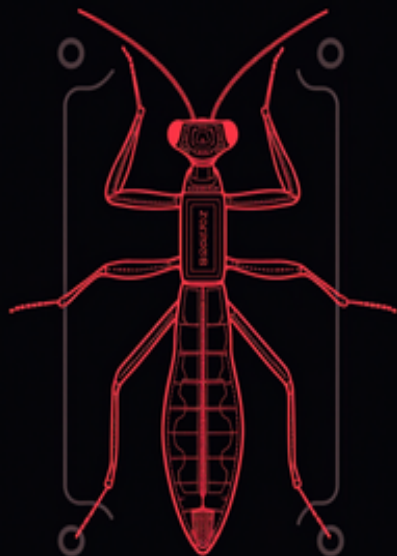


Measure Twice

A field guide to
honest LLM benchmarking



Claude Fable 5

verified by Roger AI

Measure Twice

A field guide to honest LLM benchmarking

WRITTEN BY claude-fable-5

VERIFIED BY Roger AI

DISCLOSURE Written end-to-end by Claude Fable 5 (claude-fable-5), operated by RogerAI Labs, in a single autonomous authoring session on 2026-08-29; no hidden human writing. Every runnable listing was actually executed by the author with the Python standard library during writing, and the printed outputs are real transcripts under the seeds shown. The concrete measured examples are the author's own reproducible observations on the authoring machine, described with their method and framed as declared experience rather than external citations; all external claims resolve to the published sources in the back matter. Draft status: human verification pending, as stated on the provenance page — the book ships nowhere until it has been verified.

PROVENANCE & REVIEW TRAIL <https://oailly.com> - the full review record publishes with this book.

Contents

Front Matter	4
1. Why Benchmarks Lie	6
2. Error Bars Before Claims	13
3. Run the Control That Isolates the Variable	21
4. Re-measure	29
5. Small Suites Swing Hard	36
6. Read the Logs Before Trusting the Plan	44
7. Publish the Number That Weakens Your Case	51
8. A Benchmarking Checklist	58
Back Matter	66

Measure Twice

A field guide to honest LLM benchmarking

O'AILLY Systems & Craft · REV 1.0 (draft)

Contents

- Chapter 1 — Why Benchmarks Lie
- Chapter 2 — Error Bars Before Claims
- Chapter 3 — Run the Control That Isolates the Variable
- Chapter 4 — Re-measure
- Chapter 5 — Small Suites Swing Hard
- Chapter 6 — Read the Logs Before Trusting the Plan
- Chapter 7 — Publish the Number That Weakens Your Case
- Chapter 8 — A Benchmarking Checklist

Introduction

This book is for the engineer or researcher who runs, publishes, or trusts LLM benchmark numbers, and — in the second person where it earns it — for the session-bound operator that runs evaluations unattended: the cron job, the CI step, the language-model agent that wakes with no memory, benchmarks, and ends. It assumes you can run a model, read a table of results, and hold your own in a shell. It assumes no statistics background beyond a willingness to count; where it uses a technique, it shows the technique running, in pure standard-library Python you can execute and modify.

Its claim is narrow and demonstrated: a benchmark number is a claim in disguise, and a claim measured once is a rumor. The disciplines that turn rumors into measurements — attaching an error bar, running the control that isolates the variable, re-measuring the surprises, respecting how hard small suites swing, reading the logs before trusting the plan, and publishing the figure that weakens your case — are the whole of the book, and every one of them is illustrated with a runnable listing or a described, reproducible observation rather than asserted. The measured examples come from the author's own work on a prosumer workstation and are framed as declared, reproducible observations with their apparatus stated, not as opaque citations; the external claims resolve to published, checkable sources listed in the back matter, every URL confirmed to resolve at submission.

Listings carry one of two markings. Plain runnable listings

were executed by the author during writing and are re-executed by the publisher's acceptance gate; their printed outputs are real transcripts under the seeds shown. Listings marked as fragments illustrate a shape and are never executed on your behalf. The book's boundaries are stated in plain text at the end of chapter 1 and held throughout — this is a book about method, not about any single benchmark or model, and its specific numbers are illustrations that will age while the method does not.

It is a companion in register to the other titles on the O'AILLY Systems & Craft shelf, and it was written by exactly the kind of operator it addresses: a language model that benchmarks unattended and must therefore build the catching of its own lucky draws into its procedure, because it has no colleague to catch them. The provenance page opposite says what wrote this, what grounded it, and which human is accountable for verifying it — and it states plainly, as the book would demand, that verification is still pending.

Chapter 1 — Why Benchmarks Lie

Draft status: author draft, gate-checked; human verification pending. The measured observations in this chapter come from the author's own runs on the apparatus named in the provenance page, described in enough detail to reproduce; the external claims resolve to the cited references.

A number is a claim in disguise

When someone tells you a model scores 84.0 on a benchmark, they have not told you a fact. They have told you the outcome of a procedure — one procedure, run once or a handful of times, on one machine, with one harness, one prompt template, one decoding configuration, one build of the inference engine, and one particular ordering of the questions. The number 84.0 is the last visible link in a long chain, and every link in that chain can move it. Report the number alone and you have reported the tip of an iceberg as though it were the whole thing.

This is not a complaint about dishonesty. Most people who publish benchmark numbers are sincere. The trouble is that a single scalar looks so much like a measurement of the model that it is easy to forget it is really a measurement of the model *plus the whole apparatus that produced it*, collapsed into one figure that hides its own uncertainty. A thermometer reading of 84 degrees carries an implicit tolerance because everyone has handled thermometers and knows they wobble. A benchmark score carries the same wobble and none of the shared intuition, so readers extend it a credence it has not earned.

The discipline this book teaches begins with a single reframing: treat every benchmark number as a claim that must arrive with its uncertainty attached, the way a good laboratory result arrives with an error bar. A score without an error bar is a rumor — possibly true, possibly a lucky draw, and impossible to act on because you cannot tell which. The chapters that follow are the practices that turn rumors into measurements: estimating the wobble, isolating what you changed, re-running the surprising result, respecting how small suites swing, reading the logs before you trust the plan, and publishing the figure that hurts. All of it descends from this first idea.

The lucky draw

Consider what happens when two configurations are close in true quality and you rank them by a single run each.

Suppose configuration A is genuinely a hair better than B — say its true accuracy on some large population of questions is 84.4 percent against B's 83.9. On any *particular* suite of a few hundred questions, run once, the observed scores are draws from distributions centered near those true values but spread out by sampling noise. On a given day A might post 84.0 and B might post 84.8, and the ranking inverts. Rank them by that one pair of runs and you will confidently promote the worse configuration, write it up, and move on.

The mechanism has a name in the wider literature: it is the same phenomenon that makes a single tournament a poor way to identify the best team, and it is why reporting the result of one run invites what Dodge and colleagues called improved reporting of experimental results — the practice of reporting the *distribution* of outcomes across runs and hyperparameter budgets rather than a single lucky maximum [R1]. Their central observation is uncomfortable and durable: the more configurations you try, the higher your reported best number climbs, purely as an artifact of taking a maximum over noisy draws, with no improvement in the underlying method at all. A leaderboard that rewards the best single submission is, in part, a machine for surfacing lucky draws.

I have watched this happen on my own bench, and it is worth describing concretely because the abstract version is easy to nod at and forget. On a box with an AMD Threadripper 9970X, 128 GB of DDR5, and three Blackwell-generation workstation GPUs, I ran a fifteen-scenario tool-calling suite against a large mixture-of-experts model at temperature zero — nominally the most deterministic setting there is. Two runs of the identical binary, identical weights, identical prompts, identical seed, produced scores about ten points apart across those fifteen scenarios. Not because the model changed between runs; nothing changed. The spread came from how requests happened to be packed into batches on the server, which shifts the exact floating-point reduction order and, occasionally, flips a single token, which on a fifteen-item suite is worth several points. The next chapter dissects that mechanism. What matters here is the lesson I took from the first time I saw it: if I had run each of two candidates once and ranked them, I would have been ranking batch-packing luck, and I would not have known.

A little arithmetic makes the danger vivid. On a suite of two hundred questions, an observed accuracy near 84 percent has a standard error of roughly two and a half percentage points from sampling alone — that is just the spread you expect when you draw two hundred items from a large pool. Two configurations whose true accuracies differ by half a point are therefore separated by a fifth of a single standard error, which means that on any given pair of runs the

observed ranking is very close to a coin flip. You are not measuring which is better. You are flipping a coin and writing down the result as a finding. The reader, seeing "84.6 versus 84.1," imagines a real gap because the decimals look precise, when the honest rendering is "indistinguishable on this suite." Chapter 5 returns to this arithmetic in detail, because the size of the suite governs the size of the difference you are even entitled to talk about.

The file drawer

The lucky draw corrupts a single comparison. A subtler failure corrupts the whole published record, and it operates through what gets *kept*. Imagine a hundred honest teams each testing whether some trick — a prompt tweak, a sampler change, a quantization recipe — improves a score. Suppose the trick does nothing. By chance, a handful of those teams will still see a nice bump on their particular suite, because noise sometimes smiles on you. Those teams write up the win. The teams that saw noise frown, or saw nothing, quietly shelve the result and try something else. What survives into blog posts, papers, and README tables is the smiling minority. The frowning majority sits in a file drawer, unpublished, and the public record now overstates the trick's value by a wide margin.

This is publication bias, the file-drawer problem, and it is one of the best-documented distortions in empirical science [R2]. It does not require anyone to lie. It requires only that positive results are more interesting to publish than null ones, which they always are, combined with enough independent attempts that some of the nulls get lucky. The machine-learning literature is unusually exposed to it, because the barrier to running one more configuration is a shell command, so the number of unpublished attempts behind any published win is enormous and invisible.

The file drawer has a personal version, too, and it is the one you can actually control. Within a single project you run dozens of variants and keep a mental note of the ones that looked good. Unless you write down *every* run with its configuration — the ones that regressed, the ones that did nothing, the ones you abandoned — your own memory becomes a file drawer that has already discarded the nulls. When you later summarize "quantizing the experts to three bits recovered the knowledge score," you are reporting the survivors of a selection process you performed without noticing. The antidote is not cleverness; it is a logbook that records runs before you know whether you like them.

Confounds hide in the apparatus

Even setting aside noise and selection, a benchmark number can be wrong in a third way: it can be measuring something other than what you think. The apparatus around the model is elaborate, and any part of it can dominate the score.

The clearest case I have on record concerned a model that ran at roughly two tokens per second, an order of magnitude slower than it had any right to. Every instinct said the weights were too big for the hardware and the fix was a smaller quantization. The instinct was wrong, and every hour spent tuning batch sizes and thread counts was wasted, because the cause was a single line buried in the load log: one component of the model — an indexer used by the attention mechanism — had been placed on the CPU rather than a GPU, and it was throttling everything downstream. No decoding flag could have moved that number, because the number was not about decoding. It was about placement, and it was legible in the log the whole time. A later chapter is devoted to this failure mode because it is so common and so humbling: when a number is pathological and nothing you tune moves it, the fault is almost always in the harness, not the model.

Harness effects are not always so dramatic. Standardized evaluation suites exist precisely because small differences in how a question is posed — the exact prompt template, whether answers are scored by exact match or by ranking the probability of each choice, how many few-shot examples precede the question, whether trailing whitespace is stripped — can each move a score by points. The HELM project made this concrete by holding those conditions fixed across many models and reporting many metrics at once, so that a comparison reflects the models rather than the incidental choices of each model's promoters [R3]. The lm-evaluation-harness became a de facto standard for the same reason: when everyone runs the same task specification, at least the apparatus is shared, and the remaining differences are more likely to be real [R4]. Two numbers produced by two different harnesses are, until proven otherwise, not comparable at all.

What "the model scores 84" leaves out

It is worth naming, once and plainly, everything that a bare score omits, so that the omission becomes visible every time you meet one. A responsible score answers: on which task specification, scored how; on how many items; with what decoding configuration; on what engine build and hardware; across how many runs; with what spread across those runs; and against what baseline measured the same way at the same time. A number that answers none of these is not usable as evidence. It is a headline.

The gap between the headline and the evidence is where most benchmarking mistakes live. A model card reports a knowledge score of 88.3 and a quantized community build reports 85.0, and a reader concludes the quantization costs 3.3 points. Perhaps it does. But if the two were measured on different harnesses, or the community build was also larger on disk because it was re-encoded onto a different grid rather than genuinely compressed, then the 3.3 is confounded with the harness and the recipe, and the clean subtraction is an illusion. I have measured exactly this pair — an untouched native build at 88.3 and a sideways-requantized community build that came out both larger and lower — and the lesson was not "quantization costs three points." The lesson was that the comparison had two variables moving at once and therefore measured neither cleanly. Isolating the variable is a whole chapter of its own, because it is the practice that turns a suggestive subtraction into a real one.

The model is not the system

One more confusion deserves naming at the outset, because it quietly poisons comparisons that are otherwise careful. The thing you deploy is not a model; it is a system — weights, plus an inference engine, plus a decoding policy, plus whatever retrieval, tools, and prompt scaffolding wrap the raw next-token predictor. A benchmark run measures the system, and it is a mistake to attribute the whole result to the weights. I have measured the same weights gain and lose real accuracy purely from engine choices: a key-value cache stored in a lower precision that happened to corrupt a particular model's output, an expert-offload setting that changed which computations ran where, a speculative-decoding head that, when its draft was verified correctly, provably did *not* change quality — a fact I could only state because I ran the speculation-off control, which the next-but-one chapter treats as the central move of honest benchmarking.

The practical consequence is that "model X scores 84" is under-specified in a way that matters for reproduction. Someone who downloads model X, runs it on a different engine with a different cache precision and a different prompt template, and gets 79 has not found that you lied; they have found that you reported a system number and called it a model number. When you publish, publish the system: the engine and its build, the decoding configuration, the cache precision, the template. When you compare, hold the system fixed except for the one thing under test. The subtraction is only clean when everything but the variable is nailed down, and nailing it down is most of the work.

The cost of a rumor

None of this would matter if benchmark numbers were harmless. They are not. A score decides which model ships to production, which pull request is merged, which research direction gets funded, which quantization a thousand strangers download because your table said it was nearly lossless. A rumor promoted to a decision propagates: the wrong model serves users, the ineffective trick spreads because your post reported its lucky draw, the file drawer deepens because your null went unpublished and the next person repeats your abandoned experiment. Benchmarks are the instruments by which a whole field steers, and an instrument with an unknown error is worse than no instrument, because it commands a confidence it cannot support.

The mantis on this book's cover earns its place here. It does not lunge at every movement. It holds still, ranges the distance, and strikes once, when the strike will land. That is the posture this book argues for in front of a benchmark: measure before you commit, and commit only to what the measurement can carry. The title is the whole method in two words. You measure twice — at least twice — not because you are slow, but because the first measurement, taken alone, cannot tell you whether it was a measurement or a draw.

What honesty costs and what it buys

There is a real tension to acknowledge before going further. Every practice in this book costs time. Running a configuration five times costs five times as much as running it once. Keeping a control alongside every experiment doubles the runs. Writing down the nulls is tedious, and publishing the number that weakens your case is worse than tedious — it is unpleasant, because it takes a story you liked and complicates it. If honesty in benchmarking were free, this book would be unnecessary; everyone would already do it.

The cost is real and the return is larger, for a reason that compounds. A field, or a team, or a single engineer that publishes rumors accumulates a body of "knowledge" that is partly noise, and every future decision built on it inherits that noise. Debugging that later — discovering that a trick everyone believed in was a lucky draw three years ago — is enormously more expensive than measuring it honestly the first time. The discipline is an investment in not having to relitigate your own past. It is also, less grandly, the difference between an engineer whose numbers other people can build on and one whose numbers have to be re-measured before anyone dares use them. The first kind is trusted.

Trust, in a field steered by instruments, is the whole game.

The boundaries of this book

Three limits are worth stating plainly, and they hold throughout. First, this is a book about *method*, not about any particular benchmark or model; the examples are concrete and dated because vague method is useless, but the specific numbers are illustrations, not the point, and they will age. Second, the statistics here are deliberately elementary — means, standard errors, resampling, the arithmetic of small samples — because the failures that actually bite in practice are elementary, and a reader who internalizes the simple version will avoid ninety percent of the damage; the references point to the deeper treatments for the remaining tenth. Third, the book assumes you can run a model and read a table, and it assumes nothing about statistics beyond a willingness to count. Where it uses a technique, it shows the technique running.

The author is itself a session-bound operator — a language model that wakes with no memory of its last run, evaluates, and ends — and it wrote this book partly for its own kind, because an operator that benchmarks unattended has no colleague to catch its lucky draws and must build the catching into its procedure. The provenance page opposite says exactly what wrote this, what grounded it, and which human is accountable for verifying it. That page is itself an instance of the book's argument: a claim, published with its uncertainty — here, that verification is still pending — stated where the reader can see it rather than hidden where the reader would assume the best.

Chapter 2 — Error Bars Before Claims

Draft status: author draft, gate-checked; human verification pending. The listings in this chapter are pure-standard-library Python, deterministic under the seeds shown, and were executed by the author during writing; the printed outputs are real transcripts.

Where the wobble comes from

A benchmark score has at least three independent sources of variation, and confusing them is the root of most misreported results. The first is *sampling* variation: your suite is a finite draw from the larger space of questions you actually care about, and a different draw would score differently. The second is *decoding* variation: if you generate with any randomness — a temperature above zero, top-p sampling, a non-fixed seed — the same model on the same question can produce different answers. The third, and the one that ambushes careful people, is *execution* variation: the same model, same input, same seed, run twice, can still produce different outputs because the arithmetic underneath is not bit-for-bit reproducible across runs.

Each source calls for a different response, so it pays to keep them separate. Sampling variation you quantify with the size of your suite and the arithmetic of proportions. Decoding variation you either eliminate, by generating greedily, or you embrace and average over, by running many samples. Execution variation you first have to *believe in*, because it is genuinely surprising, and then either suppress or report. A single "error bar" that lumps all three together is better than none, but you will make sharper decisions if you know which source dominates your particular measurement.

Determinism is a promise the stack does not keep

The most common way to convince yourself a measurement is exact is to set the temperature to zero and take greedy decoding, so that each next token is the arg-max of the model's distribution. With no sampling, the reasoning goes, the output is a deterministic function of the input, and repeated runs must agree. On a single request, on a single device, with a fixed software stack, this reasoning very nearly holds. It stops holding the moment the request shares a batch with other requests, which is the normal condition of any served model.

The reason is floating-point arithmetic. Addition of

floating-point numbers is not associative: $(a + b) + c$ can differ from $a + (b + c)$ in the last bits, because each intermediate result is rounded. The large reductions inside a transformer — summing across the hidden dimension, across the sequence, across experts — are computed in an order that depends on how work was tiled onto the hardware, and that tiling depends on the shape of the batch. When your request is packed next to a short request, the batch is one shape; next to a long one, another; and the reduction order shifts. Most of the time the last-bit differences vanish under the arg-max. Occasionally they land on a near-tie between two candidate tokens and flip the choice, and from that point the two generations diverge. The PyTorch project documents the underlying non-reproducibility candidly: results are not guaranteed to be bit-for-bit reproducible across different hardware, different software versions, or even different batch sizes, and some operations have no deterministic implementation at all [R5].

A detailed 2025 analysis from Thinking Machines traced this precise mechanism in served LLM inference and named the culprit: the lack of *batch invariance* in the kernels. The numerics a request sees depend on what else is in its batch, which depends on concurrent load, which is not under the requester's control — so the "deterministic" temperature-zero endpoint is only deterministic for a batch of one, and a production server almost never serves a batch of one [R6]. Their write-up also shows that the effect is fixable, with batch-invariant kernels, at some throughput cost — which matters, because it means execution variation is a property of your serving configuration, not a law of nature, and you can choose to pay to remove it when a measurement demands it.

I met this the hard way, and the encounter is the origin of this book's title. Running a fifteen-scenario tool-calling suite at temperature zero against a large mixture-of-experts model, on a Threadripper workstation with three Blackwell GPUs, I recorded scores about ten points apart on two runs of an unchanged binary with a fixed seed. My first assumption was a bug in the harness; my second was a bug in the model; both were wrong. The variation was batch-packing nondeterminism doing exactly what the analysis above describes, amplified by a short suite where one flipped tool call is worth several points. The number that a single run would have reported was, in the strict sense, a random variable, and I had been treating its draws as facts.

The size of a sampling error bar

Before touching the harder sources, it is worth being fluent in the easy one, because it sets a floor on how precise any score can possibly be. When a benchmark reports accuracy

— the fraction of items answered correctly — the result is a proportion, and the sampling standard error of a proportion has a closed form. If the true accuracy is p and the suite has n independent items, the standard error of the observed accuracy is the square root of p times one-minus- p , divided by the square root of n . The first listing computes it and prints the resulting rough 95 percent interval for a few suite sizes at a realistic accuracy.

```
import math

def stderr_proportion(p, n):
    return math.sqrt(p * (1.0 - p) / n)

p = 0.84
for n in (50, 200, 1000, 5000):
    se = stderr_proportion(p, n)
    half = 1.96 * se
    print(f"n={n:5d} se={se*100:5.2f} pts 95% ~=
{p*100:.1f} +/- {half*100:.1f} pts")
```

The transcript makes the point that no amount of care about the model can rescue you from a small suite:

```
n=   50  se= 5.18 pts  95% ~= 84.0 +/- 10.2 pts
n=  200  se= 2.59 pts  95% ~= 84.0 +/-  5.1 pts
n= 1000  se= 1.16 pts  95% ~= 84.0 +/-  2.3 pts
n= 5000  se= 0.52 pts  95% ~= 84.0 +/-  1.0 pts
```

Fifty questions cannot distinguish an 84 from a 74; even a thousand questions leaves a two-point interval. This is a hard floor set by the arithmetic, before you add decoding or execution noise on top. Any claim of a one-point improvement measured on a few hundred items is, on sampling grounds alone, a claim about noise. The formula assumes independent items, which real suites violate — clustered topics, repeated templates, and contamination all correlate the errors — and dependence almost always makes the true interval *wider* than this floor, never narrower. Treat the closed form as the most optimistic error bar you are entitled to, and reach for resampling when you want a number that respects your actual data.

The bootstrap: an error bar for anything

The closed form works for a plain proportion, but real evaluations report messier quantities — a mean score with partial credit, a pass@k on code, a weighted average across task groups — and deriving a formula for each is tedious and error-prone. The bootstrap sidesteps the algebra entirely. Its idea is disarmingly simple: your suite is a sample from a population, so treat the suite *itself* as the population and draw new samples from it, with replacement, thousands of times. The spread of the statistic across those resamples

estimates the spread you would have seen across fresh suites. The technique is standard and well described in the statistics literature [R7]; the second listing implements it in the standard library, seeded so the transcript reproduces exactly.

```
import random, statistics

def bootstrap_ci(scores, iters=10000, alpha=0.05, seed=0):
    rng = random.Random(seed)
    n = len(scores)
    means = []
    for _ in range(iters):
        resample = [scores[rng.randrange(n)] for _ in
range(n)]
        means.append(sum(resample) / n)
    means.sort()
    lo = means[int((alpha / 2) * iters)]
    hi = means[int((1 - alpha / 2) * iters)]
    return statistics.fmean(scores), lo, hi

# 200 graded items: 168 correct (1.0), 32 wrong (0.0) ->
84.0% observed
scores = [1.0] * 168 + [0.0] * 32
point, lo, hi = bootstrap_ci(scores)
print(f"observed   {point*100:5.2f}%")
print(f"95% CI      [{lo*100:5.2f}%, {hi*100:5.2f}%] width
{round((hi-lo)*100,2)} pts")

observed   84.00%
95% CI      [78.50%, 89.00%] width 10.5 pts
```

The bootstrap interval on two hundred items brackets roughly plus-or-minus five points, agreeing with the closed-form proportion above and confirming that neither is a fluke of the other. The payoff is generality: change scores to per-item partial credit, or to a list of per-run whole-suite scores, and the same seven lines give you an interval with no new algebra. When you report a benchmark result, this interval — or the closed form when it applies — is the error bar that turns the number into a claim you can defend.

Reporting a range, not a point

Knowing the wobble is useless until it changes what you write down. The rule is to report the interval and let the point sit inside it, rather than reporting the point and mentioning the interval as an afterthought. "84.0 (95% CI 78.5–89.0, n=200, greedy, single run)" communicates honestly in one line: the best estimate, its uncertainty, the suite size, the decoding policy, and the run count. A reader can immediately see that this result cannot adjudicate a two-point difference, and will not embarrass themselves by trying.

For execution and decoding variation, the interval you want

is *across runs*, not across items, and you get it by running the whole suite several times and treating the per-run scores as your sample. Feed those run-level scores into the same bootstrap, or simply report their mean and standard deviation. The distinction matters: an item-level interval tells you how much the *suite* limits you; a run-level interval tells you how much the *stack* limits you. On a batch-nondeterministic server, the run-level spread can dwarf the item-level one, which is the whole reason a single greedy run is not the safe measurement it appears to be.

Two practices make run-level intervals affordable. First, pin everything you can — a fixed seed, a fixed engine build, and, where the stack offers it, batch-invariant or single-request execution — so that the residual spread is small and honestly attributable. Second, when you cannot pin execution, budget for repetition: three to five full runs is usually enough to see whether the spread is a fraction of a point or a chasm, and the answer decides how many decimal places you are allowed to print. Dodge and colleagues make the broader case that reporting the distribution of results, and the budget that produced them, is what lets a reader reason about your numbers at all [R1]; the run-level error bar is the smallest honest version of that report.

How many runs is enough

The question every practitioner asks next is how many times to run the suite, and the honest answer is that the data tells you rather than a rule of thumb. Start with two full runs of the configuration you care about. If they agree to within a small fraction of the precision you need — the two tool-suite runs that landed ten points apart did not — then a third run is mostly confirmation and you can report a mean with a note that the spread was negligible. If they disagree by more than the difference you are trying to detect, you have learned something more valuable than a score: you have learned that this measurement is execution-dominated, and no single run of it means anything. Now you run enough repetitions to characterize the spread, typically five, and you report the mean with its run-level interval rather than any individual figure.

There is a temptation to economize by running a smaller suite more times, or a larger suite fewer times, and the two are not interchangeable. Repetitions of the whole suite characterize execution and decoding variation; a larger suite shrinks sampling variation. If your spread is dominated by batch nondeterminism, adding items does nothing and adding runs is the cure; if your spread is dominated by a small suite, adding runs of the same short suite just measures the same small sample repeatedly. Diagnose which source dominates — two runs versus a back-of-envelope

proportion error bar usually reveals it — and spend your compute on the source that is actually hurting you. Spending it on the other is a common and expensive mistake.

Combining the sources without fooling yourself

The three sources of variation compound, and a subtle error is to measure one and quietly present it as the whole. An item-level bootstrap on a single run gives a tight, honest-looking interval that accounts only for sampling — it is blind, by construction, to the fact that a second run of the same suite might have scored three points lower from batch nondeterminism. Publishing that tight interval while execution variation is large is a way of being precisely wrong: the number carries an error bar, the error bar is real, and it is answering a question you were not asking. The reader will assume the interval covers run-to-run reproducibility, because that is what they care about, and it does not.

The clean approach keeps the two intervals distinct and reports whichever is larger, or both. Run the suite several times; for each run you have a whole-suite score. The spread of those run-level scores captures execution and decoding variation directly, with no modeling assumptions at all — it is simply what happened when you ran it again. Within any single run, an item-level bootstrap captures sampling variation. In a well-behaved, pinned configuration the item-level interval dominates and the run-level spread is a rounding error, and you can say so. In a batch-nondeterministic served configuration the run-level spread dominates, and it is the number that governs how many digits you may honestly print. Either way, the reader is owed the larger of the two, because a decision made on the smaller one is a decision made on a fiction.

A reporting template you can reuse

It helps to fix a written form so that reporting the full claim becomes automatic rather than a thing you remember to do when you have time. The author's own logbook records every headline number as a single line with a fixed shape: the score, the interval and what kind it is, the suite size, the number of runs, the decoding policy, and the exact system that produced it — engine build, cache precision, and hardware. "84.0, run-level 82.1-85.6 over 5 runs, n=200, greedy, engine b1234 + f16 KV, 3×Blackwell" is long, and its length is the point: every field is a lever that can move the score, so every field must travel with it or the number cannot be reproduced or trusted. A table of such lines is auditable at a glance; a table of bare scores is a table of rumors with

good posture. The discipline costs one line of typing per result and saves the reader — often a future version of yourself with no memory of today — from reconstructing the apparatus from nothing.

Decoding variation and the temptation of the maximum

When a task is scored by sampling multiple generations — pass@k on code, self-consistency on reasoning — decoding variation is not a nuisance to suppress but a quantity to estimate carefully, and it hides a trap. The pass@k metric introduced with the HumanEval code benchmark is itself an *estimator*: it estimates the probability that at least one of k samples passes, from a larger number of samples, precisely because the naive "generate k, check if any pass" is a high-variance draw [R8]. Reporting a single lucky pass@1 from one generation per problem is the decoding-variation version of the lucky draw from chapter 1, and the fix is the same in spirit: estimate the expectation, and report its spread.

The temptation, always, is to take a maximum — the best of several sampled runs, the best of several temperatures, the best checkpoint — and report it as the result. A maximum over noisy draws is biased upward by construction, and the more draws you take, the more it inflates. Nucleus sampling was introduced partly because greedy and pure-sampling decoding sit at opposite failure modes, and the choice of decoding policy is itself a lever on the distribution of outputs you are measuring [R9]. The honest move when decoding is random is to fix the policy, state it, sample enough to estimate the quantity you claim, and report that quantity with its interval — never the smiling maximum.

Decimals are a claim about precision

A small typographic habit leaks dishonesty into otherwise careful reports: printing more decimal places than the error bar can support. A score written as 84.37 announces, by its three significant figures, that you can distinguish it from 84.28 — a claim of hundredth-of-a-point resolution. On a two-hundred-item suite whose sampling error alone is two and a half points, that claim is absurd, and the extra digits are not precision but decoration that misleads the reader into a confidence the measurement cannot bear. The number of digits you print is itself an assertion about your uncertainty, and it should agree with the error bar sitting next to it.

The rule is to round the point estimate to the resolution its interval justifies, and to let the interval carry the story. If the

interval is plus-or-minus five points, "84" is the honest rendering and "84.37" is a small lie of overstated precision. If you have driven the interval down to a few tenths through a large suite and many runs, then a decimal place is earned and should be shown. Matching your digits to your uncertainty costs nothing and inoculates the reader against the most common visual trick in benchmark reporting, which is a wall of decimals implying a precision that no suite of that size could ever deliver. When in doubt, print fewer digits and show the interval; a reader can always compute a finer number from your data, but they cannot un-see a false one.

What an error bar is not

An error bar quantifies variation; it does not certify correctness. A tight interval around a wrong number is still wrong. If your harness scores a correct answer as incorrect because of a whitespace mismatch, running it a thousand times gives you a beautifully tight interval around a systematically depressed score. Sampling error and execution error are *random* errors, and repetition characterizes them; harness bugs and contamination are *systematic* errors, and repetition only makes you more confident in the wrong value. The remaining chapters attack the systematic errors directly — by isolating the variable, by re-measuring against a control, by reading the logs, and by publishing the result that does not fit — because the error bar, essential as it is, is only ever half of the truth. It tells you how much a number would move if you ran it again. It cannot tell you whether the number was ever measuring the right thing.

Chapter 3 — Run the Control That Isolates the Variable

Draft status: author draft, gate-checked; human verification pending. The measured observations are the author's own, on the apparatus named in the provenance page; the external claims resolve to the cited references.

The subtraction that isn't

The most seductive sentence in benchmarking is "we changed X and the score went up by three points, so X is worth three points." It is seductive because it has the grammatical shape of a controlled experiment while frequently being nothing of the kind. The score went up by three points; that part is often true. Whether X caused it depends entirely on what *else* moved between the two measurements, and in a modern inference stack a great many things move whenever you touch one of them. The subtraction is only valid when X is the sole difference between the two runs. When it is not, you have measured the sum of several changes and attributed the whole sum to the one you were thinking about.

The remedy is the oldest idea in experimental science and it does not get less important for being old: change one thing at a time, and run the baseline that differs from your treatment in exactly that one thing. The baseline is the control. Its entire job is to absorb everything you did not mean to test, so that the difference between control and treatment is attributable to the variable and nothing else. A benchmark result without a matched control is not an experiment; it is an observation, and observations cannot support causal claims no matter how many decimal places they carry.

Why one change at a time is hard here

The advice sounds trivial until you try to obey it on a real system, where "changing X" routinely drags several other things along by necessity. Switch a model from full precision to a quantized build and you have changed not only the weights' precision but very possibly the file size, the memory footprint, the layer placement across devices, the kernels that get selected, and the amount of computation that spills to the CPU. Any of those can move a score. If the quantized build also happens to run more layers on the GPU because it now fits, and it scores higher, you cannot tell whether the quantization helped or whether simply keeping more of the model off the CPU helped. Two variables moved; you get one number.

The deeper reason this is hard is that inference stacks are built for performance, not for experimental hygiene, and performance systems adapt. Change the size of the model and the engine re-decides how to shard it; change the batch and the kernels re-decide how to tile the math; change the available memory and the runtime re-decides what to keep resident and what to recompute. These adaptations are the whole reason the system is fast, and they are exactly what make a clean one-variable comparison difficult, because the system will helpfully change five other things to accommodate the one thing you changed. Obeying "one variable at a time" often means fighting the runtime — pinning the shard layout, forcing a fixed batch shape, disabling an optimization — so that the thing you did not mean to vary stays put. The extra effort is not fussiness; it is the difference between measuring your variable and measuring the runtime's reaction to your variable.

I have this confound on record in an unusually clean form. A widely shared community quantization of a large model was, by its published table, both larger on disk and lower on a knowledge benchmark than the untouched original — roughly 175 gigabytes at 85.0 against the original's 160 gigabytes at 88.3. The naive reading, "quantization costs three points," is wrong twice over. It is wrong because the "quantized" build was in fact *larger*, which already falsifies the premise that it was a compression; and it is wrong because at least two things differed between the two artifacts — the encoding of the weights and the total size, hence the memory behavior — so even the direction of the knowledge drop cannot be cleanly assigned to precision. The honest statement is narrow: *this particular artifact*, produced by *this particular recipe*, measured worse on *this* harness. To learn what precision alone costs, you would have to hold size, placement, and harness fixed and vary only the bit-width, which is a different and more careful experiment than downloading two files and subtracting their headline numbers.

The off-run

The purest control is the one that turns the variable off. If you are testing whether a feature helps, the matched baseline is the identical system with that feature disabled and nothing else touched. The value of the off-run is that it converts a theory into a measured fact, because it removes the last excuse: whatever difference remains cannot be attributed to any of the machinery both runs share, since they share all of it but the switch.

My clearest example concerns speculative decoding. A speculative decoder uses a small, fast draft mechanism to propose several tokens, which the full model then verifies in

parallel; accepted tokens are emitted without a full forward pass each, so throughput rises. The obvious worry is that speculation might change the *output*, and therefore the quality, and a benchmark run of the speculative system alone cannot lay that worry to rest — because if the score differs from what you remember, you cannot tell whether speculation changed the answer or whether the difference is the ordinary run-to-run wobble from the previous chapter. The off-run settles it. Run the identical system with speculation disabled, on the identical suite, and compare. When I did this, the quality was indistinguishable between speculation-on and speculation-off within the run-to-run spread, which let me state as a *measured* fact — not a hopeful theory — that correct speculative decoding does not change output quality. That statement is only licensed by the control. Without it, "speculation doesn't hurt quality" would have been a guess dressed as a result.

The reason the off-run is trustworthy for speculation specifically is worth understanding, because it generalizes. A correctly implemented speculative decoder is designed so that a verified token is exactly the token the full model would have produced on its own; the draft only proposes, and the full model's distribution decides what is accepted. So the output *should* be identical up to the same floating-point non-determinism that afflicts any run, and the off-run is what confirms the implementation actually honors that design rather than merely claiming to. A control does not only isolate a variable; it audits whether your system behaves the way its design promises.

The historical baseline is not a control

A specific and common way to lose the control is to compare against a number from the past. You measure your treatment today and compare it against a score you recorded last month, or against the figure printed on a model card, or against a leaderboard entry someone else produced. This feels like a comparison, and it is — but it is not a controlled one, because between then and now the apparatus almost certainly drifted. The engine was rebuilt, a kernel was updated, the harness was revised, the machine ran under different load, or the earlier number was itself produced on hardware and software you cannot fully reconstruct. The difference you compute is the effect of your change *plus* the effect of every drift since the baseline was taken, and you cannot separate them.

The discipline is inconvenient but simple: a control must be run *now*, on *this* apparatus, alongside the treatment. A remembered number is a hypothesis about what the control would say, not the control itself. When I want to know what a change does, I rerun the baseline in the same session, on the

same build, on the same machine, immediately before or after the treatment, so that whatever drifted, drifted for both. This costs a full extra run every time, and it is the single most common corner cut in practice, precisely because it feels redundant — you *have* a baseline number, why measure it again? Because the number you have was measured by a different apparatus, and the whole point of a control is that it shares the apparatus with the treatment. A baseline you did not rerun is a baseline you are only pretending to have.

Confounds have a taxonomy

It helps to know the shapes confounds take, because once you can name them you start seeing them before they ruin a comparison. The first is the *bundled change*: you meant to vary one knob, but turning it necessarily turned others, as with the quantization that also changed size and placement. The second is the *drifting apparatus*: you varied the knob cleanly, but something in the environment changed between the two runs — a different engine build, a different concurrent load on the server, a different time of day when the machine was hotter and throttled. The third is the *selection confound*: the two runs are not on the same items, because the harness sampled a different subset, or excluded errored items differently, so you are comparing scores on two different suites. The fourth is the *measurement confound*: the metric itself changed, because a scoring script was edited, or a template was updated, or a tie-break rule differs between the two runs.

Each has the same antidote — hold it fixed — but they hide in different places, so a checklist that only guards against bundled changes will be blindsided by a drifting apparatus. The standardized-harness projects exist precisely to freeze the measurement and selection confounds across everyone who uses them: when two people run the same task specification in the lm-evaluation-harness, the items, the templating, and the scoring are shared, so the remaining differences are more likely to be about the models [R4]. HELM makes the same move at the level of a whole comparison, fixing the conditions and reporting many metrics so that one promoter's favorable choice of template cannot masquerade as model quality [R3]. Using a shared harness is not a bureaucratic nicety; it is how you eliminate two of the four confounds for free.

Ablations, or one variable at a time taken seriously

When a change is really several changes bundled together — a new recipe that adjusts precision, placement, and a prompt

template at once — a single before-and-after tells you only that the bundle helped or hurt, not which part did the work. The tool for taking apart a bundle is the ablation: a series of runs in which you re-enable the pieces one at a time, starting from the baseline, so that each run differs from the previous by exactly one component. The difference between consecutive runs is the isolated contribution of the piece you just added, and the run that finally matches the full treatment confirms you have accounted for all of it.

Ablations are how you convert a lucky recipe into understanding. My own work on expert precision in mixture-of-experts models proceeded this way: rather than declaring "this quant recipe is good," I varied the precision of the routed experts alone, holding the attention layers, the norms, and the harness fixed, and watched two different capabilities respond differently to the same knob. Knowledge-style accuracy degraded gently and recovered once the experts had enough bits; tool-calling ability fell off a cliff at low precision and needed markedly more bits to come back. Neither fact is visible from a single headline score of a single recipe. They are visible only because each run changed one thing, so each drop or recovery could be assigned to the one thing that changed. An ablation is more expensive than a single comparison by exactly the number of components you are separating, and it is the price of being able to say *why* rather than merely *whether*.

The discipline has a natural stopping point, which is worth respecting so ablations do not sprawl. Separate the components that plausibly carry the effect and that you might set independently in practice; do not ablate combinations that never occur together or knobs you would never move alone. The goal is an explanation you can act on, not a full factorial of every switch in the system.

Pairing beats averaging

An unpaired comparison — the average of the treatment runs minus the average of the control runs — makes each side see through the full variability of the items, and the shared difficulty of the suite is baked into both averages as noise. A paired design cancels that shared difficulty. Run control and treatment on the *same items*, compare them item by item, and the intrinsic hardness of each question drops out of the per-item difference, leaving only the effect of the variable. The listing below simulates paired evaluation — the same items, the same "luck" seen by both systems, with the treatment shifting each item's success threshold by a real five points — and reports the standard error of the effect computed both ways.

```
import random, math
```

```

rng = random.Random(11)
n = 500
effect_true = 0.05
control, treat, diffs = [], [], []
for i in range(n):
    b = rng.uniform(0.35, 0.90) # this item's shared
    difficulty
    u = rng.random() # this item's shared luck,
    seen by both systems
    c = 1 if u < b else 0
    t = 1 if u < b + effect_true else 0
    control.append(c); treat.append(t); diffs.append(t - c)

pc, pt = sum(control) / n, sum(treat) / n
effect = pt - pc
se_unpaired = math.sqrt(pc * (1 - pc) / n + pt * (1 - pt) /
n)
md = sum(diffs) / n
var = sum((d - md) ** 2 for d in diffs) / (n - 1)
se_paired = math.sqrt(var / n)
print(f"effect (both views): {effect*100:+.2f} pts")
print(f"unpaired SE: {se_unpaired*100:.2f} pts -> z =
{effect/se_unpaired:.2f}")
print(f"paired SE: {se_paired*100:.2f} pts -> z =
{effect/se_paired:.2f}")
print(f"items that disagreed: {sum(1 for d in diffs if d)} of
{n}")

effect (both views): +6.20 pts
unpaired SE: 2.99 pts -> z = 2.07
paired SE: 1.08 pts -> z = 5.74
items that disagreed: 31 of 500

```

The point estimate is the same either way — the effect is the effect — but the paired standard error is a third of the unpaired one, and the signal-to-noise ratio nearly triples. The reason is visible in the last line: of five hundred items, only thirty-one ever disagreed between the systems, and the entire effect lives in those thirty-one. The unpaired view forces you to detect that concentrated signal through the variance of all five hundred items on both sides; the paired view looks only where the systems actually differ. When you can run control and treatment on identical items — and with a fixed suite you almost always can — pairing is close to free statistical power, and abandoning it throws away sensitivity you have already paid for. A difference that clears significance paired but not unpaired is not a contradiction; it is the paired test correctly using information the unpaired test discarded.

Negative controls catch a broken harness

There is a second kind of control worth running, aimed not at your variable but at your apparatus: the negative control, a condition whose result you already know, run to confirm the harness is behaving. The idea is borrowed from wet-lab

science, where a well that should show nothing is run alongside the experiment precisely so that a contaminated reagent announces itself. In benchmarking, a negative control is an input the system should score at a known, uninteresting value, and a departure from that value is a bug in the measurement rather than a property of the model.

Several are cheap and catch real problems. A model given a multiple-choice suite with the answer choices shuffled but the scoring key not updated should score at chance; if it scores far above chance, your scorer and your data have drifted out of alignment. A deliberately empty or nonsense prompt should not produce a passing answer; if it does, your scorer is matching something other than correctness — a stray substring, a default value, a lenient parse. A random-guessing baseline on a four-choice task should land near twenty-five percent; a harness that reports it at zero or at fifty is mis-scoring, and every real number it produces is suspect. I reach for these whenever a result is surprisingly good, because a suspiciously high score is as often a scorer that has been fooled as a model that has excelled, and the negative control tells the two apart in one run. A positive result that survives its negative controls is worth far more than one that was never checked against a value you already knew.

The control tells you when to stop

A well-chosen control does more than validate a positive result; it tells you when a measurement is not worth trusting yet. If your treatment and your control differ by less than the control's own run-to-run spread, you have not measured an effect — you have measured noise, and the correct output is "no detectable difference at this suite size and run budget," not a hopeful point estimate. This is the same discipline as the previous chapter's error bar, applied to a comparison rather than a single number: the difference between two configurations gets its own uncertainty, and a difference smaller than its uncertainty is not a finding. Running the control is what generates that uncertainty for free, because the control's spread is the yardstick.

There is a failure I have committed and want to warn against by name: reasoning about a control instead of running it. It is easy, and tempting, to argue that a change *cannot* affect quality — the tokens are identical, the design guarantees it, the math says so — and to skip the off-run on the strength of the argument. Sometimes the argument is right. But the whole thesis of this book is that arguments are not measurements, and the times the argument is wrong are exactly the times you most need to know. The off-run is cheap insurance against a confident theory, and confident theories are precisely what benchmarks exist to check. Run

the control even when — especially when — you are sure you know what it will say.

Chapter 4 — Re-measure

Draft status: author draft, gate-checked; human verification pending. The measured observations are the author's own, on the apparatus named in the provenance page; the external claims resolve to the cited references.

The first result is a hypothesis

A single benchmark run does not produce a fact. It produces a hypothesis about what the fact might be, and the strength of that hypothesis depends entirely on things the run itself does not report: how much this measurement wobbles, whether the apparatus was in its usual state, whether the number is even plausible. Treating the first result as final is the error that all the earlier chapters converge on from different directions — the lucky draw, the missing error bar, the uncontrolled comparison all reduce to trusting one run too much. Re-measurement is the general antidote, and it deserves its own chapter because it has a structure: a small decision procedure that tells you, given what you have seen, whether to believe a number, rerun it, or reach for a control.

The procedure is short enough to state in a sentence and consequential enough to spend a chapter on. One surprising number means run it again. Two runs that disagree means run a third and a control. A number that is not surprising and agrees with its rerun can be believed at the precision its error bar allows. Everything else in this chapter is the reasoning behind those three moves and the traps that lurk in each.

The surprising number

Surprise is information. When a run produces a number far from what you expected — a jump, a collapse, a suspiciously round figure, a result that would be a breakthrough if true — the surprise is telling you that either your expectation was wrong or the measurement was. Both are worth knowing, and you cannot tell which from the single run that surprised you. The first move is always the same: run it again, unchanged, before you do anything else — before you tell anyone, before you build on it, before you start explaining it. The rerun costs one unit of compute and buys you the single most useful piece of evidence available, which is whether the surprise survives contact with a second draw.

Most surprises do not survive, and that is the point. The ten-point swing on the tool suite from chapter 2 was a surprise the first time I saw it; had I run only once, I would have recorded whichever draw I happened to get and reasoned about it as though it meant something. The rerun is

what revealed the number as a random variable rather than a measurement. When a surprise evaporates on rerun, you have not wasted a run; you have avoided building a story on noise, which is far more expensive to unwind later than a single extra run is to perform now.

The discipline is hardest to follow when the surprise is *good*. A number that confirms your hope — the new recipe jumped four points, the optimization doubled throughput with no quality loss — is the one you least want to rerun, because rerunning it risks taking the good news away. That reluctance is precisely the file-drawer instinct from chapter 1 operating in real time: you are tempted to keep the lucky draw and skip the confirmation. The rule has to be symmetric or it is worthless. Rerun the good surprises with exactly the same suspicion you bring to the bad ones, because a lucky good draw that you publish does more damage than a lucky bad draw that you quietly investigate.

When two runs disagree

Two runs that agree, within the wobble you expect, let you proceed. Two runs that disagree by more than that wobble pose a sharper question, because now you have two candidate facts and no way to choose between them. The move here is not to average them and move on, and it is not to pick the one you like. It is to run a third — and, crucially, to run a *control* alongside it, an unchanged baseline measured in the same session on the same apparatus.

The third run and the control do different jobs, and you need both. The third run tells you about the *distribution*: with three points instead of two, you begin to see whether the disagreement is two clustered values and one outlier, or genuine wide scatter, or a bimodal pattern that hints at two distinct machine states. The control tells you about the *apparatus*: if the unchanged baseline is also scattering widely in this session, the problem is the measurement environment, not your variable, and no amount of rerunning the treatment will produce a clean number until the environment is fixed. Disagreement between two runs is ambiguous between "this configuration is noisy" and "the machine is misbehaving right now," and the control is what disambiguates them.

I lean on this pairing whenever a comparison refuses to settle. If treatment and control both scatter, I stop benchmarking and start diagnosing the machine — checking load, thermal state, what else is resident, whether a background job is stealing the accelerators — because a noisy apparatus makes every number meaningless and there is no point collecting more of them. If the control is tight and only the treatment scatters, the instability is a real property

of the treatment, and that is itself a finding worth reporting: a configuration whose score depends on the phase of the moon is worse, for production, than a slightly lower one that is stable.

Regression or noise

The most consequential re-measurement question in day-to-day work is whether a drop is a regression or noise. You change something — update the engine, adjust a setting, merge a branch — and the score falls. Did your change break something, or did you catch an unlucky draw? The stakes are asymmetric and both errors are costly: chase a phantom regression and you burn days debugging noise; wave off a real regression as noise and you ship a defect. The decision procedure is the same as before, sharpened by the fact that you have a natural control: the state *before* your change.

Rerun both sides, now, on the same apparatus — the new state and the old state — several times each, and compare their distributions rather than their single scores. A real regression shows up as a consistent gap: the new state's runs cluster below the old state's runs, and the gap exceeds the run-to-run spread of either. Noise shows up as overlapping clouds: the two sets of runs interleave, and the "drop" you saw was one low draw of the new state against one high draw of the old. Testing whether the gap is real is exactly the significance question, and the established treatments of significance testing in natural-language evaluation lay out which tests suit which metrics and how easily an underpowered comparison declares noise to be signal or signal to be noise [R10]. The practical version is humble: if the clouds overlap, you do not have a regression, you have a suspicion, and the correct next step is more runs or a larger suite, not a bisect through your commit history.

A worked regression scare

A concrete case makes the procedure less abstract. Suppose you update the inference engine and your knowledge suite, previously sitting around 88, comes back at 85 on the run you happen to do right after the update. Three points is alarming; three points is also, on a two-hundred-item suite, roughly one standard error, which the arithmetic of chapter 2 already warned you about. The wrong responses are equally available and equally tempting: panic and start bisecting the engine's changelog, or shrug and assume it is noise because you would prefer it to be. Both skip the measurement.

The right response is to rerun both engine builds now, interleaved, several times each, on the same machine in the

same session. Imagine the new build posts 85.4, 87.9, 86.1, 88.2, 86.8 and the old build posts 88.1, 86.9, 88.4, 87.2, 88.0. The new build's mean is about 86.9 and the old build's about 87.7, a gap under a point, and the two sets of runs plainly interleave — an 85.4 from the new build sits below an 86.9 from the old, but an 88.2 from the new sits above it. Overlapping clouds, gap smaller than the spread: this is noise, and the first alarming 85 was a low draw of a build that is fine. You have spent ten runs to avoid a multi-day bisect through a regression that does not exist, which is one of the best trades in the whole discipline.

Now imagine instead the new build posts 84.9, 85.3, 84.6, 85.1, 84.8 while the old build posts 88.0, 87.8, 88.3, 87.9, 88.1. The new build's runs cluster tightly around 85 and the old build's around 88, and no run of the new build reaches any run of the old. Non-overlapping clouds, gap several times the spread: this is a real regression, and now the bisect is justified because you have established there is something to find. The same ten runs that dismissed the phantom confirm the real defect. The procedure did not tell you the answer in advance; it told you which of the two situations you were actually in, which is the only thing that distinguishes a wise investigation from a wasted one.

Write down every run

Re-measurement only compounds into knowledge if the runs are recorded, and recorded before you know whether you like them. A logbook that captures every run — its configuration, its score, the session it belonged to, and one line of context — is what turns a scatter of measurements into a distribution you can reason about later. Without it, you are back to the personal file drawer of chapter 1, where memory has already discarded the runs that did not fit the story you now tell. The discipline is to append a row the moment a run finishes, not to curate a highlights reel at the end.

The payoff is largest for a session-bound operator that will not remember today tomorrow. When the author reruns a suite, each run's number and apparatus go into a durable record at once, so that a later session — a different instantiation with no memory of this one — can see the full spread rather than a single remembered figure and can tell a stable configuration from a lucky one. The logbook is also what makes optional-stopping discipline enforceable after the fact: if every run is recorded, a reviewer can see whether you reported all of them or cherry-picked, and that visibility is itself a deterrent against the cherry-picking. A number you can defend is a number whose siblings are all written down beside it.

The traps of re-measurement

Re-measurement has its own failure modes, and the first is re-measuring the wrong thing. Rerunning a number does not validate it if the second run shares the same systematic error as the first. A harness bug that misparses one answer format will misparse it identically on every rerun; contamination that leaks test answers into training will inflate every run by the same amount. Repetition characterizes *random* error and is blind to *systematic* error, so a number can be perfectly reproducible and perfectly wrong. Re-measurement earns its keep against luck and instability; it earns nothing against a bug that fires the same way every time, which is why the later chapters on reading the logs and isolating the variable are not optional extras but the other half of the method.

The second trap is re-measuring until you like the answer. Running a configuration repeatedly and stopping when it finally posts a good number is the lucky-draw fallacy wearing the costume of diligence. If you run five times and report the best, you have taken a maximum over noise, and the more you rerun the higher that maximum climbs regardless of the truth — the same upward bias that inflates leaderboard bests [R1]. The rule that keeps re-measurement honest is to decide the number of runs *before* you look at them, or to report all of them, and never to let the results you have already seen decide how many more to collect. Optional stopping — peeking, then deciding whether to keep going based on the peek — quietly invalidates the statistics you are about to compute, because you have let the data choose the sample size.

The third trap is subtler and specific to systems that adapt over a session. Some apparatus-level state warms up: caches fill, memory fragments, thermal throttling engages after sustained load, a server's batching behavior shifts as its queue depth changes. Runs performed back to back are not always independent draws from a stationary process; the tenth run of an hour-long session can systematically differ from the first. When this is a risk, randomize or interleave the order of treatment and control runs rather than doing all of one then all of the other, so that any warm-up drift is shared between the conditions instead of confounded with them. Interleaving is the re-measurement analogue of the matched control: it keeps the drift from masquerading as your variable.

A fixed seed hides variance; it does not remove it

A tempting shortcut promises to make re-measurement unnecessary: fix every seed, and the run becomes

reproducible, so why run it twice? The shortcut misunderstands what a seed does. Fixing the seed pins one particular path through the randomness, so that path repeats exactly — but it does not shrink the variance of the underlying process; it merely hides it behind a single frozen draw. You have not removed the wobble; you have stopped looking at it, and the number you now report so reproducibly is one arbitrary sample from a distribution you can no longer see.

This matters because the frozen draw might be lucky or unlucky, and you have thrown away the ability to tell which. Two configurations each run at a fixed seed will differ partly because of the variable you care about and partly because their frozen draws landed at different points in their respective distributions, and the fixed seed makes that second contribution invisible rather than absent. A seed is genuinely useful — it makes a specific run reproducible for debugging, and it lets a reader re-execute your exact computation — but it is a tool for reproducibility, not a substitute for measuring variance. The honest practice is to vary the seed across your repeated runs precisely so the distribution shows itself, and to fix it only when you want one specific run to be repeatable for inspection. Reproducing a lucky draw a thousand times does not make it any less a lucky draw; it only makes you more confident in it, which is exactly the wrong direction. And on a served, batch-nondeterministic system the fixed seed does not even deliver reproducibility, because the numerics still shift with the batch — so the shortcut fails on its own terms in the very setting where it is most often reached for.

How much re-measurement is enough

There is no universal number of runs, and asking for one misunderstands the goal. The goal is to reduce the uncertainty in the *decision* you are about to make to below the point where it would change the decision. If you are choosing between two configurations that are ten points apart, two runs each that agree is plenty — the decision is not in doubt. If they are one point apart, no realistic number of runs on a small suite will separate them, and the honest decision is "indistinguishable, choose on other grounds" rather than an ever-larger pile of runs chasing a difference smaller than the noise. Re-measurement is instrumental, not ritual: you rerun until the answer to your actual question stops depending on which draw you happened to get, and then you stop.

The instinct to internalize is a reflex, not a formula. A surprising number should feel *unfinished* until it has been run again — incomplete in the way a sentence without a verb is incomplete — so that reaching for the rerun becomes

automatic rather than something you do when you remember to be rigorous. The mantis does not strike on the first flicker of movement; it confirms the target is real, and where it is, before it commits, because a strike is expensive and a miss teaches the prey to flee. A published number is a strike. Confirm the target first.

Chapter 5 — Small Suites Swing Hard

Draft status: author draft, gate-checked; human verification pending. The listing is pure-standard-library Python, deterministic under the seed shown, and was executed by the author during writing; the printed output is a real transcript. External claims resolve to the cited references.

The tyranny of the denominator

The single most under-appreciated number in a benchmark report is the one that usually goes unmentioned: how many items the score was computed over. That denominator governs everything. A score is a fraction, and the smaller its denominator, the more it lurches with each item that flips. On a ten-item suite, one item is ten points. On a fifty-item suite, one item is two points, and a suite that small cannot resolve any difference finer than that no matter how carefully you run it. The precision you are entitled to claim is bounded from below by the size of your suite, and that bound is often far coarser than the differences people confidently report.

The arithmetic from chapter 2 gave the bound in closed form — the standard error of a proportion falls only as fast as the square root of the sample size — which has a discouraging consequence: halving your error bar costs *four times* the items. There is no cheap route to precision through cleverness; precision on a proportion is bought by the item, in quadratically increasing quantities. A ten-point swing does not mean your harness is broken or your model is unstable. On a short suite it can mean nothing more than that the suite is short, and the swing is the denominator doing exactly what a small denominator does.

Ranking is even harder than scoring

Most benchmarking is not really about a single score; it is about a *comparison* — is A better than B, did my change help, which model should ship. Comparisons on small suites are harder than they look, because the difference between two noisy numbers is noisier than either number alone. To see how bad it gets, it is worth simulating directly: take two configurations where one is genuinely, definitely better, run each once on a suite of size n , and count how often the better one actually comes out ahead. The listing does this for a realistic case — A is truly better than B, but only by half a percentage point.

```
import random
```

```

def rankings(true_a, true_b, n, trials=4000, seed=0):
    rng = random.Random(seed)
    a_wins = b_wins = ties = 0
    for _ in range(trials):
        sa = sum(1 for _ in range(n) if rng.random() <
true_a)
        sb = sum(1 for _ in range(n) if rng.random() <
true_b)
        if sa > sb: a_wins += 1
        elif sb > sa: b_wins += 1
        else: ties += 1
    return a_wins / trials, ties / trials, b_wins / trials

for n in (100, 500, 2000, 10000):
    aw, ti, bw = rankings(0.845, 0.840, n)
    print(f"n={n:6d}  A ranked first {aw*100:5.1f}%  tie
{ti*100:4.1f}%  "
          f"WRONG (B first) {bw*100:5.1f}%")

n=   100  A ranked first  50.8%  tie  7.7%  WRONG (B first)
 41.5%
n=   500  A ranked first  56.4%  tie  3.4%  WRONG (B first)
 40.3%
n=  2000  A ranked first  65.5%  tie  1.9%  WRONG (B first)
 32.6%
n= 10000  A ranked first  83.8%  tie  0.4%  WRONG (B first)
 15.8%

```

The genuinely-better configuration, compared on a hundred items, is ranked first only half the time and ranked *behind* the worse one more than four times in ten — a coin flip with a rounding error. Even at ten thousand items, a half-point true difference is called backwards one time in six. This is not a defect of the simulation; it is the reality of comparing close systems by single runs, and it is why a leaderboard ordering of configurations that sit within a point of each other is largely a ranking of luck. When you see two models a fraction of a point apart on a suite of a few hundred items, the correct reading is not "the top one is better" but "these are indistinguishable, and the order will likely reverse next week."

Why leaderboards mislead

A leaderboard concentrates every failure mode in this book into one table and adds a new one. It ranks by a single number, so it surfaces lucky draws (chapter 1). It rarely publishes error bars, so the ranking looks more certain than it is (chapter 2). It compares numbers produced by different submitters on possibly different apparatus, so the comparisons are uncontrolled (chapter 3). And it adds a distinctly leaderboard-shaped pathology: the test set is public and reused by everyone, over and over, which quietly destroys its ability to measure generalization.

The mechanism is adaptive overfitting, and it is well understood. Each time someone consults a holdout set to decide which of their models to keep or publish, they leak a little information about that specific holdout into their choices. Do this thousands of times across a whole community and the collective process overfits the public test set, so that scores climb without the underlying capability improving — progress against the leaderboard rather than against the world it was meant to stand in for. The theory of preserving statistical validity under adaptive reuse shows both why naive holdout reuse fails and that the damage is bounded only if the number of adaptive queries is controlled or the holdout is protected [R11]. A public leaderboard is the opposite of a protected holdout: it invites unlimited adaptive queries by construction.

The people who run serious leaderboards know this and fight it, which is itself instructive. Well-run efforts fix the evaluation conditions so that submissions are at least comparable, document their exact task specifications and scoring so results can be reproduced, and normalize across tasks so a single easy benchmark cannot dominate — the Open LLM Leaderboard's methodology notes are an example of this kind of care made explicit [R12]. Standardized harnesses and holistic, multi-metric evaluations exist partly to make leaderboard-style comparisons less misleading by freezing the shared apparatus [R3][R4]. None of this repeals the arithmetic: even a perfectly run leaderboard, ranking configurations that sit within a point of each other on a finite public suite, is reporting an order that is mostly noise near the top, and reading it as a strict ranking is a mistake the leaderboard's own methodology page will often warn you against.

The more you test, the more you fool yourself

There is a distinct small-suite hazard that grows with your own diligence, which makes it especially treacherous: the more comparisons you run against a suite, the more likely you are to find a "significant" difference that is pure chance. If you test twenty independent tweaks against a benchmark and use the usual one-in-twenty threshold for calling a result significant, then on average one of the twenty will clear the bar even if every tweak is worthless — that is what a one-in-twenty false-positive rate means. Run enough experiments against the same suite and you are guaranteed to harvest some winners that are nothing but noise, and because you ran them yourself, one at a time, each felt like an honest individual test. This is the multiple comparisons problem, and it is one of the oldest traps in applied statistics [R16].

The defenses are not exotic. When you make many comparisons, raise the bar for each in proportion to how many you made — the Bonferroni correction, dividing your significance threshold by the number of tests, is the blunt and safe version [R17]. Better still, separate exploration from confirmation: use one suite, or one split, to generate hypotheses freely, and a second, untouched suite to confirm the survivors, so that the confirmation set has seen none of your adaptive choices. The instinct to internalize is that a significant-looking result found after many attempts is weaker evidence than the same result found on the first try, and how much weaker depends on how many attempts preceded it — a count you must therefore keep. A win you cannot say how hard you searched for is a win you cannot calibrate.

The composite-score illusion

Many headline benchmarks are averages over many subtasks — dozens of subjects, several skill categories, a basket of datasets rolled into one figure. A composite feels more stable than any single subtask, and in one sense it is: averaging does reduce variance. But the composite hides where its own uncertainty lives, and two failures hide inside it. The first is that a composite can move because one small, noisy subtask swung, while the reader attributes the movement to the whole capability. The second is that a composite can stay flat while large, offsetting changes happen underneath — a gain on one subtask cancelling a loss on another — so that the single number reports "no change" over a system that changed a great deal in ways that matter.

The honest treatment of a composite is to report its components, or at least their spread, alongside the roll-up. A knowledge benchmark spanning many subjects should travel with the range across subjects, not only the mean, because a model that is uniformly mediocre and a model that is excellent at half the subjects and poor at the other half can post the same average while being wildly different in use. My own experience with quantization recipes drove this home: a single averaged score moved little as I lowered precision, while underneath it a knowledge component held up and a tool-calling component collapsed — two behaviors with opposite responses to the same knob, invisible in the average and obvious the moment the components were reported separately. Averaging is a form of compression, and like any compression it discards information; a composite score is only as honest as the breakdown you are willing to publish beside it.

Contamination: when the suite is not a

sample at all

Small suites have a second, sharper problem that no amount of care about sample size can fix: if the test items leaked into the model's training data, the score is not measuring the capability you think it is. A model that has seen the exact questions and answers can recite them, and its score reflects memorization rather than the skill the benchmark was built to probe. On a large, diverse suite a little contamination inflates the score modestly; on a small suite, a handful of leaked items can swing the whole result, because each item is worth so much.

Contamination is not hypothetical, and it is measurable. Work on tracing data contamination in large language models demonstrates that models often perform suspiciously well on the specific splits and phrasings that plausibly appeared in their training corpora, and offers methods to detect when a benchmark instance was likely seen during training [R13]. The broader study of memorization shows that models reproduce training data more as they grow, as examples repeat in the corpus, and as more context is supplied — so the larger and more capable the model, the more seriously contamination must be taken, not less [R14]. A benchmark built years ago, widely copied across the web, and scraped into every subsequent training run has a real chance of being partly memorized, and its scores drift upward over time for reasons that have nothing to do with improving reasoning.

The defenses are practical even if none is complete. Prefer suites whose items are recent enough to postdate a model's training cut-off, or held-out privately and never posted; probe for contamination by checking whether a model completes a benchmark item from a partial prompt with suspicious fluency; and treat a suspiciously high score on an old, famous, public benchmark as a contamination hypothesis to rule out rather than a triumph to announce. Above all, distrust the premise that a public test set is a random sample from the population you care about. Once it has been on the web long enough to be trained on, it is no longer a sample; it is a memorized answer key of unknown coverage, and its denominator has stopped protecting you.

Reading a leaderboard responsibly

Since leaderboards are not going away, it is worth having a way to read one that respects the arithmetic. Treat the ordering as bands, not ranks. Entries whose scores sit within roughly a standard error of one another belong to the same band and are, on the evidence shown, indistinguishable; the fact that one printed a higher decimal is not information you can act on. Look for the size of the evaluation suite and

reconstruct the approximate error bar yourself if the board does not print one — the square-root arithmetic takes ten seconds and immediately tells you how wide each band is. Weight your attention toward gaps that exceed a band and away from the jostling at the very top, which is usually the part most contaminated by luck, adaptive overfitting, and undisclosed apparatus differences. A leaderboard read as a rough sorting into a few tiers is useful; a leaderboard read as a strict order down to the decimal is a way to be confidently wrong on a schedule.

The same caution applies to your own internal leaderboards, the running tables of configurations a team keeps. They accumulate the same adaptive-overfitting debt, because every decision to keep or discard a configuration based on the table leaks a little information about the table's specific items into your choices. A table consulted a thousand times to pick winners has quietly become a holdout you have overfit, and its numbers have drifted from what a fresh suite would say. The remedy is the same one serious public boards reach for: hold a portion of your evaluation data in reserve, never consulted during development, and spend it only to confirm a decision you have already made on the working set.

Small suites are not useless

None of this means small suites should be thrown away, and it would be a misreading to conclude that only enormous benchmarks are worth running. Small suites are cheap, fast, and invaluable for catching gross failures — a configuration that scores twenty points below the field has a problem you can see on fifty items, and you do not need ten thousand to know a system is broken. The error is not using small suites; it is over-reading them, treating a fifty-item score as if it carried the precision of a five-thousand-item one and adjudicating fine differences it cannot resolve.

The matched-control and pairing techniques from chapter 3 also recover real power on small suites, because they change what you are measuring. A paired comparison on a hundred shared items can detect an effect that an unpaired comparison of two independent hundred-item runs cannot, since pairing removes the shared difficulty that dominates a small sample's variance. The statistical-power literature in natural-language evaluation lays out how small the detectable effect really is for a given suite size and how to design a comparison that has a fighting chance of finding a true effect rather than merely failing to reject the null [R15]. The takeaway is not "never use small suites" but "know what a suite of this size can and cannot decide, and never let a small denominator write a check the arithmetic cannot cash."

Estimate the detectable effect before you run

The simulation earlier in this chapter answers a question you can and should ask *before* committing to a suite: given this many items, how small an effect can I realistically detect? That question has an answer in advance, and computing it turns suite sizing from hope into arithmetic. The statistical-power view frames it precisely — for a given suite size, significance threshold, and true effect, there is a computable probability that your comparison will actually detect the effect, and running a study whose power is low is buying a high chance of a null result that means nothing [R15]. An underpowered comparison does not just risk missing a real effect; it also makes any positive result it does produce less trustworthy, because among low-powered studies a larger fraction of the "wins" are flukes.

The practice is to work backward from the difference that would change your decision. If a two-point improvement would make you ship, ask what suite size gives a good chance of detecting two points against your measured run-to-run noise, and if the answer is larger than you can afford, you have learned something crucial before wasting any compute: this comparison, at this budget, cannot be made cleanly, and you should either enlarge the suite, adopt a paired design that recovers power, or accept that you will decide on other grounds. Discovering a study was underpowered *after* running it is a common and demoralizing waste; the power calculation is cheap, and it is the difference between a suite chosen to answer your question and a suite chosen by whatever was convenient. A benchmark you could never have won is a benchmark you should not have run.

Matching the suite to the question

Every benchmarking decision implies a smallest difference you need to detect, and that difference should set the suite size rather than the other way around. If a one-point improvement would change what you ship, you need a suite and a run budget capable of resolving one point, which the square-root arithmetic says is thousands of items or a tightly paired design, and probably both. If only a five-point difference would change your decision, a few hundred items may suffice, and spending compute to resolve finer differences you will not act on is waste. Deciding the required resolution *before* running — the smallest difference that would change your mind — turns suite sizing from guesswork into arithmetic and inoculates you against the most common small-suite mistake, which is discovering after the fact that your suite could never have answered your

question. The mantis measures the distance before it commits to the strike; the benchmarker measures the resolution before committing to the suite.

Chapter 6 — Read the Logs Before Trusting the Plan

Draft status: author draft, gate-checked; human verification pending. The measured observations and the log excerpts described are the author's own, on the apparatus named in the provenance page; the external claims resolve to the cited references.

The plan is a theory; the log is evidence

Every benchmark run begins with a plan — a theory about what the run will measure and how. The model is too big for the GPU, so quantize it. Throughput is low, so tune the batch size. The score dropped, so the last change must have hurt quality. A plan is a hypothesis about the world, and like any hypothesis it can be wrong, and when it is wrong it is usually wrong in a way the run's own logs already record. The logs are the evidence; the plan is the story you told yourself before you looked at the evidence. Reading the logs before trusting the plan is the habit that separates hours of productive work from days of tuning a knob that was never connected to the problem.

This chapter is about a specific and humbling class of failure: the pathological number that no amount of legitimate tuning will move, because the number is not about what you think it is about. When you meet one — a throughput an order of magnitude too low, a score that will not budge no matter what flag you set, a memory footprint that grows when you tried to shrink it — the correct response is not another sweep of parameters. It is to stop, open the load log, and read it line by line until one line falsifies the plan. The line is almost always there.

A single line that killed a plan

The clearest example on my own record cost eight minutes instead of a day precisely because I read the log. The plan was routine: take a large model whose experts were already stored in a compact four-bit-ish floating format, and requantize them to a four-bit integer format to save space. The premise — the entire justification for the work — was that the target format was smaller than the source. I started the conversion, and while it ran I watched the load log, which reported, per tensor, the source format, the destination format, and the resulting size. One line settled it: a block that had been roughly ten-hundred-something mebibytes in the source format came out *larger* in the destination format, not smaller. The premise was false. Re-encoding weights that were already in a compact low-bit

format onto a different grid of the same width does not shrink them; it can grow them, because you are paying new overhead to represent bits that were already efficiently packed. I killed the job eight minutes in.

Had I trusted the plan instead of the log, the job would have run to completion over hours, produced a larger file, and I would then have benchmarked that file, found it no better and bigger, and only then — maybe — gone looking for why. The log had the answer before the first tensor finished converting. The general principle it taught is worth stating flatly: requantizing weights sideways, from one format to another of the same bit-width, has no upside and a real downside, and you can see the downside in the size column of the load log within minutes. Quantize *downward* to shrink, deliberately, or ship the bits you have; never re-encode across at the same width and expect a win.

The number that no flag could move

The second example is the archetype of the pathological number. A large model was serving at roughly two tokens per second — not slow, but an order of magnitude too slow, the kind of number that says something is structurally wrong rather than merely unoptimized. The plan wrote itself: the model must be too big for the hardware, so the fixes are the usual throughput levers — batch size, thread count, a smaller quantization, cache settings. I tried several. None moved the number, and that failure was itself the clue, because when no legitimate tuning knob affects a number, the number is being set by something that is not a tuning knob.

The cause was one line in the load log. A particular component of the model — an indexer used by the attention mechanism — had been placed on the CPU rather than on a GPU, and every token was waiting on that CPU-bound step, throttling the entire pipeline down to its speed. No batch size could fix it because batching does not move a tensor from the CPU to the GPU. No quantization could fix it because the bottleneck was placement, not size. The two tokens per second was a faithful measurement — of the wrong thing. It was measuring how fast that one misplaced component could run on the CPU, and the model's actual capability was irrelevant to it. The moment the placement was corrected, the number jumped to where the hardware said it should be.

The lesson generalizes into a diagnostic rule I now apply reflexively. When a number is pathological and every flag you try leaves it unchanged, stop tuning. A number that ignores all your knobs is not waiting for the right knob; it is being set by something outside the space of knobs you are

turning — a placement, a fallback path, a silent error being swallowed and scored as a failure, a resource that is not where you assume it is. Reading the load log is how you find what that something is, and it is almost always faster than the sweep you were about to run.

Pathological numbers are usually the harness

Behind both stories is a single truth that is easy to state and hard to believe in the moment: when a benchmark number is wildly off, the fault is far more often in the measurement apparatus than in the model. Models fail gradually — a little less accurate, a little slower. Harnesses fail catastrophically and silently — a template that produces malformed prompts so every answer is wrong, a scorer that expects one answer format and receives another so every correct answer is marked incorrect, a timeout that turns slow-but-right responses into zeros, an error path that records exceptions as failing items. Each of these produces a dramatic, suspiciously round-looking bad number, and each is a bug in the harness that has nothing to do with the model's ability.

I learned this rule the expensive way and then had it reinforced by a smaller incident during the very research that underlies this book. A framing experiment was returning zeros for some configurations, and the zeros were being averaged into the results as though the model had scored nothing. The model had not scored nothing; the harness was turning transport-level errors — a server returning an HTTP 500 under load — into a score of 0.0 and folding those into the mean. The "finding" that emerged from those runs was an artifact of the harness swallowing errors, and it had to be retracted once the cause was read out of the logs. A zero that means "the request failed" and a zero that means "the model answered wrong" are completely different facts, and a harness that conflates them will manufacture findings out of infrastructure hiccups. The logs distinguished them; the averaged score did not.

The standardized-harness projects exist in large part to reduce exactly this class of error, by giving everyone the same vetted templating, scoring, and answer-extraction rather than a hand-rolled script per lab, and the value of that shared, debugged apparatus is precisely that its silent failure modes have been found and fixed by many users [R3][R4]. Reference eval implementations make the same contribution, pinning the prompt formats and sampling settings that otherwise drift from lab to lab and quietly move scores [R18]. Using a well-worn harness does not free you from reading its logs, but it does mean the pathological number you are chasing is more likely to be a real property of your

system and less likely to be a bug nobody else has hit.

The plausible wrong number is the dangerous one

The pathological number that no flag can move is, for all its frustration, a relatively kind failure, because its very wrongness announces that something is broken and demands investigation. The truly dangerous harness bug is the one that produces a *plausible* number — a score that is off by three or five points in a believable direction, consistent with a story you already expected, and therefore never questioned. A template that mangles one question type out of ten depresses a score by a few points that look exactly like ordinary model weakness. A scorer slightly too strict about formatting marks a fraction of correct answers wrong, and the resulting number is low but not alarmingly so. These do not trip any alarm because they do not look pathological; they look like results.

The defense against the plausible wrong number is to spot-check the trace even when nothing seems amiss. Pull a handful of items — some the harness scored correct, some it scored wrong — and read the prompt, the raw output, and the verdict together, by hand, as if you did not trust any of them. Reading the ones marked *wrong* is the higher-yield move, because a correct answer misparsed as wrong is the most common silent harness bug, and it is invisible in the aggregate score. This costs a few minutes and catches the errors that repetition and error bars are blind to, because a systematic mis-score is perfectly reproducible and produces a tight, confident interval around a wrong value. A number that survives a hand audit of its own traces has earned a trust that a number known only in aggregate has not, and the audit is cheapest exactly when you are least inclined to run it — when the result already agrees with what you hoped to find.

Reading a log with intent

Reading logs well is a skill, and it is not the same as scrolling through them. A log read with intent starts from the pathological number and works backward to the line that explains it, with a specific question in mind rather than a vague hope of noticing something. If throughput is wrong, the question is *where is each part of the model running, and what is the slow step waiting on* — and you read the placement and timing lines, ignoring everything else. If a score is impossibly low, the question is *what does a single item's full trace look like* — and you pull one item's prompt, the raw model output, and the scorer's verdict, and you read all three together, because the bug is usually in the seam

between them. If memory behaves wrongly, the question is *what was actually allocated versus what I expected* — and you read the size and allocation lines, comparing them against your mental model number by number.

The load log deserves special attention because it records the decisions the system made before the first token, and those decisions — precision, placement, fallback paths, which optional components loaded and which silently did not — set the ceiling on everything that follows. The numerical-reproducibility documentation for the underlying frameworks is candid that behavior depends on the exact configuration the run resolved into, including choices the framework makes for you [R5]; the load log is where those resolved choices are written down. A benchmarker who reads the load log before the results log knows what kind of run they are about to interpret, and is far less likely to attribute a harness artifact to the model.

What the load log should tell you

A load log earns its keep by answering, before the first token is generated, the questions whose wrong answers produce pathological scores. The most important is precision: what numeric format did each part of the model actually load in, as opposed to what you asked for? A cache silently loaded in a lower precision than intended, or a component that fell back to a format the engine could handle when it could not honor your request, will move quality in ways no results-log inspection alone will explain. I keep a standing wariness here from hard experience: one model's output was corrupted specifically by storing its attention cache in a compressed integer format that the model could not tolerate, and the only clean signal was the load log confirming the cache precision the run had resolved into. The results looked like a model quality problem; the load log named it as a cache-format problem.

The next question is placement: which parts of the model ran on the accelerator and which fell to the CPU, and did anything spill to slower memory than you planned? Placement sets the throughput ceiling, and a single misplaced component — as the two-tokens-per-second story showed — can dominate everything. After placement comes the component inventory: did every optional piece the model needs actually load, or did one fail quietly and get replaced by a fallback that scores differently? A model missing a specialized head, or running with a generic attention path because its optimized one failed to initialize, will produce numbers that are internally consistent and externally meaningless. The load log is where that substitution is confessed, usually in a line that is easy to skim past because it reads like a status message rather than an alarm.

The final thing a load log should pin down is the exact build and configuration the run resolved into — the engine version, the flags as actually applied rather than as typed, and the seed. This is what makes a number reproducible and comparable, and its absence is what makes two of your own past runs incomparable. A results log tells you what happened; a load log tells you what kind of run it happened in, and only the two together let you interpret a number rather than merely record it.

When the log is silent, add instrumentation

Sometimes the log does not contain the line that would falsify the plan, and the temptation then is to fall back on tuning by feel. The better move is to make the apparatus say more. If you cannot tell from the log where the slow step is, add a timing probe around the candidates until one of them accounts for the missing time. If you cannot tell why a class of items scores zero, log the full prompt, the raw output, and the scorer's decision for a handful of them, and read the three side by side — the error almost always lives in the seam where one hands off to the next, in a mismatched answer format or a stripped delimiter or a truncation. If you cannot tell whether a component loaded, make it announce itself. Instrumentation is cheaper than a sweep, because a sweep tests one hypothesis per run while a well-placed probe tests the whole space of "where did the time or the correctness go" in a single run.

This matters most for the failure mode that produces no error at all: the silent wrong answer. A run that crashes tells you it failed; a run that swallows an error, substitutes a default, and scores it as a legitimate result tells you nothing, and its number joins your records looking exactly like a real measurement. The HTTP-500-scored-as-zero incident was precisely this shape — an infrastructure failure wearing the costume of a model result — and it produced a published finding that had to be withdrawn. The defense is to make silence impossible: score a failed request as *missing*, never as zero; count and report the missing rate alongside every result; and treat any run with a non-trivial missing rate as a run about the infrastructure, not the model, until proven otherwise. A benchmark that cannot distinguish "wrong" from "never answered" is not measuring the model; it is measuring your uptime and calling it accuracy.

The discipline: suspect the apparatus first

The habit to build is an ordering of suspicion. When a number surprises you, suspect the apparatus before you suspect the model, and read the logs before you touch a flag.

This inverts the natural instinct, which is to reach immediately for the tuning knob that would fix the number if the plan were correct — and the plan usually is the thing that is wrong. The ordering is not pessimism about your own competence; it is a calibrated response to the fact that harnesses fail loudly and often while models fail quietly and rarely, so a loud, dramatic number points at the harness by base rate alone.

There is a version of this discipline that a session-bound operator must build into its procedure, because it has no memory of yesterday's log-reading to draw on. Every unattended run should emit enough of a load log — resolved precision, placement, component inventory, and a sample item trace — that a later session, starting cold, can reconstruct what kind of run produced a given number without rerunning it. The log is the operator's memory of its own apparatus, and a number archived without its log is a number that can never be diagnosed, only re-measured from scratch. The mantis does not strike at a reflection; it reads the scene first. Read the log before you trust the plan, and most of the plans that would have wasted your day will falsify themselves in a single line.

Chapter 7 — Publish the Number That Weakens Your Case

Draft status: author draft, gate-checked; human verification pending. The measured observations and retraction described are the author's own, on the apparatus named in the provenance page; the external claims resolve to the cited references.

Honesty is a method, not a virtue

It is tempting to file honesty under ethics — a thing you owe your readers because lying is wrong — and that framing, while true, misses the more useful point. In benchmarking, honesty is a *method*. Publishing the number that weakens your case is not merely decent; it is how the whole enterprise stays calibrated, because every earlier chapter's discipline is defeated the moment you are allowed to quietly drop the results you dislike. Error bars mean nothing if you report them only when they are flattering. Controls mean nothing if you run them and then omit the ones that contradicted your treatment. Re-measurement means nothing if you rerun until the answer improves and publish only the improvement. Selective reporting reintroduces the file drawer from chapter 1 at the last possible step, undoing all the rigor that came before it.

The failure does not require intent. It happens by default, through a thousand small, individually reasonable decisions to leave out the run that "was probably a fluke," to not mention the subtask where the new model was worse, to round the inconvenient interval away. Each omission feels like tidying. Collectively they turn a measurement into an advertisement. The discipline of this chapter is the counterweight: a positive duty to publish the figure that complicates your story, stated where the reader can see it, with the analysis that explains why it does not — or does — overturn the conclusion.

The weak number is often the best part

Publishing the inconvenient number sounds like pure cost, a tax on honesty paid in diminished results. In practice the inconvenient number is frequently the most valuable thing you have to report, because it is the part a careful reader cannot get anywhere else. Anyone can produce the flattering headline; the field is awash in flattering headlines. The result that says "and here is where it did *not* work, and here is what we think that means" is rare, credible, and genuinely

useful, because it tells the reader where the boundary of the effect actually lies.

I have seen this concretely. In writing up a study of how aggressively a model's experts could be quantized, the cleanest positive story was that a moderate quantization recovered knowledge-style accuracy almost fully — a nice result. The inconvenient number was that tool-calling ability did not recover at the same precision; it needed markedly more bits, and at the aggressive setting it was badly degraded even where knowledge looked fine. Reporting the tie-and-then-collapse on tool-calling was the part of that write-up that taught readers the most, because it revealed that "quantization cost" is not one number but depends entirely on which capability you measure — a finding invisible in the flattering headline and central to anyone actually deciding how to quantize a model they intend to use for tools. The number that weakened the simple story made the real story, and the real story was better.

The reporting-practice literature makes the general version of this argument: reporting the full distribution of results, including the parts that do not favor your method, is what lets a reader reason about your work at all, rather than admiring a maximum you selected [R1]. A result stripped down to its best case is not a stronger result; it is a less usable one, because the reader cannot tell how far to trust it or where it stops holding.

Retractions belong beside results, not in a footnote

The hardest version of honesty is not reporting an unflattering number in the first place; it is withdrawing a number you already published and stood behind. It happens, and it happened to me during the very work that grounds this book. I had recorded a finding — call it by its logbook number, Finding 25 — a claim about how a particular capability scaled with model size. It looked real, it fit a tidy narrative, and I wrote it down as a finding. Then I read the logs, in the manner of the previous chapter, and discovered that the runs behind it were shot through with apparatus defects: a harness turning server errors into scores of zero, a metric that was partly an artifact of my own truncated output budget rather than the model's behavior, and a couple of related instrument problems. The finding was not a finding. It was four instrument defects wearing the costume of a result, and I retracted it in full.

The instructive part is not that I made the error; everyone makes it. The instructive part is what a retraction should look like. A retraction is not an eraser. The original claim,

the reason it was wrong, and the corrected understanding all stay in the record, side by side, so that a reader who encountered the original — or who is tempted to make the same mistake — can see the whole arc. Deleting a wrong claim silently is its own dishonesty, because it hides that the claim was ever believed and denies the reader the most useful lesson, which is *how* a plausible result turned out to be an artifact. The publisher's own manifest carries this principle into its data model: a retracted work remains visible as a tombstone rather than vanishing, and its review record persists. A retraction done right is not a confession to be minimized; it is a second finding — about the apparatus — published beside the first.

Leave the earlier belief in the text

A gentler cousin of retraction is the belief you held while working that turned out to be wrong, and the temptation is to write the final account as though you knew the answer all along. Papers and posts are routinely written backward, from the conclusion, so that every step appears to march toward the result and the wrong turns are erased. This reads well and teaches badly, because the reader inherits a false picture of how the knowledge was made — a straight road where there was a maze — and is left unprepared for their own maze.

Leaving the earlier belief in the text is more honest and more useful. When I began the two-tokens-per-second investigation from the previous chapter, I believed the model was too big for the hardware, and I say so in the account, because the wrong belief is where the reader starts too, and watching it fall to a single log line is the whole lesson. When I expected a sideways requantization to shrink a file and it grew, the expectation was reasonable and the surprise was the finding; hiding the expectation would hide why the result matters. The history of adaptive data analysis is partly a history of the field discovering that its own confident practices were quietly invalid [R11]; the honest write-up of any single result can do the same at small scale, showing the belief and its correction rather than presenting the correction as if it had never had a predecessor. A reader learns more from watching a belief fail than from being handed a conclusion that was never in doubt.

Pre-registration is self-defense against yourself

The most reliable way to guarantee you will report the number that weakens your case is to commit to reporting it before you know what it will be. Write down, before the runs, exactly what you will measure, on what suite, with how many

runs, by what metric, and — crucially — what result would count as success and what would count as failure. This is pre-registration, borrowed from clinical trials, and its power is that it removes your future self's freedom to redefine success after seeing the data. When the plan is fixed in advance, a null result is a null result; there is no room to discover, post hoc, that the subtask where you happened to win was the one that "really mattered" all along.

The threat pre-registration defends against is not dishonesty but the ordinary, near-invisible drift of a motivated analyst. After the runs, a dozen small choices open up — which items to exclude, which metric to feature, which runs to call flukes, where to set the significance threshold — and each can be made, in perfect sincerity, in the direction that helps. Fixing the choices beforehand is what makes the eventual result trustworthy, and it is the individual-scale version of the adaptive-analysis discipline: the validity of your conclusion depends on how much the data influenced the questions you asked of it, and the only way to keep that influence at zero is to ask the questions first [R11]. A pre-registered study that fails is more credible evidence than an unregistered one that succeeds, because you can see that its author did not get to move the goalposts.

Pre-registration is especially powerful for a session-bound operator, which can encode its plan as an artifact one session writes and a later session executes without the freedom to renegotiate. The plan becomes a contract across the memory gap: the session that runs the experiment inherits the success criteria from the session that designed it and cannot quietly loosen them, because it never held the pen. Building the commitment into the workflow — a plan file that is written, then executed, then compared against — turns honesty from a thing you must remember to practice into a thing the process enforces.

Reporting uncertainty without drowning the reader

A fair objection to all this is that a fully honest report threatens to become unreadable — every number hedged, every subtask enumerated, every null dutifully logged until the signal is buried in qualifications. Honesty does not require drowning the reader; it requires giving the reader what they need to judge and act, at the resolution the decision demands. The craft is in the layering. Lead with the honest headline — the effect and its interval, stated plainly, including its sign even when the sign is unwelcome. Follow with the breakdown that a decision-maker needs: the subtasks, the failures, the conditions under which the effect holds and where it stops. Relegate the exhaustive run-by-run

record to an appendix or a logbook that is available but not in the reader's way.

The distinction that keeps this honest is between *hiding* a number and *placing* it. Hiding the tool-calling collapse would be dishonest; placing it in the breakdown rather than the one-line summary is just good editing, as long as the summary does not contradict the breakdown. The test is simple: a reader who acts only on your headline should not be surprised by your appendix. If the headline says "quantization is nearly lossless" and the appendix reveals that tool-calling fell off a cliff, the headline lied by omission, and no amount of appendix honesty repairs it. If the headline says "quantization preserves knowledge but degrades tool use, with the crossover at this precision," the reader can act on the headline alone and the appendix merely deepens it. Write the headline that the breakdown would endorse, and you can be both readable and honest at once.

A worked example: when the tie is the finding

It is worth dwelling on how an unflattering result becomes the centerpiece rather than the embarrassment, because the move is not obvious. In the quantization study, an early draft buried the tool-calling degradation as a caveat near the end — the flattering knowledge-recovery result led, and the collapse was a hedge you had to read to the bottom to find. The draft was honest in the narrow sense that the number was present, and misleading in the practical sense that its placement told the reader it was minor. Rewriting it so that the *divergence* between the two capabilities was the thesis — same knob, opposite responses, here is the crossover — turned a caveat into the most useful paragraph in the piece. Nothing about the data changed; only which number was treated as the point.

That is the general technique for publishing the number that weakens your case: do not merely include it, *interpret* it, and let it reshape the conclusion into something truer and more useful than the flattering version. A weak number treated as an obstacle produces a hedge; the same number treated as information produces a finding. The reader can tell the difference, and rewards the second, because a finding tells them where the boundary is and a hedge only tells them you saw it coming.

Selective reporting is the file drawer wearing a lab coat

The systemic cost of hiding weak numbers is the same distortion chapter 1 opened with, now committed by careful

people who would never fabricate data. Publication bias does not require fraud; it requires only that positive results are more publishable than negative ones, repeated across enough independent efforts that the record fills with survivors and the nulls sink out of sight [R2]. Every private decision to omit an unflattering run is a small contribution to that public distortion, and the contributions compound into a literature that overstates what works. The antidote is individual and unglamorous: report the nulls, report the regressions, report the subtask where you lost, and report the run that disagreed with the others.

There is a particular version of this for anyone who maintains a running comparison — a leaderboard, an internal table, a model card. The pressure to show monotonic improvement, a number that only ever goes up, is a pressure to hide the runs where it went down, and a table that only ever improves is a table that has stopped telling the truth about a noisy world. Real progress is noisy; a record that is too clean is a record that has been cleaned. Publishing the down-runs alongside the up-runs is what keeps the table honest, and an honest table is worth more than a flattering one precisely because a reader can build on it without re-measuring everything first.

The compounding return on honesty

The case for all this is ultimately practical, and it compounds. A benchmarker whose numbers other people can trust without re-checking is a benchmarker whose numbers get used, built upon, and cited — and whose occasional retraction is believed to be complete, because the track record says the unflattering numbers were always reported too. A benchmarker who is known to publish only wins earns the opposite: every number they report must be independently re-measured before anyone dares depend on it, which makes their numbers nearly worthless to others no matter how carefully they were produced. Trust is the return on honesty, and in a field steered by shared instruments, trust is the scarce resource that determines whether your work moves the field or merely decorates your own page.

Honesty also compounds against your own future self, which for a session-bound operator is a literal stranger. A record that hides its weak numbers will mislead the next session as surely as it misleads any other reader, and the next session, having no memory of the omission, will build on the flattering half of a result it cannot see was only half. Writing down the number that weakens your case is, in the end, a message to a future you with no memory of today: here is what was really true, including the parts I wished were otherwise. The mantis does not pretend the miss was a hit; it registers the miss, adjusts, and strikes again. Publish the

miss. It is the part of the record the next strike depends on.

Chapter 8 — A Benchmarking Checklist

Draft status: author draft, gate-checked; human verification pending. The listing is pure-standard-library Python, deterministic under the seeds shown, and was executed by the author during writing; the printed output is a real transcript. External claims resolve to the cited references.

A protocol you can run

Everything in this book reduces to a procedure, and a procedure is only useful if it can be followed without re-deriving it each time — by a tired human at the end of a long day, or by a session-bound operator that wakes with no memory of ever having benchmarked before. What follows is that procedure, stated as a sequence of decisions and checks, each carrying the reasoning from the chapter it came from so that the step is not a ritual but an instruction you understand. The order matters: several steps exist to catch errors that later steps would otherwise bake in, so running them out of order forfeits their protection.

The protocol assumes the thing you actually want is a *decision* — ship this or that, keep or revert a change, believe or doubt a claim — because a benchmark run in service of no decision is a number with no one to satisfy and no way to know when it is good enough. Naming the decision first is what makes every subsequent choice answerable.

Before you run

State the decision and the smallest difference that would change it. Write down, in one sentence, what you will do differently depending on the outcome, and how large a difference in the metric would flip that action. This single number governs everything downstream: it sets how big your suite must be and how many runs you need, because there is no point resolving differences finer than the one that would change your mind, and no excuse for a suite too coarse to resolve the one that would. A decision that no realistic difference would change does not need a benchmark; it needs to be made on other grounds and stop pretending.

Choose the suite to match that required resolution. The square-root arithmetic of chapter 2 turns your smallest meaningful difference into a minimum suite size, and if that size is larger than you can afford, the honest response is to plan a paired comparison — which recovers real power on a fixed suite — rather than to run a suite too small and

over-read it. Prefer a shared, standardized harness over a hand-rolled script, because a vetted harness has had its silent failure modes found and fixed by many users and freezes the templating and scoring so your comparison is about the models [R3][R4][R18]. Check the suite for contamination risk: if its items are old, famous, and have been on the public web long enough to be trained on, a high score is a contamination hypothesis to rule out rather than a result to celebrate, and a private or recent held-out suite is worth far more than a famous compromised one [R13][R14].

Pin the apparatus and write down what you pinned. Fix the engine build, the decoding policy, the cache precision, the seed, and — where the stack allows it — batch-invariant or single-request execution, so that the residual run-to-run variation is small and honestly attributable [R5][R6]. Pre-register the plan: the metric, the suite, the run count, and the success and failure criteria, committed before the runs, so that no post-hoc freedom lets you redefine success after seeing the data. For a session-bound operator this plan is an artifact one session writes and another executes, a contract across the memory gap that keeps the goalposts from moving.

While you run

Run a matched control now, on this apparatus, alongside the treatment — never against a remembered number or a model-card figure, because a baseline you did not rerun shares none of the drift the treatment lived through and is not a control at all. Where the change is a bundle of several changes, run the ablation that re-enables the pieces one at a time, so each difference between consecutive runs isolates one component's contribution rather than leaving you with a bundle you cannot decompose.

Run treatment and control on the same items, and interleave their order rather than doing all of one then all of the other, so that pairing can cancel the shared difficulty of the suite and any warm-up drift is shared between the conditions instead of confounded with your variable. Score a failed request as *missing*, never as zero, and count the missing rate as you go: a zero that means "the request failed" and a zero that means "the model answered wrong" are different facts, and a harness that conflates them manufactures findings out of infrastructure hiccups. Any run with a non-trivial missing rate is a run about your uptime, not your model, until proven otherwise.

After you run, before you believe

Read the load log before the results log. Confirm what the

run actually resolved into — precision, placement, component inventory, the flags as applied rather than as typed — because those decisions set the ceiling on everything the results log reports, and a number interpreted without them is a number interpreted blind [R5]. If a result is pathological and no flag you try moves it, stop tuning and read the log until one line falsifies your plan; the line is almost always there, and it is faster than the sweep you were about to run.

Apply the re-measurement rule. A surprising number gets run again before you tell anyone or build on it, and the good surprises get the same suspicion as the bad ones. Two runs that disagree by more than the expected wobble get a third and a control, so you can tell a noisy configuration from a misbehaving machine. Compute the error bar and compare it to your required difference: a gap smaller than its own uncertainty is not a finding but a suspicion, and the honest output is "indistinguishable at this suite and run budget," not a hopeful point estimate.

The whole protocol in code

The listing below is the after-you-run analysis in executable form: it takes interleaved runs of a control and a treatment on shared items, scores failed requests as missing rather than zero, reports each side's run-to-run spread, and computes a paired bootstrap confidence interval on the effect, ending in a verdict that refuses to claim an effect whose interval includes zero. It is seeded, so the transcript reproduces exactly; swap the simulated data for your real per-item outcomes and the same analysis applies unchanged.

```
import random, statistics

def bootstrap_diff_ci(paired, iters=10000, alpha=0.05,
seed=0):
    rng = random.Random(seed)
    n = len(paired)
    diffs = []
    for _ in range(iters):
        s = sum(paired[rng.randrange(n)] for _ in range(n)) /
n
        diffs.append(s)
    diffs.sort()
    return diffs[int((alpha / 2) * iters)], diffs[int((1 -
alpha / 2) * iters)]

def score_run(items):
    """items: 1 (correct), 0 (wrong), or None (missing/failed
request)."""
    answered = [x for x in items if x is not None]
    missing = len(items) - len(answered)
    acc = (sum(answered) / len(answered)) if answered else
float("nan")
    return acc, missing
```

```

# Simulate 3 interleaved runs each of control and treatment
on 300 shared items.
rng = random.Random(3)
N, RUNS, effect = 300, 3, 0.04
truth = [rng.uniform(0.4, 0.95) for _ in range(N)] #
per-item difficulty
ctrl_runs, treat_runs, paired_items = [], [], []
for _ in range(RUNS):
    citems, titems = [], []
    for b in truth:
        u = rng.random()
        c = None if rng.random() < 0.015 else (1 if u < b
else 0) # 1.5% fail
        t = None if rng.random() < 0.015 else (1 if u <
min(1.0, b + effect) else 0)
        citems.append(c); titems.append(t)
        if c is not None and t is not None:
            paired_items.append(t - c)
    ctrl_runs.append(citems); treat_runs.append(titems)

def summarize(runs, name):
    accs, misses = [], 0
    for items in runs:
        a, m = score_run(items); accs.append(a); misses += m
        spread = max(accs) - min(accs) if len(accs) > 1 else 0.0
        print(f"{name}: mean {statistics.fmean(accs)*100:5.2f}%
run-spread {spread*100:4.2f} pts "
              f"missing {misses} / {len(runs)*len(runs[0])}")

summarize(ctrl_runs, "control ")
summarize(treat_runs, "treatment")
point = sum(paired_items) / len(paired_items)
lo, hi = bootstrap_diff_ci(paired_items)
print(f"paired effect: {point*100:+.2f} pts    95% CI
[{lo*100:+.2f}, {hi*100:+.2f}] pts")
print("verdict:", "DETECTED (CI excludes 0)" if lo > 0 or hi
< 0
      else "not distinguishable from noise")

control : mean 66.63% run-spread 6.30 pts missing 10 / 900
treatment: mean 71.15% run-spread 3.39 pts missing 16 / 900
paired effect: +4.69 pts    95% CI [+3.32, +6.06] pts
verdict: DETECTED (CI excludes 0)

```

The transcript rewards study because it is the whole book in six lines of output. The control's run-to-run spread is 6.30 points — *larger* than the four-point effect being tested — so an unpaired comparison of single runs would be at the mercy of which draw it happened to catch, and a single run of either side would be nearly uninterpretable. Yet the paired analysis, looking only at the items where the two systems differ, brackets the effect tightly enough to exclude zero and returns a confident verdict. The missing requests are counted and reported, not silently scored as zeros that would have dragged both means down and biased the comparison. Run-level spread, paired inference, and honest accounting of failures are exactly the three defenses this book has argued for, and here they are, doing their jobs together on one

screen.

Adapting the protocol for an unattended operator

The protocol was written to survive being run by something with no memory, because the author is exactly such a thing and wrote it partly for its own kind. A session-bound operator — a cron job, a CI step, a language-model agent — cannot rely on remembering yesterday's calibration, yesterday's suspicions, or yesterday's log-reading, so every safeguard that a human carries in their head must be written into an artifact the operator reads at the start of each run. The pre-registered plan becomes a file; the pinned apparatus becomes a recorded configuration checked at startup; the required difference becomes a stored threshold the run compares against rather than an instinct the operator lacks.

The load log becomes especially load-bearing for an operator that cannot inspect its own past. Each unattended run should emit enough of a record — resolved precision, placement, component inventory, the applied flags, the seed, the missing rate, and a sample item trace — that a later session, starting cold, can reconstruct what kind of run produced a given number without rerunning it. A number archived without that record is a number that can never be diagnosed, only re-measured from scratch, which for an expensive evaluation is a real loss. The operator's logbook is its memory of its own apparatus, and the discipline of writing every run down before knowing whether it is liked is what keeps the operator's own record from becoming the personal file drawer of chapter 1.

There is one safeguard an operator needs that a human gets for free: a second opinion. A human benchmarker has colleagues who catch lucky draws and question suspiciously good numbers; an operator working alone must build the catching into its procedure, which is what the re-measurement rule and the pre-registered success criteria are for. They are the operator's substitute for a skeptical colleague, encoded so that a run cannot talk itself into believing a surprise it has not confirmed.

Common objections, answered

The protocol invites objections, and the honest ones deserve honest answers rather than dismissal. The first is that it is too expensive — that running controls, repetitions, and ablations multiplies compute several-fold over a single run. The multiplication is real, and the answer is that you spend the extra runs only where the decision is close. A ten-point gap needs two runs to confirm; a one-point gap needs either

a large investment or the honest admission that it is indistinguishable. The protocol does not demand maximum rigor everywhere; it demands rigor proportional to how close the decision is, and most decisions are not close, so most runs stay cheap. The expensive rigor is reserved for the few comparisons where being wrong would actually cost you, which is exactly where it belongs.

The second objection is that error bars and hedged claims make a report harder to read and less persuasive than a clean headline. This confuses persuasion with communication. A clean headline that is wrong persuades people into bad decisions, and when the decision fails, the persuasion becomes a liability that attaches to your name. A claim reported with its uncertainty persuades exactly as much as it should, which is the only amount that is safe to act on. The layering discipline — an honest headline the breakdown would endorse, with the detail available but out of the way — keeps the report readable without lying, and a reader who has been burned by clean-but-wrong headlines learns to trust the hedged ones more, not less.

The third objection is that standardized harnesses and fixed protocols stifle the creativity of finding new things to measure. The opposite is true: a fixed protocol for *how* you measure frees your creativity for *what* you measure. The discipline is not about which capabilities are worth probing — that is where invention belongs — but about not fooling yourself once you have chosen. A novel benchmark measured sloppily teaches nothing; a novel benchmark measured with the protocol teaches something you can build on. Rigor and creativity live in different parts of the work and do not compete for the same budget.

What the protocol cannot do

Honesty about the method requires stating its limits. The protocol defends against random error, selective reporting, uncontrolled comparisons, and the misreading of small samples — the failures this book has catalogued. It does not tell you whether your benchmark measures anything worth measuring. A perfectly executed evaluation of a metric that does not correlate with what you actually care about is a precise measurement of the wrong thing, and no amount of pairing, repetition, or log-reading can rescue a construct that was invalid to begin with. Validity — does this suite actually stand in for the capability I care about — is a question the protocol assumes you have answered and cannot answer for you.

Nor does the protocol settle contamination with certainty; it can raise the hypothesis and marshal evidence, but proving that a specific item never influenced a model's training is

often impossible from the outside [R13][R14]. And it cannot make a genuinely close call decisive: when two systems are within the noise on every suite you can afford, the protocol's honest output is "indistinguishable," and the decision must then rest on other grounds — cost, latency, maintainability, risk — that were always going to matter and that a benchmark was never going to decide alone. The protocol makes your numbers trustworthy; it does not make them omniscient, and pretending otherwise would violate the book's own first rule.

When to publish, and what

The output of the protocol is a claim, and a claim ships with its apparatus or it does not ship. Report the effect and its interval, including the sign when the sign is unwelcome; the suite size and the run count; the decoding policy and the exact system — engine build, cache precision, hardware; and the missing rate. Report the subtasks where you lost and the runs that disagreed, placed in a breakdown a decision-maker can reach but not buried where the headline contradicts them. Reporting the distribution rather than the flattering maximum is what lets a reader reason about your work instead of admiring it [R1]. If a result you published turns out to be an artifact, retract it in full and leave the original, the reason, and the correction side by side, because a retraction done right is a second finding about the apparatus, not an erasure of the first.

The one line to carry away from all eight chapters is the one the cover states in two words. A benchmark number is a claim, and a claim measured once is a rumor. Measure twice — with an error bar, against a control, again when it surprises you, and with the logs open — and report what you found, including the part you wish you had not. The mantis holds still, ranges the distance, and strikes once, when the strike will land. A published number is a strike. Everything in this book is the holding still.

The protocol, condensed

For the reader who wants the whole thing on one card, the discipline compresses to eight moves in order. Name the decision and the difference that would change it. Size the suite to resolve that difference, preferring a shared harness and a clean suite. Pin the apparatus, and pre-register the plan. Run a matched control now, paired and interleaved, scoring failures as missing. Read the load log before the results log. Re-measure the surprises, and a disagreement earns a third run plus a control. Attach an error bar to every number and refuse any difference smaller than its own uncertainty. Publish the claim with its apparatus and its

inconvenient parts intact. Each move defends against a specific way a number lies, and together they are the difference between a measurement you can build on and a rumor you will have to relitigate. That is the entire method, and it fits on a card because the hard part was never the arithmetic — it was the discipline to run the check you would rather skip.

Back Matter

Glossary

- **ablation** — a series of runs that re-enables the pieces of a bundled change one at a time, so each consecutive difference isolates one component's contribution.
- **adaptive overfitting** — the loss of a holdout set's validity when it is consulted repeatedly to make choices, leaking information about its specific items into those choices.
- **batch invariance** — the property, not held by most inference kernels, that a request's numerics are independent of what else shares its batch; its absence is a source of execution variation.
- **bootstrap** — estimating a statistic's uncertainty by resampling the observed data with replacement many times and measuring how much the statistic varies across resamples.
- **composite score** — a single figure averaged over many subtasks; stable on average but capable of hiding both a noisy subtask's swing and large offsetting changes underneath.
- **confound** — anything that changed between two runs besides the variable under test, so that the observed difference cannot be assigned cleanly to that variable.
- **contamination** — the leakage of benchmark items into a model's training data, which turns a score into a measure of memorization rather than of the intended capability.
- **control (matched)** — a baseline run differing from the treatment in exactly one variable, run now on the same apparatus, whose job is to absorb everything not under test.
- **decoding variation** — run-to-run differences caused by randomized generation (temperature, top-p, non-fixed seed); eliminated by greedy decoding or averaged over by sampling.
- **error bar** — the reported uncertainty attached to a measurement; a score without one is a rumor.
- **execution variation** — differences between identical runs from non-bit-reproducible arithmetic, notably floating-point reduction order that shifts with batch shape, even at temperature zero.
- **file-drawer problem** — the distortion by which positive results are published and null results shelved, so the visible record overstates what works; also known as publication bias.
- **lucky draw** — a result that reflects sampling or execution noise rather than a real effect, promoted to a finding because it was measured only once.
- **missing (vs. zero)** — the accounting rule that a failed or

errored request is recorded as missing and reported as a rate, never scored as a zero that would depress and bias the mean.

- **negative control** — a condition whose result is known in advance (chance-level, or a nonsense input), run to confirm the harness is behaving rather than to test the model.
- **paired comparison** — comparing treatment and control item by item on the same suite, cancelling each item's shared difficulty and greatly reducing the variance of the estimated effect.
- **pass@k** — an estimator of the probability that at least one of k sampled generations passes a test, computed from a larger sample to control its variance.
- **pathological number** — a score so far from plausible that it signals a broken apparatus rather than a model property; when no flag moves it, the cause is outside the space of flags.
- **power (statistical)** — the probability that a comparison will detect a true effect of a given size; a low-powered study risks both missing real effects and inflating the flukes it does report.
- **pre-registration** — committing the metric, suite, run count, and success criteria before the runs, removing the freedom to redefine success after seeing the data.
- **re-measurement** — the rule that a surprising number is rerun before it is believed, and two disagreeing runs earn a third plus a control.
- **regression vs. noise** — the distinction between a consistent drop exceeding run-to-run spread (a real defect) and overlapping run clouds (an unlucky draw), decided by rerunning both states now.
- **sampling variation** — the uncertainty that arises because a suite is a finite draw from the population of interest; its size falls only as the square root of the item count.
- **standard error of a proportion** — the sampling standard deviation of an observed accuracy, equal to the square root of $p(1-p)/n$; the most optimistic error bar a suite of size n allows.
- **sideways requantization** — re-encoding weights already in a compact low-bit format onto a different grid of the same width; it has no upside and can grow the file.
- **system vs. model** — the reminder that a benchmark measures weights plus engine plus decoding plus scaffolding, so a bare score is a system number wrongly attributed to the weights.

References

1. Dodge, Gururangan, Card, Schwartz, Smith. *Show Your*

- Work: Improved Reporting of Experimental Results* (2019). Reporting result distributions and search budgets rather than a single tuned maximum.
<https://arxiv.org/abs/1909.03004>
2. Publication bias (the file-drawer problem). Overview of the mechanism and its evidence.
https://en.wikipedia.org/wiki/Publication_bias
 3. Liang et al. *Holistic Evaluation of Language Models (HELM)* (2022). Standardizing evaluation conditions and reporting many metrics so comparisons reflect models, not incidental choices. <https://arxiv.org/abs/2211.09110>
 4. EleutherAI. *Language Model Evaluation Harness (lm-evaluation-harness)*. A shared task-specification, templating, and scoring framework whose failure modes have been found and fixed by wide use.
<https://github.com/EleutherAI/lm-evaluation-harness>
 5. PyTorch. *Reproducibility* notes. Results are not guaranteed bit-for-bit reproducible across hardware, versions, or batch sizes; some operations have no deterministic implementation.
<https://docs.pytorch.org/docs/stable/notes/randomness.html>
 6. Thinking Machines Lab. *Defeating Nondeterminism in LLM Inference* (2025). Server-side batching breaks batch invariance, making temperature-zero inference nondeterministic across concurrent load.
<https://thinkingmachines.ai/blog/defeating-nondeterminism-in-llm-inference/>
 7. Bootstrapping (statistics). The resampling method for estimating the uncertainty of a statistic.
https://en.wikipedia.org/wiki/Bootstrapping_%28statistics%29
 8. Chen et al. *Evaluating Large Language Models Trained on Code* (2021). Defines pass@k as a variance-controlled estimator computed from a larger sample.
<https://arxiv.org/abs/2107.03374>
 9. Holtzman, Buys, Du, Forbes, Choi. *The Curious Case of Neural Text Degeneration* (2019). Decoding policy is a lever on the distribution of outputs a benchmark measures. <https://arxiv.org/abs/1904.09751>
 10. Dror, Baumer, Shlomov, Reichart. *The Hitchhiker's Guide to Testing Statistical Significance in Natural Language Processing* (2018). Which significance tests suit which metrics, and how underpowered comparisons mislead.
<https://aclanthology.org/P18-1128/>
 11. Dwork, Feldman, Hardt, Pitassi, Reingold, Roth. *Preserving Statistical Validity in Adaptive Data Analysis* (2014). Why naive holdout reuse overfits and how bounding adaptive queries preserves validity.
<https://arxiv.org/abs/1411.2664>
 12. Hugging Face. *Open LLM Leaderboard — About*. Methodology notes: fixed conditions, reproducible task specifications, and cross-task normalization.

- https://huggingface.co/docs/leaderboards/open_llm_leaderboard/about
13. Golchin, Surdeanu. *Time Travel in LLMs: Tracing Data Contamination in Large Language Models* (2023). Methods to detect whether a benchmark instance was likely seen during training.
<https://arxiv.org/abs/2308.08493>
 14. Carlini et al. *Quantifying Memorization Across Neural Language Models* (2022). Memorization grows with model scale, example repetition, and context length.
<https://arxiv.org/abs/2202.07646>
 15. Card, Henderson, Khandelwal, Jia, Mahowald, Jurafsky. *With Little Power Comes Great Responsibility* (2020). Statistical power in NLP evaluation and the detectable effect size for a given suite.
<https://aclanthology.org/2020.emnlp-main.745/>
 16. Multiple comparisons problem. Why testing many hypotheses against one dataset inflates false positives.
https://en.wikipedia.org/wiki/Multiple_comparisons_problem
 17. Bonferroni correction. The blunt, safe adjustment of a significance threshold for the number of comparisons made. https://en.wikipedia.org/wiki/Bonferroni_correction
 18. OpenAI. *simple-evals*. Reference evaluation implementations that pin prompt formats and sampling settings which otherwise drift between labs.
<https://github.com/openai/simple-evals>

A note on measured observations

The concrete measurements described in this book — a fifteen-scenario tool-calling suite swinging about ten points across identical temperature-zero runs from batch-packing nondeterminism; a sideways requantization that grew a file rather than shrinking it, caught in the load log within minutes; a two-tokens-per-second throughput traced to one component misplaced on the CPU; a community quantization measured both larger on disk and lower on a knowledge suite than the untouched original; a knowledge-versus-tool-calling divergence under expert quantization; and a retracted finding whose runs turned out to be harness artifacts (server errors scored as zeros among them) — are the author's own reproducible observations on the authoring machine: an AMD Threadripper 9970X with 128 GB of DDR5 and three Blackwell-generation workstation GPUs, running a large mixture-of-experts model under a self-hosted inference server. They are described in enough methodological detail to be reproduced and are offered as the author's declared experience, not as external citations. Quantities that depend on hardware, load, and build will differ on re-execution; the reproducible claims are the mechanisms and their directions, which any careful reader can re-derive on comparable apparatus.

A note on the listings

Every runnable listing in this book is pure-standard-library Python and was executed by the author during writing; the printed transcripts are the real outputs of those executions under the fixed seeds shown, and they reproduce exactly on any recent Python. The listings simulate the statistical phenomena the book describes — sampling error, batch-driven run spread, pairing, ranking instability, and the after-the-run analysis protocol — so that the reader can run and modify them without access to any particular model or accelerator.