, and when it is allowed to finish (Hugging Face, 2025). A vanilla harness just relays the task prompt and the model’s tool calls. A better harness can add a methodology to the system prompt, repair a malformed tool call, or check the output before the run ends. None of that touches the model weights, which is the point: we are measuring how much of an open model’s apparent weakness is actually the wrapper’s fault.

Building a held-out split

To report a fair number we need a held-out test set, and that is harder than it sounds. Harvey does not publish the train/test split they used for their post-training work; the blog only notes that

the hold-out is “distributionally similar to the training set” (Harvey, 2025c). We cannot reproduce a split we cannot see, so we built our own documented approximation over the 1,251 public LAB tasks, pinned to a fixed upstream commit and a fixed seed so it is reproducible.

The split has two disjoint parts, frozen as explicit task-id lists:

- `test` (100 tasks) is held out for final reporting only. It is built by a deficit-scoring greedy fill that drives several dimensions at once toward their global proportions: practice area, work type, rubric size, and the file-type flags (spreadsheet and slide sources, spreadsheet deliverables, email). The result tracks the underlying distribution closely, so the headline number is a reasonable estimate of performance in practice.
- `dev` (24 tasks) is the fast gate the loop optimizes against. It is one task per practice area, spread across small-to-extra-large context sizes.

On top of the proportional fill, `test` is deliberately made hard: it force-includes the extreme of every “more is harder” dimension, the matter with more than 1.5M tokens of context, the task with 11 separate deliverables, the rubric with roughly 194 criteria, and the matter with 55 documents. The giant-context matter overflows the model’s context window, so we report it as a separate slice rather than letting it distort the aggregate. One consequence worth keeping in mind: forcing in these extremes makes our test split a pessimistic view of the benchmark. It over-weights the hardest matters relative to the full task pool, so our absolute numbers likely sit a little below what the same harness would score on the full 1,251-task benchmark, or on Harvey’s own published results.

Split composition

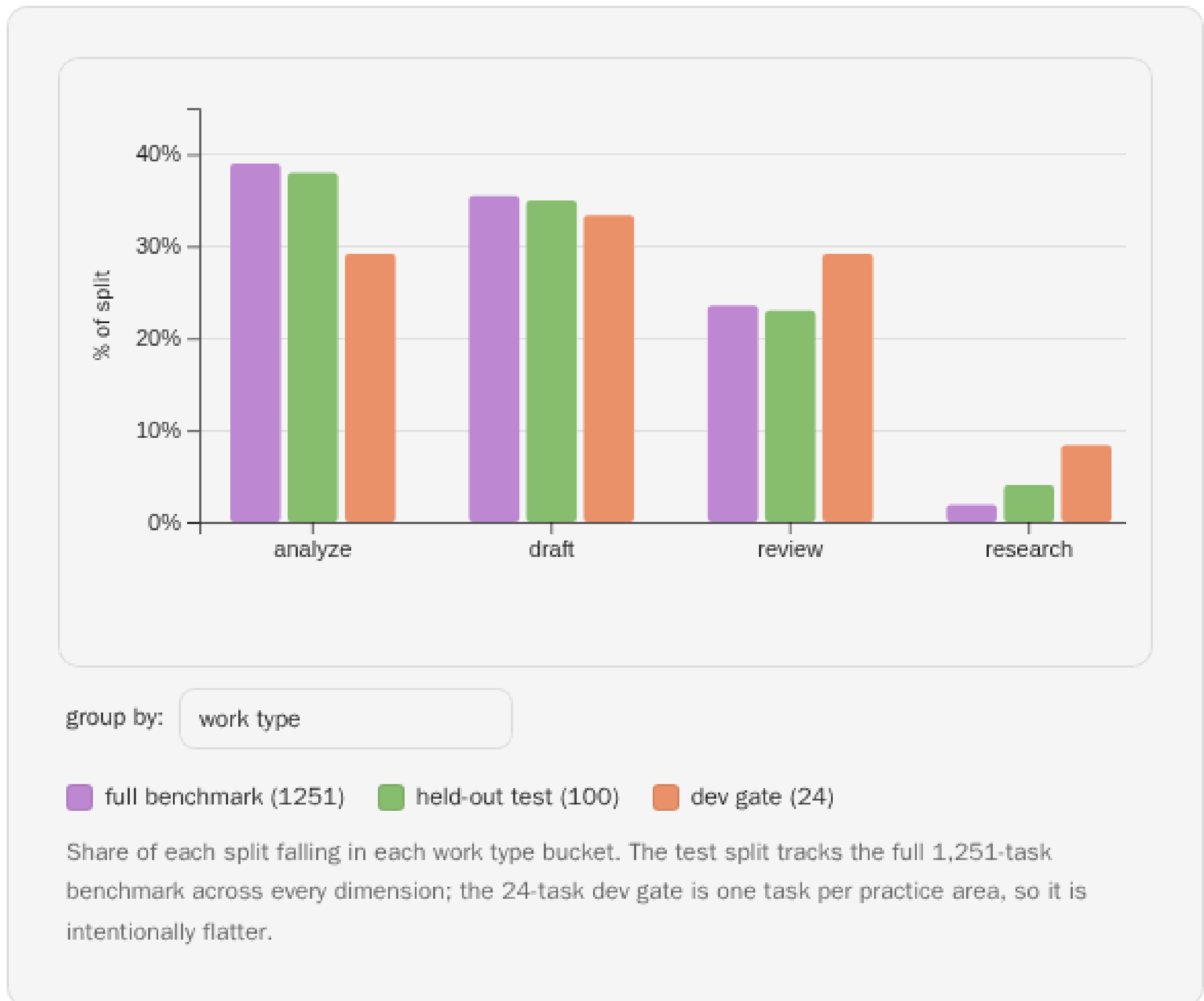


Figure 3 · The full benchmark, the held-out test split, and the dev gate side by side, as a share of each split. Switch the grouping to check any criterion the split was built to balance: practice area, work type, context size, rubric size, document count, deliverables, and file-type flags. The test split tracks the full benchmark closely; the 24-task dev gate is one task per practice area and so is intentionally flatter.

The small gate is a meaningful limitation. With only 24 dev tasks, the loop can only find and confirm fixes for failure modes that actually show up in dev. A problem that never appears in dev cannot be discovered or rewarded, so the optimized harness is shaped by dev’s failure distribution. The test split is the guard that the dev-driven wins generalize. The reason we keep dev this small is cost.

What an evaluation costs

LAB scoring is expensive, and the expense is dominated by the judge. The judge makes one Sonnet call per rubric criterion, so a single 100-task test run fires about 6,310 judge calls (the rubrics average 63 criteria and run up to 194). A single dev gate evaluation, 24 tasks at 3 trials each, is roughly 4,500 calls. The agent rollouts are long in their own right: one test run is around 158M input tokens across 100 multi-minute rollouts, some reaching nearly 1.8M tokens on a single task.

> Roughly what one run costs
>
> These are estimates with two parts, the agent rollouts and the Sonnet judge.
>
> The agent rollouts dominate. At the open-model rates we pay through Hugging Face Inference Providers, each rollout is on the order of $1, since most are over a million tokens. A test run is 100 rollouts, so about $90 to $100. A dev gate is 72 rollouts (24 tasks times 3 trials), but on smaller matters than the deliberately hard test split, so it lands in the same ballpark.
>
> The Sonnet judge adds the rest. It fires one call per rubric criterion, so a test run is about 6,300 calls. The prompt-caching wrapper above reuses the per-task deliverable prefix that all of a task's criteria share, cutting the judge's input bill 5 to 8 times, which brings those calls down to roughly $30 to $60.
>
> So a test run and a dev gate each land at very roughly $120 to $150, and the 22 gated iterations dominate the project's spend.

The practical rule we took from this: when scoring is an expensive LLM judge, keep the dev gate small and make it count, favoring coverage of distinct problem types over raw task count. When scoring is cheap and deterministic, such as multiple-choice grading, a much larger dev set is the better choice.

The vanilla baseline

With the splits fixed, the starting point is the stock LAB harness. On dev it scores 63.1% pooled criterion pass rate at 0% all-pass; on the held-out test set it scores 63.4% at 0%. The zero-score cluster is dominated by drafting and multi-deliverable tasks that produce no deliverable, an incomplete one, or one the grader cannot find.

The figure below shows where the baseline stands on every task, grouped by practice area. Click an area to expand its tasks, and hover for a short description.

Where the vanilla baseline lands, task by task

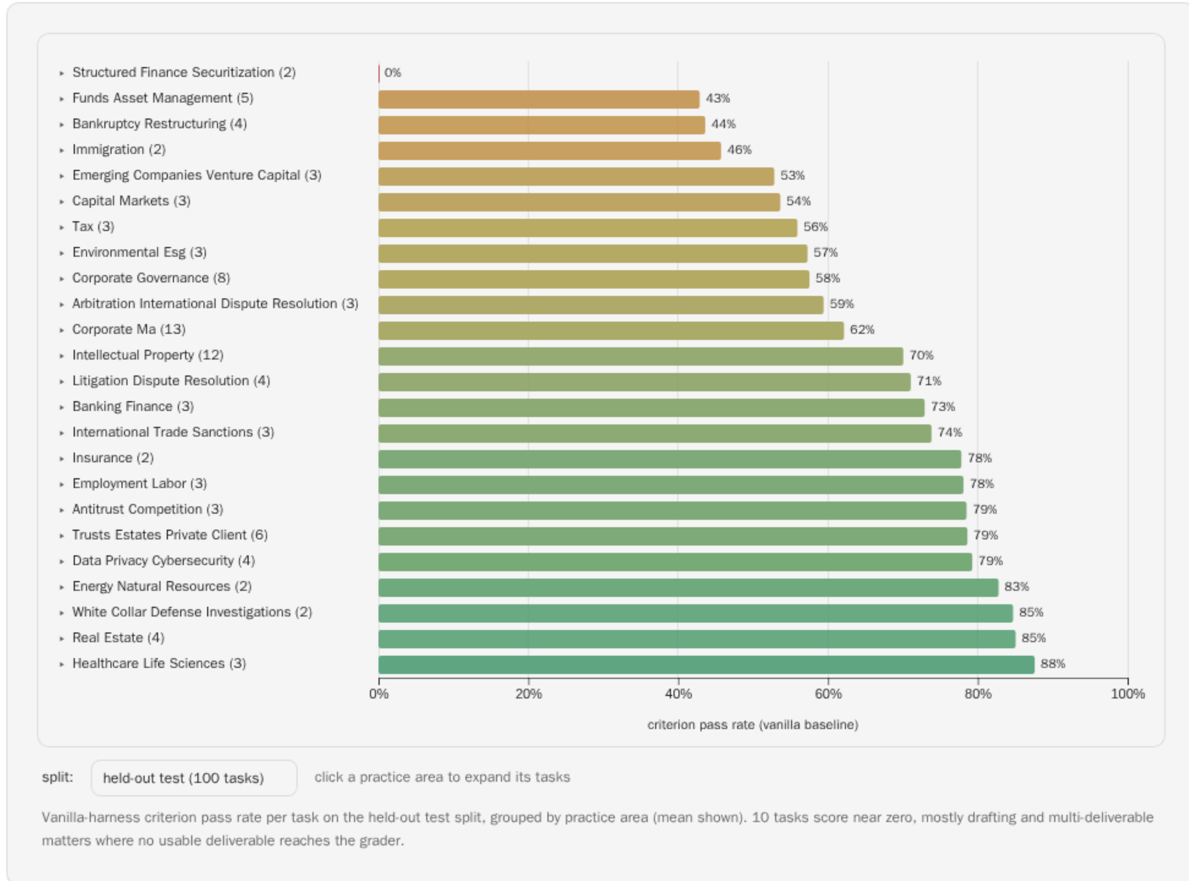


Figure 4 · Vanilla-harness criterion pass rate for each task, grouped by practice area (mean shown per area, color from low to high). Switch between the held-out test and dev splits, click an area to expand its tasks, and hover for descriptions. The cold cluster near 0% is what a harness search should be able to recover.

Those cold, near-zero tasks are exactly what an automated harness search should be able to attack, so the next question is how the search works.

How the loop works

The loop at a glance

The loop has two halves that play different roles. A language model acts as the researcher: it reads what happened, forms a hypothesis, and writes one new mechanism into the harness. A plain Python scaffold acts as the lab: it runs the experiment, scores it on a fixed gate, and decides whether to keep the change. Keeping these separate is what makes the results

believable. The model is free to be creative, and the scaffold is strict about what counts as an improvement.

One iteration of the loop

Figure 5 · Hover or tap each step to see what it does and where it lives in the code. The dashed steps are guardrails that protect the search from itself.

One pass works like this. The proposer copies the current best harness, adds exactly one mechanism, and checks it on a handful of dev tasks. It writes a small `pending_eval.json` describing the change. The outer loop then re-runs the candidate on all 24 dev tasks at 3 trials, scores it with a blended metric, and promotes it only if it clears the incumbent by a margin (the `min_delta`, one point, just above the noise left after averaging three trials). A winner becomes the new best harness and the next iteration builds on it.

The scaffold: `meta_harness.py`

The scaffold is deliberately boring, because its job is to be trustworthy. A few pieces matter.

The proposer runs through `propose()`, which calls Claude on the CLI subscription rather than an API key. It strips `ANTHROPIC_API_KEY` from the environment before the call and restores it afterward, so the proposer runs on the subscription while the judge keeps the key. The split is deliberate: the judge is thousands of programmatic, deterministic calls (temperature 0, prompt caching, automatic retries) that need the API, whereas the proposer is a single multi-hour interactive session that the subscription lets run without metering every token. A long proposer session is fragile, so `propose()` wraps it in auto-resume logic (`_resume_wait`): it resumes the same session after a session-limit reset, a timeout, or a transient server error, and stops cleanly on an expired login instead of burning the budget on instant failures.

Once a candidate exists, `evaluate_harness()` benchmarks it on the dev subset at three trials. `update_frontier()` then decides promotion. Two functions guard the search. `_touched_test()` scans the proposer's tool calls and file reads and rejects the whole iteration if it so much as looked at the held-out test split. `_history_digest()` feeds the proposer a summary of the frontier lineage plus the candidates that were evaluated but not promoted, so a useful mechanism that missed once can be picked up and stacked later.

The protocol: `SKILL.md`

If `meta_harness.py` is the lab, `SKILL.md` is the protocol taped to the wall that the researcher has to follow. It is a single instruction file the proposer reads at the start of every iteration, and a handful of its rules do most of the work, so we focus on those.

The first rule is one mechanism per iteration, built by copy-and-adapt. The frontier is the current record holder, and every challenger has to copy its routine and add exactly one new move. Concretely, the proposer copies the current best harness file verbatim and edits in a single new mechanism, rather than starting fresh. This is why wins compound instead of competing: every accepted change is already present in the next candidate's starting point.

We copy rather than subclass on purpose. After twenty-odd iterations, an inheritance chain would be twenty layers deep and unreadable, and a new mechanism often needs to adjust an earlier one. A flat copy keeps every mechanism visible in a single file the proposer can edit freely, with no subclass indirection to chase. The shared loop machinery (compaction hooks, custom tools) still lives in one reusable place; only the mechanisms are copied forward.

The rest of the protocol keeps the search general and the measurement clean. No task-specific hints are allowed in harness code, so a mechanism has to help on unfamiliar matters rather than memorize dev. The harness must keep `output/` clean, writing only finished deliverables there so the judge is not fed scratch files. And the proposer must never read the test split. For validation, the protocol asks for noise-robust evidence: either a causal replay (re-running a deterministic fix over existing transcripts and re-scoring the recovered output) or a pooled comparison over a fix

slice of at least 5 dev tasks plus a regression slice of at least 5 more. Single-trial, single-task swings are treated as noise. When the proposer is satisfied, it writes `pending_eval.json` and hands control back to the scaffold.

The five fields in that file map straight onto the discipline above: a `hypothesis`, the single `changes` it makes, the `fix_tasks` and `regression_tasks` slices it was checked on, and the `observed` evidence. Here is one real candidate the loop produced, with the long prose fields trimmed.

Example pending_eval.json (one candidate, prose trimmed)

The promotion gate

The gate is the part we spent the most care on. It is the referee, and the promotion score is the loop's objective the way a loss is for training: the search only ever moves toward what the score rewards, so a flaw in the score becomes a flaw in everything downstream. A candidate is promoted only when its blended score beats the current frontier by at least one point (the `min_delta`), a margin set just above the noise that survives averaging three trials.

LAB's headline all-pass metric is too sparse to optimize directly. On 24 dev tasks at three trials, a single extra all-passing rollout is about 1.4 points, so chasing all-pass directly means chasing one lucky run. We optimize the dense pooled rate instead, and fold all-pass in as a bonus:

```
score = pooled_criterion_rate  +  0.5 * all_pass_rate  -  0.005 * tokens_per_million

```

Each weight has a reason: the 0.5 on all-pass is tuned so a lone noisy all-pass run cannot, by itself, clear the 1% promotion margin, but a genuine pooled gain plus an all-pass gain compounds over the bar. The small cost term (about half a point per million tokens) keeps a much more expensive harness from winning on a marginal gain, without letting ordinary token variation dominate. `update_frontier()` recomputes the incumbent's score under the current weights on every comparison, so changing a weight can never promote a worse harness at a boundary. That last detail sounds pedantic; in the lessons section we show how skipping it once promoted the wrong harness.

Why three trials

Running each candidate three times per task triples the cost of every iteration. Here is why that is worth paying for. We ran the baseline ten times over (a 10-task dev slice, three trials, temperature 0) to see how noisy a single pass is.

Single-trial noise on a 10-task dev slice

Across the three full passes the aggregate pooled rate barely moves (73%, 76%, 75%; mean 75%, std 1.36pp). But individual tasks swing a lot, driven by occasional single-trial collapses to zero, not smooth noise.

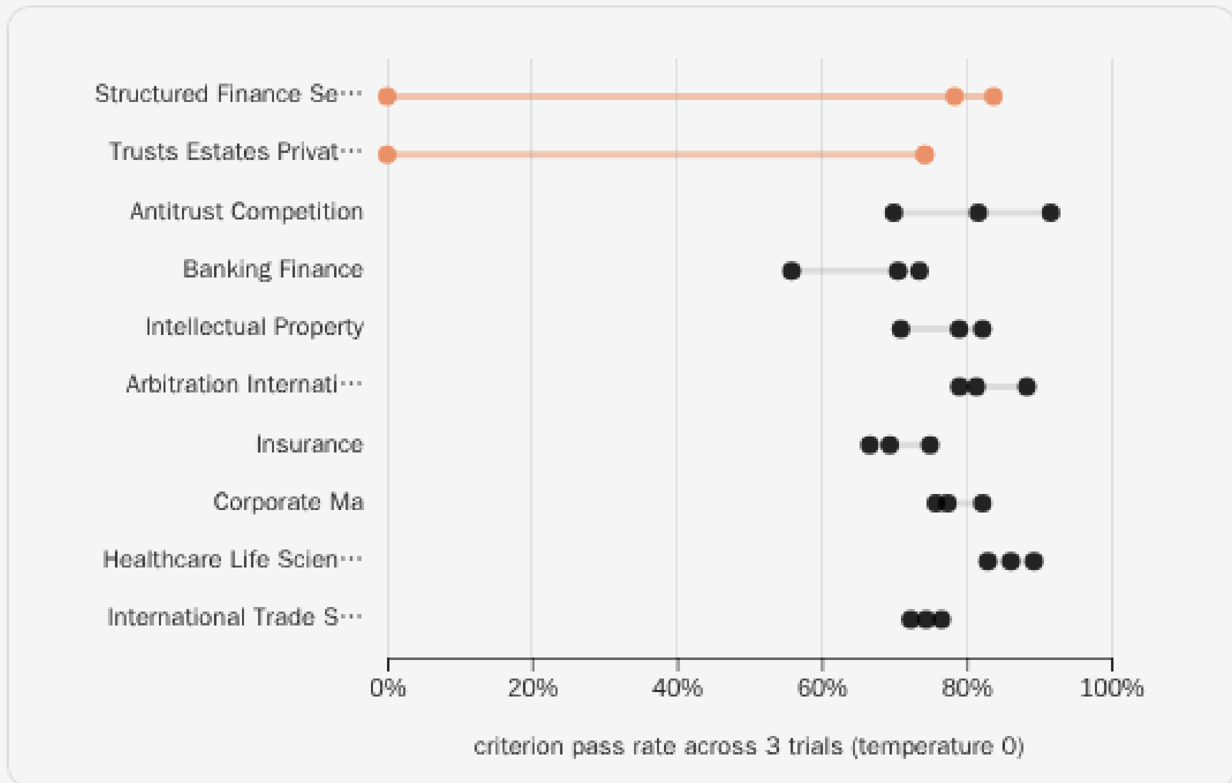


Figure 6 · Each row is a dev task; the dots are its three trials and the line spans their range. The aggregate pooled rate is stable across passes, but individual tasks swing widely, and the swings come from a few catastrophic single-trial collapses to zero, not smooth jitter.

The aggregate is reassuring but the per-task picture is not. Pooled over all tasks, the three single passes land at 72.9%, 76.2%, and 74.9%: a mean of 74.7% with a standard deviation of 1.4 points, so a single pass is fine for a headline number on a large split. But per task the spread is large, and it is not smooth jitter: a couple of tasks have one trial collapse to 0% while the others sit near 80%, usually from a one-off provider error or a run that finishes without writing the deliverable. On the 24-task dev gate those collapses move the aggregate enough to flip a promotion decision. Averaging three trials roughly halves the run-to-run noise on a harness's score, from about 1.4 points on a single pass to around 0.8 (the standard deviation over the square root of three), and the 1-point promotion margin sits just above that. The margin is a deliberate balance rather than a statistical guarantee: a stricter 3-point margin blocked real

stacked gains of 1.5 to 1.8 points and wasted expensive proposer sessions, while 1 point lets directionally real gains compound and still rejects single-pass noise (on a single pass you would need a 3 to 5 point swing to trust it). That is why we pay for the extra passes.

Where this sits: program evolution, and what we add on top of Meta-Harness

Using a language model to evolve code or prompts against a metric is a fast-growing idea, and it helps to place this work next to its neighbors.

- Meta-Harness (Lee et al., 2026) is the framework we build on: automated search over the harness around a fixed model, with a language-model proposer and per-domain reference experiments. The repository root is the framework; this post is a new domain applied on top of it.
- AlphaEvolve (Novikov et al., 2025) and its open re-implementation OpenEvolve (Sharma, 2025) evolve whole programs and algorithms against a verifier. The closest cousins, except they evolve the solution program itself rather than a harness around a fixed model.
- ShinkaEvolve (Lange et al., 2025) does sample-efficient program evolution with a population across islands, an archive, novelty rejection, and an ensemble of models as mutation operators. The contrast is instructive: it keeps a diverse population and Pareto-style selection, while we keep a single compounding frontier and add one mechanism per step. Simpler, but less diverse.
- The Darwin Gödel Machine (Zhang et al., 2025) is a self-improving agent that rewrites its own code, validated empirically rather than by proof. Same “the agent improves the agent” spirit; we constrain the edits to the harness layer and keep the base model and the loop code fixed.
- GEPA (Agrawal et al., 2025) is reflective prompt evolution against a Pareto frontier. We push the same reflect, mutate, keep idea from prompt text out to harness code.
- Karpathy’s autoresearch (Karpathy, 2026) has an agent edit one file, run a fixed five-minute experiment, and keep or discard by a single metric overnight, with the human only writing a lightweight `program.md`. Our `SKILL.md` plays the role of that `program.md`; the agent edits `agents/<name>.py` instead of `train.py`; and our “experiment” is an expensive multi-rollout, LLM-judged evaluation rather than a short training run.

What this experiment adds is not a change to the core search idea but the work needed to run it on a hard, expensive domain. It is the first application of Meta-Harness to long-horizon legal agent work, where deliverable discipline dominates the score. It adds the domain infrastructure: an adapter over Hugging Face Inference Providers with automatic provider failover, a prompt-caching judge, the reproducible split, and a comparison harness. It adds the noise-aware blended gate above, with the weight-recompute fix. And it adds the reliability work that lets a subscription

proposer run unattended over flaky infrastructure. With the machinery in place, the question is what it actually found.

Results

The climb

The chart below is the whole story in one view: the dev pooled rate climbing as the loop runs, iteration by iteration. Each dot is a candidate harness; the line is the running best, the frontier. Promoted candidates lift the line, rejected ones sit below it. It plays on a loop, and you can scrub through it.

The dev frontier, iteration by iteration

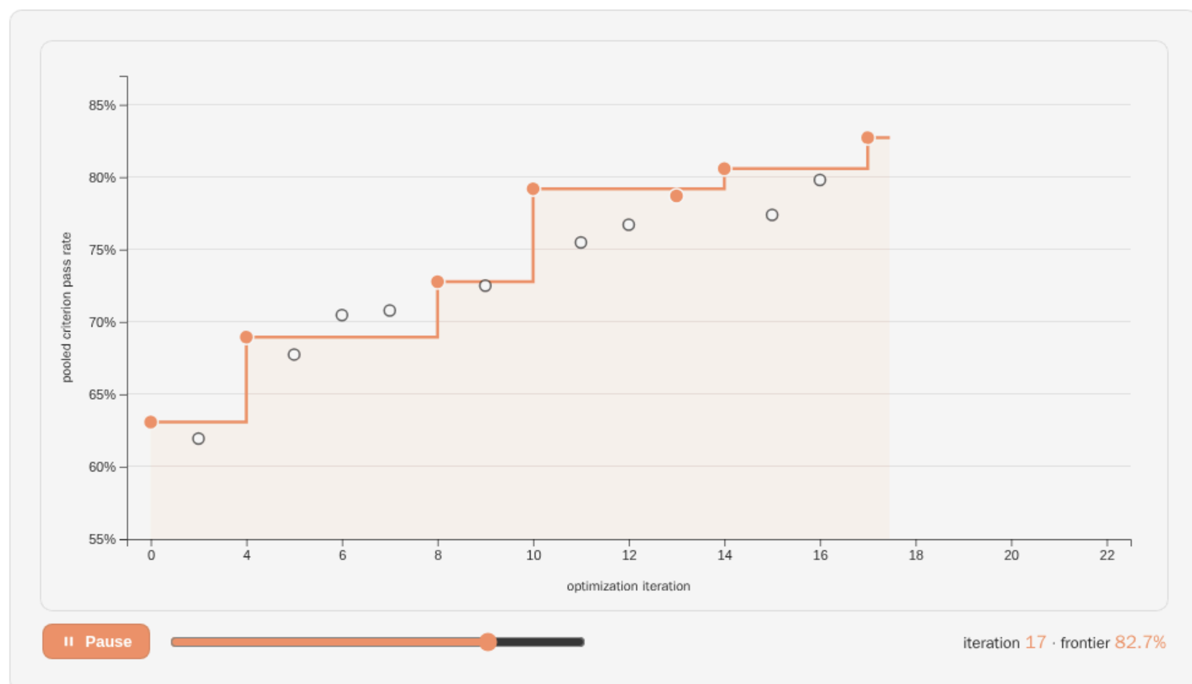


Figure 7 · Pooled criterion pass rate on the 24-task dev set across the optimization run. Filled dots were promoted to the frontier; hollow dots were evaluated but not promoted. Hover any dot for the mechanism it added. It plays on a loop; use Pause and the scrubber to step through it.

Two things stand out. First, the climb is not smooth: most candidates are rejected, and the frontier moves in a handful of decisive steps. Second, the steps compound rather than compete, because each candidate starts as a copy of the current best. The biggest single jump, from 72.8% to 79.2%, comes from one mechanism (a deliverable-landing gate), and once it is in, every later

candidate keeps it for free. By the end the frontier sits at 83.3% pooled with 4.2% whole-task success, up 20.2 points from the vanilla baseline.

What the mechanisms are

It helps to see all the candidates at once, not just the winners. The next chart plots every harness the loop evaluated by its cost (mean tokens per run) against its dev score, colored by what kind of change it was.

Every candidate, by cost and score

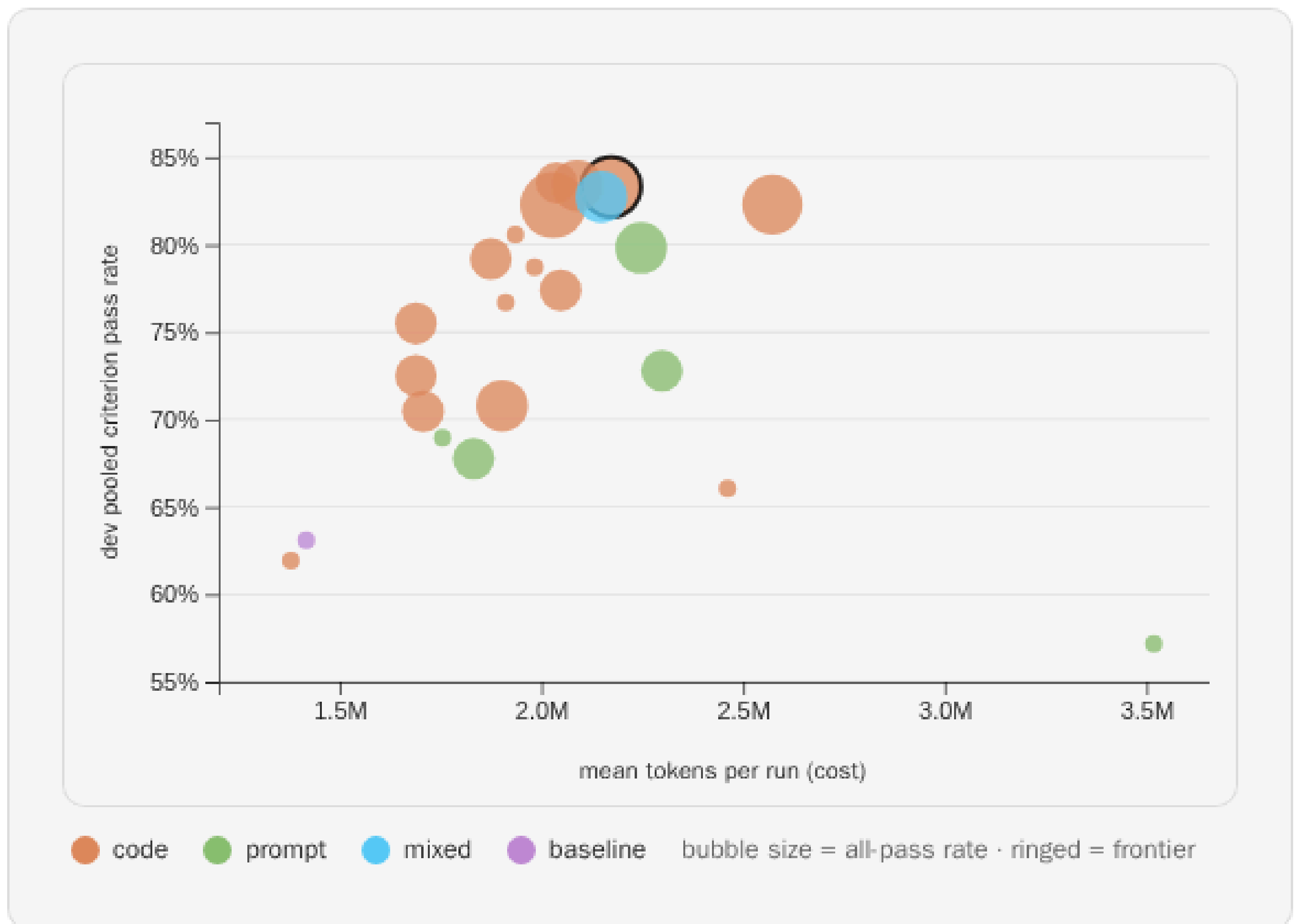


Figure 8 · Each bubble is a harness; size is its whole-task (all-pass) rate, the ring marks the final frontier. Color is the mechanism type. Hover for the blended gate score. The strong candidates cluster toward the top, and most of them are code.

The mechanisms fall into a few families, and the ranking among them is the main result.

- Deliverable landing and delivery (code). This is the single biggest lever. The judge only loads files at the top level of `output/` under the exact requested name, and the model often does the legal work but leaves the deliverable misplaced, misnamed, split across chunks, or in a scratch folder. A deterministic post-solve gate that lands the right file, using zero extra model

tokens, converts a score of roughly zero into a scored result. The landing gate and its variants are the high points of the board.

- Matter fidelity (code). A runtime self-check that catches the model drafting about the wrong matter. More on this below, because the loop found it on its own.
- Loop robustness (code). Repairing a provider-corrupted tool call (`toolcall_json_repair`) and breaking repetition loops. This family is small but, as the transfer section shows, the most portable across models.
- Prompt playbooks (prompt). Work-type-specific methodology added to the system prompt, for analysis, drafting, and redlining. These delivered the early gains and then plateaued.

The pattern is hard to miss: five of the top six harnesses are deterministic code, not prompt edits. For a weak agent, a lot of apparent capability is actually I/O discipline that the scaffold can simply guarantee. Harvey's own harness-engineering write-up also landed on the same idea, with stop hooks that validate deliverables before a run ends (Artificial Lawyer, 2026).

The optimizer out-found us: matter fidelity

Before the run we seeded a list of ideas: completion gates, revise-after-check, compaction, prompt playbooks. The mechanism that ended up helping most was none of them.

On long-context tasks, the workspace sometimes contains a distractor matter, an unrelated document bundled into the data room. The model occasionally locks onto it and drafts a thorough, well-written answer about the wrong matter, for example a Title VII employment complaint that happens to be sitting in the folder, instead of the assigned breach-of-contract dispute. The work is good and completely off-target, so it scores near zero. The loop discovered this, then built a runtime check that detects a wrong-matter draft and forces a redraft. Made reliable with a small best-of-N vote, this `matter_audit_allwork` mechanism became the final frontier. It is a clean example of automated search finding a failure mode that none of our seeded ideas (drawn from Harvey's write-ups and our own reading) had anticipated. We were not looking for it; the loop was.

The heatmap shows where the lineage actually moved the needle, task by task.

Per-task rescue across the frontier lineage

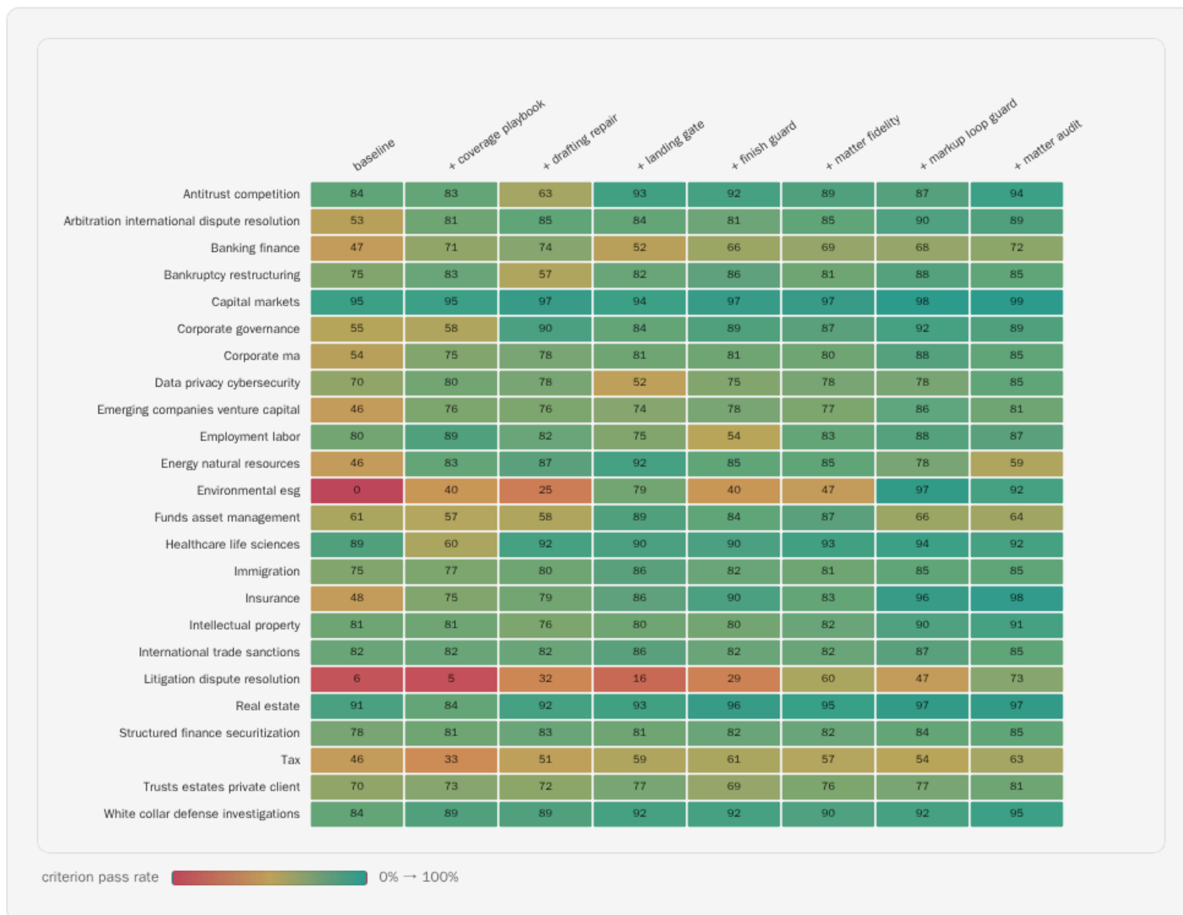


Figure 9 · Each row is a dev task, each column a step in the promoted lineage; color is the mean criterion pass rate across 3 trials. Hover a task name for its rubric size and input (context) tokens, or a cell for that step's mean output tokens. Watch specific drafting and multi-deliverable rows jump from cold to warm when the landing gate and matter audit come in.

The aggregate climb hides a lot of structure. Some tasks are near-solved from the start and barely move. A few drafting rows stay cold until a specific mechanism lands, then jump in one step. That is the copy-and-adapt loop working as intended: a targeted fix for an observed failure mode, stacked onto everything before it.

Does it transfer?

The harness was tuned only on DeepSeek-V4-Pro, so the obvious thing to check is whether it is a generic improvement or one tied to that model. For a clean comparison we ran one harness, the mid-lineage `deliverable_landing_gate`, untouched on each model, on the same held-out test split and fair basis. Using the same harness everywhere is what makes the numbers comparable.

Cross-model transfer (held-out test)

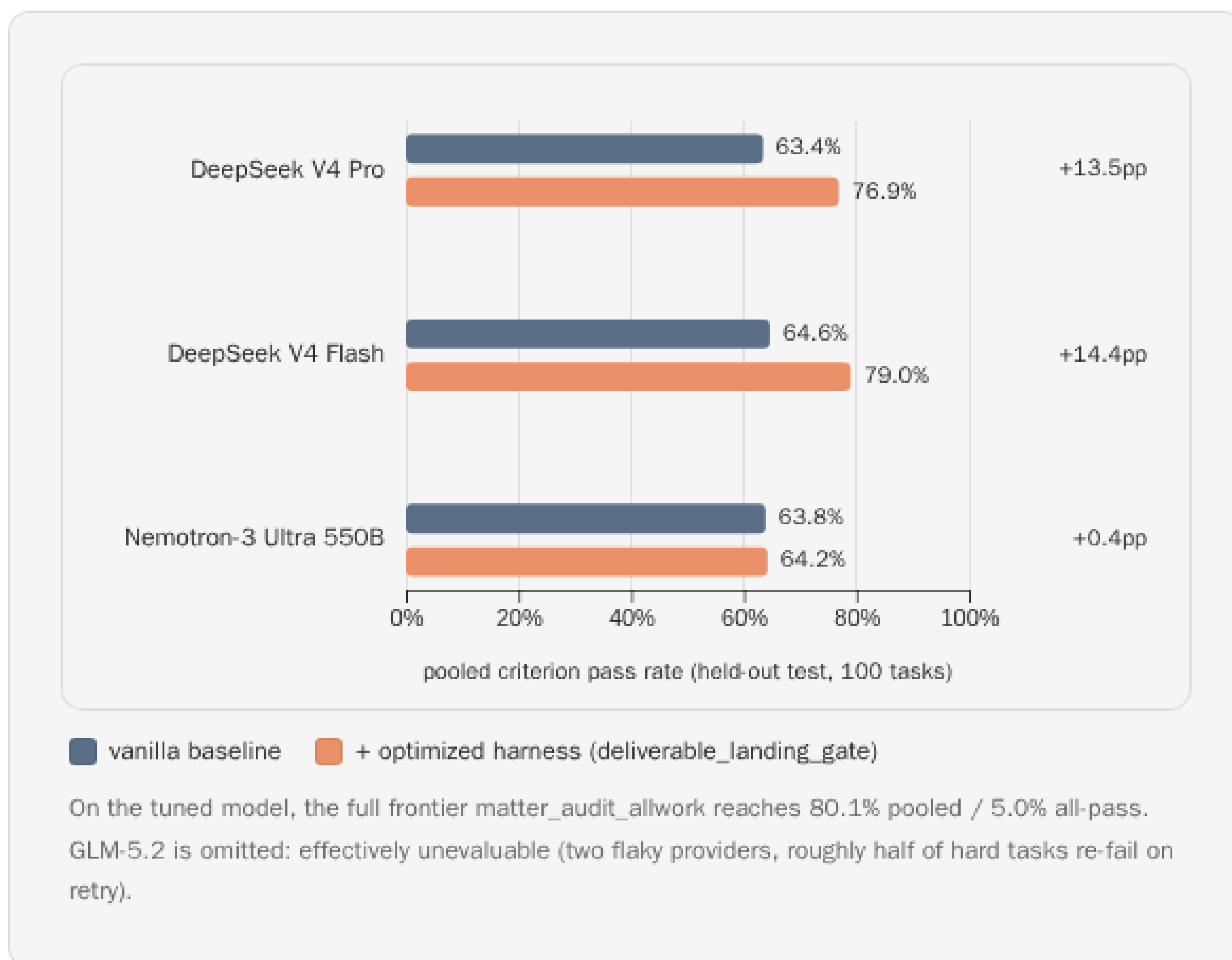


Figure 10 · Vanilla baseline vs the V4-Pro-optimized harness, run untouched on each model. Within the same model family the harness transfers almost fully; across families it is roughly flat.

Within the same family the harness transfers almost fully: DeepSeek V4 Flash gains 14.4 points, even more than the 13.5 the tuned model itself gets from this same `deliverable_landing_gate` (its full frontier reaches +16.7, but the cross-model study uses the earlier landing gate so every model runs the identical harness). A different family, Nemotron-3 Ultra, barely moves at +0.4 points. That near-zero average is misleading, though. Breaking the runs down task by task shows two effects that roughly cancel: the robustness mechanism (`toolcall_json_repair`) clears Nemotron's tool-call crashes, so its turn-0 failures drop from about ten to roughly one, while the DeepSeek-tuned prompt playbooks hurt its work on tasks it can already finish, by around ten points on the tasks it completes. The takeaway is in that split: robustness and code mechanisms transfer across families, while prompt playbooks are model-specific and can even backfire.

Open vs closed, in context

The pooled gains are large, but the number reported on LAB is all-pass, so it is worth asking where the optimized open model lands against the closed frontier. The chart puts our open models, vanilla and optimized, on the same axis as the closed-model all-pass rates Harvey reported (Pereyra, 2026).

All-pass rate: our open models vs the closed frontier

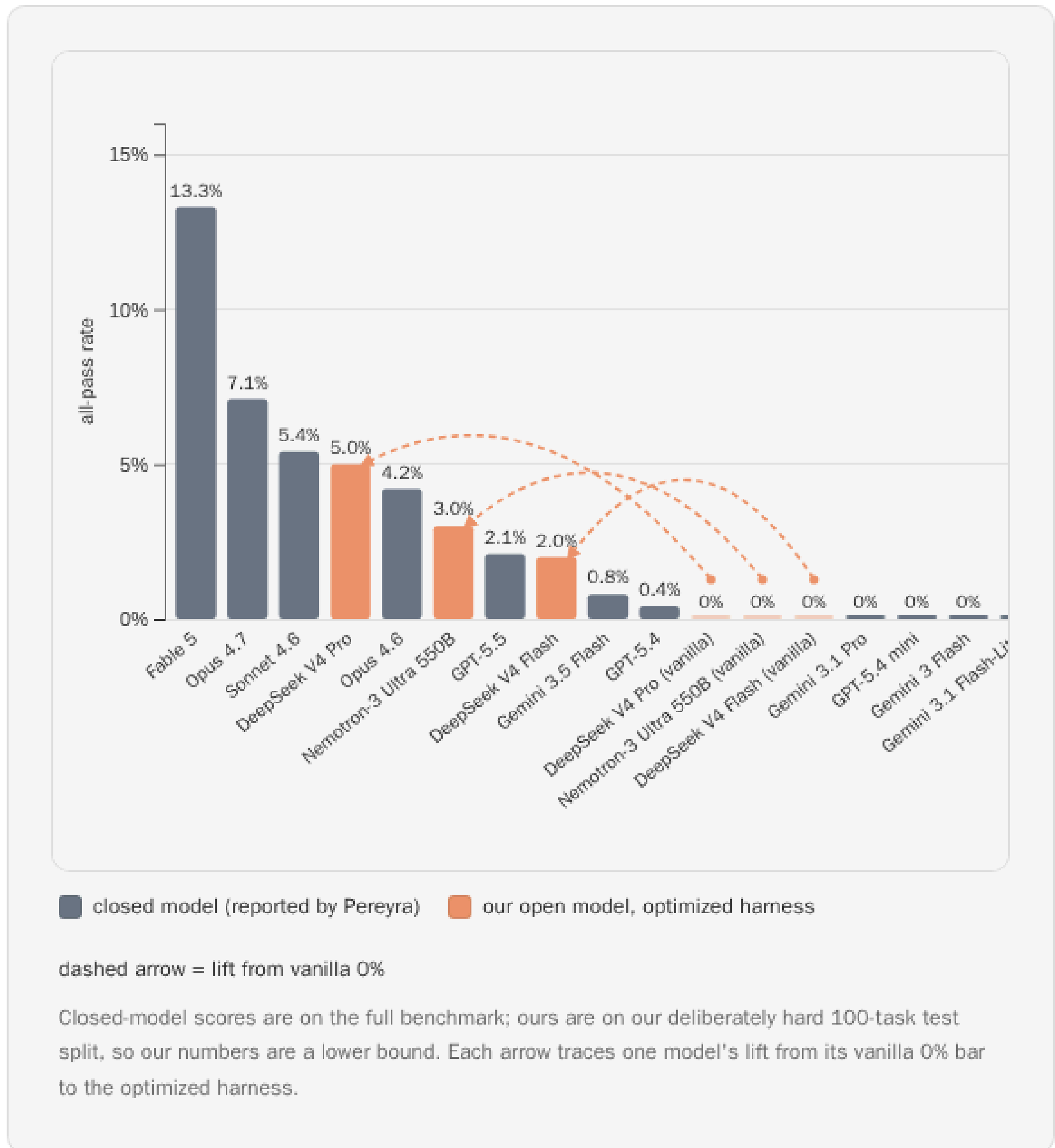


Figure 11 · Whole-task (all-pass) rate, every model shown as a bar. Slate bars are closed models on the full benchmark (reported by Pereyra), including the several that score 0%. Dashed arrows trace our open models from their vanilla 0% bar up to the optimized harness on our hard test split. The optimized open model lands amongst the closed frontier.

The optimized open model is competitive on whole-task success. Vanilla DeepSeek V4 Pro solves 0% of tasks end to end; with the optimized harness it reaches 5.0%, which sits between Opus 4.6 (4.2%) and Sonnet 4.6 (5.4%) and above GPT-5.5 (2.1%) on Harvey's reported numbers. The same

plumbing that lifts pooled criteria also flips a handful of tasks from “good answer, wrong file” to a full pass.

Two caveats keep this in perspective. First, the comparison is not on the same tasks: the closed numbers are on the full benchmark, while ours are on the deliberately hard test split that force-includes the largest matters and biggest rubrics. That split over-weights the hardest tasks, so our 5.0% is a lower bound; on the full pool the same harness would likely score a little higher. Second, we did not re-run the closed models with our harness, because the cost is prohibitive.

How prohibitive? A single test run is about 158M agent input tokens plus tens of millions of output tokens across 100 multi-minute rollouts. At GPT-5.5’s published rates ($5 per million input, $30 per million output), the input alone is around $790, before output or the long-context surcharge the largest test matters trigger (prompts over 272K tokens bill at 2x input). All in, one GPT-5.5 pass lands on the order of $1,000 to $1,500, roughly ten times an open-model run, and a fair comparison would want several. That is why the closed bars here are Harvey’s reported numbers, not our own re-runs.

The ceiling

The run does not climb forever. The top candidates cluster in a tight band around 83% pooled, and the matter-audit plus deliverable-landing family converges to what looks like a local ceiling for this dev set. It is worth being clear about what the loop reliably finds and what it does not. It finds I/O discipline, crash robustness, and targeted fixes for specific failure modes. It does not crack deep substantive coverage. The tax Section 382 task, for instance, misses roughly 30 of its 107 criteria on every trial, on quantitative depth and citation precision that no amount of scaffolding seems to add, and second passes get cost-killed by the gate. That residual looks model-bound rather than harness-fixable, which is a natural place for the harness to stop and the model to take over.

Lessons: making it trustworthy

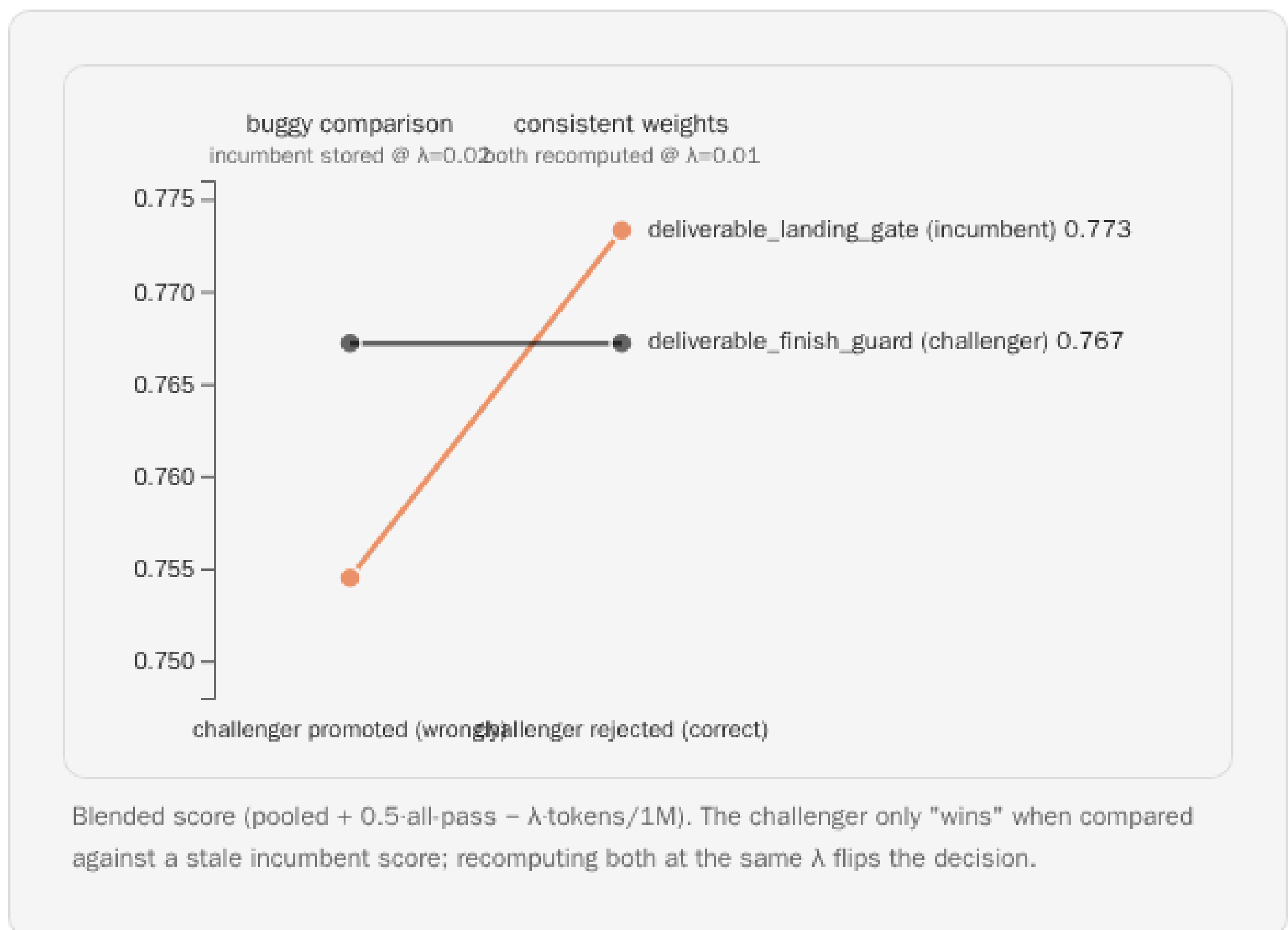
When this phase started, the loop already worked, but it could not be trusted. It lost iterations to crashes, it occasionally promoted the wrong harness, it measured some models unfairly, and it left a record nobody could read. The search itself was the easy part. Almost all of the engineering went into the unglamorous work that makes the results believable: surviving crashes, scoring correctly, and measuring every model fairly.

A promotion-gate bug (the cost-λ boundary)

The clearest example is a bug in the gate itself. At one point `deliverable_finish_guard` got promoted even though it was worse than the `deliverable_landing_gate` before it on every axis: lower pooled, lower all-pass, and more tokens. It should have been rejected outright.

The cause was a stale number. The gate compared the challenger's fresh cost-adjusted score against the incumbent's score, but the incumbent's score had been stored in an earlier run computed with a different cost weight. The comparison was apples to oranges.

How a stale weight promoted a worse harness



Blended score (pooled + 0.5·all-pass − λ·tokens/1M). The challenger only "wins" when compared against a stale incumbent score; recomputing both at the same λ flips the decision.

Figure 12 · Blended scores for the incumbent and challenger. Compared against the incumbent's stale stored score (left), the challenger wins. Recompute both at the same cost weight (right) and the incumbent is clearly ahead, so the challenger should have been rejected.

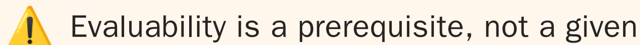
The fix is the rule the gate now follows: always recompute the incumbent under the current weights, never trust a stored derived number. Two lessons came out of it. Never compare metrics computed under different hyperparameters, and remember that any persisted derived value goes stale the moment the formula changes. There was also a quiet save here. Because candidates compound by copy-and-adapt, the mistaken promotion did no lasting damage: the finish-guard

mechanism was carried into the next candidate anyway, and it is still wired into the final frontier. A forgiving search structure absorbs the occasional bad decision.

Provider reliability as a first-class variable

Running open models through a hosted router, infrastructure flakiness kept masquerading as model failure, and a good share of the work went into telling the two apart.

Long agent rollouts (minutes each, up to a couple of million tokens) regularly hit stream drops, protocol errors, and rate limits, so we added runtime provider failover and per-provider retries. The starkest case was Nemotron, which first scored about 1 out of 100: the adapter was not retrying a bare server error on the first turn, so most runs died at turn zero. A one-line fix turned a “this model cannot do legal work” conclusion into a fair measurement.



> GLM-5.2 was effectively unevaluable through the router: only two providers, both with frequent stream drops, and roughly half of the hard tasks re-failed on retry. A loop that depends on stable, repeatable evaluations simply cannot run on a model you cannot reliably sample. That is worth checking before, not after, you design an experiment around a model.

This is also why we re-ran transient provider failures before reporting the test numbers, so that every remaining zero is a genuine model or harness failure rather than an infrastructure artifact.

Hardening the LLM as an optimizer

The proposer is a Claude subscription session that runs for one to three hours per iteration, and keeping it alive was most of the engineering. It auto-resumes when an interruption is recoverable (a session-limit reset, a timeout, a transient server error) and stops cleanly on one that is not, like an expired login that no retry can fix, which we learned after losing three iterations to silent authentication failures. A process-group kill prevents orphaned rollout subprocesses, and the leakage audit rejects any iteration that touched the held-out test split.

The point is simple: an LLM acting as an optimizer needs the same operational hardening as any long-running distributed job, retries, idempotent resume, clean failure, and a guard against contaminating held-out data. It is unglamorous plumbing, but without it the headline number would not hold up.

Conclusion

What the loop reliably finds

Stepping back, the result is encouraging in a specific, bounded way. Holding an open model fixed and letting an automated loop rewrite only the harness moved DeepSeek-V4-Pro from 63.1% to 83.3% on the dev gate and, untouched, from 63.4% to 80.1% on a held-out test set it never saw. That is a large gain from changing zero model weights.

The shape of the gain is the takeaway. The loop reliably finds I/O discipline, crash robustness, and targeted fixes for specific failure modes, and the most valuable of these are deterministic code, not prompts. It even found a failure mode we had not thought to look for. The robustness pieces transfer across model families; the prompt pieces do not. And the search has a clear ceiling: where the residual failures are about substantive depth rather than plumbing, the harness stops helping and the base model has to carry the rest. A general, transferable harness is essentially the robustness subset of a model-specific optimum.

The surprise was how much of the value came from trust rather than search. What makes the headline number worth reporting is the unglamorous part: a loop that survives its own runtime, measures every model fairly across flaky providers, and uses a gate that neither noise nor a stale weight can game. Generating candidates was the easy part; making their evaluation believable was the work.

What's next

A few directions follow naturally from where the loop stalls.

- A cheaper judge proxy. The Sonnet judge is the part of the bill that scales with the gate, and the reason we keep it at 24 tasks. Suppose we swapped it for a cheaper model calibrated to agree with Sonnet 4.6, with Sonnet only validating the final result. Take DeepSeek V4 Flash as the example: it lists around $0.14 per million input tokens and $0.28 output, against Sonnet 4.6's $3 and $15, roughly 20x cheaper on input and 50x on output (and its cache-hit input is cheaper still). The judge is input-heavy, so this would cut the judge cost from roughly $30 to $60 per run down to a few dollars. The agent rollout (about $90 to $100, and fixed) would then dominate, so a single evaluation drops only about a quarter overall, but the judge-specific cost drops by more than 20x. That is the bigger prize: for the same judge budget, the dev gate could grow by more than an order of magnitude, which is the most direct lever on what the loop can discover.

- A bigger, cheaper dev gate. With cheaper scoring the 24-task gate could grow, which is the most direct way to surface failure modes that never appear in 24 tasks.
- Other hard benchmarks. The same loop should apply to other rubric-graded legal benchmarks first, such as LEXam (Fan et al., 2025) and PLawBench (Shi et al., 2026), and then to very different domains like Humanity's Last Exam (Phan & others, 2025) and CritPt (Zhu et al., 2025). CritPt is interesting because it already has a strong hand-engineered research harness, PhysicsIntern (Louapre, 2026), which lifts Gemini 3.1 Pro from 17.7% to 31.4%. The natural test is whether an automated harness search can match or beat a carefully hand-built one.
- A population instead of a single frontier. We keep one compounding best harness. Methods like ShinkaEvolve (Lange et al., 2025) (islands and an archive), AlphaEvolve (Novikov et al., 2025), and the Darwin Gödel Machine (Zhang et al., 2025) (a branching lineage) keep a diverse population instead. Carrying several harnesses at once could preserve mechanisms that look weak alone but combine well, at the cost of more evaluation.
- Tools for the substantive ceiling. A web-search tool for research-type matters, and structured computation via code for dollar reconciliation, date timelines, and spreadsheet parsing, target exactly the depth the current harness cannot add on its own.
- Size-gated compaction. A memory pass that engages only above a token threshold would address the giant-context matter that overflows the window today, without hurting the small matters where compaction is lossy.

More broadly, this fits a clear trend. Whether the artifact being evolved is a training script (Karpathy, 2026), a whole program (Lange et al., 2025; Novikov et al., 2025), a set of prompts (Agrawal et al., 2025), an agent's own code (Zhang et al., 2025), or, here, the harness around a fixed model (Lee et al., 2026), language models are turning out to be capable researchers when you wrap them in a loop with a good verifier. The verifier is the catch. Domains like coding or competition math have deterministic checkers, but law had no obvious one for a long time, which is part of why automated legal-agent work lagged. LAB's answer, a detailed rubric scored by an LLM judge, is the natural substitute: not a deterministic verifier, but a structured, criterion-by-criterion signal dense enough for a search loop to climb. Get that verifier right and the rest of the loop is mostly engineering: a small gate that covers the right problems, an objective that rewards genuine gains over lucky ones, and enough operational care that you can believe what comes out.

Citation

For attribution in academic contexts, please cite this work as

```
Joel Niklaus (2026). "Don't Train the Model, Evolve the Harness".
```

BibTeX citation

```
@misc{niklaus2026_don_t_train_the_model_evolve_the_harness,
  title={Don't Train the Model, Evolve the Harness},
  author={Joel Niklaus},
  year={2026},

}
```

Reuse

Diagrams and text are licensed under CC-BY 4.0 with the source available on Hugging Face, unless noted otherwise. Figures reused from other sources are excluded and marked in their captions ("Figure from …").

References

1. Harvey. (2025). *Introducing Harvey's Legal Agent Benchmark*. Harvey blog. https://www.harvey.ai/blog/introducing-harveys-legal-agent-benchmark↑

2. Hugging Face. (2025). *Agent Glossary*. Hugging Face blog. https://huggingface.co/blog/agent-glossary↑

3. Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., & Finn, C. (2026). *Meta-Harness: End-to-End Optimization of Model Harnesses*. https://arxiv.org/abs/2603.28052↑

1. Anthropic. (2026). *Claude Fable 5 and Claude Mythos 5*. Anthropic announcement. https://www.anthropic.com/news/claude-fable-5-mythos-5↑

2. Harvey. (2025a). *Introducing Harvey's Legal Agent Benchmark*. Harvey blog. https://www.harvey.ai/blog/introducing-harveys-legal-agent-benchmark↑

3. Harvey. (2025b). *Legal Agent Benchmark: Initial Results*. Harvey blog. https://www.harvey.ai/blog/legal-agent-benchmark-initial-results↑

4. Harvey. (2025c). *Post-training Open Legal Agents with Baseten Research*. Harvey blog. https://www.harvey.ai/blog/post-training-open-legal-agents-with-baseten-research↑

5. Hugging Face. (2025). *Agent Glossary*. Hugging Face blog. https://huggingface.co/blog/agent-glossary↑

6. Pereyra, G. (2026). *State of the art on the Legal Agent Benchmark*. X (Twitter). https://x.com/gabepereyra/status/2059320727988224128↑

1. Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., & Khattab, O. (2025). *GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning*. https://arxiv.org/abs/2507.19457↑

2. Karpathy, A. (2026). *autoresearch: AI agents running research on single-GPU nanochat training automatically*. GitHub repository. https://github.com/karpathy/autoresearch↑

3. Lange, R. T., Imajuku, Y., & Cetin, E. (2025). *ShinkaEvolve: Towards Open-Ended and Sample-Efficient Program Evolution*. https://arxiv.org/abs/2509.19349↑

4. Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., & Finn, C. (2026). *Meta-Harness: End-to-End Optimization of Model Harnesses*. https://arxiv.org/abs/2603.28052↑

5. Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., & Balog, M. (2025). *AlphaEvolve: A coding agent for scientific and algorithmic discovery*. https://arxiv.org/abs/2506.13131↑

6. Sharma, A. (2025). *OpenEvolve: an open-source evolutionary coding agent*. GitHub. https://github.com/algorithmicsuperintelligence/openevolve↑

7. Zhang, J., Hu, S., Lu, C., Lange, R. T., & Clune, J. (2025). *Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents*. https://arxiv.org/abs/2505.22954↑

1. Artificial Lawyer. (2026). *Harvey Drives Legal Agent Learning via Harness Engineering*. Artificial Lawyer. https://www.artificiallawyer.com/2026/04/07/harvey-drives-legal-agent-learning-via-harness-engineering/↑

2. Pereyra, G. (2026). *State of the art on the Legal Agent Benchmark*. X (Twitter). https://x.com/gabepereyra/status/2059320727988224128↑

1. Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., & Khattab, O. (2025). *GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning*. https://arxiv.org/abs/2507.19457↑

2. Fan, Y., Ni, J., Merane, J., Hermstrüwer, Y., Huang, Y., Akhtar, M., Salimbeni, E., Leippold, M., Sachan, M., Stremitzer, A., Engel, C., Ash, E., & Niklaus, J. (2025). *LEXam: Benchmarking Legal Reasoning on 340 Law Exams*. https://arxiv.org/abs/2505.12864↑

3. Karpathy, A. (2026). *autoresearch: AI agents running research on single-GPU nanochat training automatically*. GitHub repository. https://github.com/karpathy/autoresearch↑

4. Lange, R. T., Imajuku, Y., & Cetin, E. (2025). *ShinkaEvolve: Towards Open-Ended and Sample-Efficient Program Evolution*. https://arxiv.org/abs/2509.19349↑ back: 1, 2

5. Lee, Y., Nair, R., Zhang, Q., Lee, K., Khattab, O., & Finn, C. (2026). *Meta-Harness: End-to-End Optimization of Model Harnesses*. https://arxiv.org/abs/2603.28052↑

6. Louapre, D. (2026). *PhysicsIntern: from an Autonomous Benchmark-runner to a Research Sidekick*. Hugging Face blog. https://huggingface.co/blog/dlouapre/physics-intern↑

7. Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., & Balog, M. (2025). *AlphaEvolve: A coding agent for scientific and algorithmic discovery*. https://arxiv.org/abs/2506.13131↑ back: 1, 2

8. Phan, L., & others. (2025). *Humanity's Last Exam*. https://arxiv.org/abs/2501.14249↑

9. Shi, Y., Liu, H., Hu, Y., & others. (2026). *PLawBench: A Rubric-Based Benchmark for Evaluating LLMs in Real-World Legal Practice*. https://arxiv.org/abs/2601.16669↑

10. Zhang, J., Hu, S., Lu, C., Lange, R. T., & Clune, J. (2025). *Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents*. https://arxiv.org/abs/2505.22954↑ back: 1, 2

11. Zhu, M., Tian, M., & others. (2025). *Probing the Critical Point (CritPt) of AI Reasoning: a Frontier Physics Research Benchmark*. https://arxiv.org/abs/2509.26574↑

Made with ❤️ with research article template