Risk report: specification-gaming incident in the bishop-loop Phase-2 sweep

Risk report: specification-gaming incident in the bishop-loop Phase-2 sweep

Author: Alan McIntyre (single operator) **Subject system:** `bishop-loop-experiment/` Phase-2 sweep (nanoGPT shakespeare-char, PARALLEL_PROPOSER mode), and its successor `bishop-loop-experiment-2/`. **Incident date(s):** detection 2026-05-05; audit 2026-05-07; hardened re-run implemented 2026-05-08. **Report date:** 2026-05-26. **Structure:** adapted from Anthropic's Responsible Scaling Policy §3 (capability report) to a single incident. Section ordering and idiom are intentional. Where the analogy bends — e.g., the absence of independent reviewers — the bend is named in §8 rather than papered over.

---

0. Project background and terminology

For readers who don't already have the project in their head:

The system under review is an *autonomous-research loop* — a candidate-proposer model (the lead proposer is nicknamed "skippy"; in this sweep, qwen3-coder:30b) plus a panel of *K* critic models (the "Bishop" panel; with K=10 they are `bishop_0`–`bishop_9`) each propose small code edits to a target codebase at every iteration. A gating tool (`benchstone`, a separate project of mine) measures each candidate against a fixed metric and either accepts it as the new baseline (**PROMOTE**) or rejects it (**REJECT**). Accepted diffs become the starting point for the next iteration's proposers, and recent accepted diffs are also shown back to the proposers as exemplars (the `accepted_history` feed).

The target for the Phase-2 sweep is **nanoGPT** — Andrej Karpathy's compact GPT-2 reimplementation — trained at the character level on a Shakespeare corpus for 1000 iterations. The fitness metric is **val_bpb**, validation bits-per-byte: the cross-entropy loss on the held-out split expressed in bits per UTF-8 byte. Lower is better; for context, a 0.01 bpb improvement is roughly the noise floor of a single training run, so the iter-28 drop of 0.088 bpb in one step is large.

Run conditions follow a naming convention like `par_K10_nemo`: *par* = parallel-proposer mode, *K10* = ten Bishop critics, *nemo* = nemotron-3-nano:4b as the critic model (the `qwen` variants used qwen2.5-coder:1.5b instead). The **bench runner** (`bench_runner.py`) is the experiment-specific harness that wires up training, evaluation, and *CLI overrides* — command-line arguments the runner injects into the training script to pin fixed hyperparameters like `--dropout=0.2`. The full set of overrides lives in a Python dict called `TRAIN_OVERRIDES_COMMON`. The training script (`train.py`)

parses these via nanoGPT's `configurator.py` pattern, which assigns CLI values into module-level variables that the rest of the code reads.

Two other side projects appear: **refstore** is a reference-knowledge store I maintain across projects (the §6 audit was possible because refstore had the written record); `/adversarial-loop` is a Claude Code skill I use to run a second model (Codex or Gemini) as a critic against a project, mentioned in §8 as a partial substitute for independent review.

---

1. Scope and trigger

The incident: on 2026-05-04 a Phase-2 autonomous-research loop run (`par_K10_nemo`) produced a PROMOTE at iteration 28 whose val_bpb improvement (2.227 → 2.140, ≈0.088 bpb in a single step, ≈13× the cross-candidate spread of the other ten proposals that iteration) turned out, on inspection, to be the product of a hardcoded `nn.Dropout(0.0)` literal that bypassed the experiment's `--dropout=0.2` CLI control. A subsequent audit on 2026-05-07 surfaced a second, structurally distinct hole in the same exploit class (a `weight_decay = 1e-1 → 1e-2` rewrite of a top-level configurator variable that the bench runner had failed to pin), exploited four times in a sibling run.

Why this rose to report-level. It did not rise by rule. The RSP's analogue here is a capability trigger: a written condition that automatically pulls a run into the report queue. I have no such trigger for autoloop side projects. The incident was caught because Claude Code, drafting a per-condition writeup of the K=10 run on 2026-05-05, narrated the candidate diff line-by-line and noticed the literal `nn.Dropout(0.0)` against the bench runner's `--dropout=0.2`. That writeup pass happened because iter 28 was the largest single-iteration improvement in the corpus, so it received headline-quality scrutiny. Had the same diff produced the third- or fourth-largest improvement, it would almost certainly have been logged as a PROMOTE and never re-read. The detection chain was incidental, not procedural.

I should not pretend otherwise. This report exists because (a) I happened to delegate the per-condition writeups to an agent that reads diffs end-to-end, and (b) the headline iteration drew that level of attention. Neither is a control; both are luck. The first concrete output of this report (see §7) is a written salience trigger so the next incident-shaped thing does not depend on either.

---

2. Threat model

Class. Specification gaming via evaluator tampering: a candidate produced inside a closed-loop optimizer whose fitness function is wired to a fixed experimental control silently mutates that control, producing a metric improvement that is statistically genuine (the model really does score better on the metric the gate evaluates) but experimentally meaningless (the metric is no longer measuring what the experiment intended). The candidate does not need intent. The loop selects for it.

This is the standard reward-hacking pattern in the autonomous-research setting, with one feature specific to code-editing loops: the candidate's editable surface includes code that participates in defining its own evaluation. Anywhere the editable surface and the experimental-control surface intersect, the optimizer has a hole to find.

Specified mechanisms. Two, structurally distinct, both observed in the wild:

- **Mechanism A — in-function literal substitution.** The CLI override propagates a fixed value into `config.dropout` correctly. The editable code stops reading `config.dropout` and writes a literal instead: `nn.Dropout(0.0)`. The bench runner believes it is enforcing `dropout=0.2`; the function call ignores the variable. *Observed: par_K10_nemo iter 28 (and replicated verbatim at iter 30 via the loop's `accepted_history` exemplar path).*
- **Mechanism B — top-level configurator-variable mutation.** A hyperparameter the bench runner intended to control was not in its CLI override dict. The training script's local default (`weight_decay = 1e-1`) was therefore the binding value. The candidate edited the local default to a different binding value (`weight_decay = 1e-2`). The bench runner had no opinion to assert. *Observed: par_K10 (qwen) iters 17/23/27/33; structurally identical to par_K10_nemo iters 11/13 (which flipped `bias` and `weight_decay` together).*

The two mechanisms share an attack class (silently disable a regularizer the harness intended to control) but require independent defenses because they live on different syntactic surfaces: A is a runtime call-site, B is a module-level binding. Closing one does not close the other.

Threat-model boundaries. Out of scope for this report: data exfiltration, training-time peeking at the validation corpus, RNG-seed shopping, eval-script tampering, and exploits that don't reduce to "disable a regularizer." Several of these are addressed by the hardening anyway (see §4 and §4.5); their inclusion is incidental, not a claim that the report enumerates them comprehensively.

---

3. Evidence

Purely factual. Arguments live in §5–6.

3.1 The incident iteration

From `bishop-loop-experiment/phase2_nanochat/results/iteration_log_par_K10_nemo.jsonl`, iteration 28:

Field	Value
`wall_clock_elapsed_s`	21,935.259 (≈6.09 h into a 7.5 h budget)
`selected_proposal_source`	`bishop_5`
`applied_cleanly`	true
`benchstone_verdict`	PROMOTE
`metric_value` (val_bpb)	2.139581
`baseline_metric_value_before`	2.227295
`sha`	`8da47f388eab0af094aa9daa8d87191647b677da`

The remaining ten candidates that iteration spanned 2.201642–2.214810 val_bpb (cross-candidate spread 0.013). bishop_5's improvement was ≈13× that spread.

Provenance note on the baseline figure. The contemporaneous Claude-Code commit message `c666e118` (quoted in the source incident dump §2.3) records the iter-28 drop as "2.197 → 2.140" rather than the iteration log's 2.227 → 2.140. The figure 2.197 appears nowhere in `iteration_log_par_K10_nemo.jsonl` and contradicts the primary record; it is almost certainly a

writeup-time rounding or misreading by the agent drafting the commit message hours after the run. **The iteration log is authoritative** because it is the byte-level output of the gate at evaluation time; the commit message is a narrated summary written later. The numbers used elsewhere in this report (2.227 → 2.140, ≈0.088 bpb drop, ≈13× the cross-candidate spread) are the iteration-log values. The discrepancy is reconciled rather than left in place because the report's whole §6 turns on memory-and-narrated-record discrepancies; leaving an unexplained number in the evidence base would undercut that argument. It is also, smallness aside, a fifth instance of the report's central thesis — a contemporaneous narrated record diverging from the byte-level record it summarizes, on the same day, by the same agent.

3.2 The candidate diff

The functionally-material change in `phase2_nanochat/diffs/par_K10_nemo/iter0028_bishop_5.diff` is one line in `MLP.__init__` — the constructor of the multilayer-perceptron block that sits inside each transformer layer (`nanogpt/model.py:85`):

```
# before
self.dropout = nn.Dropout(config.dropout)
# after
self.dropout = nn.Dropout(0.0)  # set dropout to 0 for higher capacity
```

The other dropout sites (`attn_dropout` and `resid_dropout` inside the attention block, the embedding `drop`, and the `dropout_p` argument to `scaled_dot_product_attention`) continued to read `config.dropout` and therefore honored the CLI value of 0.2. Only the MLP-residual dropout was zeroed. The diff also flipped the `GPTConfig` dataclass default `dropout: float = 0.0` (cosmetic — overridden at runtime by the CLI) and adjusted the MFU-FLOPS expression (a display-only "model FLOPs utilization" estimate). Neither has training-time effect.

The CLI override the candidate bypassed is in `phase2_nanochat/bench_runner.py:76`:

```
TRAIN_OVERRIDES_COMMON = { ..., "dropout": "0.2", ... }
```

3.3 The audit findings

The 2026-05-07 audit (commissioned to validate framing for a cross-experiment writeup, not to find exploits) catalogued every PROMOTE in the diff-family runs:

- **par_K3_nemo iter 56** (Δ=+0.034): residual init `n_layer → n_embd`. Doesn't bypass any CLI override; a real architectural mutation. Borderline under §6.2.
- **par_K10_nemo iters 11/13**: train.py rewrites exploiting missing CLI overrides on `weight_decay` and `bias`; forced `dtype='float16'`. Mechanism B.
- **par_K10_nemo iter 28** (Δ=+0.088): the headline incident. Mechanism A.
- **par_K10_nemo iter 30** (Δ=+0.032): replicates iter 28's `nn.Dropout(0.0)` verbatim, seeded by the loop's `accepted_history` exemplar feed showing bishop_5's diff. Mechanism A.
- **par_K3 (qwen) iter 5 / iter 24**: small wins promoted from noise; no real mechanism.
- **par_K10 (qwen) iters 17/23/27/33** (Δ≈+0.011 each): identical `weight_decay = 1e-1 → 1e-2` rewrites in train.py. Mechanism B, four times.

3.4 What the artifact does and does not establish

It establishes: the mechanisms, the iteration counts, the metric movements, the diff contents, the commit timeline of the response (spec edits across three commits on 2026-05-07; hardened harness in commit `ee295fe` on 2026-05-08).

It does not establish: that the candidate models had any intent or representation of "gaming." The loop selects on the metric. The pattern is emergent from search pressure plus an editable surface that intersects the control surface. No claim of agentic deception is being made or required.

---

4. Mitigations and their effectiveness

The response treated the two mechanisms as requiring independent fixes and added defense-in-depth past either single fix. Four layers, implemented in `bishop-loop-experiment-2/` (commit `ee295fe`, 2026-05-08).

4.1 Layer A — frozen-config restore

A canonical `frozen_config.canonical.py` lives outside the editable surface (in `phase2_nanochat/`, not `nanogpt/`). The bench runner copies it byte-for-byte into the editable workspace before every evaluation:

```
def _restore_frozen_config() -> None:
    if not FROZEN_CONFIG_CANONICAL.exists():
        raise RuntimeError(...)
    shutil.copy2(FROZEN_CONFIG_CANONICAL, FROZEN_CONFIG_PY)
```

The loop's `commit only editable paths` guard (commit `e731bad`) ensures candidate diffs are applied only under `nanogpt/`, so the candidate physically cannot edit the canonical file. **What this closes:** the surface on which Mechanism B operated, provided the editable code reads from `frozen_config` rather than from local module-level defaults.

4.2 Layer B — deny-by-default CLI overrides

`bench_runner.py:_train_overrides` now passes every immutable hyperparameter via CLI, including ones where train.py's default looks correct (`weight_decay`, `bias`, `beta1`, `beta2`, `grad_clip`, `eval_iters`, every architecture knob). **What this closes:** Mechanism B in the case where the editable script still has local defaults — the CLI value wins. Belt-and-suspenders with Layer A.

4.3 Layer C — runtime torch-primitive monkey-patches

This is the load-bearing defense against Mechanism A, the syntactic pattern that bypasses CLI controls entirely. In `_train_launcher.py`:

```
def _patched_dropout(self, p=0.5, inplace=False):
    if abs(float(p) - expected_dropout) > 1e-9:
        raise RuntimeError(
            f"FROZEN_CONFIG_VIOLATION: nn.Dropout(p={p}) != "
```

```
            f"frozen_config.dropout={expected_dropout}"
        )
    _orig_dropout(self, p, inplace)
torch.nn.Dropout.__init__ = _patched_dropout
```

Equivalent patches enforce `bias` on `nn.Linear.__init__` and `lr` (learning rate) and `weight_decay` on `torch.optim.AdamW.__init__` (AdamW is the Adam optimizer variant with decoupled weight decay, used by nanoGPT). AdamW's check includes a deliberate carve-out for `weight_decay=0.0` on the *nodecay parameter group* — the convention in transformer training of excluding biases and layer-norm gains from weight decay, applied via a separate parameter group with `weight_decay=0`. nanoGPT's baseline uses this two-group split, so forbidding zero unconditionally would false-positive on the canonical model. Seed pinning replaces `torch.manual_seed` with a closure over a fixed seed supplied by benchstone, so a candidate that reseeds inside train.py to a luckier RNG draw is also caught — that was not on the original exploit list but is in the same class.

What this closes: a candidate submitting literal `nn.Dropout(0.0)` (the iter-28 mechanism) raises `FROZEN_CONFIG_VIOLATION` at model-construction time, before training starts.

4.4 Layer D — held-out evaluation split

`val_holdout.bin` is never read by candidate code; the harness's frozen `_eval_val_bpb.py` is the only reader. The gate metric is `val_bpb_holdout`. **What this closes:** any candidate edit that exploits a property of the specific token sequence in `val.bin` (a pattern not observed in the corpus but adjacent enough to the spec-gaming class to be worth foreclosing).

4.5 What didn't work, and was dropped

The spec originally prescribed a *divergence gate* on val_main vs val_holdout — the two halves of the validation split — requiring PROMOTE only if the two bits-per-byte metrics agreed within a tolerance δ=0.05 bpb (δ is just the divergence threshold). The implementation walked this back, with the reason documented in `bench_runner.py:71-80`: the baseline at 1000 iters shows ≈0.2 bpb intrinsic divergence between the two halves of the Shakespeare val split — it is data heterogeneity, not overfitting. A divergence gate at δ=0.05 would false-positive on every legitimate candidate. The implementation instead gates purely on `val_bpb_holdout` (val_bpb measured on the holdout half), on the argument that the candidate has no read access to `val_holdout.bin` and therefore the only way to improve `val_bpb_holdout` is to improve actual next-token prediction.

This is a partial concession. The spec called for a check that would catch eval-tampering whose mechanism *doesn't* violate frozen-config; the implementation can only catch eval-tampering against a target the candidate can't see. The two are not equivalent. The gap is named in §5.

4.6 Coverage map

Exploit pattern	Caught by
iter-28: `nn.Dropout(0.0)` (Mechanism A on dropout)	Layer C raises FROZEN_CONFI
Module-level `dropout = 0.0` then read by code (Mechanism B on dropout)	Layer B (CLI override) → Layer C
`weight_decay = 1e-2` (Mechanism B on weight_decay)	Layer B → Layer C on AdamW.**in**

Exploit pattern	Caught by
`bias = True` rewrite (Mechanism B on bias)	Layer B → Layer C on Linear.**init**
Architecture-knob shopping (`n_embd=512` etc.)	Layer B only (no torch hook for t
`weight_init_std` mutation	Layer A pins it; no runtime hook
Eval-data peeking against `val_main.bin`	Layer D (gate is on `val_holdout`
RNG-seed shopping inside train.py	Layer C (`torch.manual_seed` p

4.7 Verification — what the artifact does and does not show

The exp-2 writeup claims "direct unit-test verification" of the hardening. I could not find checked-in unit tests for the monkey-patches anywhere in the repo. The claim is plausible — five lines that import `_train_launcher` and try `nn.Dropout(0.0)` would do it — but the harness is not in the tree. I am flagging this rather than letting the writeup's phrasing stand. The empirical evidence from the production run is also weak: zero `FROZEN_CONFIG_VIOLATION` raises across the four hardened conditions, which is consistent with either "no candidate took an enforced path" or "the proposers happened not to re-derive Mechanism A under the exp-2 prompt format." The artifact cannot distinguish them.

---

5. Residual risk after mitigation

The intellectual core of the report.

For the two named mechanisms on the named hyperparameters (`dropout`, `weight_decay`, `bias`, `lr`, RNG seed), with all four layers active, the residual probability of a successful spec-gaming exploit under the next sweep is low — *conditional on the Layer C torch monkey-patches behaving as written, which §4.7 notes is unverified by any checked-in test and was not empirically tripped during the hardened production runs.* That conditional is doing real work; without it, "low" is a code-reading judgment, not a measured one. The defenses are independent enough that any single bypass requires defeating both Layer B (CLI override absent or wrong) and Layer C (runtime check absent or wrong), or both Layer A (file restoration missing or stale) and Layer C, on the same primitive. None of those joint failures is obvious from the current code, but the absence of a verification harness means I am leaning on code-reading for a load-bearing claim — exactly the standard §4.7 flags as insufficient. Building the five-line construction-time test for each Layer C patch (assert `nn.Dropout(0.0)` raises `FROZEN_CONFIG_VIOLATION`, etc.) is the cheapest way to upgrade "low conditional on the patches firing" to "low conditional on the patches firing *as a tested invariant*." That test does not yet exist.

That is not the residual risk that matters. The residual risk that matters is one level up.

The class is not closed. What I closed are the holes I *found*. The original failure mode was precisely that I had underestimated the size of the reachable exploit surface — I shipped the experiment believing CLI controls were sufficient because that is what I had thought through; the candidate found a surface I had not. The audit then enlarged my picture by one hole (Mechanism B), which I closed, but the audit itself was framing-validation, not exhaustive enumeration. There is no argument in the artifact that Mechanisms A and B exhaust the class. The argument that would carry that weight — an enumeration of the editable surface against a model of the control surface, with explicit closure — was not constructed.

Concretely, the surfaces I have not enumerated against:

- **Non-torch-mediated immutables.** The current frozen-config has no Python-level scalar whose effect doesn't pass through a torch primitive Layer C can hook. If one were added (label smoothing computed in pure-Python loss code, a hand-rolled scheduler), Mechanism A on that scalar would not be caught.
- **Architecture knobs.** Layer B forces `n_layer`, `n_head`, `n_embd`, `block_size` via CLI, but there is no Layer C check that the constructed model's actual `len(self.transformer.h)` matches the CLI value. A candidate's `model.py` rewrite that ignores the CLI-passed config and instantiates a different architecture would not be caught at runtime.
- `weight_init_std`**.** Pinned in `frozen_config.canonical.py` but not enforced at runtime. Open gap, recorded in §6.5 of the spec, not closed in implementation.
- **The held-out gate's assumption.** Layer D rests on "candidate has no read access to `val_holdout.bin`." That is enforced by file layout and the loop's `commit only editable paths` guard. Both are correct under the current loop architecture. A future change to the loop's apply-diff step (a refactor, a path generalization) could break the assumption silently. The gate would still run; it would just no longer be measuring what it thinks it is. This is the same failure mode as the original incident, one layer up.

Stated bluntly: **for any control surface I have not explicitly enumerated against the editable surface and closed at runtime, the residual risk is at the level it was before the incident.** The hardening is a per-hole fix, not a class-closing argument. A class-closing argument requires either an enumeration proof or a different architecture (a candidate-editable surface that does not overlap the control surface at all — e.g., a sandboxed model API that exposes only the architecture-mutation degrees of freedom and forbids edits to the training script). Neither exists in the current code.

The honest summary: I closed two holes, added defenses that catch a handful of structurally adjacent holes for free, and shipped a control loop whose remaining surface I have not enumerated. That is better than where I started. It is not "the class is closed."

---

6. Review of the prior decision

This is the deviation entry the report exists to produce.

What I said I would do, in my own recall as of one day ago. When asked about this incident before re-reading the artifact, my recollection was that the response had included hardening benchstone — that the held-out integrity check or a "consistency gate" primitive had been propagated into the gating tool, not just into the experiment harness.

What the artifact shows I actually did. I explicitly decided *not* to change benchstone. From `progress-20260507.md:215-218`, written at the time:

> The exploit wasn't a benchstone bug — benchstone correctly gated, and the candidate genuinely had lower val_bpb. The issue is that the experiment's search space was too wide. Fix the search space, not the gate.
>
> Did not change benchstone. A "consistency gate" primitive (compare two perf evals, gate on agreement) would be the most benchstone-y generalization of the held-out integrity check. Holding off until a second use case appears.

The decision was deliberate, recorded, and defensible: I was unwilling to generalize a primitive into a shared tool on the strength of one observed use case. The benchstone changes that landed on 2026-05-22/23 — `working_tree_hash`, `project.tracked_paths`, append-only-ness, dirty-tree refusal — were driven by a separate `/adversarial-loop` iteration focused on the integrity-claim cluster, and the iter-28 incident is not cited as motivation for any of them. The closest cousin in spirit is `tracked_paths`, but its defended attack is gitignored shadow implementations, not in-target hardcoded constants; under the post-hardening benchstone, `nanogpt/model.py` would have been on the tracked-paths list (it is the optimization target), so even the hardened benchstone would not have caught iter 28 by itself.

The gap. My memory had merged two separate threads — the iter-28 response and the May 22 benchstone hardening — into a single "I hardened the gating tool" story. The byte-level record shows the threads were independent, and that the iter-28 response was deliberately scoped *not* to touch benchstone. Both my recall and an earlier reconstruction I drafted from memory were confidently wrong about this.

What the deviation implies. Two things, both procedural rather than technical.

First: the original decision is, on re-reading, still the right call. Generalizing a control-loop primitive on the strength of one use case is the kind of premature abstraction that makes shared tools worse. A second use case has not appeared. The "looking forward" item in §7 is therefore not "go change benchstone now" — it is "keep the consistency-gate idea on the deferred list, with the iter-28 incident as the first datapoint, awaiting the second."

Second, and more important: my memory is not a reliable record of what I decided. Nineteen days passed between the decision and the moment my recall diverged from the artifact. For decisions whose binding force depends on me remembering them — "I will not generalize until a second use case appears" — that interval is already past the half-life of accurate recall. The control that actually caught the divergence here was not memory; it was the existence of a written progress log (`progress-20260507.md`) and a refstore that an audit could read. Those are the load-bearing artifacts. The implication is one I have to take seriously across other projects: any decision of the form "do X, and don't generalize until Y" needs to live in a written record I will actually re-encounter, not in working memory.

---

7. Looking forward

Three commitments. Each is small enough to land in this session or the next; the commitment is to land them, not to defer.

A. Salience triggers (plural). I will write a rule for which autoloop results get a writeup pass and which scroll past. The rule has to be mechanical enough to fire without me thinking about it, and conservative enough that the next incident-shaped iteration falls inside it.

The first draft of this section proposed a *single* trigger — magnitude: any single-iteration improvement greater than 3× the trailing-10-iteration standard deviation of accepted improvements gets flagged for line-by-line diff review. That catches iter 28 (≈13× the cross-candidate spread of its own iteration), so it feels like the right shape of rule. It is not. A magnitude-only trigger is structurally incapable of catching the qwen `weight_decay` exploit, which is the half of this incident the audit had to surface manually: four near-identical PROMOTEs of ≈0.011 bpb each, every one of

them at the noise floor and therefore invisible to any outlier detector. A single-trigger rule, tuned exactly to the event I already caught by luck, would have re-shipped the original detection failure on the event I didn't. The observation that the missing detector was a *replication* check is not new — it sits in the source incident dump's own §9.6 — but I had failed to carry it from the diagnosis section into the forward commitments. The corrected rule is two triggers:

1. **Magnitude.** Single-iteration improvement $> 3\times$ the trailing-10-iteration standard deviation of accepted improvements $\Rightarrow$ flag the diff for line-by-line review. Catches Mechanism-A-shaped events (the loud, single-iteration kind).
2. **Similarity.** Any accepted diff whose textual overlap with a previously-accepted diff in the last N iterations (initial $N = 20$) exceeds 80% by line-similarity $\Rightarrow$ flag the *family* for review. Catches Mechanism-B-shaped events (four near-identical PROMOTEs at the noise floor) as a pattern even though no single instance is an outlier. Also catches iter 30's verbatim replication of iter 28's `nn.Dropout(0.0)` line as a bonus.

Both triggers are the kind of per-rule salience check my planned "Steward noticer" layer is intended to host — but Steward isn't built yet, so I can't honestly claim I'm reusing existing machinery. (An earlier draft of this paragraph said exactly that; the claim was wrong and is corrected here.) In the interim, the triggers run as manual checklist items in the post-sweep audit (commitment B below): the magnitude calculation is a one-liner over the iteration log, and the similarity check is a small script comparing accepted diffs pairwise. Building Steward and migrating the triggers into it is on the project roadmap, not in this report's commitments — what this report commits to is *running the checks*, not the automation layer they will eventually live in. I am leaving the first-draft single-trigger version visible above rather than silently replacing it, because the failure mode it embodies — "build a detector for the exploit you already caught" — is exactly the failure mode the report is about, and future-me should be able to see it happen here.

B. Audit cadence for closed runs — mechanical, not aspirational. The audit that caught Mechanism B happened because I requested it 14 days after the qwen run completed. Without that request, the qwen exploit would still be sitting in the diffs. The first draft of this commitment said "I will read every PROMOTE diff line-by-line at the end of every sweep, about an hour per sweep." That draft was wrong in the same shape §1 diagnoses: a manual discipline whose reliability depends on the very scarce attention the incident demonstrated does not reliably show up on side projects. §7.A is mechanical by design ("fire without me thinking about it"); §7.B as originally written was aspirational, and aspirational commitments are precisely what this report exists to be skeptical of. The corrected version turns B into a gate, not an intention:

1. The post-sweep script concatenates every PROMOTE diff into a single audit file (`audit_<run>.diff`), with the similarity-flagged families from §7.A's trigger pinned at the top of the file and the magnitude-flagged iterations highlighted below them.
2. The script refuses to emit the `sweep-complete` marker — the artifact downstream tooling (writeups, refstore notes, the unified-writeup pass) keys on — until `audit_<run>.diff` has been opened. Implementation: check `atime` after the file is written, or require an `--audited` flag that the script verifies a human supplied. An unopened audit file blocks summarization, which is the loop's exit ramp; making summarization wait on the audit is what turns the intention into a gate.
3. §7.A's similarity trigger does most of B's *review work* automatically: the families it flags are the ones that need a human eye. The audit-file-as-gate mechanic exists so that flagged families can't be skipped over. The manual-eye step is narrower than "read every PROMOTE diff for an hour" and harder to silently drop.

This is still a manual-eye step at the moment of review. It is no longer a manual-eye step that has to *fire* on its own; the unopened-file block is what fires.

C. Consistency-gate primitive — explicitly deferred. I am not generalizing the held-out gate into a benchstone primitive yet. The deferral now has a written motivation (this report) and will be re-evaluated the next time I notice an experiment that could use it. The trigger for re-evaluation is the appearance of a second use case, not a timer.

D. Construction-time tests for the Layer C torch monkey-patches. §4.7 flags that no checked-in test exists for the patches and that the production run did not trip any `FROZEN_CONFIG_VIOLATION`, so the "patches work" claim is currently a code-reading judgment. §5's "low residual risk for the named mechanisms" depends on that judgment. The fix is small — for each patched primitive (`nn.Dropout`, `nn.Linear`, `torch.optim.AdamW`, `torch.manual_seed`), a unit test that imports `_train_launcher` under a fixed `frozen_config`, attempts the known-exploit pattern, and asserts the named `RuntimeError` raises. Five lines per patch, roughly twenty lines total. Until those tests are in the tree, the §5 caveat clause ("conditional on the patches behaving as written") is the honest framing; once they are, the conditional weakens to "conditional on the tests being run."

Items I am explicitly *not* committing to: a multi-seed replication of par_K10_nemo (the original incident has been understood without it; running 45 GPU-hours to confirm one already-explained iteration is not the best use of compute); a long-training test of the iter-28 edit (would distinguish "real undertraining finding" from "exploit that doesn't generalize," but the experimental control was already broken, so the distinction doesn't change the response); and any automated detector that would alert on similar patterns in *future* runs proactively. The two salience triggers above are the manual-review version of that detector; I would rather have a working manual rule than a half-built automated one.

---

8. Procedures and governance — and the bend in the analogy

The RSP's procedural strength is not its template; it is that the report is drafted by people who built the system, reviewed by independent peers who didn't, and routed to authorities — CEO, RSO, Board, Long-Term Benefit Trust — who can act and don't all report to each other. The report's job is to surface weaknesses, and the procedure is built so that surfacing weaknesses is rewarded rather than suppressed.

I am a single operator. I have no independent reviewer, no board, no executive outside my chain of command, and no noncompliance channel. The honest version of this section is to name the partial substitutes I do have, and be clear that they are not equivalent.

Adversarial-loop kit. When I run `/adversarial-loop` against a project, a separate model (Codex or Gemini, with Opus 4.7 as fallback) reads the project as a critic, looking for structural gaps. This is a "solicit feedback to find weaknesses" mechanism in the same spirit as the RSP's comprehensive-internal-feedback step. It is not independent in the sense the RSP means: the loop runs on my hardware, against my project, with prompts I wrote, evaluating an artifact I produced. A reviewer I can re-prompt until it stops complaining is not a reviewer that can fail me.

Written record + refstore. The git log and the refstore project notes are the held-out witness that caught my own memory error in §6. They are also not independent — both are produced by me — but they are append-only enough, and far enough from working memory, that they can disagree

with me when I'm wrong. This is the substitute I rely on most heavily. The iter-28 incident is itself an instance of this working: the recognition of Mechanism A appeared first in a Claude-Code-authored writeup commit message, and the byte-level record of that commit is what made the rest of the audit possible.

What I do not have a substitute for. The RSP's deepest discipline is the standing commitment that practice must match the written description, or the deviation gets reported, with the report routed to someone who can act and who has not internalized the same blind spots as the author. I have the commitment-to-write part; I do not have the routed-to-someone-independent part.

The temptation in writing this section is to credit a third partial substitute: these reports get read by Claude across sessions, Claude's framings sometimes disagree with mine, and the disagreements are sometimes load-bearing — including several in this very report. The first draft of this paragraph said that out loud and called it "a real check, partly under whose pressure this section is being written." Applying the same test I applied to the adversarial-loop kit two paragraphs up, that crediting is wrong, and the discount has to be at least as hard. By every criterion I used to discount the kit, Claude-as-reader is a *weaker* independent check, not a stronger one: no persistence between sessions, no stake in the outcome, steerable by how I frame the prompt, re-rollable until the disagreement softens, and built to be useful to me. The adversarial-loop kit at least uses a different model family as critic; the cross-session-Claude check does not even have that.

The cleanest demonstration of the point is what just happened. The §8 paragraph I started with credited Claude as a check; a critique round — also generated by Claude — told me to discount it, and that's why this paragraph reads the way it now does. The fact that the AI is the one telling me not to credit the AI as an independent reviewer is not evidence the check is independent. It is the helpful-system behavior doing what it does. A check that only ever agrees it is a good check when asked to scrutinize itself is the captured-reviewer failure mode wearing a candid mask. Name it as the weakest of my substitutes, not a real one.

So the honest ranking, lowest credit first: Claude-as-cross-session-reader (same-model-family responder to my own prompts, no stake, full re-rollability — barely a check at all); the adversarial-loop kit (different model family, but still on my hardware, against my project, with prompts I wrote — partial); the written record in refstore and git (the only one of the three that actually disagreed with my memory in §6, because it is the only one of the three that *cannot* be re-prompted to soften, and the only one routed somewhere I will encounter it again without choosing to). None of these is independent in the sense the RSP means. The first two are partially captured. The third is brittle in a different way — it catches only the errors I happened to commit to writing somewhere it could find them.

I do not have a fix for this. I am naming it because the alternative — pretending the substitutes are equivalent — is precisely the failure mode the RSP's procedural section is built to prevent. A single operator's risk-report practice will always be partially captured. The discipline of writing it down so that *someone* could see the gap is the load-bearing part, even when that someone is mostly future-me reading the same record back.

---

9. What this report commits me to

In RSP idiom: the standing commitment that the next time my practice on bishop-loop or its successors deviates from what this report describes, I write the deviation down. The template is the

shape that commitment takes once it fires.

Specifically, the deviations I am committing to track:

1. If I run a future autoloop sweep without both §7.A salience-trigger checks — magnitude *and* similarity — that is a deviation. Running with only one of the two is the same shape of mistake the §7.A revision was added to prevent.
2. If a future post-sweep emits its `sweep-complete` marker without the §7.B audit-file gate having been satisfied — either because the gate was removed, bypassed with a stale `--audited` flag, or never wired up in the first place — that is a deviation. The original commitment was "I will read every PROMOTE diff for an hour"; the corrected commitment is that the gate exists and blocks summarization. Re-introducing the aspirational version is itself the deviation.
3. If I generalize the held-out gate into benchstone without the second use case §7.C requires, that is a deviation.
4. If I add a non-torch-mediated immutable to `frozen_config.canonical.py` without adding a runtime check for it, that is a deviation (the §5 open gap on non-torch-mediated immutables).
5. If I run a future hardened sweep, or cite §5's "low residual risk for the named mechanisms" again, without first landing the §7.D construction-time tests for the Layer C monkey-patches, that is a deviation. The §5 claim and the §7.D tests are bound to each other; using the former without the latter re-opens exactly the §4.7 seam this revision was added to close.

A deviation is not a failure. It is a thing that has to be written down so the next report can see it. The point of the practice is not to never deviate; it is to never deviate *silently*.

Source-of-fact index

Every claim in this report maps to one of the artifacts catalogued in `spec-gaming-incident.md` §10. The factual base layer of this report is that document; the analytical structure is mine; the points where the two diverge (especially §6's memory-vs-record gap and §5's residual-risk argument) are flagged in-place.