PROFORMA GLOBAL RESEARCH

Research Posture: Why We Let the Model Choose

Matt Rollings, Founder and Principal, Proforma Global

Whitepaper · 2026-05-18

Executive summary. We make every decision inside training the same way: a reinforcement-learning policy decides it, not a human picking a value. Every hyperparameter, every threshold, every gate, every architectural commitment. The numbers a person picks are there to bootstrap and nothing more. Once a decision class matures in our stack, it is a set of scalars learned by the same reward signal that trains the model itself.

This paper is about the discipline that follows from that one commitment: the Fire-Aim-Fire iteration loop, the post-step retrospective, the append-only findings ledger, and a research culture that treats negative results as first-class work.

The thesis has been tested. The foundational experiment, documented in the next paper, attached a small reinforcement-learning policy (the *REINFORCE Sidecar*) to the forward-pass control surface of a standard training loop, and it beat an otherwise-identical hand-tuned baseline by 42% on the primary held-out evaluation at matched compute. That number anchors everything else in the series: the Sidecar paradigm extended to more control surfaces, the same primitive pushed upstream into the model's own architecture, and a forensic framework that reads what each step of the progression did to the body of the trained model. The three technical papers that follow document those extensions. This one explains why they look the way they do.

1. The thesis: hyperparameters are a tax on the data

Modern transformer training is held together by hundreds of numbers a human chose: the learning rate, the weight decay, the gradient clip, the attention head count, the hidden width, the temperature on every softmax, the epsilon in every Adam denominator, the clamp on every saturating function. Every one of them encodes a prior, a guess about what regime training will land in and how it should respond.

That guess is wrong almost everywhere. A learning rate sized for early training runs too hot once the loss landscape sharpens. Clip the gradient tight enough to suppress an exploding step and you have clipped it tight enough to throttle a healthy one. A hidden width held uniform across depth asserts that the data has no preferred depth structure, which is false for every domain we have measured.

The standard response is a hyperparameter search: run a sweep, pick the best, lock it in. A sweep delivers one static configuration, optimized for an average over the whole run, and training is never stationary. The configuration that produces the best final checkpoint is rarely the one you would have wanted at step 1, or step 1,000, or step 100,000. A scalar trained against the same reward signal that trains the weights does not have that problem.

Our thesis is therefore simple and absolute: **every hyperparameter, threshold, gate, and routing decision should be learned, not hand-picked.** Human-picked values are bootstrap only. Once the path exists for the model's own reward signal to flow into a decision, the human's number is replaced by a learnable scalar, and the human's authority over that knob is permanently transferred to the model.

This is the central engineering commitment of the research program rather than a methodological flourish, and everything downstream of it (what to build, what to instrument, what to publish, what to revisit) follows from it. The commitment has been tested. The next paper documents the foundational result: a learned policy on the forward-pass control surface beat a hand-tuned baseline by 42% at matched compute, and that number is what licenses extending the same paradigm to more control surfaces, more reward signals, and eventually the model's architecture itself.

2. Knob-ownership migration

The posture is easiest to see in the trajectory of a single project's hyperparameters over its lifetime. Humans chose everything at the start: the global learning rate, weight decay, layer count, hidden width, head count, gradient clip, every cap and threshold. The active stack now learns the following per (layer, tensor) through reward-driven updates:

- per-layer learning rate scales (each weight matrix's effective LR rides a REINFORCE-driven gain)
- per-tensor weight decay
- per-layer hidden width and intermediate width (mutated mid-training by a Gaussian policy with learnable mean and learnable variance per axis)
- backward-pass behavior: gradient skip schedules, truncation depths, and projection directions

The next set is wired and still under iteration. The plumbing exists, but the policies have not yet beaten their hand-set defaults:

- per-layer attention head count and head dimension (extension of the architectural-mutation policy class to head axes: the first attempt failed because the reward signal could not see the higher recovery cost of head-axis surgery. A per-axis reward window is in flight)
- the exploration magnitude on the architectural-mutation policies (sigma is learned per axis alongside mu. An analogous sigma-learnability path is being added to the older sidecar policies, where sigma is still a hand-set constant in several places)

The reverse direction, taking a knob away from the model and giving it back to a human, has happened zero times. Once a parameter becomes learnable, it stays learnable. The trajectory is one-way.

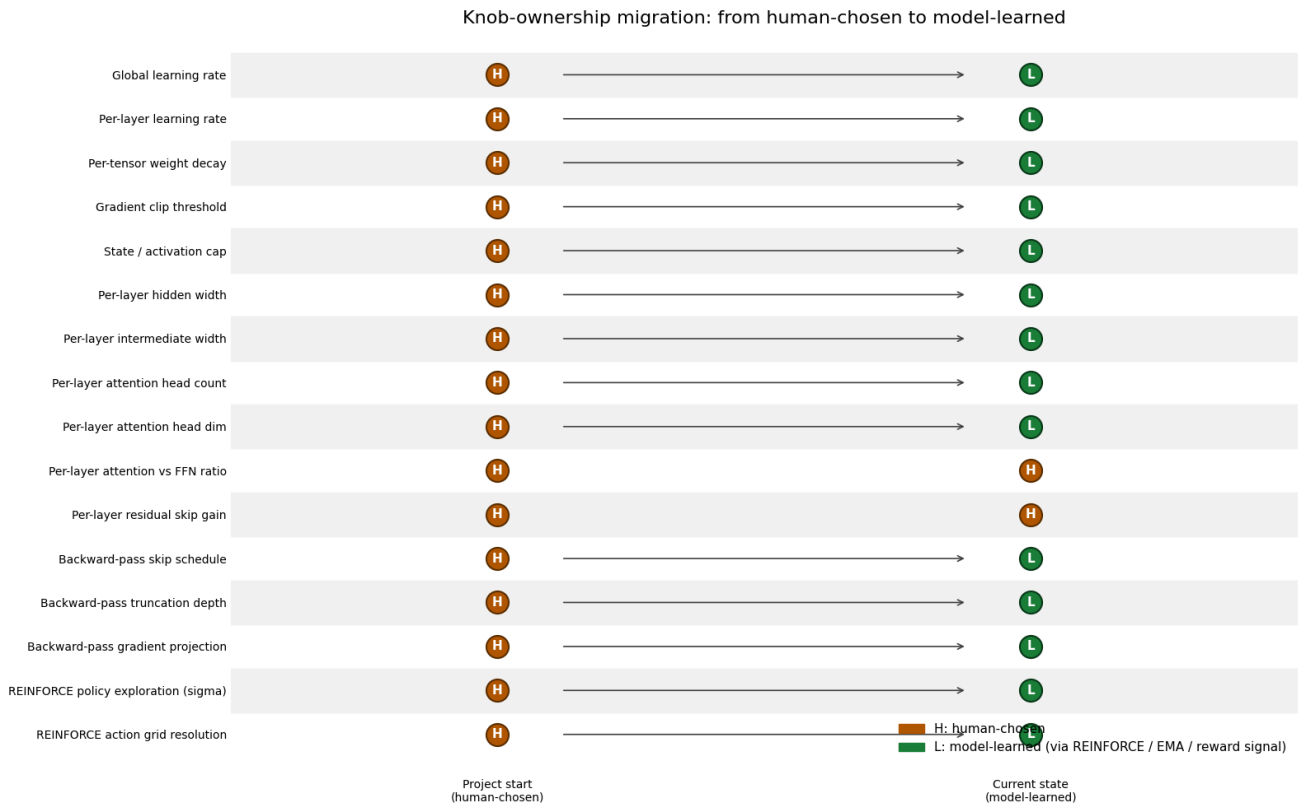

Figure 1. Knob-ownership migration over the project's lifetime. The arrows are the work.

The chart understates the change. The "global learning rate" row, for instance, does not capture that the current stack carries roughly one learnable LR scalar per (layer, weight matrix). A single migration row stands in for hundreds of scalars whose values the data now sets, not us. The human-chosen rows that remain are not settled preferences but the next migrations on the queue, sequenced by which reward path is cheapest to build.

3. Expose plumbing, never decide for the model

One rule drives every migration: **expose plumbing, never decide for the model.** When we add a feature, we add the path by which the model can express a decision. We don't encode the decision ourselves. The discipline runs in two directions.

The first direction is replacement. When we encounter a hand-picked number in existing code (a clamp, a blend weight, a temperature), we ask whether the model has any way to change it based on experience. If the answer is no, we wire one. The number becomes a learnable scalar, the reward signal turns into its update rule, and the value we started with is now just an initializer. The migration is permanent.

The second direction is construction. When we build something new, we expose ideas (observables, optional routing branches, gates that can be opened or closed) rather than prescribing what should happen with them. A canonical example: rather than fixing a state-norm cap at a value chosen to "feel safe", we expose a learnable threshold per layer, initialize it dormant (high enough that healthy training never triggers it), and wire a reward-driven path that pulls it down only if the model decides the post-projection state is more useful than the pre-projection state. The cap exists. Its value is the model's to choose.

The dormant-by-default pattern is load-bearing, so it earns its own treatment. We once added a cap pre-tuned by hand to a value we thought was "tight enough to catch the explosion we feared." A controlled comparison against the same code with the cap initialized at 50× that value (effectively disabled) showed the pre-tuned version was 14% worse on the primary training metric at the same compute budget. The cap was impeding training before any explosion occurred. A hand-picked cap value is the same mistake as any hand-picked threshold: the mechanism is useful, but the value should be initialized so it does not bind, and a reward path should own pulling it down.

The same logic applies to every gate we have added. Initialize a gate at the value the researcher believes is correct and you have already chosen its value. Initialize it dormant with a reward path that can pull it open, and the model gets to tell you whether the gate should exist at all.

4. Fire -> Aim -> Fire

None of this migration is free. Every new learnable scalar needs three things to hold: the reward signal must actually reach it through a gradient path that doesn't degenerate, the policy's own hyperparameters need to be sane enough not to collapse exploration, and the telemetry has to

let us confirm the lever is doing what we think it is. That is where the iteration discipline comes in.

We call the loop Fire-Aim-Fire, and it is scoped more carefully than the name suggests. It is not a build-order rule, and it does not mean "finish one feature before you start the next." Several features can be in flight at once. What the loop governs is the obligation that attaches to a feature once it is running: **before you call it useful, you have to instrument it deeply enough that the data can render a verdict.**

The three phases:

1. **Fire.** Build a minimum viable version of the feature on a fresh fork. Don't gold-plate it or pre-optimize. The goal is to get the path running end-to-end so the reward signal can start moving against it.
2. **Aim.** Add the observability the model needs to tell us whether the feature is garbage or gold. This is the phase that gets cut under time pressure, and we keep explicit standing rules against cutting it. A feature that goes out without the telemetry to verify it hasn't really been delivered, only deployed.
3. **Fire (again).** Iterate on what the telemetry says: keep it, kill it, or tweak it. A feature that survives the loop has been *proven* useful against a measurement we trust. One that doesn't enters the findings ledger as a constrained region of the search space.

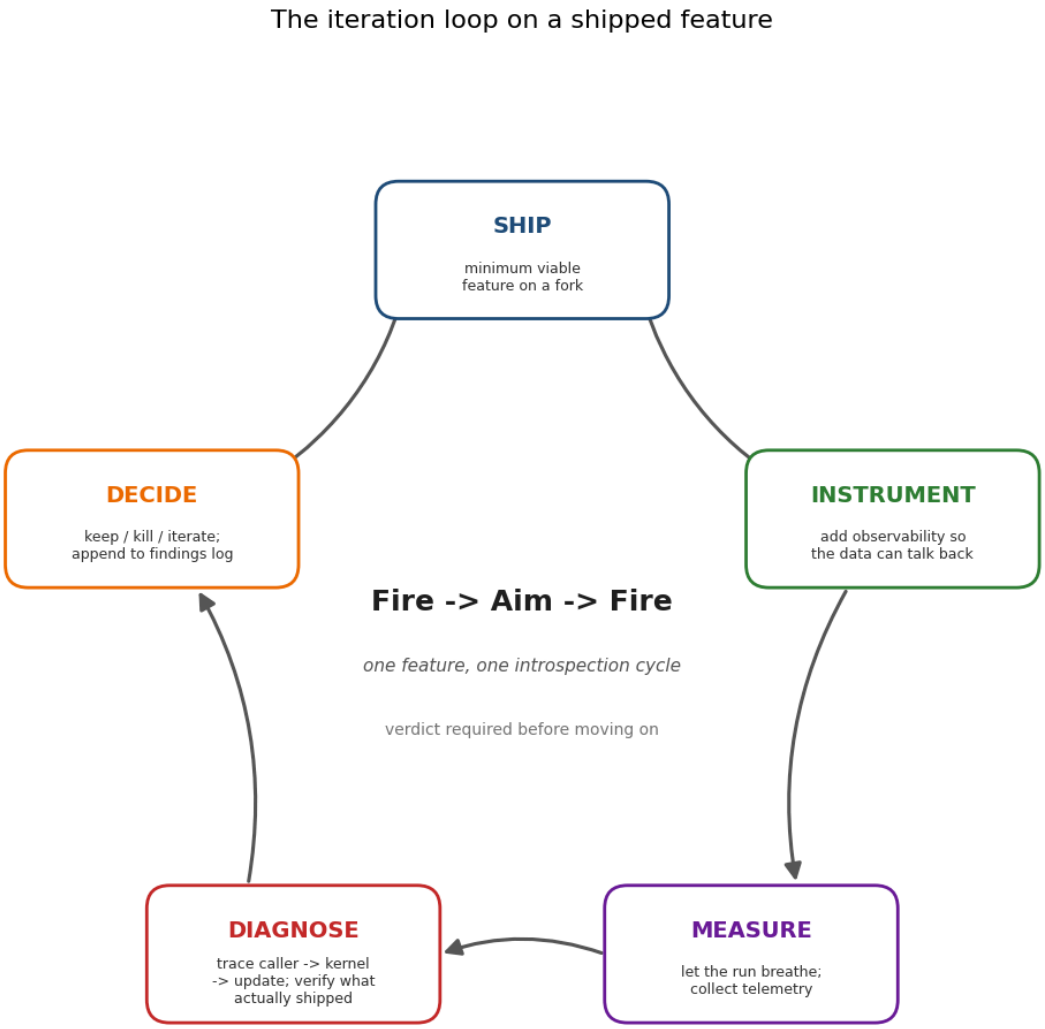

Figure 2. The iteration cycle on a single feature. Several features run the cycle in parallel. What matters is that each one completes it.

Two consequences flow from this scoping. First, "we built X" and "we verified X works" are different claims, and we are ruthless about not eliding them. A prior release delivered a multi-lever feature whose deploy header confidently announced four new control paths. A source-level audit later found that two of the four wrappers were aliases of one underlying function, one of the calls was a stub, and the net effect was that gradient buffers were being multiplicatively passed through the same scale four times per step. The metric regression was real, and it lasted across several later releases, each of which trusted the original header and

chased the wrong cause. The rule that came out of it: before you call a multi-lever feature working, trace the full caller -> dispatch -> kernel -> update path in code and show the trace. Deploy headers lie, so we read the source.

Second, the introspection phase is where most of the work happens. The model's training process is the diagnostic instrument, and our job is to expose enough of its state to read it. When a feature looks like it isn't working, the first hypothesis is usually that we aren't measuring the right thing yet, not that the feature is bad. A feature with no observability can't be evaluated, and you can't responsibly kill what you haven't evaluated. Both verdicts, kept and killed, cost the same instrumentation budget.

5. The post-step retrospective

The iteration loop is supported by a smaller, faster discipline applied after every meaningful step of work. Three questions, every time:

1. **Did this step actually work the way we wanted?** Compare what happened to what was predicted. If they diverge, the divergence is the most valuable data of the step. Do not paper over it.
2. **Are we using the proper mechanism, or routing around it?** When a fix or a new feature requires bypassing the project's existing infrastructure ("the proper mechanism is broken so let me edit in place"), the bypass is almost always the wrong move. The proper mechanism is load-bearing for reasons that may not be visible from the current task. If the proper mechanism is broken, fix the proper mechanism. Do not establish a precedent for bypassing it.
3. **Is the next action actually necessary, or are we manufacturing work?** A great deal of researcher activity is defensive completeness: running one more analysis, dumping one more CSV, building one more heatmap to "see what it shows." Write down the decision an analysis serves before you run it. If two outcomes of the analysis would lead to the same architectural change, the analysis is not actionable, and the time is better spent on a probe that does branch behavior.

The third question deserves an aside, because it cuts against a common research instinct. We once proposed a tensor-level attribution analysis at fine granularity: twelve numbers per layer across twenty-four layers. It was technically possible and would have produced a striking heatmap, and we killed it before running it, because the architectural insertions it was meant to inform were already settled by a much coarser grouping (attention versus feed-forward), and the fine-grained version would have led to the same decision either way. The right aggregation

level for an analysis is the one at which the answer changes the next action. A 288-cell heatmap that resolves to "same as the three-bucket version" is a credibility risk: it suggests more rigor than it delivers.

6. The findings ledger and the memory system

A research program of this kind generates lessons faster than any single researcher can hold them. Worse, lessons learned in one session evaporate by the next unless they are written down in a form the next session will actually read. We treat this as an engineering problem and solve it with two layered artifacts.

The first is an **append-only findings ledger**. Every material observation from a training run (a confirmed bug, an unexpected convergence pattern, a discovered structural commitment by the model, a hypothesis that survived or died) gets a dated entry. The format is strict: a timestamp, a short headline, and sections for what was confirmed broken, what was confirmed working, what surprised us, and what the action items are. Nothing is overwritten. If a later finding revises an earlier one, the revision is a new dated entry that cites the prior one, not an edit to the old one. The ledger's job is to make it impossible for a future session to re-derive the same lesson from the same logs and call it a discovery.

The second is the **promoted memory file**. A finding that survives a few iterations and is durable enough to function as a design rule (not merely a data point about a specific run) is promoted to a named memory file that loads on every session. These files are short, prescriptive, and dated. They carry incident citations: the specific failure that justified the rule, so a future session that is tempted to violate the rule can see exactly which historical mistake it would be recreating.

The combination is what matters. The ledger gives the memory files their provenance, and the memory files keep the ledger from becoming a graveyard of lessons no future session reads. Together they are the institutional memory of the project, and they are why these lessons compound instead of dissipating.

7. Honest failure analysis is the strongest credibility signal

Most ML research labs publish wins. Negative results are written up reluctantly, if at all, and are typically buried inside an appendix. This is methodologically backwards. **Negative results constrain the search space.** A confirmed dead end is, from the perspective of any subsequent

researcher, more valuable than yet another paper claiming a modest gain on a contested benchmark: the dead end is a region that no longer needs visiting, and the constraint compounds across the field.

We treat negative results as first-class outputs. Three illustrative examples from this project, paraphrased to respect the disclosure boundary:

Hand-picked variable-width architectures lose to uniform at matched parameter counts. A multi-week run of experiments built variable-width transformer specs, narrowing or widening the hidden dimension across depth on several plausible templates, and compared them against a uniform-width baseline at the same total parameter count and the same compute. The variable-width specs lost on both convergence speed and per-step throughput. The throughput loss had a specific cause: tensor-core alignment penalties on non-standard dimensions. The convergence loss had no single cause and did not come back with tuning. The conclusion was not that uniform is better full stop, but that uniform beats any hand-picked profile we tried, which means the next thing to try is letting the model pick the profile. That next experiment, run with a learned per-layer width policy, converged to a roughly 3.6× sawtooth profile no human spec had proposed, and it *also* underperformed the uniform baseline at matched compute. The forensics on *why* it underperformed turned out to be the real value of the experiment: mid-training architectural surgery was lobotomizing the optimizer's hidden state, and we would not have known to look without the negative result. The methodology and the diagnostic are the positive result, not the profile the policy found. Both failures, the hand-picked specs and the first learned policy, are part of one finding, and publishing a sanitized version of either would misrepresent the trajectory.

Hardcoded exploration magnitude in a reward-driven policy is an anti-pattern. An early REINFORCE-style policy ran with a fixed exploration parameter: the standard deviation of the Gaussian noise added to each action. We picked the value by hand at what seemed a reasonable order of magnitude. It was too aggressive for the signal regime the policy was working in, the policy could not compensate because the standard deviation was not on its gradient, and it never converged to a useful behavior. The fix was to make both the action mean *and* the action standard deviation learnable per granular unit, using the standard REINFORCE gradient expressions for each. The deeper lesson now sits in a memory file: a reward-driven policy has to learn *every* hyperparameter that materially affects its behavior, not just its action mean. The exploration magnitude is the one most commonly missed, and we now treat it as non-negotiable.

Aggressive default caps impede early training and should be initialized dormant. Section 3 already covered this one. The negative result quantified the cost at 14% worse on the primary metric at the same compute, and the design rule that followed ("initialize caps so they don't fire

during healthy training, and let observation and reward pull them down when needed") is now a load-bearing convention on every cap-style mechanism in the project.

Each of these is a published-as-internal-memory loss that compounds into a constraint on what we will and will not try next. A researcher who joins this project inherits not just the current code but a map of the regions already searched and ruled out. That map is, in our view, the most underrated artifact of any research program.

8. What this posture costs and what it buys

We owe a frank accounting of the tradeoffs. Letting the model own a knob is more expensive than picking a value. The reward path has to be wired, the policy needs its own learnable hyperparameters, the telemetry has to be instrumented, and the variance in early training is higher than a tuned baseline because the policy is exploring rather than executing. A research culture optimized for fast headline numbers would not make these choices.

The compensation is that every learned knob is mostly self-solving from there on. We tune it less, and what tuning remains is a different kind of work: it moves from picking values to picking initializations, action spaces, and reward functions. The reward signal absorbs regime shifts as training progresses, and the same code path adapts to a different model size, a different dataset, or a different compute budget with no re-sweep. A hand-tuned hyperparameter is a recurring liability, while a learned one is an engineering investment you make once and revisit only occasionally.

The compounding is the point. Each migration of a knob from human to model frees the researcher to work on the next migration. The architectural changes documented in the technical papers that follow (dynamic per-layer width and head-count mutation, reward-driven backward-pass control, cross-layer routing structures discovered rather than specified) are reachable only because the cumulative effect of prior migrations has freed enough researcher attention to build them.

9. How to read the technical papers

The three technical papers that follow describe specific systems built under the posture documented here. They form a deliberate progression rather than three parallel threads, and the order matters.

Paper 2: ***Reward-Driven Training Control*** documents the foundational result: the REINFORCE Sidecar paradigm, the +42% over vanilla at matched compute, the constraints on that result (forward-pass-only, scoped control-channel set, single reward signal), and a worked example of an additional control channel (per-(layer, tensor) learned weight decay) that demonstrates the kind of incremental gain that becomes available once the paradigm is in place. Read it first if you are evaluating the technical credibility of the research program. The 42% number is the entry-level validation of the entire research line.

Paper 3: ***Self-Discovering Architectures*** takes the Sidecar paradigm one level up the stack: instead of learning the dials inside the training loop, it applies the same primitive (Gaussian REINFORCE with learnable variance) to the model's own architecture. The machinery works and the diagnostic methodology is mature, but the headline competitive result is not yet in hand, because mid-training architectural surgery exposes an optimizer-state coupling the training-control work never hit. We document that honestly, because the methodology is the durable output and the competitive number will follow it.

Paper 4: ***Emergent Layer Roles and Functional Specialization*** is the forensic capstone. Each rung of the progression (vanilla baseline, Sidecar with foundational channels, Sidecar with additional channels, architectural mutation) changes the body of the trained model in measurable ways. This paper develops the methodology for reading those changes. The interpretability framework is general. It applies to any decoder-only transformer and requires nothing beyond a finished checkpoint and a brief gradient trace.

The technical papers will be terser than this one, because they assume the posture documented here:

- They won't justify, lever by lever, why the lever is learned rather than chosen. The default is learned.
- Negative results get no apology. Where an approach was ruled out, the writeup names what was ruled out, on what evidence, and what was tried next.
- No feature is claimed to work without showing the path by which the claim was verified. Deploy headers do not count as verification.
- Each one is specific about what the model chose. The most interesting result in a given system is rarely the headline metric. It is the configuration the model converged to under reward pressure, which is reliably one no human researcher would have proposed.

The takeaway here is not that "Proforma uses learned hyperparameters," but that we treat every hand-picked number as a temporary scaffold around a missing reward path, and that the whole research program is built around removing those scaffolds one at a time, deliberately, with the discipline to verify each removal and the memory to keep the lessons.

We let the model choose, and our job is to build the paths that make choosing possible. The rest is bookkeeping.

DOWNLOAD PDF

Work with Proforma Global

The thinking in this paper is public; the methods that turn it into a working system are not. If it fits a problem your team is working on, that is what we bring to an engagement. Start a conversation: **info@proforma.global**.